Project No: FP7-231620
Project Acronym: HATS

Project Title: Highly Adaptable and Trustworthy Software using Formal Models

Instrument: Integrated Project
Scheme: Information & Communication Technologies
Future and Emerging Technologies

Deliverable D2.2.b
Final Report on Feature Selection and Integration

Due date of deliverable: (T0+24)
Actual submission date: 1st March 2011

Start date of the project: 1st March 2009
Duration: 48 months

Organisation name of lead contractor for this deliverable: CWI

Integrated Project supported by the 7th Framework Programme of the EC

<table>
<thead>
<tr>
<th>Dissemination level</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PU</td>
<td>Public</td>
</tr>
<tr>
<td>PP</td>
<td>Restricted to other programme participants (including Commission Services)</td>
</tr>
<tr>
<td>RE</td>
<td>Restricted to a group specified by the consortium (including Commission Services)</td>
</tr>
<tr>
<td>CO</td>
<td>Confidential, only for members of the consortium (including Commission Services)</td>
</tr>
</tbody>
</table>
Executive Summary:
Final Report on Feature Selection and Integration

This document summarises deliverable D2.2.b of project FP7-231620 (HATS), an Integrated Project supported by the 7th Framework Programme of the EC within the FET (Future and Emerging Technologies) scheme. Full information on this project, including the contents of this deliverable, is available online at [http://www.hats-project.eu](http://www.hats-project.eu).

Deliverable D2.2.b reports an investigation of possible techniques that may help bring the abstract notion of features down to the concrete level of program code. These include platform modeling, feature modeling, views, feature Petri nets, context-oriented programming, multiple dispatch, trait-based programming and delta modeling. We summarize these techniques and refer to publications that provide more detailed exposition.

As a main contribution of this deliverable, we propose a generalization of delta modeling. The approach is influenced by earlier delta modeling publications, by the AHEAD methodology and by trait-based programming. The published delta modeling formalism gives rise to a new design methodology, which is to become part of the HATS methodology.

List of Authors

Frank de Boer (CWI)
Dave Clarke (KUL)
Michiel Helvensteijn (CWI)
Radu Muschevici (KUL)
José Proença (KUL)
Ina Schaefer (CTH)

Reader’s Guide

We now inform the reader of several decisions made with regard to the distribution of content between this deliverable and related deliverables.

In practice, Tasks 1.2 and 2.2 were conducted in close conjunction, due to the close relation between feature modeling and delta modeling. Consequently, the results of these tasks are relevant to both Deliverables D1.2 and D2.2.b. To offer a clear picture of these results, the content (where the deliverables would otherwise overlap) was split up as follows: D1.2 is devoted to the Full ABS Language description and related implementations, whereas D2.2.b describes the broader context of the research and specific scientific contributions, including those related to feature modeling.

Also, to maintain the status of a self-contained document, this deliverable includes all information reported in Deliverable D2.2.a (First Report on Feature Selection and Integration). This final report builds on the older document and includes more recent developments. Where necessary, the old information was updated to reflect the current state of the work and reorganized to streamline the structure of this document. The following is a list of major changes between D2.2.a and D2.2.b:
• The paper Abstract Delta Modeling [30] (as described in D2.2.a) was published and presented at GPCE ’10. Additionally a journal version of the paper [31] was submitted for a special issue of MSCS. The journal version introduces a formalism for nested deltas and adds new theorems and complete proofs.

• The running example from D2.2.a, and the sections drawing from it, were updated to better illustrate important issues such as conflicts, nesting and mutually exclusive features.

• D2.2.b contains information on feature modeling.

• D2.2.b contains information on feature model views.

• D2.2.b contains information on feature petri nets.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bibliography</td>
<td></td>
<td>48</td>
</tr>
<tr>
<td>Glossary</td>
<td></td>
<td>57</td>
</tr>
<tr>
<td>A</td>
<td>Abstract Delta Modeling</td>
<td>61</td>
</tr>
<tr>
<td>B</td>
<td>Abstract Delta Modeling (journal version)</td>
<td>74</td>
</tr>
<tr>
<td>C</td>
<td>Pure Delta Oriented Programming</td>
<td>122</td>
</tr>
<tr>
<td>D</td>
<td>Towards a Theory of Views for Feature Models</td>
<td>131</td>
</tr>
<tr>
<td>E</td>
<td>Reconciliation of Feature Models via Pullbacks</td>
<td>144</td>
</tr>
<tr>
<td>F</td>
<td>Feature Petri Nets</td>
<td>169</td>
</tr>
<tr>
<td>G</td>
<td>Modular Modelling with Feature Petri Nets</td>
<td>178</td>
</tr>
<tr>
<td>H</td>
<td>A Semantics for Context-oriented Programming with Layers</td>
<td>194</td>
</tr>
<tr>
<td>I</td>
<td>How should context-escaping closures proceed?</td>
<td>201</td>
</tr>
<tr>
<td>J</td>
<td>Multiple Dispatch in Practice</td>
<td>208</td>
</tr>
<tr>
<td>K</td>
<td>Implementing Software Product Lines using Traits</td>
<td>231</td>
</tr>
<tr>
<td>L</td>
<td>A Programming Language with Records and Traits</td>
<td>239</td>
</tr>
<tr>
<td>M</td>
<td>A Model-Based Framework for Automated Product Derivation</td>
<td>258</td>
</tr>
<tr>
<td>N</td>
<td>Variability Modelling for Model-Driven Development of Software Product Lines</td>
<td>267</td>
</tr>
<tr>
<td>O</td>
<td>A Programming Language for Software Product Lines</td>
<td>276</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

A software family or software product line (SPL) is a family of software systems with well-defined commonalities and variabilities. These software systems are called products. Feature models are often used to express the variability of SPLs, i.e. how different products in the product line differ in terms of features. Features represent increments in software functionality. A product is uniquely identified by a single valid feature configuration, i.e., a legal combination of features from the feature model.

It is useful to make a distinction between two phases of product line engineering. The first phase is feature modeling. This phase concretizes the relevant product functionality into features and captures the set of possible products by specifying relations between them. At this phase, a feature is merely a name and no behavior is attached to it yet. The second phase is feature integration, in which it is formally specified what each feature would contribute to the behavior of a final product and how different features may interact when they are selected together.

1.1 Feature Modeling

The purpose of feature models (Chapter 2) is to model the variability of an SPL. The variations of an SPL are modelled as separate “features”. Features and their mutual relations are represented in a feature model. A feature can represent any aspect of the system that is considered important. In contrast, invariant aspects are typically not modelled. Feature models are not intended to model the complete software development process, such as the requirements, design or implementation, but only variable aspects of the process.

As software development efforts involve increasingly larger feature models, scalable techniques are required to manage their complexity, and modularity techniques are required to cater for the different interests of the various stakeholders involved in the project. We describe how different stakeholders with common interests can describe the relationships between their set of features, and how to guarantee that the feature models described by the stakeholders are compatible and can be combined. One approach to managing both complexity and increasing modularity is using views (Section 2.2). A view of a feature model decomposes features up to some appropriate level of detail. With a view, only parts of a feature model are visible to each stakeholder, possibly with some details abstracted away.

1.2 Feature Integration

On the feature modeling level of abstraction, a feature is merely a label. To attach semantics to features, the code base for an SPL should be organized such that it is possible to relate a feature to its corresponding code. We call this Feature Integration (Chapter 3). Ideally, given a valid feature configuration, one should be able to automatically generate the corresponding product. How to organize SPL code is a non-trivial problem with multiple competing objectives, especially in situations where features can interact on an implementation level. We wish to be able to generate all possible products, preferably without
introducing code duplication or compromising modularity. Task 2.2 of HATS investigates several techniques which may help to achieve this goal. We describe them here.

*Context-oriented programming* (COP) \[68\] (Section 3.1) groups code modifications—generally methods or functions—into layers. Method calls are dispatched through a collection of layers. A collection of layers can be used to model a feature. Feature selection then corresponds to layer invocation, providing a link between the feature modeling level and the concrete executable level.

Multi-methods \[21\] and their *multiple dispatch* semantics (Section 3.2) are an alternative to the single-dispatch semantics of method calls of languages such as Java, C# and C++. Multiple dispatch uses the run time types of more than one argument to a method call to determine which method body to run. Multi-methods can be used to provide feature integration because they offer a means for refining method dispatch. A more specialised collection of multi-methods can be added to a code base to modify the semantics of the application, without having to actually modify existing code.

*Traits* \[53\] (Section 3.5), a construct for fine-grained behavior reuse in object-oriented languages, are considered as a compositional implementation technique for SPLs. This approach may be used for code reuse between different feature modules (code modules implementing specific features).

To address the challenge of modelling a software product line with multiple features, which may or may not be included in any given product, we introduce *Feature Petri Nets* \[89\]. A Feature Petri Nets is a Petri net \[59\] variant used to model the behaviour of an entire software product line. For this purpose transitions are annotated with *application conditions* \[103\], which are propositional formulas over features that reflect when the transition is enabled. The advantage of Feature Petri Nets is that they enable the superposition of the behaviour of the various products (given by different feature selections) in the same model.

*Delta modeling* \[103, 109, 18\] (Section 3.4.1) bridges the gap between features and program code. In a delta modeling code base one can distinguish between a *core* implementation, containing the code common to each product, and *deltas*, containing code specific to some feature configuration(s). Deltas can make changes to the core in order to integrate one or more features. Each delta has an application condition, specifying for which feature configurations it should be applied. When two or more otherwise independent features interact in the implementation, or when two features *should* interact, but do not do so by default, conflict resolving deltas can be used to solve the problem elegantly (e.g. with a minimum of code duplication and overspecification). For greater modularity and encapsulation, delta models can be *nested*, i.e. have a delta that is itself a delta model (Section 3.4.1).

Note that both feature petri nets and delta modeling use the notion of application condition, and in both it has the same meaning, representing the set of feature configurations to which a given element applies.

We give a more thorough description of each of these techniques in their corresponding sections. Related work is discussed at length in Section 4.3. Our conclusions are drawn in Chapter 4.

The main contribution of this deliverable is a generalization of the delta modeling technique and its formalism (Section 3.4.1), which is especially suited for efficient development of code for large SPLs. It avoids some of the disadvantages in earlier approaches and gives rise to a new design methodology for SPLs which supports isolated development of unrelated features and a modular way to express conflict resolution and desired interaction between features.

### 1.3 List of Appendices

This section lists all the papers in the appendices, indicates where they were published, and explains how each paper is related to the main text of this deliverable.

**Appendix A: Abstract Delta Modeling**

This paper \[30\] formalizes abstract delta modeling and proves various properties about it which help practical application. A specific instantiation of delta modeling is proposed: that of deltas acting on object oriented programs. We also add method wrapping and prove that unambiguity of such models is efficient to check.
We then compare our approach to other published approaches by encoding them in ours. An intuitive understanding of the concepts in this paper can be found in Section 3.4.1.

This paper was written by Dave Clarke, Michiel Helvensteijn and Ina Schaefer and was published in GPCE ‘10: Generative Programming and Component Engineering.

Appendix B: Abstract Delta Modeling (journal version)

This article [31] is an extension of [30]. Among other improvements, it adds nested delta models to the formalism and offers complete proofs of all theorems.

This article was written by Dave Clarke, Michiel Helvensteijn and Ina Schaefer and is submitted to a special issue of MSCS.

Appendix C: Pure Delta Oriented Programming

This paper [104] explores the possibility of using deltas only to model a product line, i.e., a core product is not used or presumed empty. This ensures greater modularity in the design.

This paper was written by Ina Schaefer and Ferruccio Damiani and was presented at FOSD ’10.

Appendix D: Towards a Theory of Views for Feature Models

This paper [32] introduces a theory of views for feature models, encompassing both view compatibility and view reconciliation for plugging together different views of an SPL. Views are regarded as partial abstraction functions, and our theory is based on the categorical notion of pullbacks in the categories of sets with functions and with partial functions. Information from this paper may also be found in Section 2.2.

This paper was written by Dave Clarke and José Proença and was published in FMSPLE’10: Formal Methods in Software Product Line Engineering.

Appendix E: Reconciliation of Feature Models via Pullbacks

This paper [97] is an extension of [32]. The main improvement is the generalisation of compatibility and reconciliation for multiple views. Information from this paper may also be found in Section 2.2.

This technical report was written by José Proença and Dave Clarke, and is also available at http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW601.pdf.

Appendix F: Feature Petri Nets

This paper [89] proposes two lightweight extensions to Petri nets: Feature Petri Nets provide a framework for modelling and verifying software product lines; and Dynamic Feature Petri Nets provide additional support for modelling dynamic software product lines. Information from this paper may also be found in Section 3.3.

It was written by Radu Muschevici, Dave Clarke and José Proença and was published in FMSPLE’10: Formal Methods in Software Product Line Engineering.

Appendix G: Modular Modelling with Feature Petri Nets

This paper [88] is a continuation of the work started in [89]. It proposes a way of modelling the behaviour of software product lines in an incremental, modular fashion using Feature Petri nets.

Appendix H: A Semantics for Context-oriented Programming with Layers

This article [33] explores the semantic foundations of an object-oriented incarnation of context-oriented programming using a minimal extension to Featherweight Java [71], containing layers and scoped layer activation and deactivation. Context-oriented programming is one of the techniques we have explored for realizing feature integration. Information from this paper may also be found in Section 3.1.
This paper was written by Dave Clarke and Ilya Sergey and was published in COP ’09: International Workshop on Context-Oriented Programming.

**Appendix I: How should context-escaping closures proceed?**

This article [29] explores higher-order context-oriented programming in a minimal language based on the Lambda Calculus. This paper identifies and proposes a few solutions to the problem of interpreting ‘proceed’—the command for invoking a previous version of a method—whenever it escapes the environment in which it is defined within a closure. Context-oriented programming is one of the techniques we explored for realizing feature integration. Information from this paper may also be found in Section 3.1.

This paper was written by Dave Clarke, Pascal Costanza and Éric Tanter and was published in COP ’09: International Workshop on Context-Oriented Programming.

**Appendix J: Multiple Dispatch in Practice**

This article [90] presents an empirical study on using multiple dispatch in practice, considering six languages that support multiple dispatch. The goal was to better understand the uses and abuses of multiple dispatch, in order to evaluate whether providing multiple dispatch is suitable for new programming languages, in general, and for the ABS language, in the context of the HATS project. Information from this paper may also be found in Section 3.2.

This paper was written by Radu Muschevici, Alex Potanin, Dave Clarke and James Noble and was submitted to Transactions on Software Engineering.

**Appendix K: Implementing Software Product Lines using Traits**

This paper [19] explores the use of traits [53] as a compositional implementation technique for SPLs. It presents a formal core calculus for a new class-based programming language FRTJ together with a type system to provide static guarantees on the product implementations. In FRTJ, traits are complemented by records, the counterpart of traits with respect to fine-grained state reuse. Traits are a mechanism of behavior reuse that we considered using to realize feature integration. Information from this paper may also be found in Section 3.5.

The article was written by L. Bettini, F. Damiani and I. Schaefer and was published in OOPS ’10: Proceedings of Object-Oriented Programming Languages and Systems.

**Appendix L: A Programming Language with Records and Traits**

In this paper [20] the prototypical implementation of a new class-based programming language FRTJ is described, wherein traits are complemented by records, the counterpart of traits with respect to fine-grained state reuse. Traits are a mechanism of behavior reuse that we considered using to realize feature integration.

This article is a technical report and was written by L. Bettini, F. Damiani, I. Schaefer and F. Strocco.

**Appendix M: A Model-Based Framework for Automated Product Derivation**

In this article [105] the concept of delta modeling is first introduced with the goal to define a structure for the design and the implementation of an SPL that is capable of automated product derivation. More information on delta modeling may be found in Section 3.4.

This paper was written by I. Schaefer, A. Worret and A. Poetzsch-Heffter, and was published in MAPLE ’09: Proceedings of Workshop in Model-based Approaches for Product Line Engineering.

**Appendix N: Variability Modelling for Model-Driven Development of Software Product Lines**

In this article [103] delta-modeling is shown to support the model-driven development of SPLs. Model-driven development relies on a stepwise refinement of models from one modeling level to the next. On each
modeling level, only the relevant system aspects have to be considered, so that design complexity can be reduced significantly. This technique is useful for both the design and the implementation of SPLs. More information on delta modeling may be found in Section 3.4.

This paper was written by Ina Schaefer. It was published in VaMoS ’10: International Workshop on Variability Modelling of Software-intensive Systems.

Appendix O: A Programming Language for Software Product Lines

In the article [18], the conceptual idea of delta-modeling is mapped to the programming language level by an extension of Java with core and delta-modules, called FDeltaJ. More information on delta modeling may be found in Section 3.4.

This technical report was written by Lorenzo Bettini, Viviana Bono, Ferruccio Damiani and Ina Schaefer and is also available from http://www.cse.chalmers.se/~schaefer/FDJ_TechReport.pdf.
Chapter 2

Feature Modeling

This chapter is divided into two main parts. We survey the literature on feature model notations and semantics in Section 2.1 and we introduce a theory of views for feature models in Section 2.2.

In the literature survey, we start by recalling feature models in Section 2.1.1, including the various definitions that have been given for the term “feature” and the systems for classifying such features. Section 2.1.2 surveys feature modeling concepts, such as feature attributes, properties and operations related to feature model evolution (specialisation/generalisation, views, configuration); it also briefly investigates the field of feature model analysis by presenting its objectives. Existing feature modeling languages (both textual and diagrammatic) are presented in Section 2.1.3, where an important focus is on works aimed at providing formal semantics for (initially rather informal) feature diagrams. For a more in-depth analysis of feature modeling languages and their semantics, we refer to Schobbens et al. [109]. Benavides et al. [16] provide a more comprehensive literature survey of the efforts to analyse feature models.

The second part of this chapter describes a theory of views that encompasses both view compatibility and view reconciliation for plugging together different views of an SPL. Section 2.2.2 describes the compatibility and reconciliation concepts, and Section 2.2.3 presents a more practical perspective regarding this approach. We discuss related work in Section 2.2.4.

2.1 Literature survey

2.1.1 Feature Models

Feature models (FM) were introduced with the concept of Feature Oriented Domain Analysis (FODA) [73]; their purpose is to model the variability of a software product line. The variable parts of an SPL are modelled as separate “features”. Features and their mutual relations are represented in a feature model. Features can represent any aspect of the system that is considered important. In contrast, aspects that are invariant are typically not modelled. Feature models are not intended to model the complete structure of a software system, such as the requirements, design or implementation, but only such artifacts that are variable.

Features are the building blocks of feature models, representing variable aspects of an SPL. By combining the features of a feature model in different ways, while respecting the composition restrictions (rules, constraints) imposed by the feature model, one obtains a set of products. A product is therefore a valid combination of features, and a feature model denotes the set of valid products for the SPL that the feature model represents.

As feature models are frequently used at the level of requirements engineering, features often refer to particular requirements. However they can also refer to architectural components or to pieces of code (see for example feature oriented programming [96]). This suggests that variable aspects of a product line must be traceable throughout all development phases, from requirements elicitation and specification, through system architecture and design, to implementation and testing.
Features

The original definition for feature was given by Kang et al. [73] as

“A prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems”.

The above definition was refined in subsequent works to address the needs for stakeholders other than end-users to control aspects, qualities or characteristics of a system relevant to them. Other stakeholders involved in a system development effort include, for example, software architects, developers, managers, marketing people, vendors, customers, contractors, standardisation bodies and investors. An early definition that takes stakeholders into account was given by Simos et al. [111] in the context of Organisation Domain Modeling (ODM):

“A feature is a distinguishable characteristic of a concept that is relevant to some stakeholder of the concept.”

This definition or re-phrased variations have been widely adopted by the research community.

Other definitions of “feature” include the following, some of which were taken from the overview provided by [35].

- “Anything users or client programs might want to control about a concept” [13]
- A “logical unit of behaviour specified by a set of functional and non-functional requirements” [22]
- “A product characteristic from user or customer views, which essentially consists of a cohesive set of individual requirements” [28]
- “An increment in product functionality” [14]
- “A structure that extends and modifies the structure of a given program in order to satisfy a stakeholder’s requirement, to implement and encapsulate a design decision, and to offer a configuration option” [7]
- “An aspect valuable to the customer” (an aspect of a system that a customer is willing to pay for) [100]
- “Features represent important product characteristics” [103].

Feature Classifications

In this section we survey approaches to classifying features according to various criteria.

FODA/FORM The original feature analysis and modeling proposal introduced features as a means to capture a user’s or customer’s understanding of the general capabilities of applications. It relates features to the following stages in the development process:

- capabilities of applications in a domain from the end-user’s perspective
- operating environments in which applications are operated (hardware, operating system, network, peripherals)
- application domain technology (e.g., navigation methods in the avionics domain) based on which requirement decisions are made
- implementation techniques (e.g., synchronisation mechanisms used in the design)
FODA \[73\] only details the modeling of capability-related features which are classified further into functional (related to the “services” provided by the application) operational (related to how user interaction with the application occurs) and presentation (related to what and how information is presented to the user) features.

The Feature-oriented Reuse Method (FORM) \[72\] further details the above classification, stating that “a feature model should cover all four categories of features for a domain”. The classification of capability-related features introduced by FODA is simplified, distinguishing only between functional and non-functional. The latter category covers features that describe intended use or expected performance (e.g., “For hobbyist use”, “For entrepreneurial use”, “Graphical interface”, “Textual interface”, “Maximum 100 users”).

Composition Feature diagrams are trees that are used to represent feature models graphically, with each node representing a feature. Several authors \[13, 15, 109, 117\] distinguish between the intermediate tree nodes, used purely for decomposition, and leaf nodes, which represent the assets of interest in the scope of the feature model. Only the latter are considered features in the sense that they form the set on which products are based. Different notions are used throughout the literature to distinguish between these two types of nodes:

- abstract (compound, parent feature, decomposable, non-terminal)
- concrete (primitive, child feature, subfeature, terminal)

A consequence of this classification is for instance that an “abstract” feature must have at least one mandatory subfeature – otherwise the model would allow configurations that include the abstract feature itself.

Earlier feature diagram notations (e.g., \[73, 61, 43, 4, 109\]) do not make the above distinction. Such feature diagrams can be re-factored into trees that conform to the above classification \[117\].

Czarnecki’s Classification Czarnecki \[44\] offers a finer grained classification, breaking features down into four categories that are more closely related to how features are implemented. A concrete feature is said to correspond to a single component and most readily resembles Kang’s definition of a functional feature. Aspectual features, such as logging and synchronisation, affect a number of components (and can be modularised using aspect technology). Abstract features, such as performance requirements, usually map to some configuration of components. Grouping features may represent a variation point and map to a common interface of plug-compatible components, or they may have a purely organisational purpose with no requirements implied.

2.1.2 Feature Modelling Concepts

Attributes

Features describe aspects of a system on a very abstract level, by giving only a name or keyword. Some authors therefore advocate augmenting feature models with feature attributes. Czarnecki et al. \[42\] introduced attributes as a way to represent a choice of values from a finite or an infinite domain (which defines an attribute’s type). Attributes also contribute to a more concise representation of a feature model: while it is possible to model a finite number of discrete attribute values as alternative sub-features, this can considerably increase the size of the model.

Benavides et al. \[17\] make the distinction between attributes that are directly related to a feature, and derived attributes (also called “extra-functional features”), which are attributes whose value is computed (using a formula) based on the values of other attributes. Derived attributes are also associated with a feature. Benavides \[15\] defines attributes as “any characteristic of a feature that can be measured”.

13
Properties

Properties in feature models can be seen as characteristics of products described by a product line that are consequences of fundamental configuration decisions, that is, the selection of features and assignment of values to attributes. Stakeholders might nonetheless be interested to control some properties by specifying their values or bounds. For example, the memory consumption of a program is a consequence of the number and size of features included, and therefore can be seen as a property.

The need to represent properties in feature models has been acknowledged; for example, Kang et al. [73, 72] and Chastek et al. [27] mention the need to deal with non-functional capabilities or features, such as the intended use of a product or its expected performance.

Constraints

A constraint, also called a composition rule [73], is a connective between two or more features in a feature model. Constraints effectively restrict the set of valid combinations of features in a model.

Following is a list of constraints that are typically used in feature models. Let $A, B$ and $B_1 \ldots B_n$ be features in a feature model, and given $A$ is a selected feature.

- $B$ is mandatory or required: $A \Rightarrow B$.
- $B$ is optional: $A \Rightarrow B \lor \neg B$.
- All features in the set $\{B_1 \ldots B_n\}$ are required: $A \Rightarrow B_1 \land B_2 \land \ldots \land B_n$.
- Any one feature in the set $\{B_1 \ldots B_n\}$ is required: $A \Rightarrow B_1 \lor B_2 \lor \ldots \lor B_n$.
- Any subset the feature set $\{B_1 \ldots B_n\}$ is required: $A \Rightarrow \{f \in C \mid C \subseteq \{B_1 \ldots B_n\}\}$.
- Mutual exclusion: out of two features $\{A, B\}$, at most one can be selected: $\neg (A \land B)$.

By establishing dependency relations between some features, constraints are used to define the set of valid products of a software product line.

Views

Griss et al. [61] briefly mention the advantage of having different views on a feature model: Each view is a diagram that shows the feature model at a certain level of detail. A less detailed view would show for example only the names of features, while a more detailed view would also show meta-attributes of each feature, such as a textual description, the category according to a classification system (context, representation, operational), binding times, etc.

A more radical insight is that “different stakeholders perceive differently what is variable” [94]. By considering developers and customers as the two main stakeholders in a software development effort, Pohl et al. [94] introduce the concept of internal and external variability: External variability is visible to the customer while an internal variability is hidden. Internal variability is often entailed by external variability and represents finer-grained variation points at lower levels of abstraction (e.g., technical details).

Views are also proposed to contain the overburdening complexity of large feature models in industrial settings. Using Reiser and Weber’s multi-level feature trees [98], a system is modelled by a hierarchy of feature diagrams, each is adapted to suit the focus of a particular stakeholder. This is done by cloning a “reference” feature tree, where initially all features of the new tree reference their counterparts in the reference tree. The cloned tree is then adapted by applying deviations. In this context, adding a child feature is termed refinement, while removing features is called reduction.
Refinement The notion of refinement is related to the idea of representing feature models at different levels of abstraction. Where a view abstracts away from details, a refinement introduces more details. Hoefner et al. [69], for example, define a refined feature model as a model in which each product has (at least) all the features of some product in the initial feature model, but possibly additional ones.

Evolution: Specialisation and Generalisation

It is generally recognised that a feature model will evolve over time to reflect refinements during the development process, error corrections or changing requirements. In this context, studies have focused on particular ways in which a feature model can evolve. Specialisation is hereby defined as transforming a feature model into a new feature model that describes a proper subset of the products described by the original feature model [44, 117]. Generalisation is understood as going in the opposite direction, with the new set of products describing a superset of the original. Other types of editing to feature models have been identified [117]: refactoring does not change the set of products, while arbitrary edits can add and remove products at the same time.

Configuration

A configuration is the blueprint of a valid product, an element from the set of products defined by a feature model [44, 117]. A configuration corresponds to a set of features and, in the case of extended feature models, attribute assignments. The configuration process is also referred to as full specialisation [44], since it can be seen as specialising a feature model until it loses all variability. Staged configuration [44] (also known as multi-step configuration [124]) is the process of reducing the number of potential products down to a single one in subsequent specialisation stages.

Analysis

Benavides et al. [16, Section 5] contains a comprehensive list of questions and problems that motivate the (automatic) analysis of feature models. We list some common objectives here.

Debugging Satisfiability: Does a FM contain at least one valid configuration? Does it contain contradictory constraints? Does a FM contain “dead” features?

Measuring How many valid configurations (products) are in a feature model (variation degree)? How many products include a given feature (commonality of a feature)?

Refactoring Are two FMs equivalent (notions of equivalence, redundancy, normal form)? Is a FM a generalisation/specialisation of another FM? How efficient is reasoning about FMs in the light of large and evolving models?

Configuration Which is the set of products that satisfies certain user constraints (e.g., “has feature A and not feature B”)? Does a FM reflect feature interactions?

2.1.3 Feature Modelling Languages

Graphical Representation

Feature Diagrams (FD) The most common notation to represent feature models graphically are feature diagrams (FD). An FD is a tree with features as nodes and related features connected through edges. Features and relations are decorated with additional symbols to express feature variability (for example, features can be denoted optional and combinatorial restrictions can be placed on features that share the same parent node). In most cases, the FD is superposed with a graph structure which allows the specification of “cross-tree constraints”. These are used to specify dependencies, such as mutual exclusion and requirement, between features in different sub-trees (see Section 2.1.2).
The tree structure itself is adequate to illustrate a hierarchy, with more abstract/general features near the root and more detailed/accurately specified features towards the leaves. A feature diagram can be seen as a hierarchical model of all products defined by an SPL.

Existing feature diagrams for feature modeling include FODA [73], FORM [72], FeatuRSEB [61], van Gurp et al. [44], PLUSS [54], Generative Programming [43], Extended GP notation [42], Riebisch et al. [100], Czarnecki et al.’s Cardinality-Based Feature Models [44, 45], and Benavides et al. [17]. See also Figure 2.1.

![Feature Model Genealogy](image)

Figure 2.1: Feature Model Genealogy (taken from K. Kang’s VaMoS 2010 Keynote).

**Criticism** Feature diagrams are an intuitive and simple way to represent feature models. However, it is not clear why the hierarchical aspect that comes with the tree structure is important (if feature models are about variability, not structure); what kind of relation(s) do edges represent (is-a, part-of, implication/requires). The tree structure is often incidental: the same feature model can be described by different trees. The necessity for cross-tree constraints suggests that feature diagrams are not the best way to represent a feature model (or that FDs are a poor visual notation, see [87]), e.g., expressing that features A and B are mutually exclusive can be done in at least two ways: by having A and B as two alternative features with a common parent, or as a cross-tree constraint.

**Product Tables** A different representation of product configuration space is to use simple tables [126]. Here, the available features are listed vertically and the products horizontally. Allocating features to products is done by checkmarking their intersection.

**The Orthogonal Variability Model** An Orthogonal Variability Model (OVM) is defined by Pohl et al. [94] Chapter 4] as “a model that defines the variability of a software product line”. The basic elements
of OVM are variation points and variants (which are the possible choices at each variation point). Both variation points and variants can be declared mandatory or optional. For a set of variants related to the same variation point one can define a range for the number of variants selectable from this group. Moreover, two types of constraints (“requires” and “excludes”) can be placed on pairs of variants, variation points or mixed pairs.

The OVM emphasizes a clear distinction between variation points and variants, which are structural elements similar to abstract and concrete features (Section 2.1.1). OVMs are similar to (cardinality or multiplicity based) feature models; their graphic notation (trees) is similar to that of feature diagrams.

Textual Representation

Graphical representations become hard to maintain for large feature diagrams, and cannot capture elegantly constructs like attributes and constraints, which are of utmost importance in industrial-sized feature models. To overcome these limitations, and to improve portability among different tools, textual representations were also developed for feature models.

Van Deursen and Klint developed the Feature Description Language (FDL), a textual representation of FODA-style feature diagrams. They also provide a feature diagram algebra with rules for transforming an FD into a normal form, count the number of products and check whether a given logical constraint over features satisfies a particular feature model.

More recently, TVL (text-based variability language) has emerged as a language for describing attribute-enhanced feature models. TVL presents itself as a scalable and comprehensive alternative to FDL. It uses a succinct notation, offering mechanisms for modularisation of concerns, and integrating most of the feature model constructs proposed in the last decades.

Formalisation and Semantics

Several publications report on the challenges to retrofit feature models with formal semantics.

The PReCISE research group at the University of Namur, Belgium surveyed feature diagram variants and proposed a formal semantics for each of these.

Classen et al. define a feature formally in the context of requirements engineering. Features are regarded as expressions of problems to be solved by the products of the software product line. A feature is a triplet \( f = (R, W, S) \) where \( R \) represents the requirements the feature satisfies, \( W \) the assumptions the feature makes about its environment and \( S \) its specification.

Höfner et al. define a product family algebra as a commutative idempotent semiring, with features as indecomposable elements in the algebra.

A widely adopted approach to formalisation found in literature is to transform feature diagrams to other formalisms for which reasoning techniques and tools exist. This is done mainly with the goal of providing tool-support for the analysis of feature models; Section 6 provides a classification and an overview of these approaches, of which we mention the most notable. Czarnecki et al. give a formal semantics for cardinality-based feature diagrams by defining an abstract syntax with a context-free grammar. Batory builds on the work of Mannion to connect feature models to iterative tree grammars and uses propositional formulas to express feature diagrams. Benavides et al. map feature diagrams to Constraint Satisfaction Problems (CSP) and propose using CSP tools for automatic reasoning. Höfner et al. propose to represent feature diagrams as idempotent semirings, for which they define a product family algebra. Wang et al. propose to map feature diagrams to well-defined W3C Semantic Web OWL ontologies.

2.2 Feature Model Views

Task 2.2 of the HATS project narrows the gap between the highly abstract level of feature modelling and concrete executable models. In this section we focus mainly on the first part—how to analyse and transform
abstract representations of features models—which suits the requirement analysis of product lines phase of the HATS methodology, described in Chapter 3 of Deliverable D1.1B [47]. As software development efforts involve increasingly larger feature models, scalable techniques are required to manage their complexity, and modularity techniques are required to cater for the different interests of the various stakeholders involved in the project. We describe how different stakeholders with common interests can describe the relation between their set of features, and how to guarantee that the feature models described by the stakeholders are compatible and can be combined.

Feature models are usually depicted using feature diagrams [73, 14], and a variety of increasingly expressive notations have been presented [13, 43, 64, 54, 61, 42, 44, 100]. We take the general perspective of a feature model as a collection of products [109, 17, 16]. Using categorical notions [84], we show a novel way of combining feature models that mutually refer to each other.

One approach to both managing complexity and increasing modularity is using views. A view of a feature model shows the decomposition of a concept up to a certain level of detail. With a view, only part of a feature model is visible to each stakeholder, possibly with some details abstracted away. Full details of this work can be found in papers “Towards a Theory of Views for Feature Models” [32] (Appendix D) and “Reconciliation of Feature Models via Pullbacks” [97] (Appendix E).

2.2.1 Preliminaries

We introduce some basic mathematical models and other relevant formal definitions. The notation \( \mathcal{P} F \) denotes the the power set of \( F \). Following Schobbens et al. [109], we define products are sets of features, and product lines as sets of products.

**Definition 2.2.1 (Product and Product Line)** Given a set of features \( F \). A product is defined as a subset of \( F \), and a product line is defined as a set of products for \( F \). We write \( \mathcal{F} \) and \( \hat{\mathcal{F}} \) to denote the set of all products and product lines, respectively. The set of all products and product lines are defined
\[
\mathcal{F} \overset{\text{def}}{=} \mathcal{P}(F) \quad \hat{\mathcal{F}} \overset{\text{def}}{=} \mathcal{P}(\mathcal{P}(F)).
\]

**Example 1** Consider the following set of features, associated to options available in different mobile phones:
\[
\{3g, \text{alarm}, \text{wifi}, \text{email}, \text{maps}, \text{gps}, \text{Internet}, \text{Apps}\}.
\]

As a convention, in the remaining of this document we start the name of a feature with an uppercase letter to denote the feature being an abstraction. For simplicity, we write only the first letter of each feature in true type font, and denote products by the consecutive letters of its features. For example, \( 3wA \) denotes the product \( \{3g, \text{wifi}, \text{Apps}\} \) (with an abstract feature Apps), and \( 3, 3wA, 3wem \) denotes a product line with three products.

A partial function \( f : F \to G \) from one feature set to another can be lifted to functions over products and product lines, written as \( \overline{f} \) and \( \hat{f} \), respectively.

**Definition 2.2.2 (Lifting)** Given a partial function \( f : F \to G \), define the lifting of \( f \) to products and product lines as \( \overline{f} \) and \( \hat{f} \), respectively, as follows (where \( \text{def}(f) \) is the range of \( f \)).
\[
\overline{f} : \mathcal{F} \to \mathcal{G} \quad \hat{f} : \hat{\mathcal{F}} \to \hat{\mathcal{G}}
\]
\[
\overline{f}(p) \overset{\text{def}}{=} \{ f(a) | a \in p \cap \text{def}(f) \} \quad \hat{f}(P) \overset{\text{def}}{=} \{ \overline{f}(p) | p \in P \}.
\]

Views remove distinction between features, e.g. mapping different varieties of televisions to a single feature TV, and hiding features that are not relevant for some stakeholder.

**Definition 2.2.3 (View)** For any partial and onto function \( g : F \to G \), we say that \( g \) is a feature view, \( \overline{g} \) is a product view, and \( \hat{g} \) is a product line view. Furthermore, for any \( P \in \hat{\mathcal{F}} \) we say that \( \hat{g}(P) \in \hat{\mathcal{G}} \) is a view on \( \hat{G} \). Similarly, a view of products for such a \( g : F \to G \) is defined as \( \overline{g} : \mathcal{F} \to \mathcal{G} \).
2.2.2 Compatibility and Reconciliation

Consider two stakeholders working with feature sets \( F \) and \( G \), respectively, who wish to combine their views. If the views are compatible, then it will be the case that some of the features in \( F \) are abstractions of features in \( G \), and vice versa. It may also be that some features in \( F \) do not appear in \( G \), and vice versa. This means that we can partition \( F \) and \( G \) each into two parts and construct a view between pairs of partitions, one going in each direction. The desired structure is called a view partition and is illustrated below. Here \( F_r \) and \( F_a \) partition \( F \). Similarly for \( G \). The functions \( v \) and \( w \) are the view functions, mapping more refined elements in \( F_r \) to their abstraction in \( G_a \), and from \( G_r \) to \( F_a \), respectively.\(^1\)

\[
\sigma = \begin{array}{ccc}
F_r & v & F_a \\
\hline
G_a & \rightarrow & G_r
\end{array}
\]

**Definition 2.2.4 (View partition)** Define a view partition to be a tuple \( \sigma = (F_r, F_a, G_r, G_a, v, w) \), where \( F_r, F_a, G_r, G_a \) are sets of features such that \( F = F_r \cup F_a \), \( G = G_r \cup G_a \), \( F_r \cap F_a = \emptyset = G_r \cap G_a \), and \( v : F_r \rightarrow G_a \) and \( w : G_r \rightarrow F_a \) are partial onto functions.

**Example 2** Recall the set of features from Example 1. We consider two different stakeholders, each concerned with a different subset of these features:

\[
F = \{\text{Internet, alarm, email, maps}\} \quad \quad G = \{3g, wifi, Apps, gps\}.
\]

In \( F \) we abstract 3g and wifi as Internet, and in \( G \) we abstract alarm, email and maps as Apps, and gps is not related to \( F \). The corresponding view partition, using the short notation introduced before, is

\[
\sigma = (aem, I, 3wg, A, \{a \mapsto A, e \mapsto A, m \mapsto A\}, \{3 \mapsto I, w \mapsto I\}).
\]

A view partition describes (1) which features in \( G \) are abstractions of \( F \) via the view \( v \), (2) which features in \( F \) are abstractions of \( G \) via the view \( w \), and (3) which elements from \( F \) are ignored by \( G \) and vice-versa. Given a view partition we define compatibility and reconciliation of features, products, and product lines. Feature models are compatible only if, when abstracting as many as possible elements from \( F \) and elements from \( G \) using the view functions, they yield the same models. When two models are compatible, they can be reconciled into a single model by selecting the most concrete features from each model. This selection is characterised by the pullback\(^{[3]}\) of the abstraction functions. We present here only the definitions of the operations, and the formal details can be found in Appendix D. Conditions that guarantee associativity of the reconciliation operator are presented in Appendix E. In the following we write \( v_a \) and \( w_a \) to represent the views \( v \) and \( w \) extended with the identity to all abstract elements. Let \( \sigma = (F_r, F_a, G_r, G_a, v, w) \) be a view partition with \( F = F_r \cup F_a \) and \( G = G_r \cup G_a \). We define compatibility and reconciliation as follows.

**Definition 2.2.5 (Compatibility of feature models)** For any \( a \in F, p \in \overline{F}, P \in \overline{F}, b \in G, q \in \overline{G}, \) and \( Q \in \overline{G} \), the compatibility of features, products, and product lines is defined as follows:

\[
\begin{align*}
& a \sim b \overset{\text{def}}{=} v_a(a) = w_a(b) \\
& P \sim Q \overset{\text{def}}{=} \bar{v}_a(P) = \bar{w}_a(Q).
\end{align*}
\]

**Definition 2.2.6 (Feature reconciliation)** Given features \( a \in F \) and \( b \in G \):

\[
\oplus : F \times G \rightarrow F_r + G_r
\]

\[
a \oplus b \overset{\text{def}}{=} \begin{cases} 
  a & \text{if } a \in F_r \text{ and } b \notin G_r \\
  b & \text{if } a \notin F_r \text{ and } b \in G_r \\
  \bot & \text{otherwise.}
\end{cases}
\]

\(^1\)Subscripts \( r \) and \( a \) indicate the source set (more refined features) and target set (more abstract features) of view functions of a view partition.
Definition 2.2.7 (Product reconciliation) Given products \( p \in \mathcal{F} \) and \( q \in \mathcal{G} \) such that \( p \sim q \):
\[
\oplus : \mathcal{F} \times \mathcal{G} \to \mathcal{F} r + \mathcal{G} r \\
p \oplus q \overset{\text{def}}{=} (p \cap F_r) \cup (q \cap G_r).
\]

Definition 2.2.8 (Product line reconciliation) Given \( P \in \hat{\mathcal{F}} \), and \( Q \in \hat{\mathcal{G}} \):
\[
\oplus : \hat{\mathcal{F}} \times \hat{\mathcal{G}} \to \hat{\mathcal{F}} r + \mathcal{G} r \\
P \oplus Q \overset{\text{def}}{=} \{ p \oplus q \mid p \in P, q \in Q, p \sim q \}.
\]

The reconciliation of two elements \( x \oplus y \) exists only when \( x \) and \( y \) are compatible, that is, when \( x \sim y \). Two features \( a \) and \( b \) are compatible if one feature abstracts the other. In Example 1, \( 3 \sim I \) but not \( 3 \sim A \) nor \( A \sim I \). Two products are compatible if every feature from each product has a compatible feature in the other product. In our case, \( 3wA \sim Ie \) but not \( 3wA \sim I \). Finally, two product lines are compatible if every product from each product line has a compatible product in the other product line, and the reconciliation \( \oplus \) joins every possible combination of compatible products. We exemplify the reconciliation of two product lines below.

Example 3 We reconcile the following products lines, over \( F \) and \( G \), respectively.
\[
P = \{ I, Ie, Iem \} \hspace{1cm} Q = \{ 3, g3wA \}.
\]

When calculating \( P \oplus Q \), the compatibility relation ignores the gps feature, while the reconciliation operator preserves it. \( P \sim Q \) because every product in \( F \) has a compatible product in \( G \) and vice-versa. For example, \( Ie \sim g3wA \), and \( Ie \oplus g3wA = g3we \). Thus,
\[
P \oplus Q = \{ I \oplus 3, Ie \oplus g3wA, Iem \oplus g3wA \} = \{ 3, g3we, g3wem \}.
\]

2.2.3 Constructing a View Partition

In the previous section, we assumed a view partition was given, though not necessarily the product line. Now we start from the other direction and address whether it is possible to find an appropriate view partition given a product line and two views. Let \( P \in \hat{\mathcal{E}} \) be a product line, and \( \hat{f} : \hat{\mathcal{E}} \to \hat{\mathcal{F}} \) and \( \hat{g} : \hat{\mathcal{E}} \to \hat{\mathcal{G}} \) define two views. The following condition guarantees the existence of a view partition.

Definition 2.2.9 (View compatibility) Two view functions \( f : E \to F \) and \( g : E \to G \) are compatible, denoted \( f \sim g \), iff for every \( a \in F \) and \( b \in G \),
\[
f^{-1}(a) \ # \ g^{-1}(b),
\]
where \( X \ # Y \) if \( X \cap Y = \emptyset \), \( X \subseteq Y \), or \( Y \subseteq X \).

Intuitively, Definition 2.2.9 states that \( f \) and \( g \) treat the features in \( E \) in terms of compatible abstractions. That is, feature sets \( E_a = f^{-1}(a) \) and \( E_b = f^{-1}(b) \) are either incompatible, namely \( E_a \cap E_b = \emptyset \), or one is an abstraction of the other, namely \( E_a \subseteq E_b \), or vice versa.

The following theorem states that when two views functions are compatible, then a view partition exists to make the two views of a product line compatible.

Theorem 2.2.10 Let \( \hat{f} : \hat{\mathcal{E}} \to \hat{\mathcal{F}} \) and \( \hat{g} : \hat{\mathcal{E}} \to \hat{\mathcal{G}} \) be two views. If \( f \sim g \), then there exists a view partition \( \sigma \) such that \( \forall P \in \hat{\mathcal{E}} \cdot f(P) \sim \sigma g(P) \). Specifically, the following gives such a \( \sigma = (F_r, F_a, G_r, G_a, v, w) \), where \( R = g \circ f^{-1} \):
\[
F_a \overset{\text{def}}{=} \{ x \in F \mid |R(x)| > 1 \} \hspace{1cm} G_a \overset{\text{def}}{=} R(F_r) \\
F_r \overset{\text{def}}{=} F \setminus F_a \hspace{1cm} G_r \overset{\text{def}}{=} G \setminus G_a \\
v : F_r \to G_a \overset{\text{def}}{=} \{ R(x) \mid x \in F_r \} \\
w : G_r \to F_a \overset{\text{def}}{=} \{ R^{-1}(x) \mid x \in G_r \}.
\]
Ultimately, one would expect all pairs of views to be compatible. The problem stems from the fact that the two incompatible views have different ‘vocabularies’.

Example 4 (Overlapping views) Given features \{3g, gsm, wifi, gps\}. Consider view functions:

\[
\begin{align*}
f &= \{3g, wifi \mapsto Internet; gsm, wifi \mapsto Location\} \\
g &= \{gsm, 3g, wifi \mapsto Radio; gps \mapsto gps\}
\end{align*}
\]

Observe that \(f \not\supseteq g\), because \(f^{-1}(Location) \neq g^{-1}(Radio)\) does not hold. Consequently, there is no view partition guaranteeing the compatibility of product lines obtained by \(f\) and \(g\). The cause is the following conflict:

\[
\text{Location} \leftrightarrow \text{Internet} \\
\text{gps} / \text{wifi} \leftrightarrow \text{gps} \\
\text{gsm} / \text{3g}
\]

Location is neither an abstraction of Radio nor vice versa, thus the calculations in Theorem 2.2.10 fail to yield a view partition.

Methodology We propose the following simple methodology for two different stakeholders \(F\) and \(G\) who want to develop in parallel a single product line consisting of two (or more) interconnected subsystems, without knowing a priori a global set of features common to them. Let \(F\) and \(G\) be also the sets of features that each of stakeholder is interested in. The agreement phase takes the following steps:

1. The stakeholders find feature names that intersect functionality, and store this data in a relation \(R\).
2. \(R\) is checked for compatibility between \(F\) and \(G\) by following the construction in Theorem 2.2.10 and verifying that the result is a view partition.
3. If the previous step fails, then the incompatibility must be resolved by finding the pairs of elements from \(F\) and \(G\) that are not abstractions in either directions, such as Location and Radio in Example 4, and adapting the product lines until the previous step holds.
4. For every new version of the feature model from \(F\) or \(G\)’s perspective, check their compatibility using the obtained view partition to know whether a reconciliation of the two of them exists.

The final reconciliation can be either calculated for the full product line feature model, or alternatively by combining per-product in both \(F\) and \(G\)’s views.

2.2.4 Related Work

Griss [61] briefly mentions the advantage of having different views on a feature model, where views are feature diagrams that display different levels of detail. A more radical insight observed by Pohl et al. [94] is that “different stakeholders perceive differently what is variable”. By considering developers and customers as the two main stakeholders, Pohl et al. introduce the concepts of external and internal variability: the former is visible to the customer while the latter is hidden. Internal variability often represents finer-grained variation points at lower levels of abstraction.

Höfner et al. [69] formalise software product lines using the feature algebra model, and also describe reconciliation. Product lines are semirings with some extra properties, where features and products differ from product lines only on the properties they obey. The authors use the concrete example of sets and multisets of features to describe products, and sets of products to describe product lines, although other examples also fit their general formalisation. An abstraction of a product line can remove references to features and add new products. For example, the reduction \(\{3, g3wem\} \leadsto \{3, g3wA\}\) from Example 3 is an
abstraction only in our setting, while \( \{3, g3wem\} \rightarrow \{3, g3wem, 3\A\} \) and \( \{t, g3wem\} \rightarrow \{\emptyset, \A\} \) are abstractions only in feature algebra. In feature algebra a view is just a product line, and reconciliation of views is achieved by combining all possible products and filtering the result using a given set of requirements. Thus, reconciliation is guided by extra requirements, while in our approach reconciliation is guided by the view partition between features of the reconciled views. We explore compatibility of views, disregarded in Höfner et al.’s approach, and allow developers of different views to refer to simplified versions of each other’s views.

Existing work on views is generally presented at a different level of abstraction than our approach, such as architectural views and views on other software models [24, 113, 25, 101, 51, 102]. Solomon [113] proposes using pushouts to merge architectures when different software systems need to be merged. Other approaches use pushouts and other algebraic techniques for model synchronisation [51] and version control in model-driven engineering [102]. Curiously, our approach uses pullbacks and not pushouts for combining models. Acher et al. [1] present a technique for composing feature models out of smaller ones. Their work focuses mostly on composing feature diagrams, though some operations at the semantic level (of sets of sets of features) are provided. Our work focuses exclusively on the underlying semantics and we provide a much richer theory. Hubaux et al. [70] outline an approach to specify views in feature diagrams, where views are sets of features. They identify key problems and challenges when allowing different stakeholders to select features of a product line given a partial view of the whole system. These include how to specify views, to check view coverage (since some decisions regarding the choice of views can be deduced [37]), to deal with dependencies regarding features from different views, and to visualise views. Our approach has more semantic foundations, and complements the syntactic perspective of how to combine configurations based on views. Segura et al. [110] use graph transformations for merging feature models, at the diagrammatic level. Their merging is akin to our view reconciliation, but by performing transformations on diagrams, they also lack a clear formal semantics. The description of the synchronisation of different viewpoints represented by different models, presented by Ruiz-González et al. [101], also lacks a formal foundation, and does not apply to feature models.

Bowman et al. [25] present a framework for viewpoint consistency. Their framework covers many aspects of software models, though not feature models. Recent work along this line [49] considers model transformations across different views, where the views are represented by different kinds of models (state machines, class diagrams, etc). In our setting we work with only one kind of model.
Chapter 3

Feature Integration

3.1 Context-Oriented Programming

A promising approach of expressing variability at the code level and for providing feature integration in a relatively straightforward fashion is context-oriented programming (COP) [68]. Code modifications—generally methods or functions—are grouped into layers. Method calls are dispatched through a collection of layers, examining the most recently turned on layer first, before proceeding to earlier layers. At run-time different layers can be turned on or off, thus modifying the semantics of method dispatch by allowing method calls to be intercepted in different ways depending on the collection of activated layers.

A collection of layers can be used to model a feature [39]. Feature selection then corresponds to layer invocation, providing a link between the feature model and the concrete executable level. Layers apply modifications to the base architecture, which implements the commonality of the software product line. The base architecture includes hooks (method calls) designed specifically to be intercepted by layers, along with default behavior for cases when such method calls are not intercepted. Layers are implemented in such a fashion that they can call each other without explicitly being required to know about each other. This property is both an advantage and a disadvantage of the COP approach. It is an advantage as features can be written independently of each other, simplifying the task of feature integration. On the other hand, it may not be easy to implement layers in a desirable fashion, and problems of feature interaction may arise but may not be easy to resolve—if layers are not placed in the anticipated order, calls will not be made in the desired order. As layers can be dynamically turned on and off, COP is promising also for SPLs that evolve over time, changing software behavior at run-time. COP permits fine-grained modifications by replacing methods, along with course-grained adaptation based on layers of methods modifying many classes simultaneously.

Two research trajectories were explored in this context, resulting in the articles: “A semantics for context-oriented programming with layers” [33] (Appendix H) and “How should a context-escaping layer proceed?” [29] (Appendix I). The first article explored the semantic foundations of an object-oriented incarnation of context-oriented programming using a minimal extension to Featherweight Java, containing layers and scoped layer activation and deactivation. A type system ensuring that no dispatched method call goes unbound was also presented. In the second article we explored higher-order context-oriented programming in a minimal language based on the Lambda Calculus. This paper identified and proposed a few solutions to the problem of interpreting “proceed”—the command for invoking a previous version of a method—whenever it escapes the environment in which it is defined within a closure. Together these results could provide a starting point for the foundations of feature integration in ABS based on ideas from context-oriented programming. What remains to be done is to integrate these notions with the delta modeling approach, which offers greater flexibility.

In addition to these fundamental results, we experimented with a more behavioral variant of COP, wherein the layers and the base architecture were represented as expressions in a process algebra, and composition of layers consists of parallel composition followed by renaming and hiding of internal names in
the process algebra. This opens the possibility of using tools such as the mCRL toolkit \cite{mCRL} for simulating and analysing behavior models of software product lines. The approach also enables the analysis of open and closed models of SPLs, depending upon whether the system is closed or not by applying the renaming and hiding operations. A related approach due to Classen et al. uses a variant of labelled transition systems called *feature transition systems* \cite{Classen2003} to specify multiple systems simultaneously. In contrast our approach based on the COP way of structuring an application, Classen et al. enable only the specification of a whole product line as a single entity.

### 3.2 Multi-methods

Multi-methods \cite{multi-methods} and their multiple dispatch semantics are an alternative to the single-dispatch semantics for method calls of imperative, object-oriented languages such as Java, C# and C++. Multiple dispatch uses the run-time types of more than one argument to a method call to determine which method body to run. While several languages over the last 20 years have provided multiple dispatch (e.g., Lisp (via CLOS), Dylan, Cecil, Diesel, Nice, MultiJava), most object-oriented languages still support only single dispatch—forcing programmers to implement multiple dispatch manually when required. The paper “Multiple Dispatch in Practice” \cite{multi-dispatch-in-practice} (Appendix J) presents an empirical study of the use of multiple dispatch in practice, considering six languages that support multiple dispatch. The goal was to better understand the uses and abuses of multiple dispatch, in order to evaluate whether providing multiple dispatch is suitable for new programming languages, in general, and for the ABS language, in the context of the HATS project.

Multi-methods can be used to provide feature integration because they offer a means for refining the method dispatch. A more specialised collection of multi-methods can be added to a code base to invasively modify the semantics of the application, without having to actually modify existing code—only the collection of modules compiled together needs to be changed.

As a way of achieving feature integration, multi-methods offer a rather fine granularity when controlling the code base, per method dispatch, though they cannot be used to replace methods nor add code to within method bodies. Coarser granularity could be achieved using higher level constructs such as modules. The main concern with multiple dispatch semantics is that very little control is provided to achieve precisely the desired semantics, in particular, if this is to be based on a given feature selection. Indeed, we are unaware of whether a language implementing multi-methods has been used for software product line engineering.

Multi-methods are not often used in practice, mainly because the languages implementing multi-methods are not yet mainstream. Even in libraries and compilers for programming languages including multi-methods, the use of multi-methods is not particularly extensive. In conclusion, multi-methods offer a flexible possible mechanism for implementing software product lines, but general experience with their use in general is missing, and they may well not offer the degree of flexibility and control required for the ABS language.

### 3.3 Feature Nets

To gain traction in the specification of adaptive systems such as product lines before committing language features to ABS, KUL conducted research into models lying in the middle ground between design formalisms and programming tools, by building on the well-known Petri Nets formalism and extending it for applicability in the SPL domain. The central idea is to provide a Petri net variant capable of modeling the set of system behaviours represented by an SPL in a single, concise Petri Net model.

Petri nets are extended in two steps. We start by guiding the execution of a Petri net based on a selection of features, and later introduce a mechanism to update the feature selection based on the execution of the Petri net. We call the first model *Feature Nets* (FN), and the second *Dynamic Feature (Petri) Nets* (DFN). (Dynamic) Feature Nets open the way for efficient analysis and verification of entire SPLs using the arsenal of formally funded techniques available for Petri nets. Future work will investigate the automated translation of feature nets into ABS, and provide a higher level design notation useful for the initial prototyping phase of a product line, before breaking it down into more specific core ABS and delta modules.
This section summarises the research that went into developing (dynamic) feature nets. Full details of this work can be found in papers “Feature Petri Nets” [89] (Appendix F) and “Modular Modeling with Feature Petri Nets” [88] (Appendix G).

### 3.3.1 Feature Nets

Petri nets [59] are used to specify how systems behave. Figure 3.1 presents an example of a Petri net for a coffee machine. This has a capacity for \( n \) coffee servings; it can brew and dispense coffee, and refill the machine with new coffee supplies. If we now add a Milk feature, we need to adapt the Petri net, not only to include the functionality of adding milk, but also to control whether or not this feature is present in the resulting software product.

![Figure 3.1: Petri net model of a basic coffee machine that can only dispense coffee.](image)

To address the challenge of modeling a software product line with multiple features, which may or may not be included in any given product, we introduce Feature Petri Nets [89], or Feature Nets (FN) for short. A Feature Net is a Petri net variant used to model the behaviour of an entire software product line. For this purpose transitions are annotated with application conditions [103], which are propositional formulas over features that reflect when the transition is enabled. The advantage of Feature Nets is that they enable the superposition of the behaviour of the various products (given by different feature selections) in the same model.

![Figure 3.2: Feature Net of the product line \{\{Coffee\}\}, \{\{Coffee, Milk\}\}\] showing each product in its initial state. Each transition has an application condition attached.

Figure 3.2 exemplifies a Feature Net of a coffee machine with a milk reservoir. It considers a product line whose products are over the set of features \{\{Coffee\},\{Coffee, Milk\}\}. The conditions above each transition reflect that the three transitions on the left-hand side can be taken when the Coffee feature is present and the two transitions on the right-hand side can be taken only when the feature Milk is present. Observe that the
restriction of this example net to the transitions that can fire for feature selection \{Coffee\} is exactly the Petri net in Figure 3.1 after removing unreachable places.

In the corresponding technical report \cite{89} two semantic definitions of Feature Nets are presented. The first semantics directly models the FN for a given feature selection. The second semantics, which we recall here, is given by projecting the FN for a given feature selection onto a Petri net with only the transitions enabled. These two notions have been shown to coincide \cite{89}.

We start with some necessary preliminaries, first by defining multisets and basic operations over multisets, then by defining Feature Nets and their behaviour. Our terminology is standard for Petri nets \cite{50}.

**Definition 3.3.1 (Multiset)** A multiset over a set \( S \) is a mapping \( M : S \to \mathbb{N} \).

We view a set \( S \) as a multiset in the natural way, that is, \( S(x) = 1 \) if \( x \in S \), and \( S(x) = 0 \) otherwise. We also lift arithmetic operators to multisets as follows. For any function \( \odot : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \) and multisets \( M_1, M_2 \), we define \( M_1 \odot M_2 \) as \( (M_1 \odot M_2)(x) = M_1(x) \odot M_2(x) \).

**Definition 3.3.2 (Application condition \cite{103})** An application condition \( \varphi \) is a propositional formula over a set of features \( F \), defined by the following grammar:

\[
\varphi ::= a \mid \varphi \land \varphi \mid \neg \varphi,
\]

where \( a \in F \). The remaining logical connectives can be encoded as usual. We write \( \Phi_F \) to denote the set of all application conditions over \( F \).

**Definition 3.3.3 (Satisfaction of application conditions)** Given an application condition \( \varphi \) and a set of features \( FS \), called a feature selection, we say that \( FS \) satisfies \( \varphi \), written as \( FS \models \varphi \), iff

\[
\begin{align*}
FS \models a & \quad \text{iff } a \in FS \\
FS \models \varphi_1 \land \varphi_2 & \quad \text{iff } FS \models \varphi_1 \text{ and } FS \models \varphi_2 \\
FS \models \neg \varphi & \quad \text{iff } FS \not\models \varphi.
\end{align*}
\]

**Definition 3.3.4 (Feature Net)** A Feature Net is a tuple \( N = (S, T, R, M_0, F, f) \), where \( S \) and \( T \) are two disjoint finite sets, \( R \) is a relation on \( S \cup T \) (the flow relation) such that \( R \cap (S \times S) = R \cap (T \times T) = \emptyset \), and \( M_0 \) is a multiset over \( S \), called the initial marking. The elements of \( S \) are called places and the elements of \( T \) are called transitions. Places and transitions are called nodes. Finally, \( F \) is a set of features and \( f : T \to \Phi_F \) is a function associating each transition with an application condition from \( \Phi_F \).

Without \( f \) and \( F \), a Feature Net is just a Petri net. Sometimes we omit the initial marking \( M_0 \). We write \( \varphi_t \) for \( f(t) \), the application condition associated with transition \( t \). For conciseness, we say that a feature selection \( FS \) satisfies transition \( t \) whenever \( FS \models \varphi_t \).

**Definition 3.3.5 (Marking of a Feature Net)** A marking \( M \) of a Feature Net \((S, T, R, F, f)\) is a multiset over \( S \). A place \( s \in S \) is marked iff \( M(s) > 0 \).

**Definition 3.3.6 (Pre-sets and post-sets)** Given a node \( x \) of a Feature Net, the set \( ^{\bullet}x = \{ y \mid (y, x) \in R \} \) is the pre-set of \( x \) and the set \( x^{\bullet} = \{ y \mid (x, y) \in R \} \) is the post-set of \( x \).

**Definition 3.3.7 (Enabling)** A marking \( M \) enables a transition \( t \in T \) if it marks every place in \(^{\bullet}t\), that is, if \( M \geq ^{\bullet}t \).

We now define the behaviour of Feature Nets for a given feature selection.
Definition 3.3.8 (Transition occurrence rule for FN) Given a Feature Net $N = (S, T, R, M_0, F, f)$ and a feature selection $FS \subseteq F$, a transition $t \in T$ occurs, leading from a state with marking $M_i$ to a state with marking $M_{i+1}$, denoted by $M_i \xrightarrow{t} M_{i+1}$, iff the following three conditions are met:

$$M_i \geq \bullet t \quad \text{(enabling)}$$
$$M_{i+1} = (M_i - \bullet t) + t^* \quad \text{(computing)}$$
$$FS \models \varphi_t \quad \text{(satisfaction)}$$

The transition rule for FN is used to define traces that describe the FN’s behaviour. We now define the semantics of a Feature Net by projecting it onto a Petri net for a given feature selection.

Definition 3.3.9 (Projection) Given a Feature Net $N = (S, T, R, M_0, F, f)$ and a feature selection $FS \subseteq F$, the projection of $N$ onto $FS$, denoted by $N \downarrow FS$, is a Petri net $(S, T', R', M_0)$, with $T' = \{ t \in T \mid FS \models \varphi_t \}$ and the flow relation $R' = R \cap ((S \cup T') \times (S \cup T'))$.

One projects $N$ onto a feature selection $FS$ by evaluating all application conditions $\varphi_t$ with respect to $FS$ for transitions $t \in T$. If $FS$ does not satisfy $\varphi_t$, then $t$ is removed from the Petri net. All application conditions are also removed when projecting.

The behaviour of a Petri net $N = (S, T, R, M_0)$ is given by the set of all of its traces [59], written $\text{Beh}(N) = \{ M_0 \xrightarrow{t_1} \cdots \xrightarrow{t_n} M_n \mid M_i \subseteq S, i \in 1..n, M_{i-1} \xrightarrow{t_i} M_i \}$. We define the behaviour of a FN by considering all possible feature selections.

Definition 3.3.10 (FN Behaviour) Given an FN $N = (S, T, R, M_0, F, f)$, we define $\text{Beh}(N)$ to be the union of the behaviours for all feature selections over $F$:

$$\text{Beh}(N) = \bigcup_{FS \subseteq F} \text{Beh}(N \downarrow FS).$$

Lemma 1 (FN behaviour is the set of all traces [89]) The behaviour $\text{Beh}(N)$ of a Feature Net $N$ coincides with the set of all its traces constructed following the transition occurrence rule for Feature Nets (Definition 3.3.8).

3.3.2 Dynamic Feature Nets

Assuming that a product is composed from a static selection of features is sometimes too restrictive. As an example, we can think of a modular appliance, some of whose features can be disabled temporarily. For example, a coffee machine using fresh milk instead of milk powder allows the removal of the milk reservoir, in order to store it in the fridge. This change in the hardware configuration may entail a change in the software configuration. Modeling the presence/absence behaviour of the Milk feature may entail a significant modeling effort.

Figure 3.3: DFN modeling the ability to connect/disconnect a feature at runtime.
To model this kind of dynamic feature reconfiguration, we introduce Dynamic Feature Nets (DFN). DFN associates simple update expressions to transitions. Upon firing of a transition, updates affect the feature selection in effect.

In our example, switching the *Milk* feature on and off can be modelled by the DFN in Figure [3.3] as an independent addition to the model in Figure [3.2]. Associated to the disconnect transition is the update expression “*Milk* off”. By firing the disconnect transition, the current feature selection is updated, dropping the *Milk* feature. This action globally disables all transitions whose application condition depends on the *Milk* feature (that is, add milk and refill milk in Figure [3.2]). Conversely, firing the connect transition re-enables all transitions conditioned on the *Milk* feature.

The feature reconfiguration model can remain disconnected from the “functional” model if the user interaction of removing/reconnecting the *Milk* feature can occur independently of the state the coffee machine. Alternatively, we can assume that the reconfiguration of features depends on the functional model. Figure [3.4] shows a model where removing/reconnecting the milk reservoir is only allowed when the machine is in a waiting state, prohibiting, for example, its removal when the machine is in the process of brewing coffee.

**Figure 3.4:** DFN (initial state) of a dynamically reconfigurable product line. Whenever transition disconnect fires, feature *Milk* is switched off, disabling all transitions that are conditioned on *Milk*. It is enabled again by firing connect.

We extend Feature Nets to capture the dynamic reconfiguration of products, resulting in a more general Petri net model. In our approach we associate to each transition an update expression that describes how the feature selection evolves after the transition. The resulting model is called Dynamic Feature Nets (DFN). DFN extend Feature Nets by adding a variable feature selection to the state of the Petri net, and associating application conditions and update expressions over the feature set to the transitions. This extension enables more concise descriptions of systems based on feature models, without adding expressive power with respect to Petri nets. We now define update expressions before formalising DFN.

**Definition 3.3.11 (Update)** An update is defined by the following grammar:

\[ u ::= \text{noop} \mid a \text{ on} \mid a \text{ off} \mid u; u \]
where \( a \in F \) and \( F \) is a set of features. We write \( U_F \) to denote the set of all updates over \( F \).

Given a feature selection \( FS \in F \), an update expression modifies \( FS \) according to the following rules:

\[
\begin{align*}
FS & \xrightarrow{noop} FS \\
FS & \xrightarrow{a\ \text{on}} FS \cup \{a\} \\
FS & \xrightarrow{a\ \text{off}} FS \setminus \{a\} \\
FS & \xrightarrow{u_0} FS' \\
FS' & \xrightarrow{u_1} FS''
\end{align*}
\]

**Definition 3.3.12 (Dynamic Feature Net)** A Dynamic Feature Net is a tuple \( N = (S, T, R, M_0, F, f, u) \), where \((S, T, R, M_0, F, f)\) is an FPN and \( u \) is a function \( T \rightarrow U_F \), associating each transition with an update from \( U_F \).

We write \( u_t \) to denote the update expression \( u(t) \) associated with a transition \( t \).

**Definition 3.3.13 (DFN transition occurrence)** Given a DFN \( N = (S, T, R, M_0, F, f, u) \) and an initial feature selection \( FS_0 \subseteq F \), a transition \( t \in T \) occurs, leading from a state \((M_i, FS_i)\) to a state \((M_{i+1}, FS_{i+1})\), denoted \((M_i, FS_i) \xrightarrow{t} (M_{i+1}, FS_{i+1})\), iff the following four conditions are met:

\[
\begin{align*}
M_i & \geq \cdot t \quad \text{(enabling)} \\
M_{i+1} & = (M_i - \cdot t) + t^* \quad \text{(computing)} \\
FS_i & \models \varphi_t \quad \text{(satisfaction)} \\
FS_i & \xrightarrow{u_t} FS_{i+1} \quad \text{(update)}
\end{align*}
\]

**Definition 3.3.14 (DFN trace)** Given a DFN \( N = (S, T, R, M_0, F, f, u) \), the behaviour \( N \) exhibits by assuming a sequence of states \((M_0, FS_0) \ldots (M_n, FS_n)\), where each change of state is triggered by a transition occurrence \((M_i, FS_i) \xrightarrow{t_i} (M_{i+1}, FS_{i+1})\), is called a trace. A trace is written \((M_0, FS_0) \xrightarrow{t_0} (M_1, FS_1) \xrightarrow{t_1} \ldots \xrightarrow{t_{n-1}} (M_n, FS_n)\).

If we consider all possible traces, we obtain the behaviour of the FPN.

**Definition 3.3.15 (DFN Behaviour)** Given a DFN \( N = (S, T, R, M_0, F, f, u) \), we define \( \text{Beh}(N) \) to be the set of all traces starting with the initial state \((M_0, FS_0)\).

### 3.3 Modular modeling

For a modeling formalism to be scalable and useful in practice, it needs to facilitate a modular, incremental workflow. This is especially important for modeling software product lines: a single SPL model combines the behaviour of a set of different systems, which are often too complex to develop simultaneously.

In the following we propose two refactoring techniques that use FNs in a modular way. Both are based on the idea of modeling features as individual subsystems and later combining them to obtain a model of the entire SPL. They differ in how these sub-models are composed together. In the first approach, a single node (place or transition) is refined into a more complex FN. The second approach is based on fusing common nodes from different subsystems.

We show how each of the two techniques can be applied to construct a coffee machine product line with the three features \{Coffee, Payment, Milk\} from separate nets modeling individual features.
Compositional modeling using refinement

In the first modeling stage, each feature’s behaviour is expressed by a separate FN. The feature’s interaction with the rest of the system is modelled by the flow of tokens to an abstract node. A feature modelled using this technique can be seen as a partially specified model of the entire SPL, where the feature’s behaviour is fully specified, whereas everything else is underspecified. Composition then entails refining the abstract node into the fully specified system.

(a) The Coffee feature is at the basis of a coffee machine product line.

(b) Milk feature

(c) Payment feature

Figure 3.5: Individual Feature Nets modeling the features Coffee, Payment and Milk.

The three features of our example coffee machine are modelled as separate FN (Figure 3.5). Unless a feature’s behaviour is self-contained, it will typically interact with other features that are part of the larger system. To faithfully model such interactions we include an abstract representation of the larger system in the shape of a node. For example, the model of Milk in Figure 3.5 reflects the fact that adding milk depends on a state of the system in which a cup of fresh coffee is available. The larger system, which is not part of the feature itself, is represented abstractly by the place coffee; a token in this place would denote a state in which a freshly brewed cup of coffee is available. Similarly, Figure 3.5 models the fact that after a payment has been accepted, the overall system is able to proceed with performing other actions such as brewing and serving drinks.

Constructing a model of the whole SPL is done by stepwise refining the abstract nodes of each feature. The intuition behind node refinement is that each abstract node is replaced with a more complex Feature Net. In our example, the first step could be to refine Payment’s PROCEED transition by replacing it with the Feature Net for Coffee. In a second step, the feature Milk is refined by replacing the place coffee with the net obtained in the previous step. We now formally define refinement for places and transitions.
Refinement of Feature Nets  Notions of refinement of Petri nets are well-known techniques \[59\] that help managing the complexity of large models. Based on these, we give a definition of refinement of Feature Petri Nets, which will be used to merge separately developed Feature Nets. Let \( N_1 = (S_1, T_1, R_1, M_0, F, f) \) and \( N_2 = (S_2, T_2, R_2, M_0, F, f) \) be two Feature Nets with \( S_1 \cap S_2 = T_1 \cap T_2 = \emptyset \). Adapting Definition 2.4.1 of \[59\], we define the refinement of \( N_1 \) by \( N_2 \) by replacing a transition or a place.

**Definition 3.3.16 (Transition refinement)** Given the Feature Nets \( N_1 \) and \( N_2 \), a transition \( t \in T_1 \), and a relation \( B \subseteq (S_1 \times T_2) \cup (T_2 \times S_1) \), the transition refinement of \( N_1 \) by \( N_2 \) is the Feature Net \( N = (S, T, R, M, F, f) \), where

\[
S = S_1 \cup S_2 \\
T = (T_1 \cup T_2) \setminus \{t\} \\
M_0 = M_{01} \cup M_{02}
\]

\[
R = (R_1 \setminus \{(s,t) | s \in S_1 \} \cup \{(t,s) | s \in S_1\}) \cup R_2 \cup B \\
f = \{t' \mapsto f_1(t) \land f_2(t') | t' \in T_B \} \cup (f_1 \cup f_2) \setminus (T \setminus T_B)
\]

**Definition 3.3.17 (Place refinement)** Given the Feature Nets \( N_1 \) and \( N_2 \), a place \( s \in S_1 \), and a relation \( B \subseteq (T_1 \times S_2) \cup (S_2 \times T_1) \), the place refinement of \( N_1 \) by \( N_2 \) is the Feature Net \( N = (S, T, R, M, F, f) \), where

\[
S = (S_1 \cup S_2) \setminus \{s\} \\
T = T_1 \cup T_2 \\
M_0 = (M_{01} \cup M_{02}) \setminus \{s\} \text{ and } f = (f_1 \cup f_2) \setminus T.
\]

In **Definition 3.3.16** \( B \) relates places from \( N_1 \) and transitions in \( N_2 \) while in **Definition 3.3.17** \( B \) relates transition from \( N_1 \) and places in \( N_2 \). These relations are used to replace the arcs between nodes of \( N_1 \) and the refined node \( (t \text{ or } s) \) with arcs between nodes of \( N_1 \) and nodes of \( N_2 \).

A refinement replaces an abstract node with the FN model of the overall system. When the node being replaced is a transition, its application condition is used in conjunction with the application conditions of the transitions in \( B \). We now show how the refinement operations are applied in the examples above to construct a model of an SPL from separate sub-models. First, a model with the two features \( \text{Coffee} \) and \( \text{Payment} \) is composed. We use the following assignment.

\[
N_1 \text{ is the FN modeling } \text{Payment} \text{ (Figure 3.5a)} \\
N_2 \text{ is the FN modeling } \text{Coffee} \text{ (Figure 3.5b)} \\
t = \text{PROCEED} \\
B = \{(\text{PAID, BREW COFFEE}), (\text{SERVE, UNPAID})\}.
\]

![Diagram](https://via.placeholder.com/150)

Figure 3.6: A software product line over feature set \{\text{Coffee, Payment}\} obtained by composing the features \text{Coffee} (Fig. 3.5a) and \text{Payment} (Fig. 3.5b) by refinement of \text{Payment}'s \text{PROCEED} transition.
Refining the transition \( t \) using Definition 3.3.16 produces the FN shown in Figure 3.6. Subsequently, we can refine the Feature Net representing Milk to include the behaviour obtained in the previous step by using the assignment below.

\[
N_1 \text{ is the FN modeling Milk (Figure 3.5b)}
\]

\[
N_2 \text{ is the FN obtained in the previous step (Figure 3.6)}
\]

\[
s = \text{COFFEE}
\]

\[
B = \{(\text{ADD MILK, READY}), (\text{READY, ADD MILK})\}.
\]

Following Definition 3.3.17, refining the place \( s \) produces an FN representing the behaviour of a product line over the three features, as shown in Figure 3.7.

Figure 3.7: FN model of an SPL over the feature set \{Coffee, Payment, Milk\} obtained by composing models of individual features.

Compositional modeling using node fusion

This subsection introduces an alternative modular modeling technique based on the fusion of nodes from separate Feature Nets. In the first modeling stage, Feature Nets representing individual features are designed such that they also model their interaction with other features. Such interactions are modelled by common nodes, which can be seen as an interface to the net. Feature Nets are then composed by fusing their common nodes.

Unlike the refinement approach presented in the previous section, nets modeling individual subsystems now “know” part of the structure of the nets with which they interact. Previously this knowledge was provided by the \( B \) relation upon composition. Now it is included earlier, when modeling the Feature Nets to be composed. This approach makes composition a simple matter of fusing the Feature Nets based on the union between their elements; the composition operation is commutative and associative.

**Definition 3.3.18 (Feature Net fusion)** Let \( N = (S,T,R,M_0,F,f) \) and \( N' = (S',T',R',M'_0,F',f') \) be two Feature Nets such that \( \forall s \in S \cap S' : M_0(s) = M'_0(s) \) holds. Define Feature Net fusion \( N \oplus N' = (S \cup S', T \cup T', R \cup R', M_0 \cup M'_0, F \cup F', f^\dagger) \) where \( f^\dagger = \{ t \mapsto f_1(t) \wedge f_2(t) \mid t \in T \cap T' \} \cup f_1 \mid (T \setminus T') \cup f_2 \mid (T' \setminus T) \).
Two Feature Nets can be fused by constructing a net based on the union of their elements. Places which appear in both nets must have matching markings; the application conditions of common transitions are joined by conjunction.

![Diagram](image.png)

Figure 3.8: The highlighted transitions overlap with the transitions of the same name in Fig. 3.5a. Merging the two nets is achieved by fusing these transitions.

To illustrate a modeling technique based on Feature Net fusion, we consider the model for a Payment feature shown in Figure 3.8. Its interaction with the rest of the system is specified explicitly by modeling the flow of tokens to and from concrete transitions which are part of the Coffee feature (Figure 3.5a). Fusing these two Feature Nets via the common transitions BREW COFFEE and SERVE leads to the same composition that was reached using the refinement technique (Figure 3.6). To also include the Milk behaviour, we first construct a Feature Net similar to the one in Figure 3.5b, only that the place labelled COFFEE has to be renamed to READY. Then, composing (Coffee ⊕ Payment) ⊕ Milk results in the net shown in Figure 3.7.

### 3.3.4 Behaviour preservation

We have shown two approaches to compose a Feature Net \( N \) over features \( F \) with a second net to obtain a new larger net \( N' \) over features \( F' \). A natural question arises: is the behaviour of \( N \) preserved in \( N' \)? We formalise two notions of preservation: (1) by considering the behaviour of \( N \) and \( N' \) when projected onto each feature selection \( FS \subseteq F \) (Definition 3.3.19); and (2) by considering traces of \( N \) and \( N' \) projected onto feature selections from \( FS' \subseteq F' \) after filtering out elements of traces not involving a transition from \( N \) (Definition 3.3.20).

The major difference between these two notions is that the first one considers only the feature selections relevant for the original Feature Net, ignoring feature selections relevant for the extension. The second notion also takes into account such feature selections, but ignores the behaviour that does not involve transitions of the original Feature Net.

Recall the two composition examples presented in this section: adding Payment to the Coffee Feature Net, and adding Milk to the resulting net. In both cases we want the composition to preserve behaviour. That is, if we take the net modeling Coffee and Payment and block all transitions related to the Payment feature, the result should be a model of a machine which dispenses coffee for free. Similarly, if we take the net that was extended with the Milk option and disregard the transitions conditioned on Milk, then its behaviour should be that of a machine which takes a payment and serves plain black coffee.

We first formalise the preservation of behaviour, and then analyse the composition examples with respect to their ability to preserve behaviour in more detail.

**Definition 3.3.9** specified how to project a Feature Net over a set of features \( FS \). We say that an extension of a Feature Net \( N = (S, T, R, F, f) \) to a new net \( N' = (S', T', R', F', f') \) preserves projections if projecting onto any feature selection from \( F \) before and after the extension yields the same behaviour.
Definition 3.3.19 (Preservation of projection) The extension of $N$ to $N'$ preserves projections iff $\forall FS \subseteq F : \text{Beh}(N \downarrow FS) = \text{Beh}(N' \downarrow FS)$.

Recall the composition of the nets modeling Coffee and Payment, and the composition of the resulting net with the net modeling Milk. Let $C$, $M$, $P$, $CP$, and $CPM$, be the nets involved in these operations, presented in Figures 3.5a, b, c, 3.6 and 3.7 respectively. Extending $CP$ with $M$ preserves projection, because $\forall FS \subseteq \{Coffee, Milk\} : \text{Beh}(CP \downarrow FS) = \text{Beh}(CPM \downarrow FS)$. However, the extension of $C$ with the net $P$ does not preserve projection. This is because $\{\text{wait,full}\} \xrightarrow{\text{brew coffee}} \{\text{ready,full,refillable}\}$ is a trace of $C \downarrow Coffee$, but not of $CP \downarrow Coffee$, as the latter requires a token in the place paid in order to fire the transition BREW COFFEE. Because projecting does not restore the structure the net had before the extension—the places UNPAID and PAID are not eliminated by the projection—its behaviour is different.

Before formalising preservation based on restricted traces, we define the restriction of traces of a Feature Net to a given set of transitions $Ts$.

Definition 3.3.20 (Trace restriction) Given a Feature Net $N = (S,T,R,F,f)$ and a set of transitions $Ts$, the behaviour of $N$ restricted to $Ts$ is defined as $\text{Beh}(N) \downarrow Ts = \{tr \mid Ts \mid tr \in \text{Beh}(N)\}$, where the restriction of a single trace to $Ts$ is the sequence defined below:

$$M \downarrow Ts = \varepsilon \quad \quad (M \xrightarrow{tr} tr) \downarrow Ts = \begin{cases} t \cdot (tr) \downarrow Ts & \text{if } t \in Ts \\ tr \downarrow Ts & \text{otherwise.} \end{cases}$$

When restricting a trace to a set of transitions $Ts$ we obtain a sequence of transitions after filtering out the elements not in $Ts$. To check whether the extension of a Feature Net $N = (S,T,R,F,f)$ to a new Feature Net $N' = (S',T',R',F',f')$ preserves its restricted traces, we compare the traces before and after the extension. We consider feature selections that use features from the composed net, and compare only the transitions that $N$ can perform.

Definition 3.3.21 (Preservation of restricted traces) The extension of $N$ to $N'$ preserves restricted traces iff $\forall FS' \subseteq F' : \text{Beh}(N \downarrow FS') = \text{Beh}(N' \downarrow FS') \downarrow T$.

The extension of the net $C$ modeling Coffee with the net $P$ modeling Payment net from Figure 3.6 does not preserve projections, as seen before. When considering the restricted traces for projections over the full set of features $F'$, we reach the same conclusion: the behaviour is still not preserved. The projection $C \downarrow \{Coffee, Payment\}$, when restricted to the transitions of $C$, cannot perform the transition BREW COFFEE anymore, because the place UNPAID will never contain a token. However, $CP \downarrow \{Coffee, Payment\}$ can brew coffee after accepting the payment. This shows that, when using a transition refinement, extra attention needs to be paid to avoid discarding desired behaviour. We now show how to adapt the net modeling the Payment feature to preserve the behaviour based on restricted traces after composition; as future work we plan to explore the use of application conditions on places or on arcs between places and transitions to simplify the design of behaviour preserving transformations.

We present in Figure 3.9 the composition of the net $C$ modeling the feature Coffee with an alternative net $P'$ modeling Payment. The only difference is the transition skip, which overrides the need for paying when the feature Payment is not available. Let $CP'$ be the net from Figure 3.9. The projection $CP' \downarrow Coffee$ now yields the same traces as $C \downarrow Coffee (= C)$, after filtering all transitions not used by $C$, since tokens can flow from UNPAID to PAID with and without the Payment feature. Observe that the projection is still not preserved, since the transition skip is present in the traces of $CP' \downarrow Coffee$, but not in the traces of $C \downarrow Coffee$.

3.3.5 Related Work

Our research relates to a number of areas, specifically Petri net based formalisms, and the behavioural specification of software product lines. We highlight the most relevant works in these fields.
A range of Petri net extensions based on a modified transition occurrence rule have been proposed \cite{2, 40, 81}. Unlike our approach, these formalisms generally exceed the expressive power of Petri nets. As a general consequence, properties such as reachability, boundedness and liveness are often not decidable for these extensions, and they generally lack the full range of mathematical tools that are available to analyse normal Petri nets. Inhibitor arc Petri nets \cite{2} can test whether a place is empty by conditioning transitions on the absence of tokens. By modeling individual features as places, the presence or absence of tokens could represent whether a feature is on or off. An application condition could be encoded by including feature places in the pre-sets of transitions, thereby conditioning its firing on the presence or absence of features. Compared to our approach, this entails a more complex net, with unclear boundaries between the functional and structural aspects. Inhibitor arc Petri nets are strictly more expressive than Petri nets. Conditional Petri nets \cite{40} associate a condition to each transition characterising when the transition is enabled. But unlike our approach, the condition is over the sequence of transitions fired in the past. In contrast, FPN are closer to the application domain of variability modeling: through application conditions they refer directly to the feature model of the SPL. Conditional Petri nets \cite{40} associate a transition with a formal language over transitions. Extending the classical occurrence rule, a transition is enabled only if the sequence of transitions that occurred in the past is in that language. An FN could be encoded as a conditional Petri net by encoding application conditions in a language over the alphabet of transitions. In Open Petri Nets \cite{10}, places designated as open represent an interface towards the environment. Open nets are composed by fusing common open places, and the composition operation is shown to preserve behaviour with respect to an inverse decomposition operation. The main difference to our model, apart from the presence of application conditions, is that we also allow transitions to guide the composition.

Various formalisms have been adopted for specifying the behaviour of software product lines, with the aim of providing a basis for analysis and verification of such models. UML activity diagrams have been used to model the behaviour of SPL by superimposing several such diagrams in a single model \cite{41}. Attached to the activity diagram’s elements are “presence expressions,” which are similar to application conditions. Models of products are obtained using model-to-model transformation by evaluating the presence conditions in the light of a given feature configuration. Compared to activity diagrams, Petri nets have a stronger formal foundation, with a larger spectrum of analysis and verification techniques, although, several studies have expressed the semantics of UML diagrams using Petri nets (e.g., \cite{56}). Gruler et al. extended Milner’s CCS with a product line variant operator that allows an alternative choice between two processes \cite{63}. This calculus, referred to as PL-CCS, includes information about variability: by defining dependencies between features, one can control the set of valid configurations.

Variability is often modelled using transition systems enhanced with product-related information. Modal transition systems (MTS) \cite{79} allow optional transitions, lending themselves as a tool for modeling a set of behaviours at once \cite{58}. Generalised extended MTS \cite{55} introduces cardinality-based variability operators and proposes to use temporal logic formulas to associate related variation points. Asirelli et al. encode
MTS using propositional deontic logic formulas \[8\]. Modal I/O automata \[80\] are a behavioural formalism for describing the variability of components based on MTS and I/O automata. Mechanisms for component composition are provided to support a product line theory. These approaches do not relate behaviour to elements of a structural variability model. Featured transition systems (FTS) \[36\] are an extension of labeled transition systems. Similar to Feature Nets, transitions are explicitly labeled with respect to a feature model, and a feature selection determines the subset of active transitions. In FTS, transitions are mapped to single features. Transition priorities are used to deal with undesired non-determinism when selecting from transitions associated to different features. With application conditions, priorities are no longer required because we can negate the features in other transitions to obtain the same effect.

### 3.4 Delta Modeling

Delta modeling \[105, 103, 18, 30, 31\] is a modular compositional approach \[77\] for representing feature-based variability on the modeling and on the implementation level during the development of software product lines. Delta modeling bridges the gap between the high-level abstract language of features and the concrete level of executable models and code. In Lopez-Herrejon et al. \[82\], the notion of program deltas is introduced to describe the modification of an object-oriented program, for example, by the introduction of new fields or extension of methods.

#### 3.4.1 Abstract Delta Modeling

For the complete formalism and description of abstract delta modeling, we refer to \[30, 31\]. In this section, we clarify the intuition by means of an example.

**Example**

We introduce a product line of code editors. We describe the features of the code editor product line and then show its realization by deltas.

**Feature model** The Editor product line is described by the sets of valid feature configurations that can be realized by different editors. Figure 3.10 depicts the feature model \[73, 120\] (see also Chapter 2) of the Editor product line. In the diagram, every box represents a feature. A line with an open circle at the end stands for the optional subfeature relation. A horizontal line segment indicates exclusive choice between subfeatures. A subfeature can only be selected if its base feature is also selected. The Editor product line has the following features:

![Feature model diagram](image-url)

Figure 3.10: Feature model of the Editor product line.
Figure 3.11: Graphical representation of the delta model for the Editor product line.

- **Editor** (Ed) is the only mandatory feature of the product line representing basic editing functionality.
- **Printing** (Pr) allows the user to print the content of an editor window.
- **Syntax Highlighting** (SH) colors code for easier recognition of programming language constructs.
- **Error Checking** (EC) performs simple grammatical analysis on code and underlines certain errors. Hovering over an error gives extra information in a tooltip.
- The optional subfeature **Semantic Analysis** (SA) of the feature **Error Checking** can detect more sophisticated errors in program code.
- **Tooltip Info** (TI) gives information about code fragments by hovering over them. The features **Tooltip Info** and **Error Checking** are mutually exclusive, since both produce different kinds of tooltips.

This product line consists of 16 editors, because there are 16 possible feature configurations.

**Delta Model** Features on the level of the feature model are only labels [41]. In order to describe the realization of the different editor variants on the implementation level, we provide a delta model. Figure 3.11 depicts the delta model for implementing the Editor product line. The dashed boxes represent the deltas. Every delta contains class additions and/or updates. Deltas are decorated with their application conditions (bottom right of each box) which consists of a propositional logic formula, linking the delta modifications to the set of feature configurations. The arrows between the deltas represent the application ordering as a strict partial order. If two deltas are not ordered, it is an indication from the designer that the order in which the deltas are applied should not matter.

For the Editor product line, we assume that product generation starts from the empty product. Thus, the core product is an empty program. This way, the Editor product line is more robust to evolution of the feature model, e.g., if mandatory features become optional, as pointed out in [104]. The delta $d_1$ is applied
for any valid feature configuration, because it is annotated only with the mandatory Ed feature. It provides the basic functionality of an editor and adds this to the empty product. Deltas $d_2, \ldots, d_5$ implement the four optional features of the product line, Pr, SH, EC and TI. They are applied if their corresponding feature is selected. Delta $d_6$ implements subfeature SA of feature EC. It is applied if SA is selected, which requires EC to be selected as implied from the feature model. Thus, it is not necessary to annotate delta $d_6$ with feature EC. Delta $d_6$ not only modifies the Editor class, but adds a new class as well.

Assume that the original designer of delta $d_1$ did not think ahead and expressed both color and underlining of text with the single method font. Since $d_3$ and $d_4$ both redefine this method independently (the two deltas are not ordered), we need a conflict-resolving delta $d_8$ to combine their functionality. Delta $d_8$ is selected only if the two conflicting deltas are also selected and is applied afterwards (by the partial order) so it can resolve their conflict and provide the appropriate semantics for the combination of features SH and EC. A similar conflict is present between deltas $d_8$ and $d_6$, so we include a conflict-resolving delta $d_9$ to fix the interaction.

Even if two deltas are not in conflict, it might be necessary to add another delta to appropriately combine their functionality. For instance, when the Pr and SH features are selected together, we want any printout to contain the syntax highlighting colors as well. Delta $d_7$ implements the desired interaction between these two features. It overwrites the print method and uses the syntax highlighting information of delta $d_3$.

Note that we do not include a conflict-resolving delta for the apparent onMouseOver conflict between $d_4$ and $d_5$. This is because the EC and TI features are mutually exclusive in the feature model. Since the two deltas are never applied together, there is no conflict to resolve.

**Refactoring** Assume that delta $d_4$ should be refactored into two deltas $d_1^4$ and $d_2^4$, where the former handles the font method and the latter handles the onMouseOver method. In order not to introduce any extra orderings to the delta model, we decide to nest the two deltas obtained from the refactoring into a new delta model (see Figure 3.12).

From this example, we observe that nesting allows greater modularity and structure in delta models. Furthermore, nested delta models are more expressive than previously introduced 'flat' delta models, since certain sets of derivations can be expressed by nested delta models, but cannot be expressed with flat delta models.

### 3.4.2 Delta Semantics

Deltas do not exhibit any behavior on their own: deltas are combined using syntactical operations on, for example, (parts of) classes, and it is these classes which exhibit behavior (semantics). The behavior of the classes is specified using the techniques such as those described in Chapter 4 of deliverable D1.2 [48]. Let $C$ be a class with behavioral specification $S$, and let $B(C) = S$ be the behavior of the class $C$. The
semantic characterization of a delta $\delta$ on classes can be seen as a delta $\delta'$ on specifications iff $\delta'$ satisfies the reuse contract: $B(\delta(C)) = \delta'(S)$ which justifies reuse of the contract $S$ and class $C$. The semantic characterization of deltas and reuse contracts with deltas will be explored further in the context of Tasks 2.4 “Types for Variability” and 2.6 “Refinement and Abstraction”.

3.4.3 Related Work

In delta modeling the variability of an SPL, which is abstractly represented by a feature model [73], was, in earlier publications, captured by a core product and a set of product-deltas that are specified in terms of product models or code. The core product would represent a member of the software family for a valid feature configuration. The feature configuration for the core product would not be uniquely determined, but could be chosen by the developer of the software family. However, recent work on pure delta oriented programming [104] has shown how and why to avoid a semantically rich core product and use deltas even for mandatory features. Product-deltas specify changes to the core product or to other product-deltas to implement all members of the software family. The changes defined in a product-delta are additions, removals and modifications of elements contained in the core product or other product-deltas. An application condition, a constraint over the features in the feature model, is attached to every product-delta. This constraint determines for which feature configurations the changes specified by the product-delta have to be applied. An application condition does not necessarily refer to only one feature, but can denote any set of combinations of features. This allows dealing with combinations of features explicitly. A product for a specific feature configuration is obtained by applying the changes specified in the product-deltas with a valid application condition to the core product.

In the articles of Schaefer et al. [105, 103] the changes defined in the applicable product-deltas are reordered so that firstly all additions, secondly all modifications, and finally all removals are applied. However, this does not prevent that two modifications target the same element. By refining the product-deltas and their application conditions, conflicting combinations can be explicitly handled. However, this may lead to a number of product-deltas that is equal to the number of products in the software family. To counter this problem, a partial order between product-deltas is introduced to determine an ordering in which two conflicting modifications are to be carried out [18]. In order to derive a product for a particular feature configuration, the set of applicable product-deltas is arranged in a linear order compatible with the partial order. The product-deltas are then applied to the core product in this linear order where the changes of one product-delta are applied simultaneously. As a result, each product-delta itself may not modify the same element twice in order to avoid conflicts. After a further loosening of restrictions [30], it is now possible for unordered deltas to modify the same element, as long as a conflict resolving delta follows to mediate between and unify the two original deltas.

The delta modeling approach represents the relation between the variability specified by feature models [73], and the behavioral specification and implementation of a software family in a flexible and modular way. If the core product is a complete, valid product, it can be developed with single application engineering techniques and verified using existing standard verification techniques. The core product is not uniquely determined, which allows a very flexible design of the software family. The general structure of the application conditions allows combinations of features to be explicitly covered. This can, for instance, be used to solve the optional feature problem [74] where two optional features require additional functionality to interact. This interaction functionality can be described with an additional product-delta that is only applied for the considered combination of features. This may be seen as a sort of conflict resolving delta [30].

The principles of delta modeling are not limited to a particular modeling or implementation language. Delta modeling can be instantiated to any concrete language by defining the semantics of the change operations in the product-deltas in terms of the concrete language constructs. As a result, the granularity of the modifications described by a product-delta depends on the underlying modeling or implementation language of the products and on the semantics of the modification operations used in the product-deltas. Although delta modeling is a compositional approach to represent product line variability, according to the classification presented by Kästner et al. [77], the granularity of the modifications can be on the same level.
as annotative approaches, for example, if the modifications specified in the product-deltas are implemented by frames (cf. [105]). Because of this flexibility, delta modeling can easily be instantiated to represent feature-based variability in the ABS language on coarse-grained as well as fine-grained levels. The modular and compositional variability structure expressed within the delta modeling approach can further serve as a foundation for the development of efficient verification techniques for software families.

In “A Model-Based Framework for Automated Product Derivation” [105] (Appendix M), the concept of delta modeling is first introduced with the goal to define a structure for the design and implementation of a software family that is capable of automated product derivation. The delta modeling approach is instantiated for a variant of UML class diagrams to represent the design of a software family and for the JCoBox [106] programming language for its implementation. The application of the change operations defined in product-deltas is automated using Frame technology [11], so that it is also possible to insert single statements, for example, into existing constructors, or to add single parameters to a parameter list.

In “Variability Modelling for Model-Driven Development of Software Product Lines” [103] (Appendix N), delta modeling is shown to support the model-driven development of software families. Model-driven development relies on a stepwise refinement of models from one modeling level to the next. On each modeling level, only the relevant system aspects are considered, so that design complexity is reduced significantly. For software families, the feature-based variability specified in feature models can first be captured on a very abstract modeling level using core and delta-models. This variability structure is then preserved under model refinement by refining core and delta-models independently. The application conditions for the delta-models remain the same. If model refinement is compatible with the changes specified in the delta-models, then delta-application and model refinement commute, providing an incremental model-based development process for software families.

In “A Programming Language for Software Product Lines” [18] (Appendix O), the conceptual idea of delta modeling is mapped to the programming language level by extending Java with core and delta-modules. This extension is called FDeltaJ. A core module implements the core product as a set of standard Java classes. The product-deltas are implemented as delta-modules. The changes that can be specified by delta-modules are additions, modifications and removals of Java classes. The modification of a class includes additions, removals and renaming of fields and methods as well as modifications of the parent class. The granularity of the possible modifications in FDeltaJ is the same as other compositional approaches [77], such as feature-oriented programming [12]. A delta-module has an application condition specified by a when-clause. A partial order between delta-modules can be specified as an after-clause to avoid conflicting modifications to the same class. FDeltaJ is accompanied with a constraint-based type system that allows typing core and delta-modules in isolation and ensures that the delta-application for every valid feature configuration results in a type-safe Java program.

For future work, we are aiming at providing tool support with IDE functionalities to ease the use of delta modeling during software family development. Additionally, we are planning to compare delta-oriented programming of software families in FDeltaJ with the traditional feature-oriented programming approach, as used in AHEAD [12]. In the traditional feature-oriented programming approach, a feature module, which is in one-to-one correspondence with a feature in the feature model, can only conservatively extend products, i.e., introduce new classes and refine existing ones. Hence, delta modeling is more expressive and provides higher design flexibility. However, this flexibility may induce additional complexity that has to be dealt with in a suitable manner.

3.5 Trait-based Programming

Traits have been designed to be units for fine-grained reuse in order to counter the problems of class-based inheritance with respect to code reuse. A trait is a set of methods, completely independent from any class hierarchy. The common behavior (i.e., the common methods) of a set of classes can be factored into a trait. Traits were introduced and implemented in the dynamically-typed class-based language
Figure 3.13: Code implementing the \( Pr \land SH \land EC \) scenario using traits.

Squeak/Smalltalk \[107, 53\] Also two recent programming languages, Scala \[92\] and Fortress \[3\], incorporate forms of the trait construct.

Traits are folded into a class at compile-time and typically have no run-time presence. However, they are often used in combination with subclassing to add functionality to an existing class hierarchy.

Take the example from Section 3.4.1. The product for the \( Pr \land SH \land EC \) feature configuration may be implemented as in Figure 3.13. The basic Editor class is inherited by EnhancedEditor, which is to form the basis of composition for the three features.

Each feature is written in the form of a trait, providing the required method modifications. If only one such trait existed, or just the \( Pr \) and \( EC \) traits, very little work would need to be done, since there are no name-clashes to resolve nor features to combine. But if the \( SH \) trait is pulled in together with one or both of the other traits, the composing class needs to provide glue code in order to resolve the conflict. Glue code may also be used to wrap the functionality of traits, for example, to conform to an imposed interface.

A class is able to do this by using several trait composition operators. When incorporating a trait, a class may exclude certain methods and/or rename certain methods from that trait. For instance, the EnhancedEditor class likely renames the methods it pulls in from the feature traits, and then defines new print(), font(c : int) and onMouseOver(c : int) methods, calling the renamed methods in order to use and combine their implementations.

In “Implementing Software Product Lines using Traits” \[19\] and “A programming language with records and traits” \[20\], we explore the usage of traits \[53\] as a compositional implementation technique \[77\] for SPLs. We propose the class-based programming language FRTJ in which traits are complemented by records, the counterpart of traits with respect to fine-grained state reuse. A class in FRTJ is assembled from records, traits and interfaces, which define fields, method implementations and publicly accessible methods of the class respectively. Records and traits themselves can be composed from other existing records and traits, respectively, by explicit composition operators, including symmetric sum, field/method renaming, and field/method exclusion. A product is specified by the classes it uses for providing its functionality. An SPL is then defined by its member products and its artifact base, containing interfaces, records, traits and
classes that are used to implement the member products.

In article [19] (Appendix K) we present a formal core calculus for FRTJ together with a type system to provide static guarantees on the product implementations. The FRTJ type system requires type-checking of the records, traits and classes of an artifact base only once to guarantee that the implemented products are type-safe. Furthermore, an extension of an already type-safe artifact base can be type-checked by only considering the newly added parts.

In article [20] (Appendix L) a prototypical implementation of FRTJ is described. The granularity of the differences between products in FRTJ relies on the operations available for composition and adaptation of traits. This granularity is on a sub-method level, because traits can be adapted to wrap code around existing methods. Furthermore, they can be defined to be parametric in their required methods such that they can be instantiated to call different methods in different contexts. However, it is not possible to insert a single statement into an existing method or to extend the defined method parameters in FRTJ, which is one of the limitations of compositional approaches [77].

3.6 Should Features be Traits or Something Else?

From the beginning, traits (Section 3.5) have been suggested as a fundamental technique for implementing product line variability. This section provides a summary of the eponymous presentation given at the Task 1.2 + 2.2 joint working meeting in Leuven, October 2009. Here, we explain why traits are not sufficient for implementing SPLs, but an alternative solution is required, which we provide in the form of delta modeling. The basic idea is that by design, traits require too much code duplication to be the underlying technique used for the SPL development.

We explain this informally in this section using the example from Section 3.4.1. To save space in our trait diagrams (since they would otherwise become quite large), we use only a subset of the example, in which the features \(SA\) and \(TI\) do not exist. So our feature set is \(\{Ed,Pr,SH,EC\}\).

We show that adapting the concept of traits to be suitable for our purpose requires so many changes to the way traits behave that they will not be traits anymore. This summary assumes basic knowledge of traits and the new delta modeling approach described earlier in this chapter. In short, we will attempt to use traits to implement the reduced Editor SPL code base capable of generating all 8 possible products.

Figure 3.13 shows how traits may be used to implement the \(Pr \land SH \land EC\) scenario. One class pulls in the right traits implementing the features, and locally resolves conflicts between them.

The first problem is that we do not yet have the code for the other 7 possible feature configurations. We start by compiling the classes and traits of the example conditionally, only for the relevant feature configurations. Then we introduce a refinement operator to refine the \(Editor\) class rather than subclass it. If we used a simple subclass, it would never be instantiated by existing code; we would require significant duplication of core code. Figure 3.14 both shows this class refinement operator and introduces conditional compilation of code (the dashed boxes).

In order to implement all 8 possible scenarios, we would require 7 mutually exclusive implementations of the Editor refinement, including two identical copies of the conflict resolving glue code it can contain. In general, we would need \(2^f - 1\) such class refinements, where \(f\) is the number of features.

We can improve this situation somewhat, if traits could take some control away from the composed entity (Figure 3.15). In general, if traits are able to choose to which class they apply, we would need \(2^c\) implementations of the class refinement, where \(c\) is the number of conflicts between traits. So in this scenario, we only need 4 implementations of the \(Editor\) class refinement, while all glue code is still duplicated.

We could use composite traits and method exclusion to remedy the situation (Figure 3.16). To enable this implementation, we again need to make changes to how traits behave. For example, we now have traits that are ‘pulled’, traits that are ‘pushed’ and traits that are ‘pushed if not already pulled’. Moreover, there are still problems yet to be solved; the method exclusion lists would need to be manually maintained to always name those methods in the \(T_{SH}Editor\) trait that we are not resolving.

In short, traits are not suitable for our purpose because they were designed to leave all control of
Figure 3.14: Code implementing the $Pr \land SH \land EC$ scenario. The top arrow (with the squiggly) denotes class refinement. The dashed boxes may be seen as deltas, or simply as conditional compilation.

Figure 3.15: This is Figure 3.14 but with conditional compilation, and traits able to apply themselves to the class refinement from the outside.
composition to the composed entity, and using them any other way defeats their purpose and complicates
the work. What we need is to leave control to the outside. Delta modeling does this. It allows deltas to
change the core architecture without the core having to know about it. If we implement the editor example
with deltas, we would end up with the model from Figure 3.11. Traits may still be useful in other capacities,
but not as the underlying mechanism for variability.

Figure 3.16: Composite traits with method exclusion. Require traits to be able to “apply if not pulled”.

HATS Deliverable D2.2.b Final Report on Feature Selection and Integration
Chapter 4

Conclusion

4.1 Summary

Deliverable D2.2.b consists of an investigation of possible techniques that may help bring the abstract notion of features down to the concrete level of program code. These include feature modeling and feature views for modeling the variability of product lines and feature petri nets, context-oriented programming, multiple dispatch, trait-based programming and delta modeling for feature integration. We summarize these techniques and refer to publications where more thorough investigations are provided.

Each feature integration technique, though promising in itself, has its disadvantages when using it as the underlying mechanism for variability in ABS. Context-oriented programming does not fully bridge the gap between features and code, since it requires the user to specify a sequence of layers, rather than a set of features. Additionally, it is not yet able to elegantly handle problems with feature interaction. Multi-methods are likely an inherently unnatural tool for the task. Section 3.6 explains in some detail why traits are not sufficient. Delta modeling has turned out to be most promising.

So as a main contribution of this deliverable, we propose a generalization of delta modeling. The approach is influenced by earlier delta modeling publications, the AHEAD methodology [12] and trait-based programming. As in earlier delta modeling publications, deltas may make changes to the core architecture of a product, dependent on their application condition, “from the outside” (as opposed to traits, which leave full control to the composed entity). As in the AHEAD methodology, an ordering between deltas may mediate some of the conflicts, if any, while similarly to trait based programming, other conflicts have to be resolved explicitly.

A paper presenting an abstract formalism of the new delta modeling approach has been collaboratively written by CWI, CTH and KUL [30] and a journal version of it has also been submitted [31]. The formalism gives rise to a new design methodology which allows different features to be developed in isolation; perhaps by different people at the same time (though one feature is allowed to fully depend on another). Desired feature interaction or conflict resolution may be expressed collaboratively. The technique allows a designer to specify exactly those dependencies that are necessary. This prevents over-specification (which can cause bugs that are hard to diagnose), code duplication and combinatorial explosions of boilerplate code.

4.2 Future Work

Delta modeling tool support integrated into an IDE would be a useful contribution. This might include a visualization of the partial order in the form of a graph and functionality to filter the code-view given some set of feature constraints. Also, a code-painting IDE such as CIDE [77] may be usable for the delta modeling approach in the future.

The theory could also be expanded to include support for proof reuse, and concrete formalisms for automatic generation of other material using delta modeling, such as API documentation, end user documentation and makefiles. Also, as feature models become more expressive and offer more possibilities (e.g.,
parametric features), delta modeling application conditions need to adapt to support them.

4.3 Related Work

4.3.1 Feature Modeling

Feature models were introduced with the concept of Feature Oriented Domain Analysis (FODA) [73] to model the variability of a software product line. The variable parts of an SPL are modelled as separate “features” within a feature model. Features can represent any aspect of the system that is considered important. Features are often defined as referring to particular requirements [35], architectural components or to pieces of code [96]. Various definitions of feature appear in the literature [43, 22, 28, 14, 7, 100, 103].

Feature diagrams are trees that depict how to decompose a concept into its key aspects [73, 61, 43, 4, 109, 13, 15, 117]. Feature diagrams and models also possess constraints, called composition rules by Kang et al. [73], which connect two features of a feature model. Constraints can express which features “require” another and which are “mutually exclusive”. Some authors advocate adding feature attributes to express micro-variability and thus make feature diagrams more concise [43]. Derived attributes and extra-functional properties can be expressed using constraints over attributes [17]. Existing feature diagrams for feature modeling include FODA [73], FORM [72], FeatuRSEB [61], van Gurp et al.’s [61], PLUSS [54], Generative Programming [43], Extended GP notation [42], Riebisch et al.’s [100], Czarnecki et al.’s Cardinality-Based Feature Models [44, 45], Benavides et al.’s [17], Product Maps [123], and Orthogonal Variability Models (OVM) [94, Chapter 4]. Text-based representations of feature models also exist [120, 23, 35].

Classen et al. [35] define a feature formally in the context of requirements engineering. Höfner et al. [69] define a product family algebra as a commutative idempotent semiring, and features as indecomposable elements of that algebra. Other approaches to formalisation involve transforming feature diagrams into other formalisms such as propositional logic, constraint satisfaction, OWL ontologies or context-free grammars [45, 13, 85, 17, 69, 122]. There exist surveys of feature diagram variants and their formal semantics [108, 118, 103, 67]. Benavides et al. [16] provide a comprehensive literature review of the automated analysis of feature models.

In the Feature Integration part of Task 2.2, we abstract away from any particular formalism for feature models or diagrams, since our focus is on expressing variability on the source code and modeling level. We see feature models merely as sets of sets of features (feature model \( \subseteq \mathcal{P}(\mathcal{F}) \)) [67].

4.3.2 Feature Integration

The existing approaches to represent feature-based variability at the modeling and implementation levels can be classified into two main directions [124, 77]. Annotative approaches, which specify negative variability, and compositional approaches, which capture positive variability.

Annotative approaches consider a single artifact representing all products of a product line. On the design level, variant annotations (e.g., using UML stereotypes [126, 9, 60]) define which parts of the model have to be removed to derive the model of a concrete product. Czarnecki et al. [41] associate presence conditions to modeling elements to be removed in certain feature configurations. At the implementation level, conditional compilation, frames [125] or COLORED FEATHERWEIGHT JAVA (CFJ) [75], are all instantiations of annotative approaches. The source code of the whole product line is marked with respect to product features at a syntactic level. Given a particular feature configuration, any marked code, which does not satisfy the configuration, is removed.

Most of the above approaches focus on the annotative representation of variability within a single modeling or implementation layer. In [9], the variability of different kinds of development artifacts is represented by variant annotations. These are resolved by means of textual decision models that are separated from the actual artifacts. Similarly, in the orthogonal variability model proposed in [94], the variability of various development artifacts is captured in a separate model. Variation points are linked to a set of variants which
refer to parts of the development artifacts. In order to resolve variability, the parts of the development artifacts that are not referred to by the selected variants are removed.

Compositional approaches rely on a set of model or code fragments that are associated to features and can be composed for a particular feature configuration. A prominent example of compositional approaches to represent feature-based variability is AHEAD [12], which can be applied at the design as well as the implementation level. In AHEAD, a product is built by stepwise refinement of a base module with a sequence of feature modules. In [66, 121, 91], design-level models are constructed by aspect-oriented composition techniques. Apel et al. [114] apply model superposition to compose model fragments. In [93], a product model is first obtained by model composition and is then subsequently refined by model transformation. In [65], a set of models is represented by a base model with associated variability and a set of resolution models, which determine how modeling elements of the base model have to be replaced for a particular product model.

At the programming language level, several program modularization techniques [82], such as aspects [76], framed aspects [83], mixins [112], hyperslices [115] or traits [53, 19], are used to implement features in a compositional fashion. Also, the modularity concepts of recent languages, such as Scala [92] or NewspEAK [26], can be used to represent product features. Although the above approaches are suitable to express feature-based variability, they do not contain designated linguistic concepts for features. In the context of feature-oriented programming [12], code fragments are explicitly associated to features and can be composed for a particular feature configuration. There are various languages realizing the feature-oriented programming paradigm, such as Jak [12], FeatureC++ [5], FeatureFST [6], or Prehofer’s feature-oriented Java extension [95]. In [86], CeasarJ is proposed as a combination of feature modules and aspects extending feature-oriented programming with means to modularize crosscutting concerns.

Context-oriented programming [68] has been explicitly proposed by Costanza and D’Hondt [39] as a compositional approach for implementing software product lines. COP’s layers group together methods, which can be activated or deactivated (even dynamically) based on feature-specific application conditions. This approach, however, does not enjoy the property of obliviousness [57], which is highly debated in the aspect-oriented programming community. Obliviousness allows programmers of the core application to be unaware of the modifications being made by subsequently added aspects or layers. Developing software product lines using context-oriented programming requires significant design effort, because layers intercept large numbers of methods and the order in which layers are imposed may not be the desired order for all products in the product line.

Multi-methods [21], also compositional in nature, offer the possibility of some fine-grained control for implementing software product lines. Any generic method, that is, a collection of methods sharing the same name, can be enhanced so that more specific combinations of arguments to the generic method are dispatched to the newly added methods. In its present form this is rather undisciplined as dispatch is based on the types of objects, which are not necessarily related to the features. In our example (Section 3.4.1), it would be impossible to modify the generic method \texttt{font(* *)} which has a concrete method \texttt{font(int c)} by adding another concrete method with signature \texttt{font(int c)}.

The delta modeling [105, 103] approach is inherently a compositional approach as it structures the product line artifacts into a core product and a set of product-deltas.

### 4.3.3 Granularity

In [77], focus is put on the concept of granularity. Approaches that support coarse granularity would be limited to the manipulation of elements at the method level. Finer granularity would allow a designer to express variability at the statement level or even the expression level.

As it is pointed out in [77], one of the advantages of annotative approaches is that they allow fine-grained modifications. A major problem of these approaches is, however, that they are non-modular and rely on one representation of all possible products which may lead to scalability issues for large and complex product lines. Although compositional approaches allow structuring the product line artifacts in a modular way and, thus, increase the ability to deal with large and complex product lines, the granularity of the possible
modifications is in general restricted by the composition operations to a coarse-grained level.

Context-oriented Programming [68], being compositional in nature, supports the complete replacement of methods, but COP layers cannot be used to insert code within the body of an existing method.

Multi-methods [21] cannot replace nor can code be inserted into existing methods. They can be grouped into modules to offer coarser granularity, and different multi-methods can be in scope in different contexts.

Delta modeling is compositional and thus more inherently suited to coarse-grained modifications. However, it can be instantiated to a concrete modeling or implementation language by defining the semantics of the change operations in the product-deltas in terms of the concrete language constructs. This means that the granularity of the modifications that can be described by a product-delta depends on the underlying modeling or implementation language of the products as well as the semantics of the modification operations used in the product-deltas. Granularity of the modifications can be at the same level as annotative approaches, for example, if the modifications specified in the product-deltas are implemented by frames (cf. [105]).

In the new delta modeling formalism (Section 3.4.1), we do not specify the supported granularity, since we abstract away from the concept of code entirely. The concrete class of delta models we introduce in [30, 31] is coarse grained, but other classes of delta models could easily be developed that support access to lower levels of code.

If we do not wish to require explicit hooks [125] in the core architecture, other than the natural code boundaries (e.g., methods, classes), it would be problematic for a delta to specify exactly where a fine-grained modification is to take place. CIDE [77], for instance, still uses such hooks in the background. By the nature of the annotative approach, the developers of the core architecture have to be aware of the feature-specific code when they make changes, and thus this approach lacks the property of obliviousness [57]; this is an important trade-off to be aware of.

We wish to also emphasize the trade-off between finer granularity and greater support for formal methods. It is, for instance, extremely problematic to support proof reuse [78] when arbitrarily fine grained changes can take place in the body of a method. Since there is a focus on formal methods and static guarantees in the HATS project, we will likely choose to support only coarse-grained modifications in ABS.

4.3.4 Expressiveness

In this section we compare the new delta modeling approach with the old delta modeling approach [105, 103, 18], AHEAD [12], and trait-based programming [53, 19].

Traditional delta models [105, 103, 18] consist of a single core and a set of product-deltas with no ordering between them. As such, any conflicts between two such deltas were unresolvable, except by mutually exclusive application conditions. In order to express all possible products, we needed a lot of code duplication between such mutually exclusive deltas. Since the new delta modeling approach is able to express conflict resolution in the form of a new delta to combine the two conflicting deltas, it avoids this problem.

AHEAD feature modules [12] (their basic units of code modification) are reasonably comparable to deltas. But AHEAD takes the opposite approach to delta modeling, and allows only modifications in a linear order, each able to override changes from all that came before. Of course, there is no problem resolving conflicts this way, since all one has to do is add a new feature module on top to handle the situation. The problem here is accidentally overriding code that was working correctly, or silently masking a conflict that would otherwise receive special attention. Delta modeling does not have this problem, since it encourages different features to be designed in different, unrelated deltas. Any conflicts can be trivially detected, so designers can resolve them the way they should be.

Trait-based programming [53, 19] leaves control of composition to the entity being composed. This makes it a most useful mechanism for reusing behavior. But by binding optional features to the common behavior of the product line, there is no way to automatically derive the core product without those features. Instead of units of reusable behavior, deployable anywhere, we need modifications targeted to a single code-base, leaving control to the outside, and this is what delta modeling offers; a more detailed comparison between traits and deltas may be found in Section 3.6.
Bibliography


[53] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. ACM TOPLAS, 28(2), 2006.


[71] A. Igarashi, B.C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. ACM Transactions on Programming Languages and Systems (TOPLAS), 23(3):396–450, 2001.


Glossary

Terms and Abbreviations

ABS Abstract Behavioral Specification language. An executable class-based, concurrent, object-oriented modeling language based on Creol, created for the HATS project.

AHEAD A feature oriented programming methodology introduced in [12] wherein a linearly ordered set of feature modules may make modifications and additions to the software product.

Application Condition A condition in propositional logic (alternately represented as a set of sets of features) indicating to which feature configurations a delta is applicable.

Application Function A function mapping deltas to their application condition.

Class Refinement Like class inheritance, except that the new ‘subclass’ completely supplants the original, being instantiated everywhere the original was instantiated before.

Compatibility of feature models Condition indicating if two features, products, or product lines can be reconciled.

Conflict The condition between two incompatible, non-ordered deltas.

Conflict Resolving Delta The delta that resolves a given conflict between two other deltas. It has to be greater in the partial order than the conflicting deltas and equalize the two possible orderings between them.

Consistent Conflict Resolution A condition of a deltoid which simplifies the conflict resolver property (and thus the unambiguity property).

Context-Oriented Programming A way to express the semantics of features in software product lines. Method implementations are grouped into layers, and methods dispatch dynamically through a collection of such layers (Section 3.1).

COP Context-Oriented Programming

Core In delta modeling, the core is the basic product that may be modified by product-deltas. The core may also be seen as a delta modifying the empty product.

Delta A unit of functionality and conflict resolution in delta modeling, able to modify a product using invasive composition of code or other content.

Delta Model Generally, a means for expressing the semantics of features within product lines. In the formalism, a delta model is defined more specifically as \((D, \prec)\), a partially ordered set of deltas.

Deltoid A class of delta models which specifically targets some type of content, specifies exactly the possible operations a delta may contain and defines equality between deltas. An example is the class of delta models targeting object oriented programming language code.
**Derivative Module** A module containing exactly the code required to make multiple features interact sensibly, allowing the modules implementing the features themselves to ignore the possibility of interaction.

**Dynamic Feature (Petri) Net** A Petri net whose transitions are conditioned on application conditions. Firing a transition can modify the set of features that are selected according to an “update expression”.

**FDeltaJ** Featherweight Delta Java, a programming language with deltas based on Featherweight Java.

**FDeltaRTJ** Featherweight Delta Records Traits Java, a planned extension of FRTJ which includes deltas.

**Featherweight Java** A reduced version of Java in which almost all features are dropped to obtain a small calculus for rigorous proofs of language properties. It has often been extended to explore new language constructs such as context oriented programming, traits and deltas.

**Feature** Generally, an increment in software functionality. On the level of feature models it is merely a label with no inherent semantic meaning.

**Feature Configuration** A subset of available features. It identifies a single product in a product line if it is valid in the feature model.

**Feature Interaction** The phenomenon in which two features interact on a semantic level. The term is used for both intentional interaction of features and for unintentional and unwanted interaction.

**Feature Model** An expression of the variability within product lines. Abstractly it may be seen as a system of constraints on the set of possible feature configurations.

**Feature Module** Comparable to a delta, a feature module may contain code modifications to be applied to a software product in order to fully implement a single feature.

**Feature Net** A Petri net whose transitions are conditioned on application conditions.

**Feature Petri Net** see Feature Net.

**FM** Feature Model

**FODA** Feature Oriented Domain Analysis

**FORM** Feature-Oriented Reuse Method

**FRTJ** Featherweight Record Trait Java, a programming language with records and traits based on Featherweight Java.

**Glue Code** In the context of traits, this is the code that the composed class contains to resolve conflicts between multiple traits.

**Incompatible Deltas** The property of two deltas \(x,y\) that interfere with each other such that \(x \cdot y \neq y \cdot x\).

**Independent Features** Features which may each be included or excluded from a product independently, according to the feature model.

**Multi-method** A method with multiple dispatch semantics.

**Multiple Dispatch** An alternative to the single dispatch semantics of method calls of languages such as Java, C# and C++. It uses the run time types of more than one argument to a method call to determine which method body to run (Section 3.2).

**Nested Delta Model** An extension of delta models, which allows a delta to contain / be a nested delta model (recursively but finitely defined).
ODM  Organisation Domain Modeling

Optional Feature Problem The situation in which a number of features that are independent in the
feature model either interfere with each other on an implementation level or should interact but don’t.

Product One member of a product line, with well-defined commonalities and variabilities to other products.

Product Line Generally, a family of products with well-defined commonalities and variabilities. In the
delta modeling formalism, a product line is more specifically defined as $(\Phi, c, D, \prec, \gamma)$, a feature model,
a core product, a delta model and an application function.

Reconciliation of feature models Operation that combines two compatible features, products, or prod-
uct lines, yielding the most refined elements from each item.

Record A construct offering managed state reuse, as a complement to traits, which offer behavior reuse.

Software Family See Software Product Line.

Software Product Line A family of software systems with well-defined commonalities and variabilities.
See also Product Line.

SPL Software Product Line

Trait A construct for fine-grained behavior reuse in object oriented languages (Section 3.5).

TVL Text-based Variability Language, a textual feature description language suggested by Classen et
al. [34].

Unambiguous Delta Model The property in a delta model that every conflict is properly resolved, such
that a unique implementation is guaranteed.

Unique Implementation A delta model has a unique implementation if all linear extensions of its partial
order result in the same implementation.

View Is function defined between features, products, or product lines. This function is partial at the level
of features, and it abstracts and filters elements from a given feature model.

View Partition Describes the relation between two sets of features $F$ and $G$ using two views, one from $F$
to $G$ and the other from $G$ to $F$. 
Symbols and Notation for the Delta Modeling Formalism

These symbols and notations are used in the papers ”Abstract Delta Modeling” [30, 31].

- $\mathcal{F}$ Some global set of features that could appear in a feature model.
- $F$ A feature selection / feature configuration.
- $\Phi$ A feature model.
- $\mathcal{D}$ Some global set of deltas, possibly restricted by some concrete deltoid.
- $D$ A set of deltas from $\mathcal{D}$.
- $d, x, y, z$ Deltas from $\mathcal{D}$.
- $x \cdot y$ The delta applying first $x$ and then $y$. $\cdot$ is associative.
- $D^+$ All non-empty delta sequences with deltas from $D$.
- $x \prec y$ A partial order on a set of deltas, indicating that $x$ should be applied some time before $y$ when generating the implementation.
- $P$ A product or delta model.
- $c$ A core product.
- $\gamma$ An application function.
- $\text{derv}$ The function taking a delta model and returning all possible implementations.
- $\Lambda$ Either a delta, or a nested delta model.
Appendix A

Abstract Delta Modeling

The paper “Abstract Delta Modeling” follows.
Abstract Delta Modeling

Dave Clarke
IBBT-DistriNet, Katholieke Universiteit
Leuven, Belgium
dave.clarke@cs.kuleuven.be

Michiel Helvensteijn
CWI, Amsterdam
LIACS, Leiden University, The Netherlands
michiel.helvensteijn@cwi.nl

Ina Schaefer
Chalmers University of Technology,
Gothenburg, Sweden
schaefer@chalmers.se

Abstract

Delta modeling is an approach to facilitate automated product derivation for software product lines. It is based on a set of deltas specifying modifications that are incrementally applied to a core product. The applicability of deltas depends on feature-dependent conditions. This paper presents abstract delta modeling, which explores delta modeling from an abstract, algebraic perspective. Compared to previous work, we take a more flexible approach with respect to conflicts between modifications and introduce the notion of conflict-resolving deltas. We present conditions on the structure of deltas to ensure unambiguous product generation.

Keywords Software Product Lines; Automated Product Derivation; Delta Modeling; Conflict Resolution.

1. Introduction

A software product line (SPL) is a set of software systems, called products, with well-defined commonalities and variabilities [9, 28]. Software product line engineering aims at developing this set of systems by reuse in order to reduce time to market and to increase product quality. Automated product derivation (or software mass customization [21]) is an approach to generating individual products without the need for manual intervention during application engineering, which can be tedious and error-prone [11].

Currently, product line variability is mostly represented by feature models [17, 35]. Features are designated product characteristics or increments of product functionality [5]. A product is uniquely identified by a valid feature configuration, i.e., a legal combination of features from the feature model. On the feature model level, features are merely labels [10]. In order to be able to automatically derive a product for a particular feature configuration, a correspondence between the features on the feature modeling level and the reusable product line artifacts has to be introduced. Additionally, the product line artifacts have to be organized in such a way that they can be assembled automatically to generate a uniquely determined product.

Delta modeling [30–32] aims at bridging the gap between features and product line artifacts to develop product line artifacts capable of automated product generation. In the delta modeling approach, a product line is represented by a core product and a set of product deltas. Product deltas specify modifications to the core product to generate further products of the product line. Each delta has an application condition specifying for which feature configurations the modifications have to be carried out, connecting features on the feature modeling level with product line artifacts. A product for a feature configuration can be obtained by applying those product deltas with a valid application condition to the core product.

In this paper, we generalize the existing delta modeling approaches [30–32] and present an abstract, algebraic formalization of the delta modeling concepts. The presented abstract delta modeling approach goes beyond existing work with its novel treatment of conflicts between deltas. A conflict between deltas arises if their specified modifications are not commutative. This means that applying them in different orders results in different (composed) modifications. In previous work, deltas were either incomparable [30, 32], which required writing additional deltas for every conflicting combination, or they had to be ordered in a very restrictive way [31] to avoid conflicts explicitly. As a main contribution of this paper, we introduce the notion of conflict-resolving deltas that relax these restrictions and make delta modeling of product lines more flexible. A conflict-resolving delta, which is applied after two conflicting deltas, eliminates differences between the (composed) modifications. If for every pair of conflicting deltas a conflict-resolving delta exists, all possible sequences of deltas produce the same modification and generate a uniquely defined product. In order to ensure this result for every valid feature configuration, we provide efficient conditions requiring only the inspection of the product line directly, without having to generate and check all products.

The concepts of abstract delta modeling can be instantiated for different kinds of development artifacts, such as documentation, models or code. We demonstrate the feasibility of the approach by presenting an instantiation of abstract delta modeling for object-oriented implementations of software product lines and extend this with method wrapping. Furthermore, we show that existing formalizations of compositional product line implementations can be seen as instantiations of abstract delta modeling.

The abstract delta modeling formalism consists of a number of ingredients, which are depicted in Figure 1, along with the operations between them. At the top is (a model of) the software product line that is defined by the feature model, the core product, the product deltas specifying modifications of the core product, a partial ordering on the deltas restricting their application and the application conditions for the deltas. By specifying a feature configuration, one can produce a delta model, consisting of an ordered collection of modifications necessary to generate the respective product. The process of derivation applied to a delta model puts the specified modifications in a linear order that is compatible with the partial
ordering in order to obtain a valid, ideally unique, composed modification. The partial delta application operation applied to a delta returns a function that takes a product and produces a new product. This function can then be applied to the core product to produce the target product for a given feature configuration. The partial delta application operation applied to a delta modifies products and incorporates deltas into product lines, transferring the unambiguity properties. Section 5 presents a concrete class of deltas and illustrates our approach using an example. Section 6 compares our approach with existing algebraic approaches from the literature. Finally, Sections 7 and 8 present related work and conclusions. Appendix A contains additional results and proofs for review purposes only.

2. Abstract Delta Modeling

This section presents our approach of abstract delta modeling. We introduce product modifications and their composition as a monoid, called a deltoid. Delta models are built on top of this monoid as partially ordered collections of modifications, where the ordering constrains the possible ways such modifications can be applied. It is important that a delta model defines unambiguous modifications as these are used later to obtain distinct products. Thus, we define the notion of conflicts, leading to ambiguous modifications, and conflict-resolving deltas eliminating these. We develop conditions to ensure that a delta model is unambiguous.

2.1 Delta Models

In existing compositional approaches for implementing software product lines, such as feature-oriented programming [5] or delta-oriented programming [31], a member product of an SPL is obtained by the application of a number of modifications (deltas) \( x_1, \ldots, x_n \) to a core product \( c \), as follows:

\[
x_n \cdot \cdots \cdot x_1(c) \cdot \cdots
\]

In feature-oriented programming the core product is determined by one or more base modules. The modifications are feature modules extending and refining the core product. In delta-oriented programming the core product can be any valid product of the product line. As both approaches treat the core product as a constant element for all products in the product line, it is useful to focus on the modifications. In this setting, the above modification would be equivalently written as follows, where \( \cdot \) refers to the composition of modifications:

\[
(x_n \cdot \cdots \cdot x_1)(c).
\]

Thus, we will focus exclusively on sequences of modifications such as \( x_n \cdot \cdots \cdot x_1 \).

It may still be possible to reason about the core product if we choose to see it as a modification \( x_0 \) applied to the empty product \( 0 \), i.e. \( c = x_0(0) \). Thus:

\[
(x_n \cdot \cdots \cdot x_1)(c) = (x_n \cdot \cdots \cdot x_1)(x_0(0)),
\]

so nothing is lost by restricting our attention to modifications.

In abstract delta modeling the main object of interest is a deltoid. A deltoid consists of a set of modifications, called deltas, along with the operation for composing them sequentially. A deltoid can contain different kinds of deltas for different kinds of development artefacts (e.g., documentation, models or code) and for different levels of abstraction (e.g., when working on component level or working on class level). The concrete nature of the modifications specified in the deltas depends on the capabilities of the underlying modeling or programming languages. Deltas may be performing the changes directly or some other structure representing those changes. We abstract away from the internal details of modifications, since many different instantiations are possible.

We define the notions of deltoids and deltas as follows.

**Definition 1** (Deltoid). A deltoid is a monoid \((D, \cdot, \epsilon)\), where \(D\) is a set of modifications (referred to as deltas), and the operation \(\cdot : D \times D \rightarrow D\) corresponds to their sequential composition. \(\epsilon\) denotes the modification applying first \(y\) and then \(x\). The neutral element \(\epsilon\) of the monoid corresponds to modifying nothing.

The operation \(\cdot\) is associative but not inherently commutative, as the ordering between two deltas may be significant. We call two deltas \(x, y \in D\) incompatible if \(x \cdot y \neq y \cdot x\).

A delta model describes the collection of deltas required to build a specific product, along with a strict partial ordering on those deltas restricting the order in which they can be applied. Recall that a strict partial order is irreflexive, asymmetric and transitive.

**Definition 2** (Delta Model). A delta model is a tuple \((D, \prec)\), where \(D \subseteq D\) is a finite set of deltas and \(\prec \subseteq D \times D\) is a strict partial order on \(D\). \(x \prec y\) states that \(x\) should be applied before, though not necessarily directly before, \(y\).

The partial order between deltas represents the intuition that a subsequent delta has full knowledge of (and access to) earlier deltas and more authority over modifications to the product. This is realized by applying the deltas in a linear extension of the partial order, as shown in the following definition. A derivation is a sequential application of all deltas in a model to generate a desired product.

**Definition 3** (Derivation). Given a delta model \(P = (D, \prec)\), its derivations are defined to be

\[
\text{deriv}(P) = \text{tot}
\]

Note that \(\text{deriv}(P)\) may potentially generate more than one distinct derivation as incompatible deltas may be applied in different orderings. However, it is desirable that all possible derivations of a delta model have the same effect, as it corresponds to deriving a unique product. This motivates the following definition.

**Definition 4** (Unique Derivation). A delta model \(P = (D, \prec)\) is said to have a unique derivation if \(x_1, \ldots, x_n = \tilde{x}_1, \ldots, \tilde{x}_n\) for all pairs of linear extensions \((x_1, \ldots, x_n)\) and \((\tilde{x}_1, \ldots, \tilde{x}_n)\) of \(\prec\). Or, equivalently, \(|\text{deriv}(P)| = 1\).

2.2 Unambiguity of Delta Models

The property that a delta model has a unique derivation can be checked by brute force. This means generating all possible derivations (in the worst case, \(n!\) for \(n\) deltas), and then checking that they...
all correspond. In order to allow for a more efficient way to establish this property, we introduce unambiguous delta models which rely on the notion of conflicting deltas and conflict resolving deltas.

Two deltas are in conflict if they are incompatible and no ordering is placed upon them. Intuitively, the two conflicting deltas are independently modifying the same part of the product in different ways, meaning that multiple distinct derivations may be possible.

**Definition 5 (Conflict).** Given a delta model \( D = (D, \prec) \), \( x, y \in D \) are said to be in conflict iff the following condition holds:

\[
x \not\prec y \iff x \cdot y \neq y \cdot x \land x \neq y \neq x.
\]

**Notation.** If \( D \subseteq D \), then \( D \) and \( D \) denote the sequences and non-empty sequences of compositions of deltas from \( D \).

One way to ensure a unique derivation is to avoid conflicts by always enforcing an ordering between incompatible deltas. This is quite restrictive and might result in overspecification of the model. An alternative is to allow conflicts but to provide additional, subsequently applied, deltas to resolve them.

**Definition 6 (Conflict-Resolving Delta).** Given a delta model \( D = (D, \prec) \) and \( x, y \in D \) which are in conflict, we say that a delta \( z \in D \) resolves their conflict iff the following property holds:

\[
(x, y) \prec z \iff z \prec D, \cdot z \cdot d \cdot z \cdot y = z \cdot d \cdot y \cdot x.
\]

An unambiguous delta model is now a delta model containing a conflict-resolving delta for every conflicting pair of deltas.

**Definition 7 (Unambiguous Delta Model).** Given a delta model \( D = (D, \prec) \), we say that the model is unambiguous iff

\[
\forall x, y \in D : x \not\prec y \implies \exists z \in D : (x, y) \prec z.
\]

If a delta model is unambiguous, we can show that it has a unique derivation. In order to prove this, we need some intermediate results. Lemma 1 states that in an unambiguous delta model, any two deltas in a derivation are either ordered or commutative.

**Lemma 1.** Given an unambiguous delta model \( D = (D, \prec) \) and \( d_1 \cdot x \cdot y \cdot d_2 \in \text{deriv}(P) \), where \( x, y \in D \) and \( d_1, d_2 \in D \). Then either \( x \prec y \) or \( d_1 \cdot x \cdot y \cdot d_2 = d_1 \cdot y \cdot x \cdot d_2 \).

**Proof.** By case distinction on the unambiguity of \( P \) for deltas \( x \) and \( y \):

- Case \( x \not\prec y \). It follows that \( d_1 \cdot x \cdot y \cdot d_2 = d_1 \cdot y \cdot x \cdot d_2 \).
- Case \( x \prec y \). Immediate.
- Case \( y \prec x \). Cannot happen, otherwise \( d_1 \cdot x \cdot y \cdot d_2 \) would not be a linear extension of \( \prec \).
- Case \( \exists z \in D : (x, y) \prec z \). Firstly, from the definition of conflict-resolving delta we have that \( x, y \prec z \), hence there exist \( d'_1, d'_2 \in D \) such that \( d'_1 \cdot x \cdot y \cdot d_2 = d'_1 \cdot z \cdot d'_2 \cdot x \cdot y \cdot d_2 \). From the remaining condition on \( z \), we have \( z \cdot d'_1 \cdot x \cdot y \cdot d_2 = z \cdot d'_1 \cdot y \cdot x \cdot d_2 \) and hence \( d_1 \cdot x \cdot y \cdot d_2 = d_1 \cdot z \cdot d'_1 \cdot x \cdot y \cdot d_2 = d_1 \cdot y \cdot d_2 \).

Lemma 2 states that removing a minimal element with respect to the partial order preserves unambiguity of delta models.

**Lemma 2.** If \( P = (D, \prec) \) is an unambiguous delta model and \( w \) is minimal in \( \prec \), then \( (D \setminus \{ w \}, \prec') \), where \( \prec' \) is \( \prec \) restricted to \( D \setminus \{ w \} \), is also an unambiguous delta model.

**Proof.** If \( D, \prec \) is unambiguous, then \( \forall x, y \in D : x \cdot y \not\prec z \in D : (x, y) \prec z \). For this to be true in \( D \setminus \{ w \} \) we need to show that there are no \( x \) and \( y \) such that \( x \prec y \) and \( w \) is such that \( (x, y) \prec w \). But \( w \) cannot satisfy this condition, as it is minimal, contradicting conditions \( x \prec w \) and \( y \prec w \) of \( (x, y) \prec w \).

Lemma 3 formulates that a minimal element in the partial order can be moved to the front of any derivation from an unambiguous delta model without changing the meaning of that derivation.

**Lemma 3.** Given an unambiguous delta model \( P = (D, \prec) \). Let \( x_n \cdot \ldots \cdot x_1 \in \text{deriv}(P), \) where \( \{ x_1, \ldots, x_n \} = D, \) with \( x, \) minimal in \( \prec \). Then \( x_n \cdot \ldots \cdot x_1 = x_n \cdot \ldots \cdot x_{i-1} \cdot x_i \cdot x_{i-2} \cdot \ldots \cdot x_1 \).

**Proof.** By induction on \( i \):

- Case \( i = 1 \). Immediate.
- Case \( i > 1 \). Since \( x_{i-1} \not\prec x_i \), as \( x_i \) is minimal, Lemma 1 implies that \( x_n \cdot \ldots \cdot x_1 = x_n \cdot \ldots \cdot x_{i-1} \cdot x_i \cdot x_{i-2} \cdot \ldots \cdot x_1 \).

The following theorem states that every unambiguous delta model has a unique derivation. This reduces the effort of checking that all possible derivations of a delta model have the same effect to checking that all conflicts between pairs of deltas are eliminated by conflict resolving deltas. The proof is by induction over the size of the delta model.

**Theorem 1.** An unambiguous delta model has a unique derivation.

**Proof.** Given unambiguous delta model \( P = (D, \prec) \). Proceed by induction on the size of \( D \):

- Case \( |D| = 0 \). Immediate as \( \text{deriv}(P) = \{ e \} \).
- Case \( |D| = 1 \). Immediate as \( \text{deriv}(P) = \{ x \} \), where \( D = \{ x \} \).
- Case \( |D| > 1 \). For any two \( d_1, d_2 \in \text{deriv}(P) \), let \( d_1 = d_1 \cdot x \) and \( d_2 = d_2 \cdot x \cdot d_1 \), where \( D_1 D_2 \subseteq D \). As \( x \) is the last element of \( d_1 \), it must be minimal in \( \prec \). Thus, by Lemma 3, \( d_2 \cdot x \cdot d_2 = d_2 \cdot d_2 \cdot x \). Now by Lemma 2, \( P = (D \setminus \{ x \}, \prec') \), where \( \prec' \) is \( \prec \) restricted to \( D \setminus \{ x \} \), is an unambiguous delta model, and \( d_1, d_2 \in \text{deriv}(P) \). By the induction hypothesis, \( d_1 = d_2 \cdot d_2 \cdot d_2 \) and thus \( d_1 = d_2 \cdot d_2 \cdot d_2 \). Hence, \( |\text{deriv}(P)| = 1 \).

2.3 Consistent Conflict Resolution

Although the notion of unambiguous delta model alleviates the task of establishing that a delta model has a unique derivation, unambiguity is still quite complex to check. The reason is that the definition of a conflict resolving delta (Definition 6) quantifies over all elements of \( D \). Hence, in order to check that a delta is indeed a conflict resolver, all these sequences of deltas have to be inspected. However, for interesting classes of deltas, a simpler criterion exists to make checking ambiguity more feasible. The consistent conflict resolution property states that if a delta \( z \) resolves an \( (x, y) \)-conflict when it is applied directly after \( x \) and \( y \), it also resolves the conflict after the application of any sequence of intermediate deltas.

**Definition 8 (Consistent Conflict Resolution).** A deltoid \((D, \cdot, \varepsilon)\) is said to exhibit consistent conflict resolution iff the following condition holds:

\[
\forall x, y, z \in D : z \cdot x \cdot y = z \cdot y \cdot x \implies \forall d \in D : z \cdot d \cdot x \cdot y = z \cdot d \cdot y \cdot x.
\]

If a deltoid \((D, \cdot, \varepsilon)\) exhibits consistent conflict resolution, then a delta model \((D, \prec)\) with \( D \subseteq D \) is also said to exhibit the property.

Note that the consistent conflict resolution property is checked at the level of the underlying deltoid, rather than for any specific delta model. Hence, it has to be established only once for each deltoid and holds for all delta models based on this deltoid.

To establish the unambiguity of a delta model exhibiting consistent conflict resolution, it is sufficient to check that for each pair of conflicting deltas \( x \) and \( y \) there exists a conflict-resolving delta
z, such that \( x \prec z \wedge y \prec z \wedge z \cdot x \cdot y = z \cdot y \cdot x \). We need not quantify over all possible intermediate sequences of deltas. Consequently, unambiguity of delta models can be established much more efficiently. This is formalized in the next theorem.

**Theorem 2.** Given delta model \( P = (D, \prec) \) exhibiting consistent conflict resolution, for all deltas \( x, y, z \in D \), it is true that \( x \prec z \wedge y \prec z \wedge z \cdot x \cdot y = z \cdot y \cdot x \) \( \Rightarrow \) \((x,y) \prec z \).

**Proof.** Assume that \( P = (D, \prec) \) exhibits consistent conflict resolution. Take arbitrary \( x, y, z \in D \). From the definition of consistent conflict resolution, we have the following:

\[
z \cdot x \cdot y = z \cdot y \cdot x \implies \forall d \in D \colon z \cdot d \cdot x \cdot y = z \cdot d \cdot y \cdot x
\]

which may be substituted into \( x \prec z \wedge y \prec z \wedge z \cdot x \cdot y = z \cdot y \cdot x \) to form the definition of \((x, y) \prec z \).

\[\square\]

### 3. Reintroducing Products

Thus far, only modifications have been considered, without considering the products that we modify. Products can be reintroduced, by defining the notion of application of a delta to a product. Firstly, we select a set of products.

**Definition 9 (Products).** Let \( P \) denote a set of possible products.

Applying a delta to a product results in another product. This is captured by the notion of delta application.

**Definition 10 (Delta Application).** Delta application is an operation \(-\cdot- : D \times P \rightarrow P\).

This definition implies that one generates a product from a sequence of deltas by first composing the deltas and then applying the result to the core product. A much stronger version of delta application is possible, borrowing the notion of monoid action.

**Definition 11 (Delta Action).** A delta application operation \(-\cdot- : D \times P \rightarrow P\) is called a delta action if it satisfies the conditions \((x \cdot y)(p) = x(y(p))\) and \(\epsilon(p) = p\), for all \(x, y \in D\) and \(p \in P\).

This generalises the case when \( \cdot \) is function composition and \(-\cdot-\) is function application.

### 4. Product Lines

Using the introduced concepts of delta models, products and delta application we can now abstractly define product lines, thus providing a link from feature configurations on the feature modeling level to product representations. We will extend the concept of unambiguity to the level of product lines and provide an efficient condition to check unambiguity.

#### 4.1 Defining Product Lines

Product line variability is predominantly captured by features where a feature captures a designated product characteristic or an increment to product functionality. At the level of the feature model features are merely labels without inherent semantic meaning. A product can be characterized by the set of features it provides.

**Definition 12 (Features).** Let \( F \) denote a universal set of features.

The set of products in a product line can be represented by a feature model. Many formal descriptions \([15, 17, 35]\) agree that a feature model determines a set of valid feature configurations.

**Definition 13 (Feature Model).** A feature model \( \Phi \subseteq \mathcal{P}(F) \) is a set of sets of features from \( F \). Each \( F \in \Phi \) is a set of features corresponding to a valid feature configuration.

In order to bridge the gap between features and product line artifacts, we introduce application conditions for deltas. An application condition attached to a delta determines for which feature configuration the delta has to be applied.

**Definition 14 (Application Function and Condition).** Let \( D \subseteq \mathcal{D} \) be a set of deltas. An application function \( \gamma : D \rightarrow \mathcal{P}(\mathcal{P}(F)) \) gives the feature configurations each delta \( x \in D \) is applicable to. Thus, \( F \in \gamma(x) \) denotes that delta \( x \) is applicable for feature configuration \( F \). \( \gamma(x) \) is called the application condition for \( x \).

A product line is defined by its feature model, characterizing all member products by a set of valid feature configurations, the core product, the associated delta model, containing the modifications used to obtain further products, and the application function, associating features and deltas.

**Definition 15 (Product Line).** A product line is a tuple \((\Phi, c, D, \prec, \gamma)\), where \( \Phi \) is a feature model, \( c \in P \) is the core product, \((D, \prec)\) is a delta model and \( \gamma \) is an application function with domain \( D \).

If feature configuration \( F \) is valid according to \( \Phi \), its corresponding product is defined by the delta model containing only the deltas applicable to \( F \). Selecting such deltas gives a delta model whose derivations applied to the core product correspond to the desired product for feature configuration \( F \).

**Definition 16 (Selected Delta Model).** Given a product line \( PL = (\Phi, c, D, \prec, \gamma) \), a selected delta model for feature configuration \( F \in \Phi \), denoted \( PL|F \), is the delta model \((D', \prec')\) where \( D' = \{d \in D \mid F \in \gamma(d) \} \) is the set of applicable deltas, and \( \prec' = \prec \) restricted to \( D' \).

We now define the set of products generated from a product line.

**Definition 17 (Generated Products).** Given a product line \( PL = (\Phi, c, D, \prec, \gamma) \), the set of generated products for feature configuration \( F \in \Phi \) is defined as follows:

\[
\text{prod}(PL, F) = \{x(c) \mid x \in \text{deriv}(PL|F)\}.
\]

### 4.2 Unambiguity of Product Lines

As argued in Section 2, unambiguity of delta models is a desired property because it ensures unique derivation and, consequently, a unique generated product. We now lift unambiguity to the product level. A product line is unambiguous if every selected delta model is unambiguous. This means that every valid feature configuration yields a uniquely defined product, which is an important condition for the applicability of automated product derivation.

**Definition 18 (Unambiguous Product Line).** A product line \( PL = (\Phi, c, D, \prec, \gamma) \) is unambiguous if

\[
\forall F \in \Phi : \text{PL}|F \text{ is an unambiguous delta model}.
\]

The unambiguity of a product line can be checked by generating the selected delta models of all valid feature configurations and checking unambiguity by the criteria proposed in Section 2. However, as the set of feature configurations is often exponential in the number of features, that naive approach would be rather expensive. Instead, we propose the notion of a globally unambiguous product line that implies product line unambiguity.

We first introduce a shorthand notation for the set of feature configurations for which two deltas \( x \) and \( y \) are applicable.

**Notation.** Given a product-line \((\Phi, c, D, \prec, \gamma)\), we introduce a shorter notation for the set of valid feature configurations to which the deltas \( x, y \in D \) apply:

\[
\gamma(x, y) \triangleq \Phi \cap \gamma(x) \cap \gamma(y).
\]
A product line is globally unambiguous if for any two conflicting deltas \( x \) and \( y \) applied together for a set of feature configurations, there is a conflict-resolving delta \( z \) applicable in at least the same set of feature configurations. Thus for any selected delta model in which \( x \) and \( y \) appear together, the conflict is resolved by the same delta \( z \). Global unambiguity of a product line can be checked by inspecting the product line only once and does not require all selected delta models to be generated.

**Definition 19 (Globally Unambiguous Product Line).** A product line \( (\Phi, c, D, \prec, \gamma) \) is called globally unambiguous if and only if

\[
\forall x, y \in D : V^{x,y} = \emptyset
\]

\[
\lor \exists t : y \Rightarrow z \in D : (V^{x,y} \subseteq \gamma(z) \land (x, y) \prec z).
\]

The following theorem states that any globally unambiguous product line is also an unambiguous product line. Hence, it suffices to check global unambiguity by inspecting the product line once to ensure that all products that can be generated from the product line are uniquely determined. In the following proof, we annotate the \( f \) and \( \prec \) operators with the delta model for which they apply, e.g., \( x \rightarrow y \) and \( (x, y) \prec_p z \).

**Theorem 3.** A globally unambiguous product line is unambiguous.

Proof. Assume \( PL = (\Phi, c, D, \prec, \gamma) \) is a globally unambiguous product line. Let \( F \in \Phi \) be a valid feature configuration. We show that \( P = PL(F) = (D', \prec') \) is an unambiguous delta model.

Given arbitrary \( x, y \in D' \) (so also \( x, y \in D \)), we perform a case analysis on the global unambiguity of \( PL \) (Definition 19). Observe that \( F \in \gamma(x) \cap \gamma(y) \), otherwise \( x \) and \( y \) would not be in \( D' \). Hence \( F \in V^{x,y} \). Now for the cases:

- Case \( V^{x,y} = \emptyset \). Cannot happen, otherwise \( x, y \notin D' \).
- Case \( \lnot (x \rightarrow y) \). So also \( \lnot (x \rightarrow y) \).
- Case \( \exists z \in D' : V^{x,y} \subseteq \gamma(z) \land (x, y) \prec (D, \prec) z \). Note that \( z \in D' \), as \( V^{x,y} \subseteq \gamma(z) \). It follows that \( (x, y) \prec_p z \) (Definition 16).

So, for all \( x, y \in D' \), either \( \lnot (x \rightarrow y) \) or \( \exists z \in D' : (x, y) \prec_p z \). Hence, \( P \) is an unambiguous delta model (Definition 7).

A product line can be unambiguous, but not globally unambiguous if conflicts between two deltas \( x \) and \( y \) are resolved by different conflict-resolving deltas \( z \) for different feature configurations. For example, take a product line \( PL \) in which the only conflicting deltas \( x \) and \( y \) are applied together for feature configurations \( F \) and \( F' \). For feature configuration \( F \) only delta \( z \) resolves the conflict, \( (x, y) \prec_{F, \prec} z \), and for feature configuration \( F' \) only delta \( z' \) resolves the conflict, \( (x, y) \prec_{F', \prec} z' \), but \( z \neq z' \). Hence, the product line is unambiguous, because the conflict is resolved in all selected delta models, but not globally unambiguous, because the conflict resolving delta is not the same in each one.

## 5. A Deltoid for Object-Oriented Programs

We now present a concrete deltoid for object-oriented programs to demonstrate our approach. In this section, deltas manipulate object-oriented programs on a coarse-grained level. That is, a delta can add, remove or modify classes. Modifications of classes include addition, removal and replacement of fields and methods.

**Notation.** Let \( f : X \rightarrow Y \) denote that \( f \) is a partial function from \( X \) to \( Y \) where \( \bot \notin Y \). If \( f(x) \) is undefined for \( x \in X \), we write \( f(x) = \bot \).

**Notation.** Given a set \( X \) where \( \bot \notin X \), we introduce the notation:

\[
X - \bot = X \cup \{ \bot \}.
\]

### 5.1 Software Products

For simplicity, we abstract from a concrete programming language as well as from the concrete implementation of methods, and focus only on the structural aspects of object-oriented programs. First, we introduce the notion of identifiers for classes, methods and fields.

**Definition 20 (Identifiers).** We define a global set of identifiers \( I \), used for classes, methods and fields.

Further, we fix an abstract set of method and field definitions.

**Definition 21 (Method and Field Definitions).** We define a global set of method and field definitions \( M \).

A class is defined as a partial mapping from identifiers to method and field definitions.

**Definition 22 (Class Definitions).** The set of class definitions is the set of partial functions \( \Psi = I \rightarrow M \). One such class definition \( \psi \in \Psi \) maps some identifiers to their definition. Unmapped identifiers are not defined in the class.

As an example, consider the following class definition. Only the explicitly mentioned identifiers are considered to be defined. As we abstract from concrete method implementations, we use capital letters to refer to method implementations, where different letters represent distinct implementations.

\[
\begin{align*}
\Phi & \mapsto f() : \text{void} \{ A \}, \\
\Psi & \mapsto g() : \text{bool} \{ B \}, \\
\gamma & \mapsto i : \text{int}
\end{align*}
\]

A program is a set of classes, mapping identifiers to class definitions.

**Definition 23 (Programs).** We equate the set of products \( P \) with the set of programs in an object-oriented language: \( P = I \rightarrow \Psi \).

As an example, consider the following program definition:

\[
\begin{align*}
C & \mapsto f() : \text{void} \{ A \}, \\
\Psi & \mapsto g() : \text{bool} \{ B \}, \\
\gamma & \mapsto i : \text{int}
\end{align*}
\]

### 5.2 Software Deltas

Software deltas modify a program by adding, modifying and removing classes. A class modification includes adding, replacing and removing methods and fields, or replacing the class completely. To ensure that composition of deltas produces a closed form, we distinguish between updating a class and replacing it. A class replacement completely replaces an existing class. A class update modifies the original class at the method/field level. Modifying a class that does not exist is treated as adding a new class. The definition of a software delta captures this set of program modifications.

**Definition 24 (Software Deltas).** The set of software deltas is defined as \( D = I \rightarrow (\{(r, u) \times (I \rightarrow M^*) \}) \). Each delta \( d \in D \) is a partial function representing class modifications. \( r \) and \( u \) represent ‘replace’ and ‘update’, respectively. Mapping an identifier to \( \bot \) indicates removal from the product. \( \epsilon = \emptyset \) is the empty delta, modifying nothing.

An example software delta is:

\[
\begin{align*}
C & \mapsto (u, \{ f \mapsto \bot \}, z() : \text{void} \{ D \}), \\
D & \mapsto \bot
\end{align*}
\]
In contrast to previous work [31], the removal of an element in this concrete deltoid does not require that the element is already present, nor does addition require its absence. These simplifications ensure that every derivation of deltas is well-defined.

Now we introduce a notation we will need in the next few definitions. It is used to combine two partial functions into another partial function by some binary operation on their codomain.

**Notation.** We use the following notation to lift an operator \( \circ \) on two partial functions to the values in their codomain. For \( i \in \mathbb{I} \):

\[
(a \circ b)(i) \equiv a(i) \circ b(i).
\]

We now define sequential composition of software deltas.

**Definition 25 (Sequential Composition of Software Deltas).** The sequential composition of software deltas \( : \mathcal{D} \times \mathcal{D} \rightarrow \mathcal{D} \) is defined as

\[
y \cdot x \equiv y \circ_C x,
\]

where the operator \( \circ_C \), working on the level of class modifications, with \( f, g \in (\mathbb{I} \rightarrow \mathbb{M}^+) \), is

\[
\begin{align*}
\circ_C & \quad \perp \quad - \quad u \ g \quad r \ g \\
\perp \quad \perp \quad - \quad - \quad - \quad - \\
u \ f \quad u \ f \quad r \ f \quad u \ (f \circ_M g) \quad r \ (f \circ_M g) \\
r \ f \quad r \ f \quad r \ f \quad r \ f
\end{align*}
\]

and \( \circ_M \), working on the level of method and field definitions, with \( m, n \in \mathbb{M} \), is

\[
\begin{align*}
\circ_M & \quad \perp \quad - \quad n \\
\perp \quad \perp \quad - \quad - \quad - \\
m \quad m \quad m \quad m.
\end{align*}
\]

The options for combining methods are limited, but Section 5.4 will redefine \( \circ_M \) to allow method wrapping. The definition of software delta composition gives concrete meaning to the notion of incompatibility. Two deltas are incompatible if they map the same identifier to two different definitions.

**Lemma 4.** Software deltas are a deltoid.

**Lemma 5.** Software deltas exhibit consistent conflict resolution.

Finally, we define software delta application to apply a software delta to a program.

**Definition 26 (Software Delta Application).** Given delta \( y \in \mathcal{D} \) and product \( p \in \mathcal{P} \), software delta application is an operation \( (-)(-): \mathcal{D} \times \mathcal{P} \rightarrow \mathcal{P} \) defined as follows:

\[
y(p) \equiv y \circ_C p,
\]

where the operators \( \circ_C \), with \( f : \mathbb{I} \rightarrow \mathbb{M}^+ \) and \( g : \mathbb{I} \rightarrow \mathbb{M} \), and \( \circ_M \), with \( m, n \in \mathbb{M} \), are defined as

\[
\begin{align*}
\circ_C & \quad \perp \quad g \\
\perp \quad \perp \quad - \\
u \ f \quad f \quad f \circ_M g \\
r \ f \quad f \quad f
\end{align*}
\]

and

\[
\begin{align*}
\circ_M & \quad \perp \quad n \\
\perp \quad \perp \\
m \quad m \quad m.
\end{align*}
\]

**Lemma 6.** Software delta application is a delta action.

### 5.3 Example Product Line

We now show an example product line based on software deltas. It is a product line of editor widgets to be used for integrated development environments and other text-editing applications.

#### 5.3.1 The Core Program

The product line is based on core program \( c \) containing one class:

\[
\begin{align*}
\text{Editor} & \rightarrow \Phi \\
\text{model} & \rightarrow \text{model: Model,}
\text{draw} & \rightarrow \text{draw(): void,}
\text{getModel} & \rightarrow \text{getModel(): Model,}
\text{font} & \rightarrow \text{font(c: int): Font ( A ),}
\text{onMouseOver} & \rightarrow \text{onMouseOver(c: int): void ( B )}
\end{align*}
\]

The model field and the draw and getModel methods implement the basic functionality of the widget and are never modified. The font method specifies the proper font for each character in the editor’s text area. In the core product it may return a monospaced black font with no decoration. The onMouseOver method is an event handler for when the mouse cursor hovers over specific characters of the content. Both methods will be modified by deltas.

#### 5.3.2 The Feature Model

The editor product line is based on the features \{Editor, SH, ERR, TT\}. Editor is the mandatory base feature of the product line, implemented by the core program. SH stands for syntax highlighting of programming language constructs. It modifies the font method. ERR implements error detection. It underlines errors in code and shows relevant information in a tooltip when the mouse hovers over the error. This feature modifies the font and onMouseOver methods. TT allows the editor to show generic information in tooltips. It modifies the onMouseOver method. The feature model \( \Phi \) of the editor product line allows any combination the three optional features (Ed abbreviates Editor):

\[
\Phi = \{ \{Ed\}, \{Ed, SH\}, \{Ed, ERR\}, \{Ed, TT\}, \{Ed, ERR, TT\}, \{Ed, SH, ERR, TT\} \}
\]

There are two potential conflicts. SH and ERR both modify font. Similarly, ERR and TT both modify onMouseOver.

#### 5.3.3 Delta Model and Application Conditions

The base code for all three optional features of the editor product line can be developed in isolation. One delta is created for each, implementing that feature as a modification to the Editor class without considering potential conflicts. These deltas work as expected if their respective feature is the only one included in the product. They are depicted in the top row of Figure 2 as \( d_1, d_2, d_3 \in D \).

Because some feature configurations include interacting features, conflict resolving deltas for the two potential conflicts in our model are designed \( d_4, d_5 \in D \) in Figure 2. Delta \( d_4 \) deals with the interaction between SH and ERR, combining the coloring of SH with the underlining of ERR. Similarly, delta \( d_5 \) handles the interaction between ERR and TT.

Figure 2 also shows the application conditions \( c(d_i) \) for each delta \( d_i \in D \) in the form of propositional logic formulae, where the propositions are features. The application conditions of the conflict resolving deltas ensure that they are applied if and only if their two conflicting deltas are applied.

#### 5.3.4 Global Unambiguity

The editor product line is globally unambiguous. As the underlying deltoid exhibits consistent conflict resolution (cf. Lemma 5), this can easily be verified. There are only two pairs of deltas in conflict: \( d_1 \perp d_2 \) and \( d_2 \perp d_3 \). For both conflicts, there is a conflict resolving delta: \( (d_1, d_2) \preceq d_4 \) and \( (d_2, d_3) \preceq d_5 \). By the choice of \( \gamma \), the appropriate conflict-resolving delta is present in each feature configuration in which conflicting deltas appear.
5.3.5 Generating a Product

To illustrate the process of product generation (cf. Definition 17), we now derive the product for the feature configuration \( F = \{ SH, ERR \} \). We first generate the selected delta model \( PL \upharpoonright F = \{ D', \prec \} \), where \( D' = \{ d_1, d_2, d_3 \} \) and \( \prec \) is \( \{ (d_1, d_4), (d_2, d_3) \} \). Since \( PL \) is globally unambiguous, it is sufficient to select one derivation of the delta model to generate the uniquely defined product. We use \( x = d_1 \cdot d_2 \cdot d_1 \). Applying Definition 25, \( x \) becomes:

\[
\begin{align*}
\text{Editor} & \rightarrow \\
\{ \text{font}(c: \text{int}): \text{Font} \{ C \}, \text{onMouseOver}(c: \text{int}): \text{void} \{ E \} \}
\end{align*}
\]

Applying \( x \) to \( c \) (Definition 26) results in the product \( x(c) \):

\[
\begin{align*}
\text{Editor} & \rightarrow \\
\{ \text{model} & \rightarrow \text{model}: \text{Model}, \\
\text{draw} & \rightarrow \text{draw}(): \text{void}, \\
\text{getModel} & \rightarrow \text{getModel}(): \text{Model}, \\
\text{font} & \rightarrow \text{font}(c: \text{int}): \text{Font} \{ G \}, \\
\text{onMouseOver} & \rightarrow \text{onMouseOver}(c: \text{int}): \text{void} \{ E \} \}
\end{align*}
\]

5.4 A Deltoid for Aspect Oriented Programming

Arguably, the essence AOP is quantification (pointcuts) and wrapping (around advice). A rudimentary semantic interpretation of quantification is simply the set of all possible matching joinpoints with the same advice. Based on this simplification, we adapt the previous concrete deltoid (Section 5) to include method wrapping. We do so by modifying method bodies \( M \) in classes and deltas to have the following (abstract) grammar:

\[
\begin{align*}
M & \ni m & \triangleright b \mid w[m] & b \in B \\
W & \ni w[v] & \triangleright e[\ ] \mid w[v[\ ]] & e \in E.
\end{align*}
\]

where \( B \) is a set of basic method bodies and \( E \) is a set of primitive wrapping methods (around advice). The notation \( w[v[\ ]] \) denotes a wrapping method with a hole in it, where the hole corresponds to where the call to the original method is made, and \( w[m] \) denotes that body \( m \) is wrapped by \( w \). Methods with a hole in them do not appear in products.

Given these ingredients, only the definitions of \( \oplus_M \) and \( \ominus_M \) from Definitions 25 and 26 need to change (\( m, n \) have no hole):

6. Other Algebraic Approaches

Other algebraic approaches describing the underlying structure of software product lines exist [3, 6]. These formalise the mechanisms underlying AHEAD [5], GenVoca [4], and FeatureHouse [2]. The key difference is that our approach considers the collection of modifications for an entire product line, rather than a single product at a time. The second difference is that those approaches generally arrange deltas into modifications and introductions, whereas we assume a unified collection of deltas. Here we compare our approach with two recent proposals, namely, Apel et al.’s Quark model [3] and Batory and Smith’s Finite Map Spaces [6]. From an algebraic perspective, these two proposals are quite similar, so we consider them together. We also consider the formalization of patch theory underlying the Darcs version control system [16], which is algebraically similar.

6.1 Quarks and Finite Map Spaces

Both Apel et al. [3] and Batory and Smith [6] base the description of a product line on the following ingredients (our notation):

1. introductions: a commutative idempotent monoid \((I, +, 0)\), where \(+ : I \times I \rightarrow I\), 
2. modifications: a monoid \((M, \cdot, 1)\), where \(\cdot : M \times M \rightarrow M\).
3. an operation \(\circ : M \times I \rightarrow I\) applying modifications to introductions, typically satisfying
   - \(M\) is a monoid action over \(I\): \(1 \circ i = i\) and \((m \bullet n) \circ i = m \circ (n \circ i)\),
   - Distributivity: \(m \circ (i + j) = m \circ i + m \circ j\), and
   - \(m \circ 0 = 0\).

Introductions play a dual role. They correspond to (elements of) products, as well as acting as one kind of delta; modifications are the other kind. That is, an introduction \(i \in I\) in a delta corresponds to introducing a new element into a product and a modification \(m \in M\) corresponds to an operation modifying an existing element.

Introductions and modifications are combined to form quarks \(Q\), which correspond to our deltas. Different notions of quark and quark composition (\(\bullet : Q \times Q \rightarrow Q\)) are listed below. These cap-

![Figure 2. A visual representation of the delta model \((D, \prec)\) from the editor product line. The dashed boxes represent the deltas \(d_1, \ldots, d_5 \in D\). The ordering \(\prec\) is represented by the arrows. Deltas are decorated with their application condition \(\gamma(d_i)\).](image-url)
ture combinations of the following: local composition which applies modifications to elements (introductions) already in the product; global composition which applies to all elements of the final product; and modifiers of modifiers which modify modifications rather than elements of the product. Modifiers of modifiers consist of functions \( R : M \rightarrow M \). Batory and Smith introduce the syntactic function \( R^0 \) to recursively apply all higher-order modifications, as follows, \( R^0(m) = h(m) \), for \( m \in M \); \( R^0(i) = i \) for \( i \in I \) and \( R^0(m \cdot m') = R^0(m) \cdot R^0(m') \) and so on. We also include the operation image \( Q \rightarrow I \) used (sometimes implicitly) to extract the final product from a quark.

**Local Quark Composition (Apel et al.)**

- \( Q = I \times M \) — an introduction and a local modification
- \( (i_2, g_2) \cdot (i_1, l_1) = [(i_2 + (l_2 \odot i_1)) \cdot l_2 \cdot l_1] \)
- \( \text{image}(i, l) = i \)

**Global Quark Composition (Apel et al.)**

- \( Q = I \times M \) — an introduction and a global modification
- \( (g_2, i_2, l_2) \cdot (g_1, i_1, l_1) = ((g_2 \cdot g_1) \odot (i_2 + i_1)) \cdot l_2 \cdot l_1 \)
- \( \text{image}(g, i, l) = g \odot i \)

**Full Quark Composition (Batory & Smith)**

- \( Q = M \times I \times M \) — a global modification, an introduction, and a local modification
- \( (g_2, i_2, l_2) \cdot (g_1, i_1, l_1) = (g_2 \cdot g_1 \cdot i_2 + (l_2 \odot i_1)) \cdot l_2 \cdot l_1 \)
- \( \text{image}(g, i, l) = g \odot i \)

**Modifiers of Modifiers (Batory & Smith)**

- \( Q = (M \rightarrow M) \times M \times I \times M \) — a modifier of modifiers, a global modification, an introduction, and a local modification
- \( (h_2, g_2, i_2, l_2) \cdot (h_1, g_1, i_1, l_1) = (h_2 \cdot h_1 \cdot g_2 \cdot g_1 \cdot i_2 + (l_2 \odot i_1)) \cdot l_2 \cdot l_1 \)
- \( \text{image}(h, g, i, l) = R^0(g \odot i) \)

In many cases, quark composition \( \bullet \) forms a monoid, with the appropriate tuple of units as the unit for \( \bullet \). Delta application \( -(\cdot) : Q \times I \rightarrow I \) (Definition 10) can be defined, for example, as \( q(p) = \text{image}(q \bullet (p, 1)) \), where \( q \in Q \) is a quark and \( p \in P \) is the core product. Note that the term \( (p, 1) \) needs to be adapted depending on the notion of quark being used.

In the absence of other axioms, not all of the quarks above form a deltoid, nor is delta application always a direct action. Global quark composition and full quark composition (Apel et al.) are not associative, and delta application forms an action only for local quark composition. In addition, Apel et al.’s’ global quark composition and full quark composition produces results such as the following (for global quark composition)

\[
(i_3, g_3) \cdot (i_2, g_2) \cdot (i_1, g_1) = ((g_3 \cdot g_2) \cdot g_1) \odot ((g_3 \cdot g_2) \cdot (i_3 + i_2)) + i_1 \),
\]

which applies modifications \( g_2 \) and \( g_3 \) multiple times. To get this composition to behave, strong idempotence criteria are proposed, but these exclude modifications such as method wrapping.

We now describe how to encode local quark composition, full quark composition, and modifiers of modifiers more directly in our setting, by making introductions a kind of modification and by eliminating quarks (where possible). By ignoring the distinction between modifications and introductions, we can focus on deltas alone, and work in a simpler algebraic setting.

### 6.1.1 Encoding Local Quark Composition

Before proceeding, we note that \((0, 1)\) is the unit of \( \bullet \) for local quark composition, and that apart from the monoid laws for \( \bullet \) it also satisfies: \( (i_1, m_1) = (i_2, m_2) \) if and only if \( i_1 = i_2 \) and \( m_1 = m_2 \).

The following definition introduces deltoid \( M_l \) consisting of deltas that are modifications \( m \in M \) and introductions \( i \in I \). We show that this is equivalent to \( Q = I \times M \) with \( \bullet \) corresponding to local quark composition.

**Definition 27.** Given \( (M, \bullet, 1), (I, +, 0), \odot \) as above. Define a monoid \( M_l = ((M \cup I) \times-I, \cdot) \), where \( \cdot \) is concatenation with unit the empty sequence \( \epsilon \), subject to the following equations \( m, n \in M, i, j \in I, \mu, \nu, \eta \in M_2 \):

1. \( \epsilon \cdot \mu = \mu \cdot \epsilon = \mu \)
2. \( \mu \cdot (\nu \cdot \eta) = (\mu \cdot \nu) \cdot \eta \)
3. \( m \cdot n = m + n \)
4. \( i \cdot j = i + j = j + i = j \cdot i \)

Definition 27 forms a deltoid by taking sequences of modifications and introductions, modulo certain equations. The equations interpret various combinations of elements of \( M_2 \) in terms of the original collection of operations. The most interesting is 3, which applies a modification \( m \) to an introduction \( i \), via \( m \odot i \), and shuffles \( m \) later in the sequence to apply to subsequent introductions. Note that equations 1 and 2 are redundant and follow from the fact that \( - \) is concatenation and \( 1 = \epsilon \) its unit.

Delta action is defined inductively over the elements of \( M_2 \), applying each element of \( M_2 \) to \( I \) via the appropriate function from the original monoids.

**Definition 28.** The delta action \(-(-) : M_2 \times I \rightarrow I \) for \( M_l \) is

- \( \epsilon(p) = p \)
- \( m(p) = m \odot p \)
- \( i(p) = i + p \)
- \( (\mu \cdot \nu)(p) = \mu(\nu(p)) \)

where \( m \in M, i \in I, \mu, \nu \in M_2 \) and \( p \in I \).

The following is a monoid homomorphism from quarks to \( M_l \).

**Definition 29.** Define \([-] : Q \rightarrow M_l \) as

\[
[i, m] = i \cdot m,
\]

The mapping from \( M_2 \) to quarks defined in the following is also a monoid homomorphism.

**Definition 30.** Define \([-\cdot] : M_l \rightarrow Q \) as

\[
\langle i \rangle = (0, 1),
\langle m \rangle = (0, m),
\langle i \rangle = (i, 1),
\langle \mu \cdot \nu \rangle = \langle \mu \rangle \bullet \langle \nu \rangle.
\]

Quarks with local quark composition are isomorphic to \( M_l \) which is stated in Theorem 4, supporting that making the distinction between introductions and modifications is unnecessary.

**Theorem 4.** For all \( q, q' \in Q \) and \( \mu, \nu \in M_2 \), we have

\[
1. \langle [q] \rangle = q.
\]
2. \([\mu q] = \mu\),
3. if \(q = q'\), then \([q] = [q']\), and
4. if \(\mu = \nu\), then \([\mu] = [\nu]\).

Theorem 5 shows that not only are quarks and \(M_1\) isomorphic, their notions of delta action behave the same.

**Theorem 5.** For all \(q \in Q\) and all \(p \in I\),
\[
\text{image}(q \cdot (p, 1)) = [q](p)
\]
and for all \(\mu \in M_1\) and all \(i \in I\),
\[
\text{image}(\mu) \cdot (p, 1)) = \mu(p).
\]

### 6.1.2 Encoding Batory and Smith’s Full Quark Composition

Encoding full quark composition is straightforward. To do so, we adapt the encoding above to use quarks and composition:

\[
\langle \langle \mu \rangle \rangle (n, \nu)(p) = \langle \langle \mu \rangle \rangle (n \circ (\nu(p)))
\]

whereas

\[
\langle \langle \mu \rangle \rangle (n, \nu)(p) = m \circ \langle \langle \mu \rangle \rangle (n \circ (\nu(p)))
\]

If we instantiate \(\mu\) and \(\nu\) with \(m'\) and \(n'\) we have in the first case:

\[
\langle \langle \mu \rangle \rangle (m \circ (n \circ (\nu(p))) = \langle \langle \mu \rangle \rangle (m \circ (n \circ (\nu(p)))
\]

and in the second case

\[
\langle \langle \mu \rangle \rangle (m' \circ (n' \circ (\nu(p))) = \langle \langle \mu \rangle \rangle (m' \circ (n' \circ (\nu(p)))
\]

which are equal in general only if \(\mu\) is commutative.

### 6.1.3 Encoding Batory and Smith’s Modifiers of Modifiers

Encoding modifers of modifiers is also straightforward. We assume that such modifiers, \(h : M \to M\), are endomorphisms on the monoid of modifications (this is already implicit in Batory and Smith’s \(I^2\) function): that is, \(h(1) = 1\) and \(h(m_n \circ m_1) = h(m_n) \cdot h(m_1)\).

We can extend the previous example to apply higher-order modifiers to the global compositions as follows:

- **quarks**: \(Q = (M \to M) \times M \times M\) — a modifier of modifiers, a global modification, and a delta
- **composition**: \(\langle h_2, g_2, \mu_2 \rangle \cdot (h_1, n_1, \mu_1) = (h_2, g_2, \mu_2 \cdot \mu_1)\), and
- **delta application**: \(\langle h, g, \mu \rangle (p) = h(g) \cdot \langle \mu(p) \rangle\).

To modify this definition so that \(h\) applies along the local modifications requires lifting \(h : M \to M\) to \(h_1 : M_1 \to M_1\), defined by the following equation:

\[
h_1(m) i = h(m) i.
\]

where \(h(i) = h(1 \cdot i) = h(1) \cdot i = 1 \cdot i = i\). In this case the delta application becomes

\[
\langle h, g, \mu \rangle (p) = h(g) \cdot \langle \mu(p) \rangle.
\]

Again delta application is not an action, for the same reason as for full quark composition.

Apel et al. [3] give the signatures of an entire hierarchy of modifiers of modifiers, but provide no further details.

### 6.2 Darcs and Patch Theory

The version control system Darcs is formalised in terms of patch theory [16]. The underlying formalism has some similarities with our work. Most notable is that ‘patches’ are modelled using a semigroup with inverses. This structure is a monoid at heart, with additional properties (such as inverses) that do not entirely make sense in our setting. The most significant similarity is that they deal with conflictors (entities for resolving conflicts), which are similar to our conflict resolving deltas. Conflictors have a more complex set of properties than our conflict resolving deltas due to the added structure of their core setting. Patch theory should nonetheless offer inspiration to guide future research.

### 7. Related Work

In general, approaches to facilitating automated product generation for software product lines can be classified in two main directions [20]. Firstly, annotative approaches, such as conditional compilation, frames [37] or Colored Featherweight Java (CFJ) [18], mark a model of the complete product line with respect to product features and remove marked product parts to obtain a product for a particular feature configuration.

Secondly, compositional approaches, such as delta modeling [30–32], associate product fragments to product features, which are assembled to implement a particular feature configuration. A prominent example of this approach is AHEAD [5], which can be applied on the design as well as on the implementation level. In AHEAD, a product is built by stepwise refinement of a base module with a sequence of feature modules. Design-level models can also be constructed using aspect-oriented composition techniques [14, 25, 36]. Apel et al. [29] apply model superposition to compose model fragments. Perrouin et al. [27] obtain a product model by model composition and subsequently refinement by model transformation. In Haugen et al. [13], a set of models is represented by a base model with associated variability and resolution models determining how modeling elements of the base model have to be replaced for a particular product model.

On the programming language level, several program modularization techniques [22], such as aspects [19], framed aspects [23], mixins [33], hyperslices [34] or traits [7, 12], are used to implement features in a compositional fashion. In addition, the modularity concepts of recent languages, such as Scala [26], or NewSpeak [8], can be used to represent product features. CeasarJ [24] is proposed as a combination of feature modules and aspects extending feature-oriented programming with means to modularize crosscutting concerns.

The notion of program deltas was introduced by Lopez-Herrejon et al. [22] to describe the modifications of object-oriented programs. Schaefer et al. [32] introduced the concept of delta modeling as a means to develop product line artifacts suitable for automated product derivation and implemented with framework technology [37]. In subsequent work [30], delta modeling is extended to a seamless model-based development approach for SPLs where an initial product line representation is stepwise refined until an implementation can be generated. The conceptual ideas of delta modeling have also been instantiated on the programming language level in an extension of Java with core and delta modules allowing the automatic generation of Java-based product implementations [31].

Originally, the delta model of a product line consisted of a single core and a set of incomparable product deltas [30, 32]. Conflicts between deltas applicable for the same feature configuration were prohibited. In order to express all possible products, an additional delta covering the combination of the potentially conflict.
ing deltas had to be specified leading to product fragments. Subsequently, a partial ordering between deltas was introduced [31]. However, it was required that all conflicts were manually resolved by specifying an appropriate ordering. In contrast, in this paper, a more flexible notion of conflicts and conflict resolution is proposed that allows intermediate conflicts between deltas as long as they are eliminated later in a derivation by a conflict-resolving delta.

8. Conclusion

Delta modeling is an approach to facilitating automated product derivation for software product lines. In this paper, we generalized the conceptual ideas of delta modeling in an abstract, algebraic setting. The main contribution of this work is the novel treatment of conflicts between deltas by explicit conflict-resolving deltas. In order to ensure that for every valid feature configuration a unique product is generated, a conflict-resolving delta has to exist for every pair of conflicting deltas in the model. We presented efficient conditions that allow checking the unambiguity of a product line without requiring to generate all products.

For future work, we will be using the ideas of abstract delta modeling for the implementation of variability within the HATS ABS language [1]. In addition, we are planning to extend abstract delta modeling with a concept of hierarchy so that a delta can itself be a delta model. This will give rise to a more modular development technique for product lines based on nested delta models.

References

[12] S. Ducasse, O. Nierstrasz, N. Schärl, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. ACM TOPLAS, 28(2), 2006.
A. Lemmas and Proofs

This appendix is included for evaluation purposes only. It contains proofs and lemmas left out of the paper for space reasons.

Section 5.2

Proof of Lemma 4.

We show that $\otimes$ is associative and that $\otimes$ is its neutral element.

First, we note that if an operator $\otimes$ is associative, then operator $\circ$ is associative as well.

We show that $\oplus_M$ is associative, i.e. $(a \oplus_M b) \oplus_M c = a \oplus_M (b \oplus_M c)$ by case distinction on $a$:

- Case $a = -r$ or $a \in M$. $(a \oplus_M b) \oplus_M c = a \oplus_M (b \oplus_M c)$
- Case $a = \otimes$. $(a \oplus_M b) \oplus_M c = b \oplus_M c = a \oplus_M (b \oplus_M c)$.

Thus $\oplus_M$ is also associative. We show that $\otimes_C$ is associative, i.e. $(a \otimes_C b) \otimes_C c = a \otimes_C (b \otimes_C c)$ by case distinction on $a$:

- Case $a = -r$ or $a \in \{r\} \times (I \rightarrow M^-)$. $(a \otimes_C b) \otimes_C c = a \otimes_C (b \otimes_C c)$
- Case $a = \otimes$. $(a \otimes_C b) \otimes_C c = b \otimes_C c = a \otimes_C (b \otimes_C c)$.

Case $a \in \{u\} \times (I \rightarrow M^-)$. Provable by case distinction on the 16 possible forms of $(b, c)$. We show the most problematic case $b, c \in \{u\} \times (I \rightarrow M^-)$. We use the associativity of $\oplus_M$:

$$ (a \oplus_M b) \oplus_M c = (u f \oplus_M u g) \oplus_M c = (u f \oplus_M u g) (\oplus_M c) $$

Thus $\otimes_C$ is also associative. And then so is $\otimes$.

Neutrality of $\otimes$ in · can be shown by realizing that $\forall x \in I$. The rest follows directly from the definition of $\otimes_C$.

Proof of Lemma 5.

Take arbitrary $x, y, z \in D$. Assume that $z \cdot x \cdot y = z \cdot y \cdot x$.

This implies that $z (i) \otimes_C x (i) \otimes_C y (i) = z (i) \otimes_C y (i) \otimes_C x (i)$ for all $i \in I$.

Assume $a \oplus_M b \oplus_M c = a \oplus_M (b \oplus_M c)$. Then for arbitrary $k$, we make a case distinction on $k$:

- Case $k = -r$ or $k \in M$. $a \oplus_M b \oplus_M c = a \oplus_M (b \oplus_M c)$
- Case $a = \otimes$. $a \oplus_M b \oplus_M c = a \oplus_M (b \oplus_M c)$

Thus $a \oplus_M b \oplus_M c = a \oplus_M (b \oplus_M c)$. Then for arbitrary $k$, we make a case distinction on $d (i)$:

- Case $d (i) = -r$ or $d (i) \in \{r\} \times (I \rightarrow M^-)$. $z (i) \otimes_C x (i) \otimes_C y (i) = z (i) \otimes_C y (i) \otimes_C x (i)$
- Case $d (i) = \otimes$. $z (i) \otimes_C x (i) \otimes_C y (i) = z (i) \otimes_C y (i) \otimes_C x (i)$

Thus $\forall x, y, z \in D$ and $a \oplus_M b \oplus_M c = a \oplus_M (b \oplus_M c)$.

Proof of Lemma 6.

This proof will follow almost the same structure as the proof of Lemma 4. Take arbitrary $l, m, n$. We make a case distinction on $l$:

- Case $l = -r$ or $l \in M$. $(l \oplus_M m) \oplus_M n = l \oplus_M n$.
- Case $l \in I$. $(l \oplus_M m) \oplus_M n = m \oplus_M n$.

Thus also $(l \oplus_M m) \oplus_M n = (l \oplus_M m) \oplus_M n = (l \oplus_M m) \oplus_M n = (l \oplus_M m) \oplus_M n$.

Proof of Lemma 7.

Take arbitrary $a, b, c$. We make a case distinction on $a$:

- Case $a = -r$ or $a \in \{r\} \times (I \rightarrow M^-)$. $(a \otimes_C b) \otimes_C c = a \otimes_C (b \otimes_C c)$
- Case $a = \otimes$. $(a \otimes_C b) \otimes_C c = a \otimes_C (b \otimes_C c)$

Thus also $(a \otimes_C b) \otimes_C c = a \otimes_C (b \otimes_C c)$. Then also $(y \cdot x) (p) = (y (x (p)))$ and $y (x (p))$.

Section 6.1.1

The following lemma captures that our notion of delta action is sensible, in that it preserves the equations in Definition 27.

Lemma 7. For all $\mu, \nu \in M$ and all $p \in P$, if $\mu = \nu$ then $\mu (p) = \nu (p)$.

Proof. We need to show that this holds for each axiom in Definition 27.

1. $(1 \otimes \mu (p)) = (1 \otimes \mu (p)) = \mu (1 \otimes p) = \mu (p)$.
2. $(\mu \cdot (\nu \cdot \eta)) (p) = (\mu \cdot (\eta (p)))$.

We now proceed by induction on the shape of $\eta$:

- Case $\eta = 1$. $\mu (\nu (\eta (p))) = \mu (\nu (\eta (p)))$.
- Case $\eta = \otimes$. $\mu (\nu (\eta (p))) = \mu (\nu (\eta (p)))$.
- Case $\eta = \otimes (\eta (p)) = (\mu \cdot \otimes (\eta (p)))$.
- Case $\eta = \otimes (\eta (p)) = (\mu \cdot \otimes (\eta (p)))$. 
- Case $\eta = \otimes (\eta (p)) = (\mu \cdot \otimes (\eta (p)))$. 

2010/7/13
\[ \eta'(\eta''(p)) = ((\mu \circ \nu)(\eta'(\eta''(p)))) = ((\mu \circ \nu)(\eta'' \bullet \eta'))(p) = ((\mu \circ \nu) \bullet (\eta'' \bullet \eta'))(p). \]

3. \( (m \bullet i)(p) = m(i(p)) = m \circ (i + p) = (m \circ i) + (m \circ p) = (m \circ i) + m(p) = ((m \circ i) \bullet m)(p). \)

4. \( (i \bullet j)(p) = i(j(p)) = i + (j + p) = j + (i + p) = (j \bullet i)(p). \)

This is also equal to \( (i + j)(p) \) and \( (j + i)(p) \).

5. \( (i \bullet i)(p) = i(i(p)) = i + (i + p) = i + p = i(p). \)

6. \( (m \bullet m)(p) = m(m(p)) = m \circ (n \circ p) = (m \circ n) \circ p = (m \circ n)(p). \)

7. \( 1(p) = 1 \circ p = \epsilon(p) = 0 + p = 0(p). \)

**Lemma 8.** The function \([ - ] : Q \to M_2\) is a homomorphism, namely, \([0,1] = \epsilon \) and for all \( q', q'' \in Q \), \([q'] \bullet [q''] = [q'] [q'']. \)

**Proof.** Firstly, \([0,1] = 0 \bullet 1 = \epsilon \bullet \epsilon = \epsilon. \)

Secondly, let \( q = (i, m) \) and \( q' = (i', m'). \) On one hand, \([i, m] \bullet [i', m'] = [i + m \circ i', m \cdot m'] = (i + m \circ i') \bullet (m \cdot m'). \) On the other hand, \([i, m] \bullet [i', m'] = [i \cdot m \circ i', m \cdot m'] = i \cdot (m \circ i') \bullet (m \cdot m'). \)

**Proof of Theorem 4.**

1. \( \langle [i, l] \rangle = \langle [i, l] \rangle = \langle 0, l \rangle \bullet \langle i, 1 \rangle = (i + 1 - 0, 1 - l) = \langle i, l \rangle. \)

2. By induction on \( \mu. \)

    - Case i. \( \langle [0, m] \rangle = \langle 0, 1 \rangle = i \bullet 1 = i. \)

    - Case m. \( \langle [m, m] \rangle = \langle 0, 0, 0 \bullet m = 1 \cdot m = m. \)

    - Case \( \mu \bullet \nu. \) \( \langle [\mu \bullet \nu] \rangle = \langle \langle \mu \bullet \nu \rangle \rangle = \langle \langle \mu \rangle \rangle \bullet \langle \langle \nu \rangle \rangle = \langle \langle \mu \rangle \rangle \bullet \langle \langle \nu \rangle \rangle = \langle \langle \mu \rangle \rangle \bullet \langle \langle \nu \rangle \rangle = \mu \bullet \nu. \) using Lemma 8 and induction hypothesis.

3. Associativity is preserved as \( \bullet \) is associative. That \( (0,1) \) is the unit of \( \bullet \) is preserved by \([ - ] \) follows from the fact that \( 0 = 1 \) and \( 1 \) is the unit of \( \bullet. \)

4. Finally, if \( \langle i_1, m_1 \rangle = \langle i_2, m_2 \rangle \), then \( i_1 = i_2 \) and \( m_1 = m_2 \). Now \([\langle i_1, m_1 \rangle] = \langle i_2, m_2 \rangle = \langle \langle \langle i_2 \rangle \rangle \rangle \).

5. We need to show that the axioms in Definition 27 are preserved by \([ - ] \). (1) follows because \( \langle [0, l] \rangle = 0 \). (2) Follows because \( \bullet \) is associative. (3) \( \langle [m \bullet i] \rangle = \langle [0, m] \bullet \langle i, 1 \rangle = \langle m \circ i, m \rangle \) which is the same as \( \langle [m \circ i \bullet m] \rangle = \langle m \circ i, m \rangle = \langle 0, m \rangle \bullet \langle i, 1 \rangle \).

6. \( \langle [i \bullet m] \rangle = \langle \langle \mu \bullet \nu \rangle \rangle = \langle \langle \mu \rangle \rangle \bullet \langle \langle \nu \rangle \rangle = \langle \langle \mu \rangle \rangle \bullet \langle \langle \nu \rangle \rangle = \langle \langle \mu \rangle \rangle \bullet \langle \langle \nu \rangle \rangle = \mu \bullet \nu. \) By induction hypothesis, there exists an \( m \) such that \( \langle \langle \mu \rangle \rangle \bullet \langle \langle \nu \rangle \rangle = \langle \langle \mu \rangle \rangle \bullet \langle \langle \nu \rangle \rangle \).

7. Similarly, applying the induction hypothesis to \( \mu \in M_2 \) and \( \langle \langle \nu \rangle \rangle = \langle \langle \nu \rangle \rangle \), we obtain that there exists an \( m \) such that \( \langle \langle \mu \rangle \rangle \bullet \langle \langle \nu \rangle \rangle = \langle \langle \mu \rangle \rangle \bullet \langle \langle \nu \rangle \rangle \). By Definition 27(delta action) this equals \( (\mu \bullet \nu)(p, n) \), and we are done.
Appendix B

Abstract Delta Modeling (journal version)

The paper “Abstract Delta Modeling (journal version)” follows.
Abstract Delta Modeling

DAVE CLARKE1† and MICHIEL HELVENSTEIJN2† and INA SCHAEFER3

1 IBBT-DistriNet, Katholieke Universiteit Leuven, Belgium
dave.clarke@cs.kuleuven.be

2 CWI, Amsterdam and LIACS, Leiden University, The Netherlands
michiel.helvensteijn@cwi.nl

3 University of Braunschweig, Germany
i.schaefer@tu-braunschweig.de

Received December 2010

Delta modeling is an approach to facilitate automated product derivation for software product lines. It is based on a set of deltas specifying modifications that are incrementally applied to a core product. The applicability of deltas depends on feature-dependent application conditions. This paper presents abstract delta modeling, which explores delta modeling from an abstract, algebraic perspective. Compared to previous work, we take a more flexible approach with respect to conflicts between modifications and introduce the notion of conflict-resolving deltas. Furthermore, we extend our approach to allow the nesting of delta models, for increased modularity and simplicity. We present conditions on the structure of deltas to ensure unambiguous product generation.

1. Introduction

A software product line (SPL) is a set of software systems, called products, with well-defined commonalities and variabilities (Clements & Northrop, 2001; Pohl et al., 2005). Software product line engineering aims at developing this set of systems by reuse in order to reduce time to market and to increase product quality. Automated product derivation (or software mass customization (Krueger, 2006)) is an approach to generating individual products from the product line artifacts without the need for manual intervention during application engineering, which can be tedious and error-prone (Deelstra et al., 2005).

Currently, product line variability is mostly represented by feature models (Kang et al., 1990; van Deursen & Klint, 2002). Features are designated product characteristics or increments of product functionality (Batory et al., 2004). A product is uniquely identified by a valid feature configuration, i.e., a legal combination of features from the feature model. On the feature model level, features are merely labels (Czarnecki & Antkiewicz, 2005). In order to be able to automatically derive a product for a particular feature

† These authors are partially supported by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (http://www.hats-project.eu)
configuration, a correspondence between the features on the feature modeling level and the reusable product line artifacts has to be introduced. Additionally, the product line artifacts have to be organized in such a way that they can be assembled automatically to generate a uniquely determined product.

Feature-oriented programming (Batory et al., 2004) is a prominent approach for implementing SPLs by compositing feature modules that directly correspond to product features. Delta modeling (Schaefer et al., 2009; Schaefer, 2010; Schaefer et al., 2010) extends feature-oriented programming. In the delta modeling approach, a product line is represented by a core product and a set of product deltas. Product deltas specify modifications to the core product to generate further products of the product line. Each delta has an application condition specifying for which feature configurations the modifications have to be carried out, connecting features on the feature modeling level with product line artifacts. A product for a feature configuration can be obtained by applying the deltas with a valid application condition to the core product.

In this article, we generalize the existing delta modeling approaches (Schaefer et al., 2009; Schaefer, 2010; Schaefer et al., 2010) and present an abstract, algebraic formalization of delta modeling. The presented abstract delta modeling approach goes beyond existing work with its novel treatment of conflicts between deltas. A conflict between deltas arises if their specified modifications do not commute. This means that applying them in different orders results in different (composed) modifications. In previous work, deltas were either considered incomparable (Schaefer et al., 2009; Schaefer, 2010), which required writing additional deltas for every conflicting combination, or they had to be ordered in a very restrictive way (Schaefer et al., 2010) to avoid conflicts explicitly. As a main contribution of this article, we introduce the notion of conflict-resolving deltas that relax these restrictions and make delta modeling of product lines more flexible. A conflict-resolving delta, which is applied after two conflicting deltas, eliminates differences between the (composed) modifications. If for every pair of conflicting deltas a conflict-resolving delta exists, all possible sequences of deltas produce the same modification and generate a uniquely defined product. In order to ensure this result for every valid feature configuration, we provide efficient-to-check conditions requiring only the inspection of the product line directly, without having to generate and check all products.

This article expends our previous work on abstract delta modeling (Clarke et al., 2010) by providing additional examples and a complete description of the formalism including all proofs. Furthermore, in this article, we extend the formalism with nested delta models as additional way of imposing structure and increasing modularity. Using nested delta models, a delta can be a delta model itself. Nested delta models are processed atomically, avoiding interference with unrelated deltas in the outer model. Thus, nesting can greatly simplify delta models, by allowing modifications to be specified within separate deltas without worrying about interference from other deltas.

The concepts of abstract delta modeling can be instantiated for different kinds of development artifacts, such as documentation, models or code. We demonstrate the feasibility of the approach by presenting an instantiation of abstract delta modeling for object-oriented implementations of software product lines and extend this with method wrapping, a key operation from aspect-oriented programming (Kiczales et al., 1997) and
Abstract Delta Modeling

The abstract delta modeling formalism consists of a number of ingredients, which are depicted in Figure 1, along with the operations between them. The ingredients of a software product line are drawn from a set of products, a set of features, and a collection of deltas (or deltoid). A software product line can then be defined by a feature model, a core product, the product deltas specifying modifications of the core product, a partial ordering on the deltas restricting their application and application conditions for the deltas. By specifying a feature configuration, one can produce a delta model, consisting of an ordered collection of modifications necessary to generate the respective product. The process of derivation applied to a delta model puts the specified modifications in a linear order that is compatible with the partial ordering in order to obtain a valid, ideally unique, composite modification. The delta application operation partially applied to a delta returns a function that takes a product and produces a new product. This function can be applied to the core product to produce a product for a given feature configuration.

This paper is organized as follows. Section 2 describes the Editor product line that is used as illustrative example in this article. Section 3 presents delta models and criteria for their unambiguity. Section 4 shows how to define product lines over delta models and how to transfer the unambiguity properties. Section 5 presents a concrete class of deltas illustrating the proposed approach. Section 6 introduces nested delta models and compares their expressiveness. In Section 7, we explain the methodology for developing product
lines by delta modeling. Section 8 relates our approach to existing algebraic approaches from the literature. Finally, Sections 9 and 10 present related work and conclusions.

2. Running Example

In this section, we introduce a product line of code editors that we use as illustrative example throughout this article. First, we describe the features of the code editor product line and then show its realization by deltas.

2.1. Feature model

The Editor product line is described by the sets of valid feature configurations that can be realized by different editors. Figure 2 depicts the feature model (Kang et al., 1990; van Deursen & Klint, 2002) of the Editor product line. In the diagram, every box represents a feature. A line with an open circle at the end stands for the optional subfeature relation. A horizontal line segment indicates exclusive choice between subfeatures. A subfeature can only be selected if its base feature is also selected. The Editor product line has the following features:

- **Editor** (Ed) is the only mandatory feature of the product line representing basic editing functionality.
- **Printing** (Pr) allows the user to print the content of an editor window.
- **Syntax Highlighting** (SH) colors code for easier recognition of programming language constructs.
- **Error Checking** (EC) performs simple grammatical analysis on code and underlines certain errors. Hovering over an error gives extra information in a tooltip.
- The optional subfeature **Semantic Analysis** (SA) of the feature **Error Checking** can detect more sophisticated errors in program code.
- **Tooltip Info** (TI) gives information about code fragments by hovering over them. The features **Tooltip Info** and **Error Checking** are mutually exclusive, since both produce different kinds of tooltips.
This product line consists of 16 editors, because there are 16 possible feature configurations.

2.2. Delta Model

Features on the level of the feature model are only labels (Czarnecki & Antkiewicz, 2005). In order to describe the realization of the different editor variants on the implementation level, we provide a delta model. Figure 3 depicts the delta model for implementing the Editor product line. The dashed boxes represent the deltas. Every delta contains class additions and/or updates. Deltas are decorated with their application conditions (bottom right of each box) which consists of a propositional logic formula, linking the delta modifications to the set of feature configurations. The arrows between the deltas represent the application ordering as a strict partial order. If two deltas are not ordered, the designer indicates that the order in which the deltas are applied should not matter.

For the Editor product line, we assume that product generation starts from the empty product. Thus, the core product is an empty program. This way, the Editor product line is more robust to evolution of the feature model, e.g., if mandatory features become optional, as pointed out by Schaefer & Damiani (2010). The delta $d_1$ is applied for any valid feature configuration, because it is annotated only with the mandatory $Ed$ feature.
It provides the basic functionality of an editor and adds this to the empty product. Deltas \( d_2, \ldots, d_5 \) implement the four optional features of the product line, \( Pr, SH, EC \) and \( TI \). They are applied if their corresponding feature is selected. Delta \( d_6 \) implements the subfeature \( SA \) of the feature \( EC \). It is applied if the \( SA \) feature is selected, which requires the feature \( EC \) to be selected as implied from the feature model. Thus, it is not necessary to annotate delta \( d_6 \) with the feature \( EC \). Delta \( d_6 \) not only modifies the \textit{Editor} class, but adds a new class as well.

Assume that the original designer of delta \( d_1 \) did not think ahead and expressed both color and underlining of text with the single method \texttt{font}. Since \( d_3 \) and \( d_4 \) both redefine this method independently (the two deltas are not ordered), we need a conflict-resolving delta \( d_8 \) to combine their functionality. Delta \( d_8 \) is selected only if the two conflicting deltas are also selected and is applied afterwards (by the partial order) so it can resolve their conflict and provide the appropriate semantics for the combination of the \( SH \) and \( EC \) features. A similar conflict is present between deltas \( d_5/d_8 \) and \( d_6 \), so we include a conflict-resolving delta \( d_9 \) to fix the interaction.

Even if two deltas are not in conflict, it might be necessary to add another delta to appropriately combine their functionality. For instance, when the \( Pr \) and \( SH \) features are selected together, we want any printout to contain the syntax highlighting colors as well. Delta \( d_7 \) implements the desired interaction between these two features. It overwrites the \texttt{print} method and uses the syntax highlighting information of delta \( d_3 \).

Note that we do not include a conflict-resolving delta for the apparent \texttt{onMouseOver} conflict between deltas \( d_4 \) and \( d_5 \). This is because the \( EC \) and \( TI \) features are mutually exclusive in the feature model. Since the two deltas are never applied together, there is no conflict to resolve.

### 3. Abstract Delta Modeling

This section presents our approach of abstract delta modeling. We introduce product modifications and their composition as a monoid, called a \textit{deltoid}. Delta models are built on top of this monoid as partially ordered collections of modifications, where the ordering constrains the possible ways such modifications can be applied. It is important that a delta model defines unambiguous modifications as these are used later to obtain distinct products. Thus, we define the notions of conflict, leading to ambiguous modifications, and conflict-resolving deltas eliminating these. We develop conditions to ensure that a delta model is unambiguous.

#### 3.1. Products and Deltas

In existing compositional approaches to implementing software product lines, such as feature-oriented programming (Batory \textit{et al.}, 2004) or delta-oriented programming (Schaefer \textit{et al.}, 2010), a member product of an SPL is obtained by the application of a number of modifications (deltas) \( x_1, \ldots, x_n \) to a core product \( c \), as follows:

\[
x_n(\cdots x_1(c) \cdots).
\]
Abstract Delta Modeling

In feature-oriented programming, the core product is determined by one or more base feature modules. The modifications are specified by further feature modules extending and refining the core product. In delta-oriented programming, the core product can be any product of the product line. As both approaches treat the core product as a constant element, it is useful to focus on the modifications. The above modification would be equivalently written as follows, where \( \cdot \) refers to the composition of modifications:

\[
(x_n \cdot \ldots \cdot x_1)(c).
\]

Thus, we will focus on sequences of modifications such as \( x_n \cdot \ldots \cdot x_1 \). It may still be possible to reason about the core product if we choose to see it as a modification \( x_c \) applied to an empty product \( 0 \), i.e. \( c = x_c(0) \). Thus:

\[
(x_n \cdot \ldots \cdot x_1)(c) = (x_n \cdot \ldots \cdot x_1 \cdot x_c)(0),
\]

so usually nothing is lost by restricting our attention to modifications. A similar approach is pursued in pure delta-oriented programming (Schaefer & Damiani, 2010; Schaefer et al., 2011) where a product is only generated from deltas applied to the empty product.

In abstract delta modeling, the main object of interest is a deltoid. A deltoid consists of a set of modifications, called deltas, along with the operation for composing them sequentially and a neutral element, i.e. the empty delta. A deltoid can contain different kinds of deltas for different kinds of development artefacts (e.g., documentation, models or code) and for different levels of abstraction (e.g., when working on component level or working on class level). The concrete nature of the modifications specified in the deltas depends on the capabilities of the underlying modeling or programming languages. In the example in Section 2, we consider modifications to object-oriented programs by adding, removing and modifying classes. Deltas may be functions performing the changes directly or some other structure representing those changes. We abstract away from the internal details of modifications, since many different instantiations are possible.

Firstly, we assume a set of products, which includes possible core products, intermediate products and end-products. In the example of Section 2, this are all possible editors.

**Definition 1 (Products).** Let \( \mathcal{P} \) denote a set of possible products.

We define the notions of deltoids and deltas as follows.

**Definition 2 (Deltoid).** A deltoid is a monoid \((\mathcal{D}, \cdot, \epsilon)\), where \( \mathcal{D} \) is a set of product modifications (referred to as deltas), and the operation \( \cdot : \mathcal{D} \times \mathcal{D} \to \mathcal{D} \) corresponds to their sequential composition. \( y \cdot x \) denotes the modification applying first \( x \) and then \( y \). The neutral element \( \epsilon \) of the monoid corresponds to modifying nothing.

The operation \( \cdot \) is associative, but not inherently commutative, as the ordering between two deltas may be significant. We call two deltas \( x, y \in \mathcal{D} \) incompatible if \( y \cdot x \neq x \cdot y \).

Applying a delta to a product results in another product. This is captured by the notion of delta application.

**Definition 3 (Delta Application).** Delta application is an operation \( -(-) : \mathcal{D} \times \mathcal{P} \to \ldots \)
If \( d \in D \) and \( p \in P \), then \( d(p) \in P \) is the product resulting from applying delta \( d \) to product \( p \).

This definition implies that one generates a product from a sequence of deltas by first composing the deltas and then applying the result to a core product. A much stronger version of delta application is possible, borrowing the notion of monoid action.

**Definition 4 (Delta Action).** A delta application operation \( -(\cdot) : D \times P \to P \) is called a *delta action* if it satisfies the conditions \((y \cdot x)(p) = y(x(p))\) and \(\epsilon(p) = p\), for all \(x, y \in D\) and \(p \in P\).

Deltoids and delta actions together generalise the case when \( \cdot \) is function composition and \( -\cdot \) is function application.

Together, a product set and a deltoid (where the product set is closed under delta application) determines all possible product lines that can be created. For instance, the set \( P \) of all object-oriented programs and a deltoid \( D \) of method/class modifications could be the basis for a specific product line of object-oriented software, e.g., the Editor product line presented in Section 2.

An *empty product* might not always exist for a given product set and a corresponding deltoid. However, a product can be used as an empty product if it can be transformed into any other product in the product set by delta application:

**Definition 5 (Empty Product).** Given a product set \( P \) and a deltoid \((D,\cdot,\epsilon)\), a product \( p \in P \) can be used as *empty product* iff \( \forall p' \in P : \exists x \in D : x(p) = p' \).

The existence of an empty product according to the above definition indicates that the deltoid is sufficiently expressive to describe any product of the product set. Of course, the actual suitability of a product as the empty (core) product depends on the language in which the products are realized. For instance, for software, the empty program may be most suitable, despite the richness of the deltoid qualifying other products for that role. Aspect-oriented programming (Kiczales et al., 1997) has no notion of an empty program, because advice is woven in at pointcuts which have to be defined in a base program containing existing classes. Advice can only add statements before, after or around existing statements identified by the pointcut. Advice cannot add new classes.

Related to the notion of empty product is that of a maximally expressive deltoid. This states that for any two products there is a modification to convert one into the other.

**Definition 6 (Maximal Expressivity).** Given a product set \( P \). A deltoid \((D,\cdot,\epsilon)\) is said to be *maximally expressive* iff \( \forall p, p' \in P : \exists x \in D : x(p) = p' \).

A deltoid that is able to add, but not to remove, elements will not be maximally expressive. In addition, for a maximally expressive deltoid, every product can serve the role of empty product.

From here on, a certain product set \( P \) and deltoid \((D,\cdot,\epsilon)\) are assumed to be given for all definitions, unless otherwise specified.
3.2. Delta Models

A delta model describes the collection of deltas required to build a specific product, along with a strict partial ordering on those deltas restricting the order in which they can be applied. Recall that a strict partial order is irreflexive, asymmetric and transitive. Figure 3 depicts the delta model for the Editor product line.

**Definition 7 (Delta Model).** A delta model is a tuple $(D, \prec)$, where $D \subseteq D$ is a finite set of deltas and $\prec \subseteq D \times D$ is a strict partial order on $D$. $x \prec y$ states that $x$ should be applied before, though not necessarily directly before $y$.

The partial order between deltas represents the intuition that a subsequent delta has full knowledge of (and access to) earlier deltas and more authority over modifications to the product. This is realized by applying the deltas in a linear extension of the partial order, as shown in the following definition. A derivation is a sequential composition of all deltas from a delta model, necessary to generate a desired product.

**Definition 8 (Derivation).** Given a delta model $P = (D, \prec)$, its derivations are defined to be

$$\text{derv}(P) \triangleq \left\{ x_n \cdot \ldots \cdot x_1 \mid x_1, \ldots, x_n \text{ is a linear extension of } \prec, \text{ where } \{x_1, \ldots, x_n\} = D \right\}$$

Note that $\text{derv}(P)$ may potentially generate more than one distinct derivation as incompatible deltas may be applied in different orderings. However, it is desirable that all possible derivations of a delta model have the same effect, as this corresponds to deriving a unique product. This motivates the following definition.

**Definition 9 (Unique Derivation).** A delta model $P = (D, \prec)$ is said to have a unique derivation iff $x_n \cdot \ldots \cdot x_1 = x'_n \cdot \ldots \cdot x'_1$ for all pairs of linear extensions $(x_1, \ldots, x_n)$ and $(x'_1, \ldots, x'_n)$ of $\prec$, or, equivalently, iff $|\text{derv}(P)| = 1$.

3.3. Unambiguity of Delta Models

The property that a delta model has a unique derivation can be checked by brute force. This means generating all possible derivations (in the worst case, $n!$ for $n$ deltas), and then checking that they all correspond. In order to allow for a more efficient way to establish this property, we introduce unambiguous delta models which rely on the notions of conflicting deltas and conflict-resolving deltas.

Two deltas are in conflict if they are incompatible and no ordering is placed upon them. Intuitively, the two conflicting deltas are independently modifying the same part of the product in different ways, meaning that multiple distinct derivations may be possible. For example, in Figure 3 the deltas $d_3$ and $d_4$ are in conflict, since they both redefine the `font` method in different ways.

**Definition 10 (Conflict).** Given a delta model $P = (D, \prec)$, $x, y \in D$ are said to be in conflict iff the following condition holds:

$$x \not\prec y \triangleq y \cdot x \neq x \cdot y \land \lnot y \land \lnot x.$$
One way to ensure a unique derivation is to avoid conflicts by always enforcing an ordering between incompatible deltas (Schaefer et al., 2010). However, features with conflicting implementations are often independent in concept. Kästner et al. (2009) call the issue of how to model such situations the \textit{optional feature problem}. Imposing an ordering on the deltas of conceptually orthogonal features is often inappropriate. Some (unrelated) functionality may be inadverently and silently overwritten. Furthermore, sometimes neither of the original choices in functionality is exactly what is required, and, instead, some combination has to be used. The alternative is to allow conflicts, but to provide additional, subsequently applied, deltas to resolve them. In Figure 3, the conflict-resolving delta for the conflict between deltas $d_3$ and $d_4$ is delta $d_5$ that combines the functionality introduced by $d_3$ and $d_4$.

\textbf{Notation 1.} If $D \subseteq \mathcal{D}$, then $D^*$ denotes the sequences of compositions of deltas from $D$.

\textbf{Definition 11 (Conflict-Resolving Delta).} Given a delta model $P = (D, \prec)$ and $x, y \in D$ which are in conflict, we say that a delta $z \in D$ \textit{resolves} their conflict iff the following property holds:

$$(x, y) \triangleleft z \iff x \triangleleft z \land y \triangleleft z \land \forall d \in D^*: z \cdot d \cdot x \cdot y = z \cdot d \cdot z \cdot y \cdot x \cdot d_1.$$ 

An unambiguous delta model is a delta model containing a conflict-resolving delta for every conflicting pair of deltas. Conflict-resolving deltas take the role of \textit{derivative modules} (Kästner et al., 2009) or \textit{lifters} (Pehofer, 1997). They contain only the functionality necessary when several interacting features are selected together. The delta model of the product line presented in Figure 3 is strictly speaking not unambiguous, since $d_4$ and $d_5$ are in conflict without a conflict-resolving delta. However, since the two deltas have mutually exclusive application conditions (which we formalize in Section 4), the delta model for the Editor product line is still unambiguous.

\textbf{Definition 12 (Unambiguous Delta Model).} Given a delta model $(D, \prec)$, we say that the model is \textit{unambiguous} iff

$$\forall x, y \in D: x \not\prec y \Rightarrow \exists z \in D: (x, y) \triangleleft z.$$ 

If a delta model is unambiguous, it can be shown to have a unique derivation. In order to prove this, some intermediate results are required. Lemma 1 states that in an unambiguous delta model, any two deltas in a derivation are either ordered or commutative.

\textbf{Lemma 1.} Given an unambiguous delta model $P = (D, \prec)$ and $d_2 \cdot y \cdot x \cdot d_1 \in \text{der}(P)$, where $x, y \in D$ and $d_1, d_2 \in D^*$. Then either $x \prec y$ or $d_2 \cdot y \cdot x \cdot d_1 = d_2 \cdot x \cdot y \cdot d_1$.

\textbf{Proof.} By case distinction on the unambiguity of $P$ for deltas $x$ and $y$:

- Case $y \cdot x = x \cdot y$. It follows directly that $d_2 \cdot y \cdot x \cdot d_1 = d_2 \cdot x \cdot y \cdot d_1$.
- Case $x \prec y$. Immediate.
- Case $y \prec x$. Cannot happen, otherwise $d_2 \cdot y \cdot x \cdot d_1$ would not be a linear extension of $\prec$.
- Case $\exists z \in D: (x, y) \prec z$. Firstly, from the definition of conflict resolving delta we have that $x, y \prec z$, hence there exist $d'_{\rho}, d''_{\rho} \in D^*$ such that $d_2 \cdot y \cdot x \cdot d_1 = d'_{\rho} \cdot z \cdot d''_{\rho} \cdot y \cdot x \cdot d_1$. 

Abstract Delta Modeling

From the remaining condition on $z$, we have $z \cdot d''_2 \cdot y \cdot x = z \cdot d''_2 \cdot x \cdot y$, from which we deduce $d_2 \cdot y \cdot x \cdot d_1 = d'_2 \cdot z \cdot d''_2 \cdot y \cdot x \cdot d_1 = d'_2 \cdot z \cdot d''_2 \cdot x \cdot y \cdot d_1 = d_2 \cdot y \cdot x \cdot d_1$.

Lemma 2 states that removing a minimal element with respect to the partial order preserves unambiguity of delta models.

**Lemma 2.** If $P = (D, \prec)$ is an unambiguous delta model and $w$ is minimal in $\prec$, then $(D \setminus \{w\}, \prec')$, where $\prec'$ is $\prec$ restricted to $D \setminus \{w\}$, is also an unambiguous delta model.

**Proof.** If $(D, \prec)$ is unambiguous, then $\forall x, y \in D : x \not \prec y$ implies $\exists z \in D : (x, y) \not \prec z$. For this to be true in $D \setminus \{w\}$ we need to show that there are no $x$ and $y$ such that $x \not \prec y$ with $w$ such that $(x, y) \not \prec w$. But $w$ could not have been a conflict resolver, as it is minimal, contradicting conditions $x \not \prec w$ and $y \not \prec w$ of $(x, y) \not \prec w$.

Lemma 3 formulates that a minimal element in the partial order can be moved to the front of any derivation from an unambiguous delta model without changing the meaning of that derivation.

**Lemma 3.** Given an unambiguous delta model $P = (D, \prec)$. Let $x_n \ldots x_1 \in \text{derv}(P)$, where $\{x_1, \ldots, x_n\} = D$, with $x_i$ minimal in $\prec$. Then

$$x_n \cdot \ldots \cdot x_1 = x_n \cdot \ldots \cdot x_{i-1} \cdot x_i \cdot x_{i-1} \cdot \ldots \cdot x_1 \cdot x_i.$$

**Proof.** By induction on $i$:

— Case $i = 1$. Immediate.

— Case $i > 1$. As $x_i$ is minimal, $x_{i-1} \not \prec x_i$ holds. Lemma 1 implies that $x_n \cdot \ldots \cdot x_1 = x_n \cdot \ldots \cdot x_{i-1} \cdot x_i \cdot x_{i-1} \cdot \ldots \cdot x_1$. Now $x_i$ is in position $i - 1$, so induction gives $x_n \cdot \ldots \cdot x_{i+1} \cdot x_i \cdot x_{i-1} \cdot \ldots \cdot x_1 = x_n \cdot \ldots \cdot x_{i+1} \cdot x_{i-1} \cdot \ldots \cdot x_1 \cdot x_i$.

The following theorem states that every unambiguous delta model has a unique derivation. This reduces the effort of checking that all possible derivations of a delta model have the same effect to checking that all conflicts between pairs of deltas are eliminated by conflict-resolving deltas. The proof is by induction over the size of the delta model.

**Theorem 1.** An unambiguous delta model has a unique derivation.

**Proof.** Given unambiguous delta model $P = (D, \prec)$. Proceed by induction on the size of $D$:

— Case $|D| = 0$. Immediate as $\text{derv}(P) = \{\epsilon\}$.

— Case $|D| = 1$. Immediate as $\text{derv}(P) = \{x\}$, where $D = \{x\}$.

— Case $|D| > 1$. For any two $d_1, d_2 \in \text{derv}(P)$, let $d_1 = d'_1 \cdot x$ and $d_2 = d'_2 \cdot x \cdot d''_2$, where $x \in D$ and $d'_1, d'_2, d''_2 \in D^\ast$. As $x$ is the last element of $d'_1$, it must be minimal in $\prec$. Thus, by Lemma 3, $d'_1 \cdot x \cdot d''_2 = d'_2 \cdot d''_2 \cdot x$. Now by Lemma 2, $P' = (D \setminus \{x\}, \prec')$, where $\prec'$ is $\prec$ restricted to $D \setminus \{x\}$, is an unambiguous delta model, and $d'_1, d'_2, d''_2 \in \text{derv}(P')$. By the induction hypothesis, $d'_1 = d'_2 \cdot d''_2$ and thus $d_1 = d'_1 \cdot x = d'_2 \cdot d''_2 \cdot x = d_2$. Hence, $|\text{derv}(P)| = 1$.
3.4. Consistent Conflict Resolution

Although the notion of unambiguous delta models alleviates the task of establishing that a delta model has a unique derivation, unambiguity is still quite complex to check. The reason is that the definition of a conflict-resolving delta (Definition 11) quantifies over all elements of $D^*$. Hence, in order to check that a delta is indeed a conflict resolver, all these sequences of deltas have to be inspected. Naturally, we could restrict the checks to consider only relevant elements of $D^*$, but in this section, we propose a simpler criterion to make checking ambiguity more feasible for interesting classes of deltoids. The consistent conflict resolution property states that if a delta $z$ resolves an $(x,y)$-conflict when it is applied directly after $x$ and $y$, it also resolves the conflict after the application of any sequence of intermediate deltas.

Definition 13 (Consistent Conflict Resolution). A deltoid $(\mathcal{D}, \cdot, \epsilon)$ is said to exhibit consistent conflict resolution iff the following condition holds:

$$\forall x,y,z \in \mathcal{D}: z \cdot y \cdot x = z \cdot x \cdot y \Rightarrow \forall d \in \mathcal{D}: z \cdot d \cdot y \cdot x = z \cdot d \cdot x \cdot y.$$ 

If a deltoid $(\mathcal{D}, \cdot, \epsilon)$ exhibits consistent conflict resolution, then a delta model $(\mathcal{D}, \prec)$ with $\mathcal{D} \subseteq \mathcal{D}$ is also said to exhibit the property.

The consistent conflict resolution property is checked at the level of the underlying deltoid, rather than for any specific delta model. Hence, it has to be established only once for a given deltoid and then holds for all delta models based on that deltoid. To establish the unambiguity of a delta model exhibiting consistent conflict resolution, it is sufficient to check that for each pair of conflicting deltas $x$ and $y$ there exists a conflict-resolving delta $z$, such that $x \prec z \land y \prec z \land z \cdot y \cdot x = z \cdot x \cdot y$. We need not quantify over all possible intermediate sequences of deltas. Consequently, unambiguity of delta models can be established much more efficiently. This is formalized in the next theorem.

Theorem 2. Given a delta model $P = (\mathcal{D}, \prec)$ exhibiting consistent conflict resolution, for all deltas $x, y, z \in \mathcal{D}$ such that $x \prec z$ and $y \prec z$, if $z \cdot y \cdot x = z \cdot x \cdot y$, then $(x,y) \prec z$.

Proof. From the definition of consistent conflict resolution, we have the following:

$$z \cdot y \cdot x = z \cdot x \cdot y \Rightarrow \forall d \in \mathcal{D}: z \cdot d \cdot y \cdot x = z \cdot d \cdot x \cdot y$$

$$\Rightarrow \forall d \in \mathcal{D}^* : z \cdot d \cdot y \cdot x = z \cdot d \cdot x \cdot y.$$ 

Combined with the facts $x \prec z$ and $y \prec z$, this is precisely the definition of $(x,y) \prec z$. □

4. Product Lines

Using the introduced concepts of delta models, products and delta application, we can now abstractly define product lines, thus, providing a link from feature configurations on the feature modeling level to products. We extend the concept of unambiguity to the level of product lines and provide an efficient condition to check it.
4.1. **Defining Product Lines**

Product line variability is predominantly captured by features, where a feature captures a designated product characteristic or an increment to product functionality. At the level of the feature model features are merely labels without inherent semantic meaning. A product can be characterized by the set of features it provides.

**Definition 14 (Features).** Let $\mathcal{F}$ denote a universal set of features.

The set of products in a product line can be represented by a feature model. Many formal descriptions (Kang et al., 1990; van Deursen & Klint, 2002; Heymans et al., 2008) agree that a feature model determines a set of valid feature configurations. Figure 2 shows the feature diagram of the Editor product line.

**Definition 15 (Feature Model).** A feature model $\Phi \subseteq \mathcal{P}(\mathcal{F})$ is a set of sets of features from $\mathcal{F}$. Each $F \in \Phi$ is a set of features corresponding to a valid feature configuration.

For the sake of simplicity, our formalism does not consider extended feature models, such as cardinality-based or attributed feature models (Czarnecki et al., 2004; Czarnecki & Kim, 2005), though we believe that they can be added in a straightforward fashion.

In order to bridge the gap between features and product line artifacts, we introduce application conditions for deltas. An application condition attached to a delta determines for which feature configurations the delta has to be applied.

**Definition 16 (Application Function and Condition).** Let $D \subseteq \mathcal{D}$ be a set of deltas. An application function $\gamma : D \to \mathcal{P}(\mathcal{P}(\mathcal{F}))$ gives the feature configurations each delta $x \in D$ is applicable to. Thus, $F \in \gamma(x)$ denotes that delta $x$ is applicable for feature configuration $F$. $\gamma(x)$ is called the application condition for $x$.

A product line is defined by its feature model, characterizing all member products by a set of valid feature configurations, the core product, the associated delta model, containing the modifications used to obtain further products, and the application function, associating features and deltas. None of these elements can be inferred from the other elements. For the Editor product line presented in Section 2, the feature model is depicted in Figure 2, the core product is the empty product and the delta model is shown in Figure 3 containing the application ordering and the application conditions attached to the deltas.

**Definition 17 (Product Line).** A product line is a tuple $PL = (\Phi, c, D, \prec, \gamma)$, where $\Phi$ is a feature model, $c \in \mathcal{P}$ is the core product, $(D, \prec)$ is a delta model and $\gamma$ is an application function with domain $D$ such that $\forall x \in D : \gamma(x) \subseteq \Phi$.

If feature configuration $F$ is valid according to $\Phi$, its corresponding product is defined by the delta model containing only the deltas applicable to $F$. Selecting those deltas yields a delta model whose derivation(s) applied to the core product correspond to the desired product for feature configuration $F$.

**Definition 18 (Selected Delta Model).** Given a product line $PL = (\Phi, c, D, \prec, \gamma)$, a selected delta model for feature configuration $F \in \Phi$, denoted $PL \upharpoonright F$, is the delta model...
(D′,≺′) where D′ = {d ∈ D | F ∈ γ(d)} is the set of applicable deltas, and ≺′ is ≺ restricted to D′.

We now define the set of products generated from a product line.

**Definition 19 (Generated Products).** Given a product line PL = (Φ, c, D, ≺, γ), the set of generated products for feature configuration F ∈ Φ is defined as follows:

\[
\text{prod}(PL, F) \overset{\text{def}}{=} \{ x(c) \mid x ∈ \text{derv}(PL \upharpoonright F) \}.
\]

### 4.2. Unambiguity of Product Lines

As argued in Section 3, unambiguity of delta models is a desired property because it ensures unique derivation and, consequently, a unique generated product. We now lift unambiguity to the product line level. A product line is unambiguous if every selected delta model is unambiguous. This means that every valid feature configuration yields a uniquely defined product, which is an important condition for the applicability of automated product derivation.

**Definition 20 (Unambiguous Product Line).** A product line PL = (Φ, c, D, ≺, γ) is unambiguous if

\[∀ F ∈ Φ : PL \upharpoonright F \text{ is an unambiguous delta model}.\]

The unambiguity of a product line can be checked by generating the selected delta models of all valid feature configurations and checking then unambiguity by the criteria proposed in Section 3. However, as the set of feature configurations is often exponential in the number of features, this naive approach would be rather expensive. Instead, we propose the notion of a globally unambiguous product line that implies product line unambiguity. First, we introduce a shorthand notation for the set of feature configurations for which two deltas x and y are applicable.

**Notation 2.** Given a product-line (Φ, c, D, ≺, γ), the set of valid feature configurations to which the deltas x, y ∈ D apply is denoted:

\[V_{x,y} \overset{\text{def}}{=} γ(x) \cap γ(y).\]

A product line is globally unambiguous if for any two conflicting deltas x and y applied together for a set of feature configurations, there is a conflict-resolving delta z applicable in at least the same set of feature configurations. Thus, for any selected delta model in which x and y appear together, the conflict is resolved by the same delta z. Global unambiguity of a product line can be checked by inspecting the product line only once and does not require all selected delta models to be generated.

**Definition 21 (Globally Unambiguous Product Line).** A product line (Φ, c, D, ≺, γ) is called globally unambiguous if and only if for all x, y ∈ D such that V_{x,y} ≠ ∅, if x E y, then there exists a z ∈ D such that V_{x,y} ⊆ γ(z) and (x, y) ◁ z.

D. Clarke and M. Helvensteijn and I. Schaefer
The following theorem states that any globally unambiguous product line is also an unambiguous product line. Hence, it suffices to check global unambiguity by inspecting the product line once to ensure that all products that can be generated from the product line are uniquely determined. In the following proof, we annotate the $\mathcal{I}$ and $\mathcal{C}$ operators with the delta model for which they apply, e.g., $x\mathcal{I}_P y$ and $(x, y) \mathcal{C}_P z$.

**Theorem 3.** A globally unambiguous product line is unambiguous.

**Proof.** Assume $PL = (\Phi, c, D, \prec, \gamma)$ is a globally unambiguous product line. Let $F \in \Phi$ be a valid feature configuration. We show that $P = PL \mid F = (D', \prec')$ is an unambiguous delta model.

Given arbitrary $x, y \in D'$ (so also $x, y \in D$), we perform a case analysis on the global unambiguity of $PL$ (Definition 21). Observe that $F \in V(x, y)$, otherwise $x$ and $y$ would not be in $D'$. Now for the cases:

- Case $V(x, y) = \emptyset$. Cannot happen, otherwise $x, y \notin D'$.
- Case $\neg (x \mathcal{I}_D y)$. So also $\neg (x \mathcal{I}_P y)$.
- Case $\exists z \in D : V(x, y) \subseteq \gamma(z) \land (x, y) \mathcal{C}_{(D, \prec)} z$. Note that $z \in D'$, as $V(x, y) \subseteq \gamma(z)$. It follows that $(x, y) \mathcal{C}_P z$ (Definition 18).

So, for all $x, y \in D'$, either $\neg (x \mathcal{I}_P y)$ or $\exists z \in D' : (x, y) \mathcal{C}_P z$. Hence, $P$ is an unambiguous delta model (Definition 12).

A product line can be unambiguous without being globally unambiguous if conflicts between two deltas $x$ and $y$ are resolved by different conflict-resolving deltas $z$ for different feature configurations. For example, take a product line $PL$ in which the only conflicting deltas $x$ and $y$ are applied together for feature configurations $F$ and $F'$. For feature configuration $F$, only delta $z$ resolves the conflict, $(x, y) \mathcal{C}_{PL} z$, and for feature configuration $F'$ only delta $z'$ resolves the conflict, $(x, y) \mathcal{C}_{PL'} z'$, but $z \neq z'$. Such a product line is unambiguous, because the conflict is resolved in all selected delta models, but not globally unambiguous, because the conflict-resolving delta is not the same.

5. Deltoids for Object-Oriented Programs

In this section, we present a concrete product set and deltoid for object-oriented programs to demonstrate our approach. Deltas manipulate object-oriented programs on a coarse-grained level. That is, a delta can add, remove or modify classes. Modifications of classes include addition, removal and replacement of fields and methods, as used for the Editor product line presented in Section 2.

**Notation 3.** Let $f : X \rightarrow Y$ denote that $f$ is a partial function from $X$ to $Y$. If $f(x)$ is undefined for $x \in X$, write $f(x) = \bot$, where $\bot \notin Y$.

**Notation 4.** Given a set $X$ where $- \notin X$, define the notation:

$$X^- \equiv X \cup \{-\}.$$
5.1. Software Products

For simplicity, we abstract from a concrete programming language, as well as from concrete implementations of methods, and focus on the structural aspects of object-oriented programs. First, we introduce the notion of identifiers for classes, methods, and fields.

**Definition 22 (Identifiers).** We define a global set of identifiers \( I \), used for classes, methods, and fields.

Further, we fix an abstract set of method and field definitions.

**Definition 23 (Method and Field Definitions).** We define a global set of method and field definitions \( M \).

A class is defined as a partial mapping from identifiers to method and field definitions.

**Definition 24 (Class Definitions).** The collection of class definitions is the set of partial functions \( \Psi = I \rightarrow M \). Such a class definition \( \psi \in \Psi \) maps some identifiers to their definition. Unmapped identifiers are not defined in the class.

As an example, consider the following class definition. Only the explicitly mentioned identifiers are considered to be defined. We use capital letters to refer to method implementations, where different letters represent distinct implementations (as in Figure 3).

\[
\begin{cases}
  f \mapsto f(): \text{void} \{ A \}, \\
  g \mapsto g(): \text{bool} \{ B \}, \\
  i \mapsto i: \text{int}
\end{cases}
\]

A program is a set of classes, mapping identifiers to class definitions.

**Definition 25 (Programs).** We define the set of programs \( P \) as the set of class definitions \( \Psi \). A class modification includes adding, replacing, and removing methods and fields, or replacing the class completely. To ensure that composition of deltas produces a closed form, we distinguish between updating a class and replacing it. A class replacement completely replaces an existing class. A class update modifies the original class at the method/field level. Modifying a class that does not exist is treated as adding a new class.

As an example, consider the following program definition:

\[
\begin{cases}
  C \mapsto \begin{cases}
    f \mapsto f(): \text{void} \{ A \}, \\
    g \mapsto g(): \text{bool} \{ B \}, \\
    i \mapsto i: \text{int}
  \end{cases} \\
  D \mapsto \begin{cases}
    h \mapsto h(x: \text{int}): \text{int} \{ C \}, \\
    b \mapsto b: \text{bool}
  \end{cases}
\end{cases}
\]

5.2. OOP Software Deltas

OOP software deltas modify a program by adding, modifying, and removing classes. A class modification includes adding, replacing, and removing methods and fields, or replacing the class completely. To ensure that composition of deltas produces a closed form, we distinguish between updating a class and replacing it. A class replacement completely replaces an existing class. A class update modifies the original class at the method/field level. Modifying a class that does not exist is treated as adding a new class.

The definition of an OOP software delta captures this set of program modifications.
Definition 26 (OOP Software Deltas). The set of software deltas is defined as
\[ D = I \setminus \left( \{ r \} \times (I \to M) \cup \{ u \} \times (I \to M^-) \right). \]

Each delta \( d \in D \) is a partial function representing class modifications. \( r \) and \( u \) represent ‘replace’ and ‘update’, respectively. Mapping an identifier to \( - \) indicates removal from the product. \( \epsilon = \varnothing \) is the empty delta, modifying nothing.

The following example of an OOP software delta denotes that class \( C \) is updated by removing field/method \( f \) and adding method \( z \) and field \( i \) and that class \( D \) is removed.

\[
\begin{aligned}
C \mapsto u \\
\{ f \mapsto -, \\
z \mapsto z(): \text{void} \{ D \}, \\
i \mapsto i: \text{float}
\}
\end{aligned}
\]

\[
D \mapsto -
\]

In contrast to previous work (Schaefer et al., 2010), the removal of an element in this concrete deltoid does not require that the element is already present, nor does addition require its absence. This ensures that every derivation of deltas is well-defined.

Now we introduce some notation required in the next few definitions. The first notation is used to combine two partial functions into another partial function by some binary operation on their codomain.

Notation 5. We use the following notation to lift an operator \( \circ \) on two partial functions to the values in their codomain. For \( i \in I \):

\[
(a \circ b)(i) \overset{\text{def}}{=} a(i) \circ b(i).
\]

The following notation excludes method/field removals from a class update. It is needed when a class update is sequentially composed with a class replacement, since class replacements should not contain removals, and when a delta is applied to a product.

Notation 6. Given a class update \( f : I \to M^- \), we define \( f^* \) as \( f \), but without any method or field removals:

\[
f^*(i) \overset{\text{def}}{=} \begin{cases} 
\bot & \text{if } f(i) = - \\
\text{otherwise}
\end{cases}
\]

Now we define sequential composition of software deltas.

Definition 27 (Sequential Composition of OOP Software Deltas). The sequential composition of OOP software deltas \( \cdot : \mathcal{D} \times \mathcal{D} \to \mathcal{D} \) is defined as

\[
y \cdot x \overset{\text{def}}{=} y \uplus_C x,
\]
where the operator $\oplus_C$, working on the level of class modifications, with $e, f : \mathcal{I} \rightarrow \mathcal{M}^-$ and $g, h : \mathcal{I} \rightarrow \mathcal{M}$, is

\[
\begin{array}{c|ccc}
\oplus_C & \perp & u f & r h \\
\hline
\perp & \perp & u f & r h \\
\hline
u e & u e & r e^* & u (e \oplus_M f) & r (e \oplus_M h)^*
\end{array}
\]

and $\otimes_M$, working on the level of method and field definitions, with $m, n \in \mathcal{M}$, is

\[
\begin{array}{c|ccc}
\otimes_M & \perp & n \\
\hline
\perp & \perp & n \\
\hline
m & m & m
\end{array}
\]

The options for combining methods are limited here, but Section 5.4 will redefine $\otimes_M$ to allow method wrapping. The definition of OOP software delta composition gives concrete meaning to the notion of conflict. Two OOP software deltas are in conflict if they map the same identifier to two different definitions, that is, if they modify the same field or method of the same class in different ways.

OOP software deltas satisfy the following properties. The proofs of properties Lemmas 4 to 6 are subsumed by the proofs of properties of the AOP software deltas (Section 5.4), as they are a conservative extension of OOP software deltas. Proofs appear in the appendix.

**Lemma 4.** OOP software deltas are a deltoid.

**Lemma 5.** OOP software deltas exhibit consistent conflict resolution.

Finally, we define OOP software delta application to apply an OOP software delta to a program.

**Definition 28 (OOP Software Delta Application).** Given delta $y \in D$ and product $p \in P$, OOP software delta application is an operation $-(-) : D \times P \rightarrow P$ defined as follows:

\[
y(p) \overset{\text{def}}{=} y \otimes_C p,
\]

where the operators $\otimes_C$, with $f : \mathcal{I} \rightarrow \mathcal{M}^-$ and $g, h : \mathcal{I} \rightarrow \mathcal{M}$, and $\otimes_M$, with $m, n \in \mathcal{M}$, are defined as

\[
\begin{array}{c|ccc}
\otimes_C & \perp & h \\
\hline
\perp & \perp & h \\
\hline
u f & f^* & f \otimes_M h \\
\hline
r g & g
\end{array}
\]

\[
\begin{array}{c|ccc}
\otimes_M & \perp & n \\
\hline
\perp & \perp & n \\
\hline
m & m & m
\end{array}
\]
Abstract Delta Modeling

Lemma 6. OOP software delta application is a delta action.

OOP software deltas can be applied to construct any product from any other product.

Lemma 7. OOP software deltas are maximally expressive.

Proof. Because any method and class can be removed, all the elements not required can be removed using deltas. The new elements can be added using additional deltas. The composition of these deltas is the delta required to complete the proof.

Lemma 8. The empty program \( \emptyset \in \mathcal{P} \) is an empty product.

Proof. From Lemma 7, every product is a candidate for the empty product.

5.3. Example: Editor Product Line

We now describe our Editor product line presented in Section 2 using OOP software deltas. The Editor product line is described by the feature model depicted in Figure 2. The set of valid feature configurations is the following:

\[
\Phi = \left\{ \{Ed, Pr\}, \{Ed, Pr, SH\}, \{Ed, Pr, SH, EC\}, \{Ed, Pr, SH, EC, SA\}, \\
\{Ed, Pr, SH, TI\}, \{Ed, Pr, EC\}, \{Ed, Pr, EC, SA\}, \{Ed, Pr, TI\}, \\
\{Ed, SH\}, \{Ed, SH, EC\}, \{Ed, SH, EC, SA\}, \{Ed, SH, TI\}, \\
\{Ed, EC\}, \{Ed, EC, SA\}, \{Ed, TI\} \right\}
\]

The Editor product line uses the empty program as the core product: \( c = \emptyset \in \mathcal{P} \). The delta model of the product line is shown in Figure 3 using a graphical representation of the OOP software deltas. Each feature has its own delta \( d_i \in D \), implementing that feature without considering potential conflicts. Because some feature configurations include interacting features, conflict-resolving deltas \( (d_i, d_j \in D) \) have to be added. Delta \( d_8 \) deals with the interaction between \( SH \) and \( EC \), combining the coloring of \( SH \) with the underlining of \( EC \). Similarly, delta \( d_9 \) handles the interaction between \( SH \) and \( SA \). The desired feature interaction between \( Pr \) and \( SH \) is implemented by delta \( d_7 \). Figure 3 also shows the application conditions \( \gamma(d_i) \) for each delta \( d_i \in D \) in the form of propositional logic formulae, where the propositions are features.

The Editor product line is globally unambiguous. As the underlying deltoid exhibits consistent conflict resolution (cf. Lemma 5), this can easily be verified. There are three pairs of deltas in conflict: \( d_3 \prec d_4 \), \( d_3 \prec d_6 \) and \( d_4 \prec d_6 \). For the first, the conflict is resolved by \( (d_3, d_4) \prec d_8 \). The second and third conflict have the same resolver: \( (d_3, d_6) \prec d_9 \) and \( (d_8, d_9) \prec d_9 \). By the choice of \( \gamma \), the conflict-resolving delta is present in each feature configuration in which conflicting deltas appear.

To illustrate product generation (cf. Definition 19), we now derive the product for feature configuration \( F = \{Ed, SH, EC, SA\} \in \Phi \). We create the selected delta model \( PL \mid F = (D', \prec') \), with \( D' = \{d_1, d_3, d_4, d_6, d_8, d_9\} \) and

\[
\prec' = \left\{ (d_1, d_3), (d_1, d_4), (d_1, d_6), (d_1, d_8), (d_1, d_9), (d_3, d_8), \\
(d_3, d_9), (d_4, d_6), (d_4, d_8), (d_4, d_9), (d_6, d_9), (d_8, d_9) \right\}
\]
Since the Editor product line is globally unambiguous, it is sufficient to select one derivation of the delta model, such as \( x = d_9 \cdot d_8 \cdot d_6 \cdot d_4 \cdot d_3 \cdot d_1 \). Applying Definition 27, \( x \) is defined as follows:

\[
\begin{align*}
\text{Editor} & \mapsto \rightarrow \text{model} : \text{Model}, \\
\text{semAnalyzer} & \mapsto \rightarrow \text{semAnalyzer} : \text{SemanticAnalyzer}, \\
r & \begin{cases}
\text{getModel} & \rightarrow \rightarrow \text{getModel}() : \text{Model} \{ A \}, \\
\text{font} & \rightarrow \rightarrow \text{font}(c: \text{int}) : \text{Font} \{ O \}, \\
\text{onMouseOver} & \rightarrow \rightarrow \text{onMouseOver}(c: \text{int}) : \text{void} \{ J \}
\end{cases}, \\
\text{SemanticAnalyzer} & \mapsto \rightarrow \{ \\
\text{analyze} & \rightarrow \rightarrow \text{analyze}(m: \text{Model}) : \text{void} \{ K \}, \\
\text{getErrors} & \rightarrow \rightarrow \text{getErrors}() : \text{Errors} \{ L \}
\end{cases}
\end{align*}
\]

Applying \( x \) to \( c \) (Definition 28) results in a product that has the same form (only without the annotation \( r \)), since the core product is the empty program \( c = \emptyset \).

5.4. A Deltoid for Aspect-Oriented Programming

In AOP (Kiczales et al., 1997) and in languages based on feature-oriented programming (Prehofer, 1997; Apel et al., 2009b), delta-oriented programming (Schaefer et al., 2010), context-oriented programming (Costanza & Hirschfeld, 2005) and step-wise refinement (Batory et al., 2004), it is possible to refine a method implementation in such a way that it uses the previous method implementation. This can be thought of as method wrapping, and is realised, for example, by the original keyword in delta-oriented programming. This is a simple delta-oriented programming example illustrating the idea.

```java
class A {
    int m() { B }
}
delta D {
    modifies class A {
        modifies int m() { C; int x = original(); D }
    }
}
```

Applying delta \( D \) to class \( A \) results in a new implementation of \( m \), which effectively corresponds to the old implementation placed where the call to \( \text{original()} \) is made:

```java
class A {
    int original_m() { B }
    int m() { C; int x = original_m(); D }
}
```

To model this approach, we adapt the OOP software deltas (Sections 5.1 and 5.2) to include method wrapping by modifying method bodies \( \mathcal{M} \) in classes and deltas to have
the following (abstract) grammar:

\[
M \ni wb ::= b | w[m] | w[ ] \quad b \text{ is a normal method body}
\]
\[
B \ni m ::= b | w[m]
\]
\[
W \ni w[ ] ::= e[ ] | w[w[ ]]
\]

\[ w[ ] \] denotes a wrapping method with a hole in it, where the hole corresponds to the place where the call to the original method is made, and \( w[m] \) denotes that body \( m \) is wrapped by \( w \). Methods with a hole do not appear in products.

Given these ingredients, only the definitions of \( \oplus_M \) and \( \odot_M \) from Definitions 27 and 28 need to change. In the following, \( m, n \) denote methods without hole.

\[
\begin{array}{c|cccc}
\oplus_M & \bot & - & n & w[ ] \\
\hline
\bot & \bot & - & n & w[ ] \\
- & - & - & - & - \\
\hline
m & m & m & m & m \\
w[ ] & w[ ] & - & w[n] & w[v[ ]] \\
\end{array}
\quad
\begin{array}{c|cccc}
\odot_M & \bot & n \\
\hline
\bot & \bot & n \\
- & - & - & - \\
\hline
m & m & m \\
w[ ] & w[ ] & \bot & w[n] .
\end{array}
\]

The example above has the following correspondence with our setting:

\[ b = \text{int } m() \{ B \}, \]
\[ w[ ] = \text{int } m() \{ \text{C; int } x = [ ]; D \}, \text{ and} \]
\[ w[b] = \text{int } m() \{ \text{C; int } x = B; \text{ D} \}. \]

The AOP software deltas enjoy the same properties as the OO software deltas. Proofs of Lemmas 9 to 11 appear in the appendix.

**Lemma 9.** AOP software deltas are a deltoid.

**Lemma 10.** AOP software deltas exhibit consistent conflict resolution.

**Lemma 11.** AOP software delta application is a delta action.

**Lemma 12.** AOP software deltas are maximally expressive.

*Proof.* Follows from the fact that AOP software deltas are a conservative extension of OOP software deltas.

\[ \square \]

6. Nested Delta Models

In this section, we extend the notions of abstract delta modeling to incorporate nested delta models. Nested delta models can express the isolated, atomic application of a collection of deltas within a model by making it possible for a delta model to be a delta itself in another model. To this end, we need to redefine some base definitions, like that of a deltoid and its composition operation. Definitions 29 to 31 mutually recursively define nested deltoids, nested delta models and the notion of derivation.

A nested delta model, used in the following definition, is similar to a delta model as introduced before, except that a delta can be a delta model itself.
Definition 29 (Nested Deltoid). Given a deltoid \((D, \cdot, \epsilon)\), its corresponding nested deltoid is the monoid \((D_N, \cdot, \epsilon)\), where \(D_N\) contains exactly all elements of \(D\) and all nested delta models based on this nested deltoid (Definition 30). \(N_1 \cdot N_2\) is defined as \(\{ n_1 \cdot n_2 | n_1 \in \text{deriv}(N_1), n_2 \in \text{deriv}(N_2) \}\) for \(N_1, N_2 \in D_N\) (for deriv, see Definition 31).

Definition 30 (Nested Delta Model). Given a nested deltoid \((D_N, \cdot, \epsilon)\), a nested delta model is a tuple \((D, \prec)\), where \(D \subseteq D_N\) is a set of (nested) deltas and \(\prec\) is a strict partial order on \(D\).

The following refined notion of derivation relies on an overloaded definition of deriv.

Definition 31 (Derivations). The derivations of a delta (and hence a nested delta model) are defined as:

\[
\text{deriv}((D, \prec)) = \bigcup_{x_1, \ldots, x_n \text{ is a linear extension of } \prec \text{ where } \{x_1, \ldots, x_n\} = D} x_n \cdot \ldots \cdot x_1
\]

\[
\text{deriv}(m) = \{m\}, \quad \text{where } m \in D.
\]

The nested versions of the definitions of unique derivation (Definition 9), conflict (Definition 10), conflict-resolving delta (Definition 11) and unambiguous delta model (Definition 12) are as before, except that they use \(\cdot\) where the old definitions use \(\ast\). This means that nested delta models act like ordinary deltas. To get this result, however, we require that all delta models that are used as deltas are unambiguous, so that there is no delta containing an unresolved conflict.

For product lines containing nested deltas, we want to consider application conditions on all levels of nesting. Thus, we add application functions to nested delta models.

Definition 32 (Nested Delta Model with Application Conditions). A nested delta model with application conditions is triple \((D, \prec, \gamma)\) where the elements of \(D\) are either modifications \(m \in D\) or nested delta models with application conditions, \(\prec\) is as before, and \(\gamma : D \to \mathcal{P}(\mathcal{P}(F))\), such that if \(N = (D', \prec', \gamma') \in D\), then \(\gamma(N) = \bigcup_{d \in D'} \gamma'(d)\).

For reasons of simplicity, the last condition automatically derives the application condition of a nested delta model from that of its deltas. If at least one inner delta is applicable, then the nested delta model is applicable as well. Based on Definition 32, we can now define nested product lines:

Definition 33 (Nested Product Line). A nested product line is a tuple \(PL = (\Phi, c, D, \prec, \gamma)\), where \((D, \prec, \gamma)\) is a nested delta model with application conditions. \(\Phi\) and \(c\) have the same meaning as in Definition 17.

A selected nested delta model recursively removes from a nested delta model with application conditions those deltas that are not applicable for a given feature configuration. The definition of generated products (Definition 19) from a selected delta model is as before.
Definition 34 (Selected Nested Delta Model). Given a nested product line $PL = (\Phi, c, D, \prec, \gamma)$, a selected nested delta model for a feature configuration $F \in \Phi$, denoted as $PL \mid F$, is the nested delta model $(D, \prec, \gamma) \mid F$, where $\mid F$ is defined on nested delta models as

$$(D, \prec, \gamma) \mid F = (D', \prec')$$

where $D' = \{ d \mid F \mid d \in D, F \in \gamma(d) \}$ and

$\prec'$ is $\prec$ restricted to $D'$

$m \mid F = m.$

As an example for a nested delta model, assume that delta $d_4$ from the Editor product line presented in Section 2 should be refactored into two deltas $d_1^4$ and $d_2^4$, the first handling the `font` method and the second handling the `onMouseOver` method. In order not to introduce any extra orderings to the delta model, we decide to nest the two deltas obtained from the refactoring into a new delta model (see Figure 4).

From this example, we can observe that nesting allows greater modularity and structure in delta models. Furthermore, nested delta models are more expressive than previously introduced 'flat' delta models, since certain sets of derivations can be expressed by nested delta models, that cannot be expressed with flat delta models.

Theorem 4. Nested delta models are able to express sets of derivations that flat delta models cannot. That is, there exist nested delta models $(D, \prec)$, for which there exists no flat delta model $(D', \prec')$ such that $\text{deriv}(D, \prec) = \text{deriv}(D', \prec')$.

Proof. Consider the following nested delta model $N$. We prove that there exists no flat delta model which generates the same derivations.
We now try to find a flat delta model $N' = (\{x, y, z\}, \prec')$ such that $\text{derv}(N') = \text{derv}(N)$. We consider all possible partial orderings $\prec'$:

- $\prec' = \emptyset \implies |\text{derv}(N')| = 6$
- $\prec' = \{(A, B)\}$ s.t. $A, B \subseteq \{x, y, z\} \implies |\text{derv}(N')| = 3$
- $\prec' = \{(A, B), (B, C)\}$ s.t. $A, B, C = \{x, y, z\} \implies |\text{derv}(N')| = 1$
- $\prec' = \{(A, C), (B, C)\}$ s.t. $A, B, C = \{x, y, z\} \implies |\text{derv}(N')| = 1$

Only the last two cases are relevant. We need to find a bijection between $\{A, B, C\}$ and $\{x, y, z\}$ such that either $C \cdot B \cdot A, B \cdot C \cdot A$ or $C \cdot B \cdot A, C \cdot A \cdot B$ has a solution. But no such bijection exists, hence, there exists no flat delta model $N'$ such that $\text{derv}(N') = \text{derv}(N)$. Since, any flat delta model, trivially, is a nested delta model, this shows that nested delta models are strictly more expressive than flat delta models.

7. Methodology

This section suggests a methodology for developing software product lines by abstract delta modeling. The key goal is to structure product lines using delta modeling such that code duplication and unnecessary dependencies between deltas are minimized and a maximum opportunity for concurrent development of deltas is provided. In the following, we assume a product set $\mathcal{P}$ and deltoid $(\mathcal{D}, \cdot, \epsilon)$ and propose guidelines on how to design a well-structured product line $\mathcal{PL} = (\Phi, c, D, \prec, \gamma)$.

Product line development starts with the feature model $\Phi$ describing the set of possible products by the set of valid feature configurations. A feature model, which is a feature tree with cross-tree constraints such as ‘requires’ and ‘excludes’ relationships between features (Kang et al., 1990; van Deursen & Klint, 2002), can be easily mapped to a delta model. Since delta modeling is very flexible, it is not necessary to change the feature model to alleviate implementation conflicts. Instead, these can be resolved by conflict-resolving deltas.

The first step in product line development is to decide on a core product. If there is an existing legacy application, this can be chosen as the core product in order to save development effort, following the extractive product line development principles suggested by Krueger (2002). Having a solid core architecture, which is reusable among all products, is a favorable characteristic of a software product line (Pohl et al., 2005).
If there is no legacy code and a sensible empty product exists (Definition 5), this can be chosen the core product $c$ such that all functionality is introduced by deltas. This has advantages for the evolution of the product line, as pointed out by Schaefer & Damiani (2010), for example, when mandatory features become optional.

For creating the delta model, we begin by specifying exactly one delta $d_f \in D$ for each feature $f$, such that $d_{f_1} \prec d_{f_2}$ iff $f_2$ is a subfeature of $f_1$ (or if the feature $f_2$ "requires" feature $f_1$ by the feature model constraints, i.e., $\{ F \in \Phi \mid f_1 \in F \} \supseteq \{ F \in \Phi \mid f_2 \in F \}$).

In this way, the structure of the delta model mimics the structure of the feature model. For instance, in Figures 2 and 3, the application condition of each delta $d_f$, corresponds directly to the realized feature, $\gamma(d_f) = \{ F \in \Phi \mid f \in F \}$. Each of these deltas only implements its corresponding feature and can be developed without taking possible conflicts into account since these will be resolved later. This facilitates the development of features without requires and subfeature relationships concurrently in their corresponding deltas, for example, by different developer teams. For instance, in Figure 3, delta $d_1$ has to be developed first, but then $d_2, \ldots, d_5$ can be developed independently.

Between deltas implementation conflicts may occur, for instance, when two deltas manipulate the same program entity in different ways. An example is the $d_3 \& d_4$ conflict, in which both $d_3$ and $d_4$ redefine the font method in Figure 3. For each implementation conflict, a conflict-resolving delta has to be provided. Its application condition should be the intersection of the application conditions of the conflicting deltas. Furthermore, it has to be greater in the application ordering $\prec$ than both conflicting deltas. If two deltas with mutually exclusive application conditions contain an implementation conflict, such as deltas $d_4$ and $d_5$ in Figure 3, no conflict needs to be resolved since the two deltas will never be applied together.

Furthermore, every desired feature interaction requires a delta introducing the glue code to implement it. Structurally, a delta for a desired feature interaction is the same as a conflict-resolving delta. For example, delta $d_7$ implements the desired interaction between the Pr and SH features in Figure 3. After developing conflict-resolving and feature interaction deltas for pairs of features and deltas, we proceed iteratively by providing deltas for higher-order combinations of features, as suggested by Prehofer (1997) and Liu et al. (2006) using higher-order lifters and derivates, until all feature interactions and implementation conflicts are handled. If a delta has to be split into multiple deltas, for example, when a delta is refactored to improve code reuse or modularity, the resulting deltas should be combined into a nested delta to avoid introducing new conflicts to the original delta model.

8. Related Algebraic Approaches

Other algebraic approaches describing the underlying structure of software product lines exist (Apel et al., 2010; Batory & Smith, 2007). These formalise the mechanisms underlying AHEAD (Batory et al., 2004), GenVoca (Batory & O’Malley, 1992), and Feature-House (Apel et al., 2009b). The first difference with our approach is that we consider the collection of modifications for an entire product line, rather than a single product at a time, and thus are able to talk about conflicts and conflict resolution at the level of the
product line. The second difference is that those approaches generally arrange ‘deltas’ into introductions and modifications—introductions correspond to the core ingredients of product, whereas modifications modify existing ingredients. In contrast, we assume a single, unified collection of deltas. Here we compare our approach with two recent proposals, namely, the Quark model (Apel et al., 2010) and Finite Map Spaces (Batory & Smith, 2007). From an algebraic perspective, these two proposals are quite similar, so we consider them together. By encoding these frameworks, we demonstrate that our formalism is sufficient to express these using simpler notions, as well as providing an alternative foundation for tools based on these formalisms.

8.1. Quarks and Finite Map Spaces

Both Apel et al. (2010) and Batory & Smith (2007) base the description of a product line on the following ingredients (our notation):

- **introductions**: a commutative idempotent monoid \((I, +, 0)\), where \(+ : I \times I \rightarrow I\), of which some are ‘atomic’ and form a basis \(\mathcal{I} \subseteq I\) (in the sense of vector spaces/modules).

- **modifications**: a monoid \((M, \bullet, 1)\), where \(\bullet : M \times M \rightarrow M\).

- an operation \(\odot : M \times I \rightarrow I\) applying modifications to introductions, satisfying
  - \(M\) is a monoid action over \(I\): \(1 \odot i = i\) and \((m \bullet n) \odot i = m \odot (n \odot i)\),
  - Distributivity: \(m \odot (i + j) = m \odot i + m \odot j\), and
  - \(m \odot 0 = 0\).

- **products**: elements of \(I\), which are of the form \(\sum_{j=1}^{n} (m_j \odot i_j)\), where each \(m_j \in M\) and \(i_j \in \mathcal{I}\).

Introductions and modifications are combined to form quarks \(Q\), which correspond to our deltas. Different notions of quark and quark composition (\(\circ : Q \times Q \rightarrow Q\)) have been defined—these correspond approximately to our notion of deltoid—to capture combinations of the following operations:

- **local composition**: these apply modifications to elements already in the product;

- **global composition**: these apply modifications to all elements of the final product, and thus their application is delayed until after all introductions have been made; and

- **modifiers of modifiers**: these modify modifications rather than elements of the product.

In addition to the quark and quark composition, the unit of quark composition and an operation \(\text{image} : Q \rightarrow I\) used when extracting the final product from a quark need to specified. The image operation also applies globally applicable operations at the last minute, in some cases.

**local quark composition (Apel et al., 2010)**

- \(Q = I \times M\) — an introduction and a local modification
- \(\langle i_2, l_2 \rangle \circ \langle i_1, l_1 \rangle = \langle i_2 + (l_2 \odot i_1), l_2 \bullet l_1 \rangle\)
- unit is \(\langle 0, 1 \rangle\)
- \(\text{image}(i, l) = i\)
global quark composition (Apel et al., 2010)
— \( Q = I \times M \) — an introduction and a global modification
— \( \langle i_2, g_2 \rangle \bullet (i_1, g_1) = (g_2 \bullet g_1) \odot (i_2 + i_1), g_2 \bullet g_1) \)
— unit is \( (0, 1) \)
— image \((i, g) = i \)

full quark composition (Apel et al., 2010)
— \( Q = M \times I \times M \) — a global modification, an introduction, and a local modification
— \( \langle g_2, i_2, l_2 \rangle \bullet \langle g_1, i_1, l_1 \rangle = \langle g_2 \bullet g_1, (g_2 \bullet g_1) \odot (i_2 + (l_2 \odot i_1)), l_2 \bullet l_1 \rangle \)
— unit is \( (1, 0, 1) \)
— image \((g, i, l) = g \odot i \)

full quark composition (Batory & Smith, 2007)
— \( Q = M \times I \times M \) — a global modification, an introduction, and a local modification
— \( \langle h_2, g_2, i_2, l_2 \rangle \bullet \langle h_1, g_1, i_1, l_1 \rangle = \langle h_2 \odot h_1, g_2 \bullet g_1, i_2 + (l_2 \odot i_1), l_2 \bullet l_1 \rangle \)
— unit is \( (id, 1, 0, 1) \)
— image \((h, g, i, l) = h^b (g \odot i) \).

The function \( h^b \) used in the definition of modifiers of modifiers applies the modifiers. Given a modifier of modifiers \( h : M \rightarrow M \), Batory & Smith (2007) introduce the set of rewriting rules defining a function \( h^b \) to recursively apply all higher-order modifications. We rewrite \( h^b \) to be defined by the following set of equations, where \( m, m' \in M \), \( i, i' \in I \), and \( i \in \mathbb{I} \) is a basis element:

\[
\begin{align*}
R^b(m) &= h(m) \\
R^b(0) &= 0 \\
R^b(i) &= i \\
R^b(i + i') &= R^b(i) + R^b(i') \\
R^b(m \bullet m') &= R^b(m) \bullet R^b(m') \\
R^b(m \odot i) &= R^b(m) \odot R^b(i).
\end{align*}
\]

Note that \( h^b \) is overloaded to apply to both elements of \( M \) and \( I \).

Our observation is that this amounts to saying that \( h : M \rightarrow M \) acts like monoid homomorphism on the modifications lifted to introductions

\[
R^b \left( \sum_{j=1}^{n} m_j \odot i_j \right) = \sum_{j=1}^{n} (h(m_j) \odot i_j).
\]

For local quark composition, Batory & Smith (2007)'s full quark composition, and modifiers of modifiers, quark composition \( \bullet \) forms a monoid, with the appropriate tuple
of units as the unit for ♦. Delta application $-(-): Q \times I \to I$ (Definition 3) can be defined, for example, as $q(p) = \text{image}(q \star \langle p, 1 \rangle)$, where $q \in Q$ is a quark and $p \in I$ is the core product. Note that the term $\langle p, 1 \rangle$ needs to be adapted depending on the notion of quark being used.

In the absence of other axioms, the other quarks above do not form a deltoid, Global quark composition and full quark composition (Apel et al., 2010) are not even associative. In addition, Apel et al. (2010)’s global quark composition and full quark composition produce results such as the following (for global quark composition):

$((i_3 \cdot g_3) \star \langle i_2, g_2 \rangle) \star \langle i_1, g_1 \rangle = (((g_3 \cdot g_2) \star g_1) \circ ((g_3 \cdot g_2) \circ (i_3 + i_2)) + i_1), (q_3 \cdot g_2) \cdot g_1$.

which applies modifications $g_3$ and $g_2$ multiple times. This makes no sense if wrapping is one of the kinds of possible modifications. To get this composition to behave, strong idempotence criteria are proposed (Apel et al., 2010), but these exclude modifications such as method wrapping. Delta application is an action only for local quark composition.

We now describe how to encode local quark composition, full quark composition (Batory & Smith, 2007), and modifiers of modifiers (Batory & Smith, 2007) more directly in our setting. From our perspective, introductions play a dual role. They correspond to (elements of) products, as well as represent one kind of delta; modifications are the other kind. That is, an introduction $i \in I$ in a delta corresponds to introducing a new element into a product and a modification $m \in M$ corresponds to an operation modifying an existing element. In our encoding, we make introductions a kind of modification and eliminate quarks from the local composition variant. By ignoring the distinction between modifications and introductions, we can focus on deltas alone, and work in a simpler algebraic setting. For full quark composition and modifiers of modifiers, the notion of quark needs to be reintroduced.

### 8.1.1. Encoding Local Quark Composition

Before proceeding, we recall that $\langle 0, 1 \rangle$ is the unit of ♦ for local quark composition, and that apart from the monoid laws for ♦, we have also $\langle i_1, m_1 \rangle = \langle i_2, m_2 \rangle$ if and only if $i_1 = i_2$ and $m_1 = m_2$.

The following definition introduces deltoid $M_I$ consisting of deltas that are sequences of modifications $m \in M$ and introductions $i \in I$. We show that this is equivalent to $Q = I \times M$ with ♦ corresponding to local quark composition.

**Definition 35 (Deltoid $M_I$).** Given a monoid $(M, \cdot, 1)$, a commutative monoid $(I, +, 0)$, and an operation $\circ : M \times I \to I$ satisfying the conditions:

1. $1 \circ i = i$
2. $(m \cdot n) \circ i = m \circ (n \circ i)$
3. $m \circ (i + j) = (m \circ i) + (m \circ j)$
4. $m \circ 0 = 0$

Define deltoid $M_I = ((M \cup I)^*, \cdot, \epsilon)$, where $\cdot$ is concatenation with unit the empty sequence $\epsilon$, subject to the following equations $(m, n \in M, i, j \in I, \text{ and } \mu, \nu, \eta \in M_I)$:

1. $\epsilon \cdot \mu = \mu = \mu \cdot \epsilon$
2. $\mu \cdot (\nu \cdot \eta) = (\mu \cdot \nu) \cdot \eta$
3. $m \cdot i = (m \circ i) \cdot m$
4. $i \cdot j = i + j = j + i = j \cdot i$
5. $i \cdot i = i + i = i$
6. $m \circ n = m \cdot n$
7. $\epsilon = 0 = 1$. 


Definition 35 forms a deltoid by taking sequences of modifications and introductions, modulo the given equations. The equations interpret various combinations of elements of $M_I$ in terms of the original collection of operations. The most interesting is 3, which applies a modification $m$ to an introduction $i$, via $m \odot i$, and shuffles $m$ later in the sequence to apply to subsequent introductions. Note that equations 1 and 2 are redundant and follow from the fact that $\cdot$ is concatenation and $\epsilon$ its unit, but we include them for completeness.

Delta action is defined inductively over the elements of $M_I$, applying each element of $M_I$ to $I$ via the appropriate function from the original monoids.

**Definition 36.** The delta action $\langle \cdot \rangle : M_I \times I \to I$ for $M_I$ is defined inductively as follows:

$$
\begin{align*}
\epsilon(p) &= p \\
m(p) &= m \odot p \\
i(p) &= i + p \\
(\mu \cdot \nu)(p) &= \mu(\nu(p)).
\end{align*}
$$

where $m \in M$, $i \in I$, $\mu, \nu \in M_I$ and $p \in I$.

The following lemma captures that our notion of delta action is sensible, in that it preserves the equations in Definition 35. More precisely, if we have elements $\mu, \nu \in M_I$ that are perhaps syntactically distinct but equal by the equations, then they produce equal results when applied to an product $p \in I$—recall that we consider elements of $I$ as both introductions and ultimately as products.

**Lemma 13.** For all $\mu, \nu \in M_I$ and all $p \in I$, if $\mu = \nu$ by equation 1–7 of Definition 35, then $\mu(p) = \nu(p)$.

**Proof.** Proof appears in the appendix. \qed

We now show the equivalence of quarks and $M_I$ by producing homomorphisms in each direction and proving some theorems about them. The following is a monoid homomorphism from quarks to $M_I$.

**Definition 37.** Define $\llbracket \cdot \rrbracket : Q \to M_I$ as

$$
\llbracket \langle i, m \rangle \rrbracket = i \cdot m.
$$

**Lemma 14.** The function $\llbracket \cdot \rrbracket : Q \to M_I$ is a homomorphism. That is, $\llbracket \langle 0, 1 \rangle \rrbracket = \epsilon$ and for all $q, q' \in Q$, $\llbracket q \hat{\cdot} q' \rrbracket = \llbracket q \rrbracket \cdot \llbracket q' \rrbracket$.

**Proof.** Proof appears in the appendix. \qed

The mapping from $M_I$ to quarks defined in the following is also a monoid homomorphism, by definition.

**Definition 38.** Define $\langle \langle \cdot \rangle \rangle : M_I \to Q$ as

$$
\langle \langle \epsilon \rangle \rangle = (0, 1)
$$
Quarks with local quark composition are isomorphic to $M_I$, which is stated in Theorem 5, supporting our claim that making the distinction between introductions and modifications is unnecessary.

**Theorem 5.** For all $q, q' \in Q$ and $\mu, \nu \in M_I$, we have
1. $\langle \llbracket q \rrbracket \rangle = q$,
2. $\langle \llbracket \mu \rrbracket \rangle = \mu$,
3. if $q = q'$, then $\llbracket q \rrbracket = [q']$, and
4. if $\mu = \nu$, then $\langle \llbracket \mu \rrbracket \rangle = \langle \llbracket \nu \rrbracket \rangle$.

**Proof.** Proof appears in the appendix.

Finally, Theorem 6 shows that not only are quarks and $M_I$ isomorphic, but their notions of delta action correspond, so they will generate the same products.

**Theorem 6.** For all $q \in Q$ and all $p \in I$,

$$\text{image}(q \check{\circ} \langle p, 1 \rangle) = [q](p)$$

and for all $\mu \in M_I$ and all $i \in I$,

$$\text{image}(\langle \mu \rangle \check{\circ} \langle p, 1 \rangle) = \mu(p).$$

**Proof.** Proof appears in the appendix.

8.1.2. **Encoding Batory and Smith’s Full Quark Composition.** Encoding full quark composition is straightforward. To do so, we adapt the encoding above to use quarks $Q = M \times M_I$, where quark composition is $\langle m, \mu \rangle \check{\circ} \langle n, \nu \rangle = \langle m \cdot n, \mu \cdot \nu \rangle$ and define delta application $-(-) : Q \times I \rightarrow I$ to be $(m, \mu)(p) = m \circ (\mu(p))$, relying on delta application for $M_I$.

The full quark $\langle g, i, l \rangle$ is encoded as $\langle g, \llbracket [i, l] \rrbracket \rangle = \langle g, i \cdot l \rangle$, using the definition from the previous section. The results from the previous section can be extended to establish an isomorphism between the two forms of quark, in the obvious manner.

It is easy to show that the notions of delta application for full quark composition and this encoding coincide. On one hand, for full quark composition

$$\text{image}(\langle g, i, l \rangle \check{\circ} \langle 1, p, 1 \rangle) = \text{image}(\langle g, i + (l \circ p), l \rangle) = g \circ (i + (l \circ p)).$$

On the other hand, for our encoding:

$$\llbracket [i, l] \rrbracket (p) = g \circ ([i, l](p)) = g \circ (i \cdot l)(p) = g \circ (i(l(p))) = g \circ (i + (l \circ p)).$$
However, in the absence of other assumptions, this notion of delta application is not an action—that is \((q \bullet q')(p) = q(q'(p))\) does not hold in general—as for example:

\[
\langle (m, \mu), (n, \nu) \rangle(p) = \langle m \bullet n, \mu \cdot \nu \rangle(p) = (m \bullet n) \odot ((\mu \cdot \nu)(p)) = (m \bullet n) \odot (\mu(\nu(p)))
\]

whereas

\[
\langle m, \mu \rangle \langle (n, \mu)(p) \rangle = m \odot (\mu(n \odot (\nu(p)))).
\]

If we instantiate \(\mu\) and \(\nu\) with \(m'\) and \(n'\), such that \(m \neq m'\) and \(n \neq n'\), we have in the first case:

\[
(m \bullet n) \odot m'(n'(p)) = (m \bullet n) \odot (m' \odot (n \odot p)) = (m \bullet n) \odot m' \odot n' \odot p
\]

and in the second case

\[
m \odot (m'(n \odot n'(p))) = m \odot (m' \odot (n \odot (n' \odot p))) = (m \bullet m' \bullet n \bullet n') \odot p.
\]

However, \((m \bullet n \bullet m' \bullet n') \odot p\) equals \((m \bullet n' \bullet m \bullet n) \odot p\) are equal in general only if \(\bullet\) is commutative.

8.1.3. Encoding Batory and Smith’s Modifiers of Modifiers. Encoding modifiers of modifiers is also relatively straightforward. We assume that such modifiers, \(h : M \rightarrow M\), are endomorphisms on the monoid of modifications (this is already implicit in (Batory & Smith, 2007)’s \(R^h\) function): that is, \(h(1) = 1\) and \(h(m_2 \bullet m_1) = h(m_2) \bullet h(m_1)\), for all \(m_1, m_2 \in M\).

We can extend the previous encoding to apply higher-order modifiers to global modifications as follows:

- quarks: \(Q = (M \rightarrow M) \times M \times M_I\) consist of a modifier of modifiers, a global modification, and a delta
- composition: \((h_2, g_2, \mu_2) \bullet (h_1, g_1, \mu_1) = (h_2 \circ h_1, g_2 \bullet g_1, \mu_2 \cdot \mu_1)\), and
- delta application is \(\langle h, g, \mu \rangle(p) = h_I(g) \odot h_I(\mu)(h_I(p))\), where \(h_I : M_I \rightarrow M_I\) is \(h : M \rightarrow M\) lifted to \(M_I\), defined by the following:
  \[
  \begin{align*}
  h_I(\epsilon) &= \epsilon \\
  h_I(\mu \cdot \nu) &= h_I(\mu) \cdot h_I(\nu) \\
  h_I(\text{id}) &= \text{id} \\
  h_I(m \odot i) &= h(m) \odot h_I(i).
  \end{align*}
  \]

Note that \(h_I\) is essentially \(R^h : I \rightarrow I\) lifted from an overloaded function on \(M\) and \(I\) to a function on \(M_I\).

Delta application for Batory & Smith (2007) of quark \(\langle h, g, i, l \rangle\) to product \(p\) is:

\[
\text{image}(\langle h, g, i, l \rangle \bullet \langle \text{id}, 1, p, 1 \rangle) = \text{image}(\langle h, g, i + (l \odot p), l \rangle) = R^h(g \odot (i + (l \odot p)))
\]
\[ = R^h(g) \odot (R^h(i) + (R^h(l) \odot R^h(p))). \]

Our delta application produces the same result:

\[
((h, g, [i, l]])(p) = (h, g, i \cdot l)(h_1(p)) = h_1(g) \odot (h_1(i \cdot l))(h_1(p)) = h_1(g) \odot (h_1(h_1(l) \cdot h_1(i))(h_1(p))) = h_1(g) \odot (h_1(i) + (h_1(l) \odot h_1(i))).
\]

Again delta application is not an action, for the same reason as for full quark composition.

8.2. Discussion.

From the various examples given in the previous two sections, it is clear that not all candidate deltoids satisfy all the properties that our OOP and AOP software deltoids enjoy. Our discussion will focus on two such properties.

Firstly, composition of OOP and AOP software deltas results in a closed representation. That is, OOP software deltas are elements of \[ I \rightarrow (\{r\} \times (I \rightarrow M) \cup \{u\} \times (I \rightarrow M^-))^- \], the carrier of the deltoid, and composing two such such deltas results again in an element of this set, rather than a sequence of modifications. The key advantage of having a closed representation of deltas is that it is easier to compute when conflicts occur, simply by comparing \( f \cdot g \) and \( g \cdot f \). The closed representation induces non-trivial equations between deltas, which ideally should be axiomatised. If deltas are the actual functions performing the modifications, rather than a representation of them as is the case with our OOP and AOP software deltas, the composition of two deltas \( f \) and \( g \) is merely their composition \( g \circ f \), which is not a closed form. We may not easily be able to say anything about such a form, such as whether \( g \circ f = f \circ g \). So reasoning about conflicts becomes difficult.

Another case where nothing can be said is when the free deltoid of a set of primitive modifications is considered. In this situation, the elements of the deltoid consist of sequences of primitive modifications. The equation \( g f = f g \) holds only when \( g f \) and \( f g \) are the same sequence of modifications—this is what it means to be free. With free deltoids, there are no equations, so no reasoning possibilities regarding conflicts exist, since all swaps constitute conflicts. In addition, no conflict resolution is possible, because it will never be the case that \( z \cdot x \cdot y = z \cdot y \cdot x \) when \( x \cdot y \neq y \cdot x \). This illustrates the advantage of dealing with deltoids that have a computable representation. However, modifiers of modifiers can work more effectively with free deltoids, because there are no equations to restrict the modifiers of modifiers. It is possible, for example, to completely remove a delta. In contrast, if there were equations that need to be satisfied, this would not be possible.

The second point to discuss is the consequences of whether delta application is a delta action. Consider a free deltoid where delta application is an action. This induces
equations on the free deltoid, namely,
\[ f \equiv g \iff \forall p. f(p) = g(p). \]
In such a situation, one could work with the free deltoid modulo the induced equations. These may not, however, be easy to determine. If the deltoid is not free, one needs to ensure that the equations on the deltoid satisfy the above constraint. If delta application is not an action, then one cannot rely on the equations resulting from composition. For example, if \((x \cdot y)(p) \neq x(y(p))\), then even \(z = x \cdot y\) cannot be used when reasoning about deltas. Thus, reasoning about conflicts and conflict resolution is inherently unreliable.

As above, if modifiers of modifiers are present, then they can be more flexibly applied whenever there are few equations between modifiers, which is the case if delta application is not an action.

9. Related Work

In general, approaches to facilitating automated product generation for software product lines can be classified in two main directions (Kästner et al., 2008). Firstly, annotative approaches, such as conditional compilation, frames (Zhang & Jarzabek, 2003) or COLORED FEATHERWEIGHT JAVA (CFJ) (Kästner & Apel, 2008), mark a model of the complete product line with respect to product features and remove marked product parts to obtain a product for a particular feature configuration.

Secondly, compositional approaches, such as delta modeling (Schaefer et al., 2009; Schaefer, 2010; Schaefer et al., 2010; Schaefer & Damiani, 2010), associate product fragments to product features, which are assembled to implement a particular feature configuration. A prominent example of this approach is AHEAD (Batory et al., 2004), which can be applied on the design as well as on the implementation level. In AHEAD, a product is built by stepwise refinement of a base module with a sequence of feature modules. Design-level models can also be constructed using aspect-oriented composition techniques (Heidenreich & Wende, 2007; Völter & Groher, 2007; Noda & Kishi, 2008). Apel et al. (2009a) apply model superposition to compose model fragments. Perrouin et al. (Perrouin et al., 2008) obtain a product model by model composition and subsequently refinement by model transformation. In Haugen et al. (2008), a set of models is represented by a base model with associated variability and resolution models determining how modeling elements of the base model have to be replaced for a particular product model.

On the programming language level, several program modularization techniques (Lopez-Herrejon et al., 2005), such as aspects (Kästner et al., 2007), framed aspects (Loughran & Rashid, 2004), mixins (Smaragdakis & Batory, 2002), hyperslices (Tarr et al., 1999) or traits (Ducasse et al., 2006; Bettini et al., 2010), are used to implement features in a compositional fashion. In addition, the modularity concepts of recent languages, such as SCALA (Odersky, 2007) or NEWSpeak (Bracha, 2007), can be used to represent product features. CeasarJ (Mezini & Ostermann, 2004) and Aspectual Feature Modules (Apel et al., 2008b) are proposed as a combination of feature modules and aspects to modularize crosscutting concerns.
The notion of program deltas was introduced by Lopez-Herrejon et al. (2005) to describe the modifications of object-oriented programs. Schaefer et al. (2009) introduced delta modeling as a means to develop product line artifacts suitable for automated product derivation and implemented with frame technology (Zhang & Jarzabek, 2003). In subsequent work (Schaefer, 2010), delta modeling was extended to a seamless model-based development approach for SPLs, where an initial product line representation is stepwise refined until an implementation can be generated. The conceptual ideas of delta modeling have also been instantiated on the programming language level in an extension of Java with core and delta modules allowing the automatic generation of Java-based product implementations (Schaefer et al., 2010). In Schaefer & Damiani (2010) and Schaefer et al. (2011), a version of delta-oriented programming is proposed where products are generated only from delta modules applied to the empty product.

Originally, the delta model of a product line consisted of a single core and a set of incomparable product deltas (Schaefer et al., 2009; Schaefer, 2010). Conflicts between deltas applicable for the same feature configuration were prohibited. In order to express all possible products, an additional delta covering the combination of the potentially conflicting deltas had to be specified leading to product fragments. Subsequently, a partial ordering between deltas was introduced (Schaefer et al., 2010; Schaefer & Damiani, 2010; Schaefer et al., 2011). However, it was required that all conflicts were manually resolved by specifying an appropriate ordering. In contrast, in this paper, a more flexible notion of conflicts and conflict resolution is proposed that allows intermediate conflicts between deltas as long as they are eliminated later in a derivation by a conflict-resolving delta. The notion of conflict-resolving deltas is similar to lifters (Prehofer, 1997) or derivatives (Liu et al., 2006) in feature-oriented programming, which are used to facilitate the correct interaction between different feature modules. A delta that is applied for the combination of certain features to resolve a conflict can fill in the role of a lifter or derivative. However, deltas are more expressive than lifters or derivatives, e.g., by allowing removals of entities and by specifying complex application conditions to deal with arbitrary combinations of features.

The definition of a conflict as a lack of commutativity between modifications is also discussed in the context of program refactoring (Mens et al., 2005). The underlying formalisation uses graph transformation systems and critical pair analysis. Oldevik et al. (2009) define a conflict in a sequence of model transformations to occur if two transformations do not commute. A similar notion of conflict related to non-commutativity is observed by Apel et al. (2008a) when two aspects advise shared join points. In order to make non-commutative aspects commute, the aspects have to be refactored following a particular scheme. In contrast, in delta modeling, the conflicting deltas do not have to be changed, only a conflict-resolving delta has to be added.

On a completely different note, the version control system Darcs is formalised in terms of patch theory (Jacobson, 2009). The underlying formalism has some similarities with our work. Most notable is that ‘patches’ are modeled using a semigroup with inverses. This structure is a monoid at heart, with additional properties (such as inverses) that do not entirely make sense in our setting. The most significant similarity is that they deal with conflictors (entities for resolving conflicts), which are similar to our conflict...
resolving deltas. Conflictors have a more complex set of properties than our conflict resolving deltas due to the added structure of their core setting. Patch theory should nonetheless offer inspiration to guide future research.

10. Conclusion

Delta modeling is an approach to facilitating automated product derivation for software product lines. In this paper, we studied the conceptual ideas of delta modeling in an abstract, algebraic setting. One contribution of this work is the novel treatment of conflicts between deltas by explicit conflict-resolving deltas. Further, we extended the formalism with nested delta models to provide additional means to impose structure in a delta model and to increase modularity of delta specifications. In order to ensure that for every valid feature configuration a unique product is generated, a conflict-resolving delta has to exist for every pair of conflicting deltas in the model. We presented efficient conditions that allow checking the unambiguity of a product line without requiring that all products be generated.

For future work, we will be using the ideas of abstract delta modeling for the implementation of variability within the HATS ABS language (Hähnle, 2010). The HATS ABS is an abstract executable modeling language for adaptable, object-oriented, distributed systems. By defining delta modification operations for ABS modeling entities, the variability of an ABS model can be specified by an ABS model delta. An ABS model for a particular configuration in space or in time can be generated from a core ABS model by application of ABS model deltas. The abstract, algebraic results presented in this article, in particular regarding consistent conflict resolution, can be immediately transferred to ABS models. Finally, variants of abstract delta modeling, such as basing the framework on partial monoids with a partial composition operation, will be investigated.

References

D. Clarke and M. Helvensteijn and I. Schaefer 36


Ducasse, S., Nierstrasz, O., Schärl, N., Wuyts, R. & Black, A. 2006 Traits: A mechanism for fine-grained reuse. *ACM TOPLAS* 28 (2).


Noda, N. & Kishi, T. 2008 Aspect-Oriented Modeling for Variability Management. In SPLC.


Appendix A. Proofs

The first part of this appendix gives proofs of properties for the OO software delta and the AOP software delta (Sections 5 and 5.4). Because the AOP software deltoid is a conservative extension of the OO software deltoid, we give only proofs for the former case. Relevant proofs for the OO software deltas can be obtained by ignoring cases involving wrapping. The definitions of $B$, $\mathcal{M}$, and $\mathcal{W}$ used are those found in Section 5.4. The second part of the appendix gives proofs for properties stated in Section 8.

A.1. Useful Lemmas

The following lemmas will be useful later on. Their proofs are all straightforward from the definitions.

**Lemma 15.** If $f \in (\mathcal{I} \rightarrow \mathcal{M}^-)$ and $g \in (\mathcal{I} \rightarrow \mathcal{M}^-)$, then $(f \oplus \mathcal{M} g)^* = (f \oplus \mathcal{M} g^*)^*$.

**Lemma 16.** If $f \in (\mathcal{I} \rightarrow \mathcal{M}^-)$ and $g \in (\mathcal{I} \rightarrow \mathcal{M})$, then $(f \oplus \mathcal{M} g)^* = f \oplus \mathcal{M} g^*$.

**Lemma 17.** If $f \in (\mathcal{I} \rightarrow \mathcal{M}^-)$ and $g \in (\mathcal{I} \rightarrow \mathcal{M}^-)$, then $(f \oplus \mathcal{M} g)^* = f \oplus \mathcal{M} g^*$.

**Lemma 18.** If $f \in (\mathcal{I} \rightarrow \mathcal{M})$, then $f^* = f$. 
A.2. Proof of Lemma 9 (and hence Lemma 4)

Proof. We show that \(\cdot\) is associative and \(\emptyset\) is its neutral element. We start by working at the level of method modifications, then consider class-level modifications. First, note that if an operator \(\circ\) is associative, then operator \(\bigcirc\) is also associative. For arbitrary \(a, b, c \in M \cup \{-, \bot\}\), we show that \(\oplus_M\) is associative, i.e. \((a \oplus_M b) \oplus_M c = a \oplus_M (b \oplus_M c)\), by case distinction on \(a\):

- Case \(a = \bot\):
  \[
  (a \oplus_M b) \oplus_M c = (\bot \oplus_M b) \oplus_M c = b \oplus_M c = \bot \oplus_M (b \oplus_M c) = a \oplus_M (b \oplus_M c).
  \]

- Case \(a = -\):
  \[
  (a \oplus_M b) \oplus_M c = (- \oplus_M b) \oplus_M c = - \oplus_M c = - = - \oplus_M (b \oplus_M c) = a \oplus_M (b \oplus_M c).
  \]

- Case \(a = m\) for some \(m \in B\):
  \[
  (a \oplus_M b) \oplus_M c = (m \oplus_M b) \oplus_M c = m \oplus_M c = m = m \oplus_M (b \oplus_M c) = a \oplus_M (b \oplus_M c).
  \]

- Case \(a = w[\ ]\) for some \(w[\ ] \in W\). We make a case distinction on \(b\):
  - Case \(b = \bot\):
    \[
    (a \oplus_M b) \oplus_M c = (w[\ ] \oplus_M -) \oplus_M c = w[\ ] \oplus_M (- \oplus_M c) = w[\ ] a \oplus_M (b \oplus_M c).
    \]
  - Case \(b = -\):
    \[
    (a \oplus_M b) \oplus_M c = (w[\ ] \oplus_M -) \oplus_M c = - \oplus_M c = - = - \oplus_M (b \oplus_M c) = a \oplus_M (b \oplus_M c).
    \]
  - Case \(b = m\) for some \(m \in B\):
    \[
    (a \oplus_M b) \oplus_M c = (w[\ ] \oplus_M m) \oplus_M c = w[m] \oplus_M c = w[m] = w[\ ] \oplus_M m = a \oplus_M (b \oplus_M c).
    \]
  - Case \(b = u[\ ]\) for some \(u[\ ] \in W\). We make a case analysis one \(c\).
    - Case \(c = \bot\):
      \[
      (a \oplus_M b) \oplus_M c = (w[\ ] \oplus_M u[\ ]) \oplus_M \bot = w[u[\ ] \oplus_M \bot = w[u[\ ]] = w[\ ] a \oplus_M (b \oplus_M c).
      \]
    - Case \(c = -\):
      \[
      (a \oplus_M b) \oplus_M c = (w[\ ] \oplus_M u[\ ]) \oplus_M - = w[u[\ ]] \oplus_M - = - = - \oplus_M (b \oplus_M c).
      \]
    - Case \(c = m\) for some \(m \in B\):
      \[
      (a \oplus_M b) \oplus_M c = (w[\ ] \oplus_M u[\ ]) \oplus_M m = w[u[\ ]] \oplus_M m = w[u[m]] = w[\ ] \oplus_M m = a \oplus_M (b \oplus_M c).
      \]
    - Case \(c = v[\ ]\) for some \(v[\ ] \in W\).
      \[
      (a \oplus_M b) \oplus_M c = (w[\ ] \oplus_M u[\ ]) \oplus_M v[\ ] = w[u[\ ]] \oplus_M v[\ ] = w[u[v[\ ]]] = a \oplus_M (b \oplus_M c).
      \]

Thus \(\oplus_M\) is associative. Consequently, so is \(\bigcirc_M\).

For arbitrary \(a, b, c \in \{r\} \times (I \rightarrow M) \cup \{u\} \times (I \rightarrow M^-) \cup \{-, \bot\}\), we show that \(\oplus_c\) is associative, i.e. \((a \oplus_c b) \oplus_c c = a \oplus_c (b \oplus_c c)\) by case distinction on \(a\):
— Case \( a = \perp \).

\[(a \oplus_C b) \ominus_C c = (\perp \ominus_C b) \ominus_C c = \perp \ominus_C (b \ominus_C c) = a \ominus_C (b \ominus_C c).\]

— Case \( a = - \).

\[(a \ominus_C b) \ominus_C c = (- \ominus_C b) \ominus_C c = - \ominus_C (b \ominus_C c) = a \ominus_C (b \ominus_C c).\]

— Case \( a = r f \) for some \( f \in (\mathcal{I} \rightarrow \mathcal{M}) \).

\[(a \ominus_C b) \ominus_C c = (r f \ominus_C b) \ominus_C c = r f \ominus_C c = r f = r f \ominus_C (b \ominus_C c) = a \ominus_C (b \ominus_C c).\]

— Case \( a = u f \) for some \( f \in (\mathcal{I} \rightarrow \mathcal{M}^-) \). We make a case distinction on \( b \):

- Case \( b = \perp \).

\[(a \ominus_C b) \ominus_C c = a \ominus_C c = a \ominus_C (b \ominus_C c).\]

- Case \( b = - \). For some \( f \in (\mathcal{I} \rightarrow \mathcal{M}^-) \):

\[
(a \ominus_C b) \ominus_C c = (u \ominus_C r g) \ominus_C c = r (f \ominus_M g)^* \ominus_C c = r (f \ominus_M g)^* = u \ominus_C r g c = u \ominus_C (b \ominus_C c).
\]

- Case \( b = r g \) for some \( g \in (\mathcal{I} \rightarrow \mathcal{M}) \).

\[
(a \ominus_C b) \ominus_C c = (u \ominus_C r g) \ominus_C c = r (f \ominus_M g)^* \ominus_C c = r (f \ominus_M g)^* = u \ominus_C r g c = u \ominus_C (b \ominus_C c).
\]

- Case \( b = u g \) for some \( g \in (\mathcal{I} \rightarrow \mathcal{M}^-) \). We make a case distinction on \( c \):

- Case \( c = \perp \).

\[
(a \ominus_C b) \ominus_C c = (a \ominus_C b) \ominus_C \perp = a \ominus_C b = a \ominus_C (b \ominus_C \perp) = a \ominus_C (b \ominus_C c).
\]

- Case \( c = - \).

\[
(a \ominus_C b) \ominus_C c = (u \ominus_C u g) \ominus_C c = u (f \ominus_M g) \ominus_C c = r (f \ominus_M g)^* = u \ominus_C r g c = u \ominus_C (b \ominus_C c).
\]

The step \( r (f \ominus_M g)^* = r (f \ominus_M g)^* \) follows from Lemma 15.

- Case \( c = r h \) for some \( h \in (\mathcal{I} \rightarrow \mathcal{M}) \).

\[
(a \ominus_C b) \ominus_C c = (u \ominus_C u g) \ominus_C r h = u (f \ominus_M g) \ominus_C r h = r ((f \ominus_M g) \ominus_M h)^* = u \ominus_C r g \ominus_M h = u \ominus_C (b \ominus_C c).
\]

- Case \( c = u h \) for some \( h \in (\mathcal{I} \rightarrow \mathcal{M}^-) \).

\[
(a \ominus_C b) \ominus_C c = (u \ominus_C u g) \ominus_C u h = u (f \ominus_M g) \ominus_C u h = u (f \ominus_M (g \ominus_M h)) = u \ominus_C u (g \ominus_M h) = u \ominus_C (b \ominus_C c).
\]

Thus \( \ominus_C \) is associative. Consequently, so are \( \ominus_M \) and \( \ominus \).

Neutrality of \( \ominus \) in \( \cdot \) can be shown by realizing that \( \ominus (i) = \perp \) for all \( i \in \mathcal{I} \). The rest follows directly from the definition of \( \ominus_C \).

\( \blacksquare \)
A.3. Proof of Lemma 10 (and hence Lemma 5)

For this case, we state and prove a lemma which will prove most invaluable later on. Firstly, define (overload) \((-)^* : \mathcal{M} \cup \{d, -\} \rightarrow \mathcal{M} \cup \{\bot\}\) as \((-)^* = \bot, (\bot)^* = \bot, (m)^* = m\), for \(m \in \mathcal{M}\).

**Lemma 19.** Given \(a, d \in \mathcal{M} \cup \{-, \bot\}\).

1. Given \(b \in \mathcal{M} \cup \{-, \bot\}\). If \((a \oplus_M b)^* = a^*\), then \((a \oplus_M d \oplus_M b)^* = (a \oplus_M d)^*\).
2. Given \(b \in \mathcal{M} \cup \{-, \bot\}\). If \((a \oplus_M b)^* = a^*\), then \((a \oplus_M d \oplus_M b)^* = (a \oplus_M d)^*\).
3. Given \(b, c \in \mathcal{M} \cup \{-, \bot\}\). If \(a \oplus_M b \oplus_M c = a \oplus_M c \oplus_M b\), then \((a \oplus_M d \oplus_M b \oplus_M c)^* = (a \oplus_M d)^*\).
4. Given \(b \in \mathcal{M} \cup \{-, \bot\}\) and \(c \in \mathcal{M} \cup \{-, \bot\}\). If \((a \oplus_M b)^* = (a \oplus_M c \oplus_M b)^*\), then \((a \oplus_M d \oplus_M b)^* = (a \oplus_M d \oplus_M c \oplus_M b)^*\).
5. Given \(b, c \in \mathcal{M} \cup \{-, \bot\}\). If \((a \oplus_M b)^* = (a \oplus_M c)^*\), then \((a \oplus_M d \oplus_M b)^* = (a \oplus_M d \oplus_M c)^*\).

**Proof.**

**Case 1.**
Assume that \((a \oplus_M b^*) = a^*\), where \(b \in \mathcal{M} \cup \{-, \bot\}\). By case distinction on \(a:\)

\(- a = \bot\). Hence \(b^* = \bot\), thus \(b = \bot\) or \(b = \bot\). In the both cases \((a \oplus_M d \oplus_M b^*)^* = (a \oplus_M d \oplus_M \bot)^* = (a \oplus_M d)^*\).

\(- a = \bot\). Then clearly \((a \oplus_M d \oplus_M b^*)^* = (\bot)^* = (a \oplus_M d)^*\).

\(- a = m\) for some \(m \in \mathcal{B}\). Then clearly \((a \oplus_M d \oplus_M b^*)^* = (m)^* = (a \oplus_M d)^*\).

\(- a = w[\ ] \) for some \(w[\ ] \in W\). Hence \(b = \bot\) or \(b = \bot\). In the both cases, as with \(a = \bot\),

\((a \oplus_M d \oplus_M b^*)^* = (a \oplus_M d \oplus_M \bot)^*)^* = (a \oplus_M d)^*\).

**Case 2.**
Assume that \((a \oplus_M b^*) = a^*\), where \(b \in \mathcal{M} \cup \{-, \bot\}\). By case distinction on \(a:\)

\(- a = \bot\). Hence \(b^* = \bot\), thus \(b = \bot\). Now \((a \oplus_M d \oplus_M b)^* = (a \oplus_M d \oplus_M \bot)^* = (a \oplus_M d)^*\).

\(- a = \bot\). Then clearly \((a \oplus_M d \oplus_M b)^* = (\bot)^* = (a \oplus_M d)^*\).

\(- a = m\) for some \(m \in \mathcal{B}\). \((a \oplus_M d \oplus_M b)^* = (m)^* = (a \oplus_M d)^*\).

\(- a = w[\ ] \) for some \(w[\ ] \in W\). Hence \(b = \bot\). Thus, as with \(a = \bot\), Now \((a \oplus_M d \oplus_M b)^* = (a \oplus_M d \oplus_M \bot)^* = (a \oplus_M d)^*\).

**Case 3.**
For arbitrary \(a, b, c \in \mathcal{M} \cup \{-, \bot\}\), assume \(a \oplus_M b \oplus_M c = a \oplus_M c \oplus_M b\). Then for arbitrary \(k \in \mathcal{M} \cup \{-, \bot\}\), we show that \(a \oplus_M k \oplus_M b \oplus_M c = a \oplus_M k \oplus_M c \oplus_M b\), by case distinction on \(a:\)

\(- \text{Case } a = \bot\).

\(a \oplus_M b \oplus_M c = a \oplus_M c \oplus_M b \iff \bot \oplus_M b \oplus_M c = \bot \oplus_M c \oplus_M b \iff b \oplus_M c = c \oplus_M b\)

Hence \(a \oplus_M k \oplus_M b \oplus_M c = a \oplus_M k \oplus_M c \oplus_M b\) for all \(k\).

\(- \text{Case } a = \bot\).

Observe that \(- \oplus_M d = \bot\) for all \(d\). Thus \(a \oplus_M k \oplus_M b \oplus_M c = - \oplus_M k \oplus_M b \oplus_M c = - \oplus_M k \oplus_M c \oplus_M b = a \oplus_M k \oplus_M c \oplus_M b\) for all \(k\).

\(- \text{Case } a = m\) for some \(m \in \mathcal{B}\).
Observe that $m \oplus_M d = m$ for all $d$. Thus $a \oplus_M k \oplus_M b \oplus_M c = m = m \oplus_M k \oplus_M b \oplus_M c$ for all $k$.

— Case $a = w[ ]$ for some $w[ ] \in W$.

Consider the following table whose elements are $(a \oplus_M b \oplus_M c, a \oplus_M c \oplus_M b)$—omitting entries above the diagonal for reasons of symmetry:

<table>
<thead>
<tr>
<th>$b^c$</th>
<th>$\bot$</th>
<th>$\neg$</th>
<th>$n$</th>
<th>$v[ ]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bot$</td>
<td>$(w[ ], w[ ])$</td>
<td>$(\neg, \neg)$</td>
<td>$m$</td>
<td>$w[ ]$</td>
</tr>
<tr>
<td>$\neg$</td>
<td>$(w[m], w[m])$</td>
<td>$(w[m], \neg)$</td>
<td>$(w[n], w[n])$</td>
<td>$(w[u], [w[u][ ]], w[u][ ]])$</td>
</tr>
</tbody>
</table>

Clearly $a \oplus_M b \oplus_M c = a \oplus_M c \oplus_M b$ only when $b = \bot$, $c = \bot$ or $b = c$. It is straightforward to see in these cases that $a \oplus_M k \oplus_M b \oplus_M c = a \oplus_M k \oplus_M c \oplus_M b$ for all $k$.

Thus $\forall a, b, c \in M \cup \{\bot, \neg\} : a \oplus_M b \oplus_M c = a \oplus_M c \oplus_M b$.

**Case 4.**

Assume that $(a \oplus_M b)^* = (a \oplus_M (c \oplus_M b))^*$, where $b \in M \cup \{\neg, \bot\}$ and $c \in M \cup \{\bot\}$.

By case distinction on $a$:

— $a = \bot$. This means that $b^* = (c \oplus_M b)^*$. Case analysis on $c$ results in the following:

  — $c = \bot$: Hence $(a \oplus_M d \oplus_M c \oplus_M b)^* = (a \oplus_M d \oplus_M \bot \oplus_M b)^* = (a \oplus_M d \oplus_M b)^*$, as desired.

  — $c = m, b = m$: Hence $(a \oplus_M d \oplus_M c \oplus_M b)^* = (a \oplus_M d \oplus_M m \oplus_M m)^* = (a \oplus_M d \oplus_M m)^*$, as desired.

  — $c = \neg, b = \neg$: Hence, $(a \oplus_M d \oplus_M c \oplus_M b)^* = (a \oplus_M d \oplus_M \neg \oplus_M \neg)^* = (a \oplus_M d \oplus_M \neg)^*$, as desired.

  — $c = w[ ], b = \neg$:

    — $c = \neg, b = \neg$: Hence, $(a \oplus_M d \oplus_M c \oplus_M b)^* = (a \oplus_M d \oplus_M w[ ] \oplus_M \neg)^* = (a \oplus_M d \oplus_M \neg)^* = (a \oplus_M d \oplus_M \neg)^*$, as desired.

— $a = \neg$. Hence $(a \oplus_M d \oplus_M b)^* = (\neg)^* = (a \oplus_M d \oplus_M (c \oplus_M b)^*)^*$.

— $a = m$ for some $m \in B$. Hence $(a \oplus_M d \oplus_M b)^* = (m)^* = (a \oplus_M d \oplus_M (c \oplus_M b)^*)^*$.

— $a = w[ ]$ for some $w[ ] \in W$. Again we can deduce that $b^* = (c \oplus_M b)^*$, and perform the same case analysis as for $a = \bot$. The cases are written sufficiently generally to apply directly here.

**Case 5.**

Assume that $(a \oplus_M b)^* = (a \oplus_M c)^*$, where $b, c \in M \cup \{\bot\}$. By case distinction on $a$:

— $a = \bot$. From this we deduce that $b = c$, and hence $(a \oplus_M d \oplus_M b)^* = (a \oplus_M d \oplus_M c)^*$.

— $a = \neg$. Hence $(a \oplus_M d \oplus_M b)^* = (\neg)^* = (a \oplus_M d \oplus_M c)^*$.

— $a = m$ for some $m \in B$. Hence $(a \oplus_M d \oplus_M b)^* = (m)^* = (a \oplus_M d \oplus_M c)^*$.

— $a = w[ ]$ for some $w[ ] \in W$. From this we deduce that $b = c$, and hence $(a \oplus_M d \oplus_M b)^* = (a \oplus_M d \oplus_M c)^*$.
Now we begin the main proof. Let $\mathcal{Y} = \{(r) \times (\mathcal{I} \rightarrow \mathcal{M})\} \cup \{(u) \times (\mathcal{I} \rightarrow \mathcal{M}^{-})\} \cup \{-, \perp\}$. It is clear that for $x, y, z \in \mathcal{D}$, $z \cdot y \cdot x = z \cdot x \cdot y$ implies $z \cdot d \cdot y \cdot x = z \cdot d \cdot x \cdot y$ if and only if for all $i \in \mathcal{I}$, $z(i) \odot y(i) \odot x(i) = z(i) \odot x(i) \odot y(i)$ implies $z(i) \odot d(i) \odot y(i) \odot x(i) = z(i) \odot d(i) \odot x(i) \odot y(i)$. Note that each $x(i), y(i), z(i), d(i) \in \mathcal{Y}$.

So we must show that for $p, q, r \in \mathcal{Y}$ if $p \odot r \odot q \odot r = p \odot q \odot r$, then for all $o \in \mathcal{Y}
\begin{align*}
p \odot q \odot r \odot q \odot r = p \odot q \odot r \odot q \odot r.
\end{align*}
Assume $p \odot q \odot r \odot q \odot r = p \odot q \odot r \odot q \odot r$. Let $o \in \mathcal{Y}$.

Firstly by considering the different cases for $o$, we see that when $o = \perp$, we have that $p \odot q \odot r \odot q \odot r = p \odot q \odot r \odot q \odot r$, from which the result follows, and when $o = -$ or $o = \top$, we have that for all $x \cdot o \odot q \odot r = o$, from which the desired result again quickly follows.

Assume that $o = u \odot d$. We proceed by case analysis on $p$.

— Case $p = \perp$.

From the definition of $\odot q \odot r$, it is clear that $\perp$ is the unit of $\odot q \odot r$, thus $\perp \odot q \odot r \odot q \odot r = \perp$. The desired result follows immediately.

— Case $p = -$.

From the definition of $\odot q \odot r$, it is clear that for all $q \odot r \odot q \odot r = -$, from which the result follows immediately.

— Case $p = \top$ for some $h \in (\mathcal{I} \rightarrow \mathcal{M})$.

From the definition of $\odot q \odot r$, it is clear that for all $q \odot r \odot q \odot r = h$, from which the result follows immediately.

— Case $p = u \odot h$ for some $h \in (\mathcal{I} \rightarrow \mathcal{M}^{-})$.

Firstly observe, as before, if $q = \perp$ or $r = \perp$, then $q \odot r \odot q \odot r \odot q$, from which the desired result follows.

The following table contains $(p \odot q \odot r, p \odot q \odot r)$ for the remaining combinations, again removing symmetry.

<table>
<thead>
<tr>
<th>$q \odot r$</th>
<th>$- \quad u \quad f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u \odot e$</td>
<td>$(r, h^<em>) \ast (r, h^</em>)$</td>
</tr>
<tr>
<td>$r \odot e$</td>
<td>$(r, h \odot r(e) \ast (r, h^*))$</td>
</tr>
<tr>
<td>$r \odot f$</td>
<td>$(r, h \odot r(e)) \ast (r, h^*)$</td>
</tr>
</tbody>
</table>

Note that we can determine the values for $p \odot q \odot r \odot q \odot r$ and $p \odot q \odot r \odot q \odot r$ by replacing $h$ by $h \odot d$ in each case.

Now we perform a case analysis based on these entries:

— Case $q = r = -$. Easy.

— Case $q = u \odot e, r = -$. From the table we deduce $(h \odot u \odot e)^* = h^*$. The desired result, namely $(h \odot u \odot d)^* = (h \odot d)^*$, follows from Lemma 19(1), lifted from $\odot u \odot d$ to $\odot u \odot d$.

— Case $q = r \odot e, r = -$. From the table we deduce $(h \odot u \odot e)^* = h^*$. The desired result, namely $(h \odot u \odot d)^* = (h \odot d)^*$ follows from Lemma 19(2), lifted from $\odot u \odot d$ to $\odot u \odot d$.

— Case $q = u \odot e, r = u \odot f$. From the table we deduce $h \odot u \odot e \odot u \odot f = h \odot u \odot f \odot u \odot e$. 

The desired result, namely \((h \overline{\oplus}_M d) \overline{\oplus}_M (e \overline{\oplus}_M f) = (h \overline{\oplus}_M d) \overline{\oplus}_M (f \overline{\oplus}_M e)\), follows from Lemma 19(3), lifted from \(\oplus_M\) to \(\overline{\oplus}_M\).

- Case \(q = r, e = f\). From the table we deduce \((h \overline{\oplus}_M e)^* = (h \overline{\oplus}_M (f \overline{\oplus}_M e))^*\). The desired result, namely \(((h \overline{\oplus}_M d) \overline{\oplus}_M e)^* = ((h \overline{\oplus}_M d) \overline{\oplus}_M (f \overline{\oplus}_M e))^*\), follows from Lemma 19(4), lifted from \(\oplus_M\) to \(\overline{\oplus}_M\).

- Case \(q = r, e = f\). From the table we deduce \((h \overline{\oplus}_M e)^* = (h \overline{\oplus}_M f)^*\). The desired result, namely \(((h \overline{\oplus}_M d) \overline{\oplus}_M e)^* = ((h \overline{\oplus}_M d) \overline{\oplus}_M f)^*\), follows from Lemma 19(5), lifted from \(\oplus_M\) to \(\overline{\oplus}_M\).

Thus we have that for all \(p, q, r \in Y\), that \(p \oplus_M q \oplus_M r = p \oplus_M r \oplus_M q\) implies that \(\forall o \in Y : p \oplus_M o \oplus_M q \oplus_M r = p \oplus_M o \oplus_M r \oplus_M q\).

Hence from (***), we have the desired result.

A.4. Proof of Lemma 11 (and hence Lemma 6)

We show that \(\cdot\) is associative and \(\otimes\) is its neutral element. We start by working at the level of method modifications, then consider class-level modifications.

For arbitrary \(a, b \in M \cup \{-, \perp\}\) and arbitrary \(c \in M \cup \{-, \perp\}\), we show that \((a \oplus_M b) \otimes_M c = a \otimes_M (b \otimes_M c)\) by case distinction on \(a\):

- Case \(a = \perp\).

\[(a \oplus_M b) \otimes_M c = (\perp \oplus_M b) \otimes_M c = b \otimes_M c = \perp \otimes_M (b \otimes_M c) = a \otimes_M (b \otimes_M c)\].

- Case \(a = -\).

\[(a \oplus_M b) \otimes_M c = (- \oplus_M b) \otimes_M c = - \otimes_M c = - = - \otimes_M (b \otimes_M c) = a \otimes_M (b \otimes_M c)\].

- Case \(a = m\) for some \(m \in B\).

\[(a \oplus_M b) \otimes_M c = (m \oplus_M b) \otimes_M c = m \otimes_M c = m = m \otimes_M (b \otimes_M c) = a \otimes_M (b \otimes_M c)\].

- Case \(a = w[\ ]\) for some \(w[\ ] \in W\). We make a case distinction on \(b\).

  - Case \(b = \perp\).

  \[(a \oplus_M b) \otimes_M c = (w[\ ] \oplus_M \perp) \otimes_M c = w[\ ] \otimes_M c = w[\ ] \otimes_M (\perp \otimes_M c) = a \otimes_M (b \otimes_M c)\].

  - Case \(b = -\).

  \[(a \oplus_M b) \otimes_M c = (w[\ ] \oplus_M -) \otimes_M c = - \otimes_M c = - = w[\ ] \otimes_M - = w[\ ] \otimes_M (- \otimes_M c) = a \otimes_M (b \otimes_M c)\].

  - Case \(b = m\) for some \(m \in B\).

  \[(a \oplus_M b) \otimes_M c = (w[\ ] \oplus_M m) \otimes_M c = w[m] \otimes_M c = w[m] \otimes_M c = w[\ ] \otimes_M m = w[\ ] \otimes_M (m \otimes_M c) = a \otimes_M (b \otimes_M c)\].

  - Case \(b = u[\ ]\) for some \(u[\ ] \in W\). We make a case distinction on \(c\).

    - Case \(c = \perp\).

    \[(a \oplus_M b) \otimes_M c = (w[\ ] \oplus_M u[\ ]) \otimes_M \perp = w[u[\ ]] \otimes_M \perp = \perp = w[\ ] \otimes_M \perp = w[\ ] \otimes_M (u[\ ] \otimes_M \perp) = a \otimes_M (b \otimes_M c)\].
Case \( c = m \) for some \( m \in \mathcal{B} \).

\[
(a \oplus_M b) \odot_M c = (w[u]\ | \odot_M u[\ ] \odot_M m = w[u]\ | \odot_M m) = w[u[m]]
\]

Thus \( \forall a, b \in \mathcal{M} \cup \{-, \perp\} : \forall c \in \mathcal{M} \cup \{\perp\} : (a \oplus_M b) \odot_M c = a \odot_M (b \odot_M c) \). Consequently, also \( \forall f, g \in (\mathcal{I} \to \mathcal{M}^{-1}) : \forall h \in (\mathcal{I} \to \mathcal{M}) : (f \odot_M g) \odot_M h = f \odot_M (g \odot_M h) \) holds.

Now we move to class-level modifications. For arbitrary \( a, b \in \{r\} \times (\mathcal{I} \to \mathcal{M}) \cup \{u\} \times (\mathcal{I} \to \mathcal{M}^{-1}) \cup \{-, \perp\} \) and arbitrary \( c \in (\mathcal{I} \to \mathcal{M}) \cup \{\perp\} \), we show that \( (a \oplus_C b) \odot_C c = a \odot_C (b \odot_C c) \) by case distinction on \( a \):

- Case \( a = \perp \).

\[
(a \oplus_C b) \odot_C c = (\perp \oplus_C b) \odot_C c = b \odot_C c = \perp \odot_C (b \odot_C c) = a \odot_C (b \odot_C c).
\]

- Case \( a = - \).

\[
(a \oplus_C b) \odot_C c = (- \oplus_C b) \odot_C c = - \odot_C c = \perp = - \odot_C (b \odot_C c) = a \odot_C (b \odot_C c).
\]

- Case \( a = r \) for some \( f \in (\mathcal{I} \to \mathcal{M}) \).

\[
(a \oplus_C b) \odot_C c = (r f \oplus_C b) \odot_C c = r f \odot_C c = f = r f \odot_C (b \odot_C c) = a \odot_C (b \odot_C c).
\]

- Case \( a = u \) for some \( f \in (\mathcal{I} \to \mathcal{M}^{-1}) \). We make a case distinction on \( b \):

  - Case \( b = \perp \).

\[
(a \oplus_C b) \odot_C c = (u \oplus_C \perp) \odot_C c = a \odot_C c = a \odot_C (\perp \odot_C c) = a \odot_C (b \odot_C c).
\]

  - Case \( b = - \).

\[
(a \oplus_C b) \odot_C c = (u f \oplus_C \perp) \odot_C c = r f \odot_C c = f^* = u f \odot_C c = u f \odot_C (b \odot_C c).
\]

  - Case \( b = r \) for some \( g \in (\mathcal{I} \to \mathcal{M}) \).

\[
(a \oplus_C b) \odot_C c = (u f \oplus_C r g) \odot_C c = r (f \odot_M g)^* \odot_C c = (f \odot_M g)^* = f \odot_M g = u f \odot_C g = u f \odot_C (r g \odot_C c) = a \odot_C (b \odot_C c).
\]

Step \( \dagger \) follows from Lemma 16.

- Case \( b = u \) for some \( g \in (\mathcal{I} \to \mathcal{M}^{-1}) \). We make a case distinction on \( c \):

  - Case \( c = \perp \).

\[
(a \oplus_C b) \odot_C c = (u f \oplus_C u g) \odot_C \perp = u (f \odot_M g) \odot_C \perp = (f \odot_M g)^* = f \odot_M g = u f \odot_C g = u f \odot_C (u g \odot_C \perp) = a \odot_C (b \odot_C c).
\]

Step \( \dagger \) follows from Lemma 17.

- Case \( c = h \) for some \( h \in (\mathcal{I} \to \mathcal{M}) \).

\[
(a \oplus_C b) \odot_C c = (u f \oplus_C u g) \odot_C h = u (f \odot_M g) \odot_C h = (f \odot_M g) \odot_M h = f \odot_M (g \odot_M h) = f \odot_M (u g \odot_C h) = u f \odot_C (u g \odot_C h) = a \odot_C (b \odot_C c).
\]

Thus \( (y \odot_C x) \odot_C p = y \odot_C (x \odot_C p) \) also holds for all \( x, y \in \mathcal{D} \) and \( p \in \mathcal{P} \). Or in standard notation: \( (y \cdot x)(p) = y(x(p)) \).
A.5. Proof of Lemma 13

Proof. We need to show that this holds for each axiom in Definition 35.

1. \((\epsilon \cdot \mu)(p) = \epsilon(\mu(p)) = \mu(p) = \mu(\epsilon(p)) = (\mu \cdot \epsilon)(p)\).
2. \((\mu \cdot (\nu \cdot \eta))(p) = \mu((\nu \cdot \eta)(p)) = \mu(\nu(\eta(p))) = (\mu \cdot \nu)(\eta(p)) = ((\mu \cdot \nu) \cdot \eta)(p)\).
3. \((m \cdot i)(p) = m(i(p)) = m \circ (i + p) = (m \circ i) + (m \cdot p) = (m \circ i) + m(p) = (m \circ i)(m(p)) = ((m \circ i) \cdot m)(p)\).
4. \((i \cdot j)(p) = i(j(p)) = i + (j + p) = j + (i + p) = (j \cdot i)(p)\). This is also equal to \((i + j)(p)\) and \((j + i)(p)\), as, for example \(i + (j + p) = (i + j) + p = (i + j)(p)\).
5. \((i \cdot i)(p) = i(i(p)) = i + (i + p) = (i + i) + p = (i + i)(p)\). Also, \((i + i) + p = i + p = i(p)\).
6. \((m \cdot n)(p) = m(n(p)) = m \circ (n \cdot p) = (m \bullet n) \circ p = (m \bullet n)(p)\).
7. \(1(p) = 1 \circ p = p = \epsilon(p) = p = 0 + p = 0(p)\). 

A.6. Proof of Lemma 14

Proof. Firstly, \([0, 1] = 0 \cdot 1 = \epsilon \cdot \epsilon = \epsilon\). Secondly, let \(q = \langle i, m \rangle\) and \(q' = \langle i', m' \rangle\). On one hand, \([\langle i, m \rangle \cdot \langle i', m' \rangle] = [\langle i + m \odot i', m \bullet m' \rangle]\) \(= (i + m \odot i') \cdot (m \bullet m')\). On the other hand, \([\langle i, m \rangle] \cdot [\langle i', m' \rangle] = i \cdot m \odot i' \cdot m' = i \cdot (m \odot i') \cdot (m \bullet m') = (i + m \odot i') \cdot (m \bullet m')\).

A.7. Proof of Theorem 5

Proof.

1. \(\langle\langle\langle i, 0 \rangle\rangle\rangle = \langle i, 0 \rangle \cdot \langle i, 0 \rangle = \langle i + 0, 1 \bullet l \rangle = \langle i + 0, 1 \bullet l \rangle = \langle i, l \rangle\).
2. By induction on \(\mu\).
   - Case \(i\). \([\langle i, 0 \rangle] = \langle\langle\langle i, 1 \rangle\rangle\rangle = i \cdot i = \epsilon = \epsilon\).
   - Case \(m\). \([\langle m, 0 \rangle] = \langle 0, m \rangle\) \(= 0 \cdot m = \epsilon \cdot m = m\).
   - Case \(\mu \cdot \nu\). \([\langle\langle\mu, \nu\rangle\rangle]\) \(= [\langle\langle\mu, \nu\rangle\rangle] = \langle\langle\langle\mu, \nu\rangle\rangle\rangle\) \(= \mu \cdot \nu\). using Lemma 14 and induction hypothesis.
3. Associativity is preserved as \(\bullet\) is associative. That \(\langle 0, 1 \rangle\) is the unit of \(\bullet\) is preserved by \([-\] \) follows from the fact that \(0 \cdot 1 = \epsilon\) and \(\epsilon\) is the unit of \(\cdot\). Finally, if \(\langle i_1, m_1 \rangle = \langle i_2, m_2 \rangle\), then \(i_1 = i_2\) and \(m_1 = m_2\). Now \([\langle i_1, m_1 \rangle] = m_1 \odot i_1 = m_2 \odot i_2 = [\langle i_2, m_2 \rangle]\).
4. We need to show that the axioms in Definition 35 are preserved by \(\langle\langle\cdot\rangle\rangle\).
   (1) Follows because \(\langle\epsilon\rangle = \langle 0, 1 \rangle\) is the unit of \(\cdot\).
   (2) Follows because \(\cdot\) is associative.
   (3) \(\langle m \odot i \rangle = \langle 0, m \rangle \cdot \langle i, 0 \rangle\) which is the same as \(\langle(\langle m \odot i \rangle \cdot m) = \langle m \odot i, 1 \rangle \cdot (0, m) = \langle m \odot i, m \rangle\).
   (4) \(\langle i \odot j \rangle = \langle i, 1 \rangle \cdot \langle j, 1 \rangle = \langle i + j, 1 \rangle = \langle i + j\rangle\), which is clearly also equal to \(\langle\langle i + j\rangle\rangle\) and \(\langle\langle j + i\rangle\rangle\).
   (5) \(\langle i \odot j \rangle = \langle i, 1 \rangle \cdot \langle i, 1 \rangle = \langle i + i, 1 \rangle = (i, 1) = \langle\langle i, i\rangle\rangle\).
   (6) \(\langle m \cdot n \rangle = \langle 0, m \rangle \cdot \langle 0, n \rangle = (0 + (m \odot 0), m \bullet n) = (0, m \bullet n) = \langle m \cdot n \rangle\).
   (7) By definition we have \(\langle\epsilon\rangle = \langle 0, 1 \rangle\), \(\langle 0 \rangle = \langle 0, 0 \rangle\), and \(\langle 1 \rangle = \langle 0, 1 \rangle\).
A.8. Proof of Theorem 6

Proof. Firstly, $\text{image}(\langle i, m \rangle \cdot \langle p, 1 \rangle) = \text{image}(\langle i + (m \circ p), m \rangle) = i + (m \circ p)$ and $\langle [i, m] \rangle(p) = (i \cdot m)(p) = i(m(p)) = i + (m \circ p)$.

Secondly, we prove by induction using the stronger hypothesis that for all $\mu \in M_I$ and all $i \in I$, there exists an $m$ such that $\langle \langle \mu \rangle \rangle \cdot \langle p, 1 \rangle = \langle \mu(p), m \rangle$. Proceed by induction on form of $\mu$:

— Case $i$. $\langle \langle i \rangle \rangle \cdot \langle p, 1 \rangle = \langle i, 1 \rangle \cdot \langle p, 1 \rangle = \langle i + p, 1 \rangle$. Now $i(p) = i + p$, as desired.

— Case $m$. $\langle \langle m \rangle \rangle \cdot \langle p, 1 \rangle = \langle 0, m \rangle \cdot \langle p, 1 \rangle = \langle m \circ p, m \rangle$. Now $m(p) = m \circ p$, as desired.

— Case $\mu \cdot \nu$. $\langle \langle \mu \cdot \nu \rangle \rangle \cdot \langle p, 1 \rangle = \langle \langle \mu \rangle \rangle \cdot \langle \langle \nu \rangle \rangle \cdot \langle p, 1 \rangle$. By the induction hypothesis, there exists an $m$ such that $\langle \langle \nu \rangle \rangle \cdot \langle p, 1 \rangle = \langle \nu(p), m \rangle$. Similarly, applying the induction hypothesis to $\mu \in M_I$ and $\langle \nu(p), m \rangle \in P$, we obtain that there exists an $n$ such that $\langle \langle \mu \rangle \rangle \cdot \langle \nu(p), m \rangle = \langle \mu(\nu(p)), n \rangle$. By Definition 35 this equals $\langle \langle \mu \cdot \nu \rangle(p), n \rangle$, and we are done. 

$\square$
Appendix C

Pure Delta Oriented Programming

The paper “Pure Delta Oriented Programming” [104] follows.
Abstract

Delta-oriented programming (DOP) is a modular approach for implementing software product lines. Delta modules generalize feature modules by allowing removal of functionality. However, DOP requires to select one particular product as core product from which all products are generated. In this paper, we propose pure delta-oriented programming (Pure DOP) that is a conceptual simplification of traditional DOP. In Pure DOP, the requirement of one designated core product is dropped. Instead, program generation only relies on delta modules comprising program modifications such that Pure DOP is more flexible than traditional DOP. Furthermore, we show that Pure DOP is a true generalization of FOP and supports proactive, reactive and extractive product line engineering.

Categories and Subject Descriptors D.1.5 [Programming Techniques]: Object-oriented Programming; D.3.3 [Programming Languages]: Language Constructs and Features

General Terms Design, Languages, Theory

Keywords Software Product Line, Feature-oriented Programming, Delta-oriented Programming, Program Generation

1. Introduction

A software product line (SPL) is a set of software systems with well-defined commonalities and variabilities [12, 27]. The approaches to implementing SPL in the object-oriented paradigm can be classified into two main directions [19]. First, annotative approaches (e.g., [4, 17]) mark the source code of all products with respect to product features and remove marked code for particular feature configurations. Second, compositional approaches [23], associate code fragments to product features that are assembled to implement a given feature configuration.

Feature-oriented programming (FOP) [7] is a prominent approach for implementing SPLs by composition of feature modules. A feature module directly corresponds to a product feature. In the context of object-oriented programming, feature modules can introduce new classes or refine existing ones by adding fields and methods or by overriding existing methods. In delta-oriented programming (DOP) [29], feature modules are generalized to delta modules that additionally allow the removal of classes, fields and methods and that can refer to any combination of features. DOP requires selecting one particular product as designated core product. The core product is implemented in the core module. From this core module, all other products are generated by delta module application. Nevertheless, the requirement of the core product makes it difficult to deal with product line evolution, for instance, if the product line evolves such that the original core product is no longer a valid product. Furthermore, the uniquely determined core product prevents a true generalization of FOP by DOP, since feature module composition in FOP may start from several different base feature modules that may not correspond to valid products.

In this paper, we propose pure delta-oriented programming (Pure DOP) as a conceptual simplification of traditional DOP [29], which we will call Core DOP in the following. In Pure DOP, the requirement to chose one product as core product is dropped. Instead, only delta modules are used for product generation. Thus, we call the approach Pure DOP. A delta module can specify additions, removals classes or modifications of classes. In order to define a product line over a set of delta modules, each delta module is attached an application condition determining for which feature configurations the modifications of the delta module have to be applied. This creates the connection between the modifications of the delta modules and the product features [16]. Additionally, the delta modules can be partially ordered to ensure that for every feature configuration a uniquely defined product is generated.

The contribution of this work is twofold. First, Pure DOP relaxes the requirement of a single valid core product. This makes Pure DOP more flexible than Core DOP [29]. Pure DOP is a true generalization of FOP since every FOP product line can be understood as a Pure DOP product line which is not obvious for Core DOP. Further, Pure DOP supports proactive, reactive and extractive product line development [22] by allowing program generation from any set of existing legacy product implementations which is not directly possible with Core DOP. Second, in the presentation of (Pure) DOP given in this paper, the application conditions for delta modules, as well as the delta module ordering, are only defined when a product line is specified. In contrast, in the traditional presentation of (Core) DOP [29], application conditions and ordering are fixed for each delta module. The separation of application conditions and application ordering from the specification of the modifications in a delta module increases the reusability of delta modules and allows developing different product lines over the same set of delta modules.

The paper is organized as follows: In Section 2, we present Pure DOP of JAVA programs and show its formalization LPJ using LJ (LIGHTWEIGHT JAVA) [32] as base language for the generated products in Section 3. We show that Pure DOP is a
true generalization of POF by providing an embedding of LFJ (LIGHTWEIGHT FEATURE JAVA) [13] into LPJ in Section 4. We demonstrate that Pure DOP supports proactive, extractive and reactive SPLE in Section 5. We show that Pure DOP is a conceptual simplification of Core DOP in Section 6.

2. Pure Delta-oriented Programming

In order to illustrate the main concepts of Pure DOP, we use the expression product line (EPL) as described in [23]. The EPL is based on the expression problem [35], an extensibility problem, that has been proposed as a benchmark for data abstractions capable to support new data representations and operations. We consider the following grammar:

\[
\begin{align*}
\text{Exp} & ::= \text{Lit} \mid \text{Add} \mid \text{Neg} \\
\text{Lit} & ::= \langle \text{non-negative integers} \rangle \\
\text{Add} & ::= \text{Exp} + \text{Exp} \\
\text{Neg} & ::= \text{Exp}
\end{align*}
\]

Two different operations can be performed on the expressions described by this grammar: printing, which returns the expression as a string, and evaluation, which computes the value of the expression. The products in the EPL can be described by two feature sets, the ones concerned with data Lit, Add, Neg and the ones concerned with operations Print and Eval. Lit and Print are mandatory features. The features Add, Neg and Eval are optional. Figure 1 shows the feature model of [16] of the EPL.

**Pure Delta Modules** The main concept of pure DOP are delta modules which are containers of modifications to an object-oriented program. The modifications inside a delta module act on the class level by adding, removing and modifying classes. A class can be modified by changing the super class, by adding and removing fields and methods and by modifying methods. The modification of a method can either replace the method body by another implementation, or wrap the existing method using the original construct. The original construct expresses a call to the method with the same name before the modifications and is bound at the time the product is generated. Before or after the original construct, other statements can be introduced wrapping the existing method implementation. The original construct (similar to the Super() call in AHEAD [7]) avoids a combinatorial explosion of the number of delta modules in case the original method has to be wrapped differently for a set of optional features. Listing 1 contains the delta module for introducing the Lit feature. Listing 2 contains the delta modules for incorporating the Print and Eval features by modification of the class Lit.

**Pure Delta-oriented Product Lines** The delta-oriented specification of a product line comprises the set of product features, the set of valid feature configurations and the set of delta modules necessary to implement all valid products. Furthermore, the specification of a product line in Pure DOP associates each delta module with the set of features configurations in which the delta modules has to be applied by attaching an application condition in a when clause. The application condition is a propositional constraint over the set

```
Figure 1. Feature model for Expression Product Line
```

```
Listing 1: Delta module for Lit feature
```

```
Listing 2: Delta modules for Print and Eval features
```

```
Listing 3: Delta modules for Add, Print and Eval features
```

```
Listing 4: Delta modules for Neg, Print and Eval features
```
of features. Since only feature configurations which are valid according to the feature model are used for program generation, the application conditions attached to delta modules have to be understood as a conjunction with the formula describing the set of valid feature configurations. The application condition creates the link from the features in the feature model to the delta modules. In this way, we can specify delta modules for combinations of features to solve the optional feature problem [20].

In order to obtain a product for a particular feature configuration, the modifications specified in the delta modules with valid application conditions are applied incrementally to the previously generated product. The first delta module is applied to the empty product. All other delta modules are applied to the respective intermediate product. The modifications of a delta model are applicable to a (possibly empty) product if each class to be removed or modified exists and, for every modified class, if each method or field to be removed exists, if each method to be modified exists and has the same header as the modified method, and if each class, method or field to be added does not exist. During product generation, every delta module must be applicable. Otherwise, the resulting product is undefined. In particular, the first delta module that is applied can only contain additions.

In order to ensure that each delta module is applicable during product generation, the delta modules are ordered in the specification of a pure delta-oriented product line. The order of delta module application is defined by a total order on a partition of the set of delta modules. Deltas in the same partition can be applied in any order to the previous product, but the order of the partitions is fixed. The ordering captures semantic requires relations that are necessary for the applicability of the delta modules.

Listing 5 shows a delta-oriented specification of the EPL. In this specification, application conditions are attached to the delta modules that are required to implement the different products of the EPL. The used delta modules are depicted in Listings 1, 2, 3 and 4. The order of delta module application is defined by an ordered list of the delta module sets which are enclosed by [ ... ].

### Product Generation

The generation of a product for a given feature configuration consists of two steps, performed automatically:

1. Find all delta modules with a valid application condition; and
2. Apply the selected delta modules to the empty product in any linear ordering that is consistent with the total order on the partitioning of the delta modules.

If two delta modules add, remove or modify the same class, the ordering in which the delta modules are applied can influence the resulting product. However, for a product line implementation, it is essential to guarantee that for every valid feature configuration exactly one product is generated. This property is called *unambiguity* of the product line. For unambiguity, the delta modules in each partition must be compatible. This means that if one delta module in a partition adds or removes a class, no other delta module in the same partition may add, remove or modify the same class, and the modifications of the same class in different delta modules in the same partition have to be disjoint. Defining the order of delta module application by a total ordering on a delta module partition provides an efficient way to ensure unambiguity, since only the compatibility of each partition has to be checked.

### 3. A Kernel Calculus for Pure Delta Modules

In this section, we introduce the syntax and the semantics of LPAJ (LIGHTWEIGHT PURE DELTA JAVA), a kernel calculus for Pure DOP of product lines of JAVA programs. LPAJ is based on LJ (LIGHTWEIGHT JAVA) [32]. Thus, it is particularly suitable for comparison with the formalization of FOP in LJF (LIGHTWEIGHT FEATURE JAVA) [13].

#### LPAJ Syntax

The syntax of LPAJ, as an extensions to LJ, is given in Figure 2. Following [15], we use the overline notation for possibly empty sequences. For instance, we write “\( \overline{s} \)” as short for a possibly empty sequence of statements “\( s_1 \ldots s_n \)” and “\( \overline{DC} \)” as short for a possibly empty sequence of delta clause definitions “\( DC_1 \ldots DC_n \)”.

Definitions of named elements (like delta clause or delta subclause definitions) are assumed to contain no duplicate names (that is, the names of the elements of the sequence must be distinct). The constructs for class definitions \( cd \), field definitions \( fd \), method definitions \( md \), method signatue \( ms \) and statement \( s \) are those of LJ [32] and (of LJF [13]). The metavariable \( \delta \) ranges over delta module names. A delta module definition DMD for a delta module with the name \( \delta \) can be understood as a mapping from class names to delta clause definitions. A delta clause definition DC can specify the addition, removal or modification of a class. The adds-domain, the removes-domain and the modifies-domain of a delta module definition DMD are defined as follows:

\[
\begin{align*}
\text{addsDom}(\text{DMD}) &= \{ C \mid \text{DMD}(C) = \text{adds} \; \text{class} \; C \} \\
\text{removesDom}(\text{DMD}) &= \{ C \mid \text{DMD}(C) = \text{removes} \; C \} \\
\text{modifiesDom}(\text{DMD}) &= \{ C \mid \text{DMD}(C) = \text{modifies} \; C \}
\end{align*}
\]

The modification of a class is defined by possibly changing the super class and by listing a sequence of delta subclauses DS defining modifications of methods and additions/removals of fields and methods. A delta modifies clause DC can be understood as a mapping from the keyword extending to an either empty or singleton set of class names and from field/method names to delta subclauses.

The adds-, removes- and modifies-domain of a delta modifies-clause DC are defined as follows:

\[
\begin{align*}
\text{addsDom}(\text{DC}) &= \{ \delta \mid \text{DC} = \text{adds} \; \text{domain} \; \delta \} \\
\text{removesDom}(\text{DC}) &= \{ \delta \mid \text{DC} = \text{removes} \; \delta \} \\
\text{modifiesDom}(\text{DC}) &= \{ \delta \mid \text{DC} = \text{modifies} \; \delta \}
\end{align*}
\]

Listing 5: Pure DOP specification of the EPL

<table>
<thead>
<tr>
<th>features</th>
<th>Lit, Add, Neg, Print, Eval</th>
</tr>
</thead>
<tbody>
<tr>
<td>configurations</td>
<td>Lit &amp; Print</td>
</tr>
</tbody>
</table>
| deltas | [ DLit, \\
|        | DAdd when Add, \\
|        | DNeg when Neg ] |
|        | [ DPrint, \\
|        | DEval when Eval, \\
|        | DAddEval when (Add & Eval), \\
|        | DNegPrint when Neg, \\
|        | DNegEval when (Neg & Eval) ] |

\[ DLit, \\
DAdd when Add, \\
DNeg when Neg \]

\([ \Delta \text{Lit}, \\
\Delta \text{Add when Add,} \\
\Delta \text{Neg when Neg} ] \]

\[
\begin{align*}
\text{DC} &= \{ \text{adds cd} \mid \text{modifies C} \} \\
&\quad \{ \text{extends C} \} \{ \text{DS} \} \\
&\quad \{ \text{removes C} \} \\
\text{DS} &= \{ \text{adds fd} \mid \text{adds md} \} \\
&\quad \{ \text{modifies md} \} \\
&\quad \{ \text{modifies wmd} \} \\
&\quad \{ \text{removes a} \} \\
\text{wmd} &= \{ \text{ms} \{ 5 \text{ original} \}; \text{ 5 return y}; \} \text{ method wrapper}
\end{align*}
\]

Figure 2. LPAJ: syntax of delta modules
The modification of a method, defined by a delta modifies subclause, can either replace the method body by another implementation, or wrap the existing method using the original() call. In both cases, the modified method must have the same header as the unmodified method. The original() call may only occur in the body of the method provided by a delta modifies subclause modifies wmd.

The occurrence of original() represents a call to the unmodified method where the formal parameters of the modified method are passed implicitly as arguments. In LFJ [13], the Super() construct of AHEAD [7] is modeled in the same way.

After we have defined the notion of delta modules over LJ, we can formalize LPΔJ product lines. We use the metavariabes \( \phi \) and \( \psi \) to range over feature names. We write \( \psi \) as short for the set \( \{ \psi \} \), i.e., the feature configuration containing the features \( \psi \). A delta module table DMT is a mapping from delta module names to delta module definitions. A LPΔJ SPL is a 5-tuple \( L = (\psi, \Phi, \Psi, \Gamma, \delta_{DMT}) \) consisting of:

1. the features \( \psi \) of the SPL,
2. the set of the valid feature configurations \( \Phi \subseteq \mathcal{P}(\psi) \),
3. a delta module table \( \delta_{DMT} \) containing the delta modules,
4. a mapping \( \Gamma : dom(\delta_{DMT}) \rightarrow \Phi \) determining for which feature configurations a delta module must be applied (which is denoted by the when clause in the example),
5. a total order \( \delta_{DMT} \) on a partition of \( dom(\delta_{DMT}) \), called the application partial order, determining the order of delta module application.

To simplify notation, in the following we always assume a fixed SPL \( L = (\psi, \Phi, \Psi, \Gamma, \delta_{DMT}) \). We further assume that the SPL L satisfies the following sanity conditions.

(i) For every class name \( C \) (except Object) appearing in DMT, we have \( C \subseteq \{ s | \delta_{DMT}(s) \neq \text{null} \} \), meaning that every class is added at least once.
(ii) The mapping \( \Delta : \Phi \rightarrow \mathcal{P}(\delta_{DMT}(\Delta)) \), such that \( \Delta(\psi) \), the set of names of delta modules whose application condition is satisfied by the feature configuration \( \psi \), is injective and such that \( (\psi, \Phi, \delta_{DMT}(\psi)) = \delta_{DMT}(C) \), i.e., for every feature configuration a different set of delta modules is applied and every delta module is applied for at least one feature configuration.

In the following, we write \( dom(\delta) \) as short for \( dom(\delta_{DMT}(\delta)) \), and we write \( \delta(\psi) \) as short for \( \delta_{DMT}(\delta(\psi)) \).

LPΔJ Product Generation A LJ program can be represented by a class table. A class table CT is a mapping from class names to class definitions. A delta module is applicable to a class table CT if each class to be removed or modified exists and, for every delta modifies clause, if each method or field to be removed exists, if each method to be modified exists and has the same header specified in method modifies subclause, and if each class, method or field to be added does not exist.

Given a delta module \( \delta \) and a class table CT such that \( \delta \) is applicable to CT, the application of \( \delta \) to CT, denoted by \( \delta(\psi, \Gamma) \), is the class table CT' defined as follows:

\[
\begin{align*}
\text{addsDom}(DC) &= \{ a \mid \text{DMT}(a) = \text{adds} \cdots a \cdots \} \\
\text{removesDom}(DC) &= \{ a \mid \text{DMT}(a) = \text{removes} a \} \\
\text{modifiesDom}(DC) &= \{ m \mid \text{DMT}(m) = \text{modifies} \cdots m \cdots \}
\end{align*}
\]

Figure 3. LFJ: syntax of feature modules

The semantics of the original() call is captured by replacing the occurrence of original() in the method body specified by the modifies subclause with the body of the unmodified method.

For any given total order of delta module application, a LPΔJ SPL defines a product generation mapping. That is, a partial mapping from each feature configuration \( \psi \) to the class table of the product that is obtained by applying the delta modules \( \Delta(\psi) \) to the empty class table according to the given order. The product generation mapping can be partial since a non-applicable delta module may be encountered during product generation such that the resulting product is undefined.

Unambiguous and Type-Safe LPΔJ Product Lines A LPΔJ SPL is unambiguous if all total orders of delta modules that are compatible with the application partial order define the same product generation mapping. In an unambiguous SPL, for every feature configuration at most one product implementation is generated.

In order to find a criterion for unambiguity, we define the notion of compatibility of a set of delta modules. A set of delta modules is called compatible if no class added or removed in one delta module is added, removed or modified in another delta module contained in the same set, and for every class modified in more than one delta module, its direct superclass is changed at most by one delta clause and the fields and methods added, modified or removed are distinct. For a set of compatible delta modules, any order of delta module application yields the same class table since the alterations in compatible delta modules do not interfere with each other.

A SPL is locally unambiguous if every set \( S \) of delta modules in the partition of \( dom(\delta) \) provided by the application partial order \( \delta_{DMT} \) is compatible. If the SPL L is locally unambiguous, then it is unambiguous. Local unambiguity can be checked by inspecting the delta modules in each partition only once.

A LPΔJ SPL is type-safe if the following conditions hold: (i) its product generation mapping is total, (ii) it is locally unambiguous, and (iii) all generated products are well-typed LJ programs.

4. Generalization of FOP
In this section, we show that Pure DOP is generalization of FOP [7] by providing a mapping from LFJ [13] into LPΔJ.

4.1 Recalling LFJ
The syntax of the LFJ extensions to LJ is given in Figure 3. It is taken from [13]. A feature module definition FMD contains the

\[
\begin{align*}
\text{FMD} &::= \text{feature \( \phi \) \( (\text{cd} \text{cd}) \)} \\
\text{rcd} &::= \text{refines class \( C \) extending \( \{ \text{cd} \text{md} \} \)} \\
\text{rmd} &::= \text{refines ms \( \{ \text{Super}() \} \{ \text{ms} \text{md} \} \)} \\
\text{ct} &::= \text{returns \( y \)}
\end{align*}
\]
feature ϕ and a set of class definitions cd and class refinement
definitions rcd. Class definitions are given according to the syntax
of LJ. A class refinement definition can change the superclass, add
fields fd, provide new method definitions md and refine existing
method definitions rmd. A method refinement can wrap the existing
method body using the Super() construct.

A feature module table FMT is a mapping from feature names
to feature module definitions. A LFJ product line can be described
by a 3-tuple L = (FMT, Φ, <form(FMT)), consisting of:

1. a feature module table FMT with a feature module for each
feature of the SPL,
2. the set of the valid feature configurations Φ ⊆ P(dom(FMT)),
3. a total order <form on the set of features dom(FMT).

The product associated to a feature configuration ψ is gener-
ated by composing (see Section 3.1 of [13]) the feature modules
associated to the features in ψ according to the total order <form.
During feature module composition, newly defined classes, fields
and methods are added and class and method refinements are car-
ried out. According to [13], a LFJ product line is type-safe if all
generated products are well-typed LJ programs.

4.2 Mapping LFJ into LPAJ

A product line in FOP can be represented as a product line in Pure
DOP. The set of features and the set of valid feature configurations
in both product lines is the same. Every feature module in a LFJ
product line is mapped to a delta module where additions are
translated to adds clauses and refinements to modifies clauses. The
application condition of the delta module denotes all configurations
in which the respective feature is contained. The ordering of delta
module application is the total ordering of the feature modules.

Formally, the mapping from LFJ product lines to LPAJ product
lines is defined as follows: for a LFJ product line L = (FMT, Φ, <form), [L] denotes the corresponding LPAJ product
line ([ψ], Φ, DMT, Γ, <form) where

- ψ = dom(FMT) = dom(DMT),
- The delta module table DMT is obtained by translating each
feature module in FMT to a delta module with the same name,
according to the clauses in Figure 4,
- Γ : dom(DMT) → Φ, where Γ(φ) = {ψ | ψ ∈ Φ and φ ∈ ψ},
- <form is the total order on \{ {φ} | φ ∈ ψ \} defined by:
{φ}_1 <form {φ}_2 if and only if {φ}_1 <form {φ}_2.

The following theorem states that the LPAJ product lines
generate the same products as the LFJ product line. Hence, Pure DOP
is a true generalization of FOP.

THEOREM 4.1. If L is a type safe LFJ product line, then [L] is
a type safe LPAJ product line such that, for every valid feature
configuration ψ, the product for ψ generated by L is the same as
the product for ψ generated by [L].

Although it is possible in principle to encode FOP in Core DOP,
a straightforward embedding as for Pure DOP is not possible. This

is because a feature-oriented SPL may have several base feature
modules, while Core DOP requires exactly one core module as
starting point for product generation.

5. Pure DOP for Product Line Development

Pure DOP supports proactive, extractive and reactive product line
development [22]. In the proactive approach, the scope of the prod-
uct line, i.e., the set of products to be developed, is analyzed before-
hand. All reusable artifacts are planned and developed in advance.
The example for Pure DOP presented in Section 2 can be seen as
proactive product line development, since we start from the feature
model defining the scope of the product line and develop delta mod-
ules and a Pure DOP SPL for these products. However, proactive
development requires a high upfront investment to define the scope
of the product line and to develop reusable artifacts.

Hence, in order to reduce the adoption barrier for product line
engineering, Krueger [22] proposes the usage of reactive and ex-
tractive approaches. In reactive product line engineering, only a ba-
sic set of products is developed. When new customer requirements
arise, the existing product line is evolved. The extractive approach
allows turning a set of existing legacy application into a product
line. Development starts with the existing products from which the
other products of the product line are derived.

FOP [7, 13] supports proactive product line development well.
However, since feature modules are restricted to add or refine
existing classes, FOP does not support extractive development and
only partially supports reactive development. It is not possible to
start from an existing legacy application comprising a larger set
of features and to remove features. Moreover, in order to deal with
new requirements following the reactive approach, feature modules
might have to be refactored to remove functionalities. Also, in Core
DOP, extractive product line development is not straightforward,
since one product has to be chosen as designated core product. In
contrast, Pure DOP is flexible and expressive enough to cover all
three product line engineering approaches directly.

5.1 Reactive Product Line Engineering

In reactive product line engineering, development starts with an
initial product line that is evolved in order to deal with changing
customer requirements. Consider as initial product line the example
depicted in Listing 5. Assume now that a new feature Sub should
be introduced for representing subtraction expressions. In the new
EPL the Sub feature should be an alternative to the Neg feature.
Additionally, the Print feature should become optional and the Eval
feature mandatory. The feature diagram for the evolved product line
is given in Figure 5.

In order to realize the new Sub feature, we have to add delta
modules that introduce the corresponding data structure for sub-
traction and the associated print and the evaluation functionalities.
The respective delta modules are shown in Listing 6. The specifi-
cation for the evolved SPL is shown in Listing 7, where the op-
erator \texttt{choose}(P_1, \ldots, P_n) means at most one of the propositions
P_1, \ldots, P_n is true (see [5]).
5.2 Extractive Product Line Engineering

Extractive product line engineering starts with a set of existing legacy applications from which the other products of the product line are generated. Assume that we have already developed a product containing the Lit, Neg and Print features and a product containing the Lit, Add and Print features. Now, we want to transform these existing legacy applications into a product line according to the feature model in Figure 1.

First, the existing applications have to be transformed into delta modules that are applied initially. Listing 8 shows two delta modules adding the implementation of the two existing products, respectively. Second, in order to provide product implementations with less features, delta modules have to be specified that remove functionality from the existing products. Listing 9 shows the delta module that removes the feature Add.

Listing 10 shows the extractive implementation of the product line described by the feature model in Figure 1 starting from a product with features Lit, Neg and Print and a product with features Lit, Add, and Print introduced by the delta modules DLitNegPrint and DLitAddPrint in the first and second partitions, respectively. Their application conditions are exclusive such that for any feature configuration product generation starts with one of them. If the Add feature is not selected and the Neg feature is selected, we start with the existing product in delta module DLitNegPrint. Otherwise, we start with the existing product in delta module DLitAddPrint. If both features Add and Neg are selected, we add the Neg feature by the delta modules DNeg and DNegPrint of Listing 4. If both the Add feature and the Neg feature are not selected, we remove the Add feature by the delta module DremAdd of Listing 9. Finally, we add the evaluation functionality if the feature Eval is selected.

This example shows that Pure DOP supports extractive product line engineering by introducing the existing products in initial delta modules, by delta modules removing functionality, and by specifying the product line to generate the products from the existing products by suitable delta module application.
delta DremPrintLit { 
  modifies interface Exp { removes print }
  modifies class Lit { removes print }
}
delta DremPrintAdd { 
  modifies class Add { removes print }
}
delta DremPrintNeg { 
  modifies class Neg { removes print }
}

Listing 11: Delta modules removing the Print feature

features Lit, Add, Neg, Sub, Print, Eval
configurations Lit & Eval & choose1(Neg,Sub)
deltas 
  [ DLitNegPrint when (Add & Neg),
    DSub when Sub ] 
  [ DLitAddPrint when (Add & Neg),
    DremAdd when (Add & Neg) ] 
  [ DNeg when (Add & Neg),
    DremAdd when (Add & Neg) ] 
  [ DNegPrint when (Add & Neg & Print),
    DremAdd when Print, DremAddPrint when (Print & Add),
    DremPrintNeg when (Print & Neg),
    DSub when (Sub & Print),
    DSubEval when Sub ]

Listing 12: Pure DOP specification of the evolved extractive EPL

5.3 Combining Extractive and Reactive PL Engineering

Extractive and reactive product line engineering can be combined. An initial product line is developed from a set of existing legacy applications and evolved when new requirements arise. Consider, the product line developed using the extractive approach in Listing 10. Assume, that now the Sub feature should be added and the product line should be changed to implement the feature diagram in Figure 5. Since in this product line, the feature Print is optional, we have to provide delta modules that remove the printing functionality from the Lit, Add and Neg classes. These delta modules are depicted in Listing 11.

Listing 12 shows the specification of the evolved product line depicted in Listing 10. The generation starts again from the two delta modules DLitNegPrint and DLitAddPrint introducing the existing products. Additionally, the product line contains delta modules for adding the Sub feature (cf. Listing 6) and delta modules for removing the Print feature (cf. Listing 11).

6. Comparison with Core DOP

In the traditional presentation of DOP [29], which we refer to as Core DOP, program generation always starts from a core module containing the implementation of a selected valid product of the product line. Then, delta modules specify the changes to the core module in order to implement the other products. Moreover, in the presentation of Core DOP given in [29]:

- the feature configuration corresponding to the product implemented by the core module is specified in the code of the core module;
- the application condition of a delta module is specified in the code of the delta module by a clause of the form “when $\gamma$”, where $\gamma$ is a propositional constraint specifying the feature configurations in which the delta module has to be applied, and
- the application partial order for the delta modules is specified in the code of the delta modules using a clause of the form “after $\delta$”, which specifies that the delta module must be applied after all applicable delta modules in $\delta$ have been applied.

Pure DOP and Core DOP are indeed equivalent:

- A Pure DOP product line can be expressed as a Core DOP product line by adding an empty product to the product line and choosing it as the product implemented by the core module.
- A Core DOP product line can be expressed as a Pure DOP product line by transforming the core module into a delta module that has to be applied before any other delta module for all the valid feature configurations.

Pure DOP is a conceptual simplification of Core DOP dropping the notion of the core module and separating the specification of the application conditions and of the application ordering from the delta modules. This presents the following advantages:

- Pure DOP allows reusing delta modules for implementing different product lines (cf. Sections 2 and 5).
- Every delta module in Pure DOP containing only adds clauses can play the role of the core module. Thus, product lines with multiple base modules, that may not correspond to valid products, are possible. As a consequence, Pure DOP is a true generalization of FOP (cf. Section 4).
- Pure DOP supports the evolution of product lines. If a product line evolves such that the core product of a Core DOP product line is no longer a valid product, the core module and potentially all delta modules have to be refactored. In contrast, in pure DOP, existing delta modules can be reused for the specification of the evolved product line (cf. Section 5).

7. Related Work

The notion of program deltas is introduced in [23] to describe the modifications of object-oriented programs. In [30], delta-oriented modeling is used to develop product line artifacts suitable for automated product derivation and implemented with frame technology [36]. This approach is extended in [28] to a seamless delta-oriented model-based development approach for SPLs. In [11], an algebraic representation of delta-oriented product lines is presented. The main focus in [11] is to reason about conflicting modifications and to devise a general criterion to guarantee the unambiguity of product lines using conflict-resolving deltas. The unambiguity property presented in this work is an instance of the criterion presented in [11], but it is more restrictive since it requires to order all potential conflicts. Delta modules are one possibility to implement arrows in the category-theoretical framework for program generation proposed by Batory in [6].

Feature-oriented programming (FOP) [2, 7, 13, 34], Core DOP [29] and Pure DOP are compositional approaches [19] for implementing SPLs. For a detailed comparison between FOP and Core DOP, the reader is referred to [29]. Other compositional approaches used to implement product lines rely on aspects [18], framed aspects [24], combinations of feature modules and aspects [3, 25], mixins [31], hyperslices [33] or traits [8, 14]. In [23], several of these modern program modularization techniques are compared with respect to their ability to represent feature-based variability. Furthermore, the modularity concepts of recent languages, such as Scala [26] or Newspeak [10], can be used to represent product features.
In [1], an approach is presented that combines reactive and extractive product line engineering [22] based on aspect-oriented programming refactorings. The modification operations that can be specified in delta modules are sufficient to express before, after and around advice considered in aspect-oriented programming [21]. Delta modules do not comprise a specification formalism for modifications to be carried out at several places of a program (such as pointcuts), such that all program modifications have to be explicitly specified. Adding a pointcut-specification technique to delta modules would allow encoding AOP by DOP, which is a subject of future work. However, delta modules are more flexible than aspects by their ability to remove functionality, such that a program refactoring is not required to evolve a product line when functionality has to be removed.

8. Conclusions and Future Work

In this paper, we have proposed pure delta-oriented programming (Pure DOP) as a conceptual simplification of Core DOP [29]. An implementation of the Pure DOP programming language presented in this paper and a core calculus with a constraint-based type system are currently being developed. Following the conceptual comparison of FOP, Core DOP and Pure DOP in this paper, we are evaluating Pure DOP empirically at larger case examples and investigating the extraction of delta modules from version histories.

The concept of Pure DOP is not bound to a particular programming language. In this work, we have instantiated it for LFJ. For future work, we are aiming to use other languages for the underlying product implementations. A starting point is the trait-based calculus FEATHERWEIGHT RECORD-TRAIT JAVA (FRTJ) [8, 9]. In FRTJ, classes are assembled from interfaces, records (providing fields) and traits [14] (providing methods) that can be directly manipulated by designated composition operations. These operations make FRTJ a good candidate for implementing delta modules in an expressive way.

References


[14] S. Ducasse, O. Nierstrasz, N. Schäfl, R. Wuys, and A. Black. Traits: A mechanism for fine-grained reuse. ACM TOPLAS, 28(2):331–388, 2006.


Appendix D

Towards a Theory of Views for Feature Models

Towards a Theory of Views for Feature Models

Dave Clarke, José Proença
IBBT-DistriNet, Dept. Computer Sciences
Katholieke Universiteit Leuven, Belgium
Email: {dave.clarke,jose.proenca}@cs.kuleuven.be

Abstract—Variability in a Software Product Line (SPL) is expressed in terms of a feature model. As software development efforts involve increasingly larger feature models, scalable techniques are required to manage their complexity. Furthermore, as many stakeholders have a vested interest in different aspects of a feature model, modularity techniques are required to independently express their views of the product line abstracting away from unnecessary details. To address these issues of scalability and modularity this paper introduces a theory of views for features models, encompassing both view compatibility and view reconciliation for plugging together different views of a product line.

Keywords—software product lines; variability; formal methods

I. INTRODUCTION

Variability in a Software Product Lines (SPL) is expressed in terms of a feature model. Feature models are usually depicted using feature diagrams [1, 2], and a variety of increasingly expressive notations have been presented [3, 4, 5, 6, 7, 8, 9, 10]. The semantics of feature diagrams are generally understood in terms of sets of (multi)sets of features [11, 12], where at this level of abstraction a feature is simply a name. As software development efforts involve increasingly larger feature models, scalable techniques are required to manage their complexity, and modularity techniques are required to cater for the different interests of the various stakeholders involved in the project.

One approach to both managing complexity and increasing modularity is using views. A view of a feature model shows the decomposition of a concept up to a certain level of detail. With a view, only part of a feature model is visible to each stakeholder, possibly at with some details abstracted away. For example, one stakeholder may be interested in the user visible functionality, and thus does not need to see the detailed features required by programmers in order to to implement that functionality. Programmers in one team need not see the complete variability of the product line from the perspective of other teams.

A theory of views for features models is introduced, along with notions of view compatibility and view reconciliation, to address issues of scalability and modularity. Our work is presented quite generally, using notions from category theory, to ensure the future applicability of our results to more detailed models of software product lines. Our contributions include: a notion of view for feature models; criteria for determining whether views are compatible for a given SPL; and techniques for reconciling compatible views.

The paper is organised as follows. §II provides some motivating examples. §III introduces some preliminary concepts. §IV defines abstractions and views of feature models. §V presents our key definitions and results: view compatibility and view reconciliation and their characterisation. §VI takes a different angle, revealing some limitations in our approach. §VII applies our approach to express two notions of refinement of feature models. §VIII discusses related work. §IX presents our conclusions and future work.

II. MOTIVATING EXAMPLES

We motivate our work with a collection of small examples, where features are simple consumer items:

\[
\{tv, book, iPod, newspaper, cd\}. \tag{1}
\]

In this paper we investigate on how to combine views on product lines. A view consists of a selection of relevant features combined with the abstraction of some other features into more general features. The term reconciliation refers to this combination of views, and we use a simple example to guide the intuition behind the reconciliation process. We start by considering views over the two sets:

\[
F = \{\text{Electronic, book, newspaper}\} \tag{2}
\]

\[
G = \{tv, iPod, Paper\}. \tag{3}
\]

In F we abstract tv and iPod as Electronic, and in G we abstract book and newspaper as Paper. We do not use cd yet. As a convention, we start the name of a feature with an uppercase letter to denote it is an abstraction. A product is a set of features, and a product line is a set of possible products.

Notation. For simplicity, we use only the first letter of each feature using true type font, and denote products by the consecutive letters of its features. For example, tiP denotes the product \{tv, iPod, Paper\}. 

This research is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Methods (http://www.hats-project.eu), and the K.U.Leuven BOF-START project STRT1/09/031 DesignerTypeLab.
Example 1 (Success story). We reconcile the following products lines, over \( F \) and \( G \), respectively.

\[
P = \{E, E_b, E_{bn}\} \quad Q = \{t, t_i P\}.
\]

The reconciliation of two products or product lines \( x \) and \( y \) is denoted by \( x \oplus y \). \( P \) and \( Q \) can be reconciled only when they are compatible. Two features \( a \) and \( b \) are compatible, written \( a \prec b \), if one abstracts the other feature. In our example, \( t \prec E \) but not \( t \prec P \) or \( P \prec E \). Two products are compatible if every feature from each product has a compatible feature in the other product. In our case, \( t_i P \prec E_b \) but not \( t_i P \prec E \). Finally, two product lines are compatible if every product from each product line has a compatible product in the other product line.

Our two product lines \( P \) and \( Q \) are compatible, and the reconciliation \( \oplus \) joins every possible combination of compatible products. The reconciliation of these product lines yield:

\[
P \oplus Q = \{E \oplus t, E_b \oplus t_i P, E_{bn} \oplus t_i P\} = \{t, t_i P, t_i b\}.
\]

Example 2 (Partiality). We now include also the \( cd \) feature, which is neither refined or abstracted by any other feature.

\[
G_2 = G \cup \{cd\} \quad Q_2 = \{t, c t_i P\}
\]

When calculating \( P \oplus Q_2 \), the compatibility relation ignores the \( cd \) feature, while the reconciliation operator preserves it. In our example, \( c t_i P \prec E_b \) and \( c t_i P \oplus E_b = c t_i b \). Thus, \( P \prec Q_2 \) and \( P \oplus Q_2 = \{t, c t_i b, c t_i b\} \).

Example 3 (Incompatibility). We now extend \( Q \) with \( \text{Paper} \).

\[
Q_3 = \{t, t_i P, P\}
\]

In this example, \( P \not\prec Q_3 \), because there is no product \( p \in P \) for which \( P \prec \text{Paper} \).

III. PRELIMINARIES

We start by introducing basic mathematical models of concepts such as features, product lines, software product lines, following Schobbens et al. [13], along with other relevant formal definitions.

A. Features, Products, and Software Product Lines

A feature is a unit of variability in a software product line relevant to some stakeholder [14]. From a formal perspective, features are represented as elements of a set. Products and product lines are defined as a set of features and as a set of products, respectively. Modelling from the level of abstraction of features, we thus equate a product with a feature configuration and a product line with a feature model.

Typically, \( F, F', G, G', \ldots \) range over sets of features, \( a, b, \ldots \) over features, \( p, q, \ldots \) over products, and \( P, Q, \ldots \) over product lines. The notation \( P(F) \) denotes the power set of \( F \), namely, the set of subsets of \( F \).

Definition 4 (Product). Given a set of features \( F \). A product is defined as a subset of \( F \). The set of all products for features \( F \) is denoted \( \mathcal{P}(F) \) and is defined

\[
\mathcal{P}(F) \equiv \mathcal{P}(P(F)).
\]

Definition 5 (Product Line). Given a set of features \( F \). A product line is defined as a set of products for \( F \), that is, a subset of \( \mathcal{P}(F) \). The set of all product lines for features \( F \) is denoted \( \mathcal{F} \) and defined

\[
\mathcal{F} \equiv \mathcal{P}(\mathcal{P}(F)).
\]

B. Function Lifting

Our results will be phrased using some basic category theory [15]; we will be working in the categories \text{Set} \ of sets and functions, and \text{PFun} \ of sets and partial functions.

Given a partial function \( f : F \rightarrow G \), the domain of \( f \) is \( \text{dom}(f) = F \), the codomain is \( \text{cod}(f) = G \), the pre-image is \( \text{def}(f) = \{a \in F \mid f(a) \neq \bot\} \), and the range is \( \text{rng}(f) = \{f(a) \mid a \in \text{def}(f)\} \). A function \( f : F \rightarrow G \) is total if \( \text{dom}(f) = \text{def}(f) \), and is onto if \( \text{cod}(f) = \text{rng}(f) \). The composition of two partial functions \( u \circ v \) is defined as:

\[
(u \circ v)(a) \equiv \begin{cases} \bot, & \text{if } v(a) = \bot \\ u(v(a)), & \text{otherwise} \end{cases}
\]

A partial function \( f : F \rightarrow G \) from one feature set to another can be lifted to functions over products and product lines, written as \( \hat{f} \) and \( f \), respectively.

Definition 6 (Lifting). Given a partial function \( f : F \rightarrow G \), we define the lifting of \( f \) to products and product lines as \( \hat{f} \) and \( f \), respectively, defined as follows.

\[
\hat{f} : \mathcal{F} \rightarrow \mathcal{F} \quad \hat{f}(p) \equiv \{f(a) \mid a \in p \cap \text{def}(f)\}
\]

\[
f : \mathcal{F} \rightarrow \mathcal{F} \quad f(P) \equiv \{f(p) \mid p \in P\}.
\]

Another useful notion is the restriction of a product or product line to a smaller set of features. Restriction is one of the ingredients of our definition of view.

Definition 7 (Restriction). Let \( G \subseteq F \), \( p \in \mathcal{P} \), and \( P \in \mathcal{F} \). Define the restriction of \( p \) to \( G \), and the restriction of \( P \) to \( G \), denoted as \( p \mid G \) and \( P \mid G \), respectively, as follows:

\[
p \mid G \equiv p \cap G
\]

\[
P \mid G \equiv \{p \mid G \mid p \in P\}.
\]
IV. ABSTRACTIONS AND VIEWS

Abstraction removes distinctions between features, e.g., mapping all different varieties of television to a single feature TV. Views are then defined on top of notion of abstraction, enabling not only the merging of distinctions, but also the hiding of features. Both concepts can be expressed for features and lifted to products and product lines.

Definition 8 (Abstraction). Any onto function \( f : F \rightarrow G \) defines an abstraction \( \hat{f} : F \rightarrow \hat{G} \). A product line \( Q \) is defined as an abstraction of product line \( P : \hat{F} \) if an \( f : F \rightarrow G \) exists such that \( \hat{f}(P) = Q \). Similarly, an abstraction of products for such an \( f : F \rightarrow G \) is defined as \( \hat{f} : F \rightarrow \hat{G} \).

Define a view function as partial abstraction.

Definition 9 (View). For any partial and onto function \( g : F \rightarrow \hat{G} \), we say that \( g \) is a feature view, \( \overline{g} \) is a product view, and \( \overline{g} \) is a product line view. Furthermore, for any \( P \in \hat{F} \) we say that \( \overline{g}(P) \in \overline{G} \) is a view on \( \overline{G} \). Similarly, a view of products for such an \( g : F \rightarrow G \) is defined as \( \overline{g} : F \rightarrow \overline{G} \).

Example 10. Consider a product line with features \( F = \{ \text{sonyTV}, \text{philipsTV}, \text{sonyDVD}, \text{philipsDVD} \} \). The following function \( v : F \rightarrow G \), where \( G = \{ \text{TV}, \text{sonyDVD}, \text{philipsDVD} \} \), defines a view for a stakeholder interested only in the DVD variability and whether a TV is present or not:

\[
\begin{align*}
\text{sonyTV} & \mapsto \text{TV} \\
\text{sonyDVD} & \mapsto \text{sonyDVD} \\
\text{philipsDVD} & \mapsto \text{philipsDVD}.
\end{align*}
\]

V. VIEW COMPATIBILITY AND RECONCILIATION

We now consider the following problem. Two views of a feature model are developed independently by two stakeholders. No globally definitive feature model exists or perhaps it has become out-dated. The stakeholders want to check whether their views are compatible and if so they want to see how the combined or reconciled view looks.

Assume the stakeholders are working with feature sets \( F \) and \( G \). If the views are compatible, then it will be the case that some of the features in \( F \) are abstractions of features in \( G \), and vice versa. It may also be that some features in \( F \) do not appear in \( G \), and vice versa. This means that we can partition \( F \) and \( G \) each into two parts and construct a view between pairs of partitions, one going in each direction. The desired structure is called a view partition and is illustrated below. Here \( F_r \) and \( F_a \) are a partition of \( F \). Similarly for \( G \). The functions \( v \) and \( w \) are the view functions, mapping more refined elements in \( F_r \) to their abstraction in \( G_a \), and from \( G_r \) to \( F_a \), respectively.1 We depict a view partition below, where \( v \) abstract from \( F_r \) to \( G_a \), and \( w \) from \( G_r \) to \( F_a \).

1Subscripts \( r \) and \( a \) indicate the source set (more refined features) and target set (more abstract features) of view functions of a view partition.

\[
\begin{array}{c}
\sigma = \begin{array}{c}
\text{sonyTV} \mapsto \text{TV} \\
\text{sonyDVD} \mapsto \text{sonyDVD} \\
\text{philipsDVD} \mapsto \text{philipsDVD}.
\end{array}
\end{array}
\]

Definition 11 (View partition). Define a view partition to be a tuple \( \sigma = (F_r, F_a, G_r, G_a, v, w) \), where \( F_r, F_a, G_r, G_a \) are sets of features such that \( F = F_r \cup F_a \), \( G = G_r \cup G_a \), \( F_r \cap F_a = \emptyset = G_r \cap G_a \), and \( v : F_r \rightarrow G_a \) and \( w : F_a \rightarrow G_r \) are partial onto functions.

The set \( F_r + G_r \) consists of the most refined features of \( F \) and \( G \) (+ is categorical coproduct, i.e., disjoint union). Reconciliation of two views will produce a product line in terms of \( F_r + G_r \). On the other hand, \( F_a + G_a \) is the set of the most abstract features. These are used for checking the compatibility of two views.

Define \( v_r : F_r + G_r \rightarrow G_a \) and \( v_a : F_a \rightarrow F_a + G_a \) to be the partial onto functions that extend \( v \) with the identity for elements in \( G_r \) and \( F_a \), respectively. That is, for example,

\[
v_r(x) \overset{\text{def}}{=} \begin{cases} v(x), & \text{if } x \in F_r, \\ x, & \text{otherwise.} \end{cases}
\]

Define \( w_r \) and \( w_a \) analogously, swapping \( F \) and \( G \).

These functions play an important role in what follows. For example, function \( v_r : F_r + G_r \rightarrow G_a \) presents the view of the reconciled product line in terms of \( G \). Function \( v_a : F_a \rightarrow F_a + G_a \) recasts a view based on features \( F \) in terms of the most abstract features, namely \( F_a + G_a \).

A. View Compatibility

Given a view partition, the following definition captures what it means for the underlying views to be compatible for features, products and product lines.

Definition 12 (Compatibility). For any \( a \in F \), \( p \in \hat{F} \), \( b \in \hat{G} \), \( q \in G \), and \( Q \in \hat{G} \), and for a view partition \( \sigma = (F_r, F_a, G_r, G_a, v, w) \), define compatibility of features, products, and product lines, denoted by the binary operator \( \rangle \), as follows:

\[
\begin{align*}
\hat{a} & \rangle b \overset{\text{def}}{=} v_r(a) = w_a(b) \\
\hat{p} & \rangle q \overset{\text{def}}{=} w_r(p) = w_a(q) \\
\hat{P} & \rangle Q \overset{\text{def}}{=} \overline{v}_r(P) = \overline{w}_a(Q).
\end{align*}
\]

Observe that \( v_r(a) = w_a(b) \) is equivalent to the more intuitive condition \( v(a) = b \) if \( q \) is a compatible feature in \( q \), and vice versa. The states that \( b \) is an abstract feature corresponding to \( a \), or vice versa, or both \( a \) and \( b \) only appear in \( F_r \) and \( G_r \), respectively. Now, \( p \rangle q \) states that every feature in \( p \) has a compatible feature in \( q \), and vice versa. Similarly, \( P \rangle Q \) states that every product in \( P \) has a compatible product in \( Q \), and vice versa.

The key role of compatibility is that it provides the conditions for ensuring that two views can be reconciled.
B. Interlude: Pullbacks

The key machinery to get reconciliation to work is the categorical notion of a pullback \[15\]. A pullback in the category \(\text{Set}\) of a pair of functions \(f : A \to C\) and \(g : B \to C\) is the set \(A \times_C B = \{(a, b) \in A \times B \mid f(a) = g(b)\}\), in other words, the pairs of elements \(a \in A\) and \(b \in B\) that agree (via \(f\) and \(g\)) in \(C\). This is the canonical definition of pullback—any isomorphic set can also be the basis of a pullback. The key property of a pullback is that if there are functions \(h\) and \(k\) that make the outer ‘square’ of the diagram below commute (that is, \(k \circ f = h \circ g\)), then there exists a unique function \(\sigma\) that makes the whole diagram commute. This captures that \(A \times_C B\) is the most detailed of such objects.

A weak pullback drops the uniqueness criteria.

Although pullbacks always exist in \(\text{Set}\), we need to select specific objects as the locus of reconciliation, namely, \(F_r + G_r\) and \(F_a + G_a\), and cannot always work with pullbacks. \(\text{Theorem 13}\) describes when these objects are pullbacks (of the appropriate functions) and when we can only work with a weak pullback.

**Theorem 13.** Let \(\sigma = (F_r, F_a, G_r, G_a, v, w)\) be a view partition. The first diagram is a pullback, and the second is a weak pullback.

![Diagram](attachment:image.png)

From now on we drop references to the view partition \(\sigma\). When \(a\) and \(b\) are compatible, and at least one is in the domain of \(v\) or \(w\), then \(a\) refines \(b\) or vice-versa. The reconciliation chooses the most refined feature. The split into these three cases is justified by the following lemma.

**Lemma 15.** Let \(a \in F\) and \(b \in G\), such that \(a \in \text{def}(v)\) or \(b \in \text{def}(w)\). If \(a \prec b\) then \(a \in F_r \Leftrightarrow b \notin G_r\).

D. Product Reconciliation

Reconciliation for features can be lifted to products, by selecting the most refined features and discarding the rest.

**Definition 16** (Product reconciliation). Given a view partition \(\sigma = (F_r, F_a, G_r, G_a, v, w)\), and products \(p \in F\) and \(q \in G\) such that \(p \prec q\), define:

\[ p \oplus q \defeq (p \upharpoonright F_r) \cup (q \upharpoonright G_r). \]

The following is used to prove our characterisation property.

**Lemma 17.** \(p \prec q \Rightarrow \overline{w}(p \oplus q) = p\) and \(\overline{v}(p \oplus q) = q\).

The following corollary is justified by \(\text{Lemma 17}\) and from the fact that \(F_r + G_r\) is the pullback. This corollary characterises product reconciliation (\(\text{Definition 16}\)). The ‘outer square’ corresponds precisely to the compatibility condition (\(\text{Definition 12}\)). The fact that the inner square is a pullback guarantees the uniqueness of the function \(1 \to F_r + G_r\), which can be shown to be \(p \oplus q\), the uniquely selected element of \(F_r + G_r\).

**Corollary 18.** Given view partition \(\sigma = (F_r, F_a, G_r, G_a, v, w)\). If \(p \prec q\), then \(p \oplus q\) is the unique morphism making the following diagram commute:

\[ \begin{array}{ccc}
1 & \oplus & q \\
\downarrow & & \downarrow \\
F_r + G_r & \to & F \\
\downarrow & & \downarrow \\
G & \oplus & F_a + G_a \\
\end{array} \]

E. Product Line Reconciliation

We now lift the previous results for products and define the reconciliation of product lines as follows.

**Definition 19** (Product line reconciliation). Given a view partition \(\sigma = (F_r, F_a, G_r, G_a, v, w)\), \(P \in F\), and \(Q \in G\), define:

\[ P \oplus Q \defeq \{ p \oplus q \mid p \in P, q \in Q, p \prec q \}. \]

This operation takes all compatible pairs of products from each product line and combines them with the product...
reconciliation operation. The imposition that \( P \preceq Q \) is a sufficient condition to guarantee that, after reconciling \( P \) and \( Q \), the original values of \( P \) and \( Q \) can still be recovered, as stated by the following lemma.

**Lemma 20.** \( P \preceq Q \Rightarrow \widehat{w}_r(P \oplus Q) = P \) and \( \widehat{v}_r(P \oplus Q) = Q \).

Furthermore, when the above condition does not hold, it is not possible in general to recover the values of \( P \) and \( Q \) from \( P \oplus Q \), meaning that the \( P \) and \( Q \) would not be views on \( P \oplus Q \). The following corollary is justified by Lemma 20 and that \( F_s \oplus G_s \) is a weak pullback.

**Corollary 21.** Given view partition \( \sigma = (F_r, F_a, G_r, G_a, v, w) \). If \( P \preceq Q \), then \( P \oplus Q \) makes the diagram commute:

\[
\begin{array}{c}
P \preceq Q \Rightarrow \widehat{w}_r(P \oplus Q) = P \text{ and } \widehat{v}_r(P \oplus Q) = Q.
\end{array}
\]

This construction is somewhat unsatisfying, as it does not say conclusively that operation for product line view reconciliation is the best we can hope for. Before presenting an alternative characterisation, we take a step back to consider how our constructions look for views of a feature, product or product line, based on some given view partition.

Define the following functions:

\[
c : F_r + G_r \rightarrow F_r + G_r \quad a \mapsto w_r(a) \oplus v_r(a) \quad (10)
\]

\[
\tau : F_r + G_r \rightarrow F_r + G_r \quad p \mapsto w_r(p) \oplus v_r(p) \quad (11)
\]

\[
\hat{\sigma} : F_r + G_r \rightarrow F_r + G_r \quad P \mapsto \widehat{w}_r(P) \oplus \widehat{v}_r(P) \quad (12)
\]

The following state the relationship between a feature, product or product line and the views on it for a given view partition. Firstly, the views are compatible and secondly, reconciliation works on the nose for features and products but provides an over approximation for product lines.

**Lemma 22.** Given \( a \in F_r + G_r, p \in F_r + G_r \) and \( P \in F_r + G_r \). Then:

1) \( w_r(a) \preceq v_r(a) \)

2) \( w_r(p) \preceq v_r(p) \)

3) \( \widehat{w}_r(P) \preceq \widehat{v}_r(P) \)

The following properties capture the nature of \( \hat{\sigma} \), and thus provide some intuition about product lines reconciliation.

**Lemma 23.** \( \hat{\sigma} \) is a closure operator.\(^2\)

The following definition captures the equivalence class of product lines that have the same views in \( \hat{F} \) and in \( \hat{G} \).

**Definition 24.** Define equivalence \( \sim \subseteq F_r \times G_r \times F_r \times G_r \) as \( P \sim Q \iff \widehat{w}_r(P) = \widehat{w}_r(Q) \land \widehat{v}_r(P) = \widehat{v}_r(Q) \).

The following lemma characterises \( \hat{\sigma} \) as selecting the maximum of all possible equivalent product lines with the same views in \( \hat{F} \) and \( \hat{G} \).

**Lemma 25.** \( \hat{\sigma}(P) \) is the maximum (wrt set inclusion) of \( [P] \), the equivalence class containing \( P \).

**VI. CONSTRUCTING A VIEW PARTITION**

In the previous section, we assumed a view partition was given, though not necessarily the product line. Now we start from the other direction and address whether it is possible to find an appropriate view partition given a product line and two views. Let \( P \in \hat{E} \) be a product line, and \( \hat{f} : \hat{F} \rightarrow \hat{G} \) and \( \hat{g} : \hat{E} \rightarrow \hat{G} \) define two views in \( \hat{F} \) and \( \hat{G} \). The following condition guarantees the existence of a view partition.

**Definition 26** (View compatibility). Two view functions \( f : E \rightarrow F \) and \( g : E \rightarrow G \) are compatible, denoted \( f \sim g \), if for every \( a \in F \) and \( b \in G \)

\[
f^{-1}(a) \neq g^{-1}(b),
\]

where \( X \neq Y \) iff \( X \cap Y = \emptyset, X \subseteq Y, \) or \( Y \subseteq X \).

Intuitively, Definition 26 states that \( f \) and \( g \) treat the features in \( E \) in terms of compatible abstractions. That is, two feature sets \( E_a = f^{-1}(a) \) and \( E_b = g^{-1}(b) \) are either incompatible, that is, \( f^{-1}(a) \cap g^{-1}(b) = \emptyset \), or that one is an abstraction of the other \( f^{-1}(a) \subseteq g^{-1}(b) \), or vice versa.

The following theorem states that when two views functions are compatible, then a view partition exists to make the two views of the original product line compatible.

**Theorem 27.** Let \( \hat{f} : \hat{E} \rightarrow \hat{F} \) and \( \hat{g} : \hat{E} \rightarrow \hat{G} \) be two views. If \( f \sim g \), then there exists a view partition \( \sigma \) such that \( \forall P \in \hat{E} : \hat{f}(P) \sim \hat{g}(P) \). Specifically, the following gives such a view partition \( \sigma = (F_r, F_a, G_r, G_a, v, w) \), where \( R = g \circ f^{-1} \):

\[
\begin{align*}
F_a & \equiv \{ x \in F \mid |R(x)| > 1 \} & F_r & \equiv F \setminus F_a \\
G_a & \equiv R(F_r) & G_r & \equiv G \setminus G_a \\
v : F_r \rightarrow G_a & \equiv \{ R(x) \mid x \in F_r \} & w : G_r \rightarrow F_a & \equiv \{ R^{-1}(x) \mid x \in G_r \}.
\end{align*}
\]

The relationship between the original product line \( P \in \hat{E} \) and the reconciled views \( \hat{\sigma}(P) \) has already been described in Lemmas 22, 23 and 25.

Ultimately, one would expect all pairs of views to be compatible. That they are not points to a weakness in our theory. The problem stems from the fact that the two incompatible views have different ‘vocabularies’.

**Example 28** (Overlapping views). Given features \{te, book, iPod, newspaper, cd\}. Consider view functions:

\[
f = \{ cd, book \mapsto Data ; iPod \mapsto Electronic \}
\]

\[
g = \{ cd, iPod \mapsto Music ; book \mapsto Paper \}
\]
Observe that \( f \triangleleft g \), because \( f^{-1}(\text{Data}) \not\# g^{-1}(\text{Music}) \) does not hold. Consequently, there is no view partition guaranteeing the compatibility of product lines obtained by \( f \) and \( g \). The cause is the following conflict:

\[
\begin{array}{ccc}
\text{Data} & \xrightarrow{f} & \text{Paper} \\
\text{Electronic} & \xrightarrow{g^{-1}} & \text{iPod}
\end{array}
\]

\( \text{Data} \) is neither an abstraction of \( \text{Music} \) nor vice versa, thus the calculations in Theorem 27 fail to yield a view partition.

VII. APPLICATIONS

Two applications demonstrate the expressiveness of our approach. Although the key ingredient is an abstraction function, view reconciliation can express notions of refinement:

- **Uniform refinement**: replace a feature by a collection of products.
- **Refinement in context**: replaces a feature \( a \) by a collection of products based on the other features occurring with \( a \) in a product.

These could allow feature models to be developed in a more scalable and modular fashion.

A. Uniform refinement

First we describe this operation based on sets. We will define only a simple variant which refines only one feature, and conjecture that it generalises in a straightforward fashion.

Uniform refinement of a feature \( f \) by a set of products \( Q \), consists of taking each product containing \( f \) and replacing it by product \( p \), for each \( p \in Q \). In feature diagrams this corresponds to replacing a leaf feature \( f \) by a whole feature diagram whose underlying feature model is \( Q \).

**Definition 29 (Uniform Refinement).** Given product line \( P \in \bar{F} \), \( f \in F \), and \( Q \in \bar{G} \), where \( F \cap G = \emptyset \). Define:

\[
\text{UniR}(P,f,Q) \overset{\text{def}}{=} \{ p[f \mapsto q] \mid p \in P, q \in Q \}.
\]

\[
p[f \mapsto q] \overset{\text{def}}{=} \begin{cases} p \setminus \{ f \} \cup q, & \text{if } f \in p \\ p, & \text{otherwise.} \end{cases}
\]

The resulting refinement is a product line over features \( F \setminus \{ f \} \cup G \). This refinement can be achieved when \( \emptyset \not\in Q \) using reconciliation, as follows.

\[
\text{UniR}'(P,f,Q) \overset{\text{def}}{=} P \oplus_\sigma Q_0
\]

\[
Q_0 = Q \cup \{ \emptyset \}
\]

\[
\sigma = (F_r,F_a,G_r,G_a,v,w)
\]

where \( F_r = F \setminus \{ f \} \)

\[
F_a = \{ f \}
\]

\[
G_r = \bigcup_{q \in Q} q
\]

\[
G_a = \emptyset
\]

\[
v = \emptyset
\]

\[
w = \{ g \mapsto f \mid g \in G_r \}.
\]

The view partition \( \sigma \) is defined such that the to-be-replaced feature \( f \) is in \( F_a \) as the abstraction of the features in \( Q \). The remaining features from \( F \) are kept in \( F_r \) and are not abstracted by \( v \). Consequently, the features in \( F_r \) are preserved during the reconciliation. Correctness is given by the following lemma.

**Lemma 30.** \( \text{UniR}(P,f,Q) = \text{UniR}'(P,f,Q) \).

**Example 31.** Consider the following product line, where we write \( abc \) to represent the product \( \{a,b,c\} \):

\[ Pwd = \{ c, cd, s \} \]

\( Pwd \) describes the features of a password field. A password can have characters (feature \( c \)), digits (feature \( d \)), and symbols (\( s \)), according to the combinations in \( Pwd \). We apply the uniform refinement

\[ (c, \{ u, l \}) \]

which refines characters as lowercase and uppercase letters (features \( l \) and \( u \), respectively), and requires uppercase letters. The view partition used to calculate this refinement is:

\[ \begin{array}{c}
\{ d, s \} \\
\{ c \} \\
\emptyset
\end{array} \]

\[ v = \emptyset \]

Finally, the refinement is calculated as follows.

\[
\text{UniR}(Pwd, c, \{ l, u, lu \})
\]

\[
= \{ c, cd, s \} \oplus \{ u, lu, \emptyset \}
\]

\[
= \{ c \oplus u, c \oplus lu, cd \oplus u, cd \oplus lu, s \oplus \emptyset \}
\]

\[
= \{ u, lu, du, dlu, s \}.
\]

B. Refinement in Context

We now describe a more complex notion of refinement, where a feature \( f \) is replaced by a set of possible products depending on the context, namely the features in the same product as \( f \), in which \( f \) appears. Refinement in context is given by an operation \( \text{RiC}(P,f,k) \), where \( k : \bar{G} \rightarrow \bar{G} \), can be understood as taking the product line \( P \) and refining it by replacing \( f \) by the product \( k(c) \), whenever \( f \) appears together with context \( c \subseteq \bar{G} \).

**Definition 32 (Refinement in Context).** Given a product line \( P \in \bar{F} \), define refinement in context wrt \( f \in F \) and \( k : \bar{G} \rightarrow \bar{G} \) by:

\[
\text{RiC}(P,f,k) = \{ p[f \mapsto q] \mid p \in P, q \in k(p \cap C) \}
\]

where \( f \in F \), \( f \not\in C \subseteq F \), and \( F \cap G = \emptyset \).

We encode the same refinement as a reconciliation as follows, for the cases where \( \emptyset \not\in \text{rng}(k) \):

\[
\text{RiC}'(P,f,k) \overset{\text{def}}{=} P \oplus_\sigma Q_k
\]

\[
Q_k = \{ c \cup g \mid (c,g) \in k \} \cup \bar{G}
\]

\[
\sigma = (F_r,F_a,G_r,G_a,v,w)
\]
where \( F_r = F \setminus F_a \), \( G_r = G \cup C \), 
\( F_a = \{ f \} \cup C \), \( G_a = \emptyset \), 
\( v = \emptyset \), \( w = \{ g \mapsto f \mid g \in G \} \cup \{ c \mapsto c \mid c \in C \} \).

Correctness is given by the following lemma:

**Lemma 33.** \( R t C(P, f, k) = R t C'(P, f, k) \).

**Example 34.** Recall the password scenario from Example 31.

\[
P w d = \{ c, d, s \}
\]
where \( P w d \in F \), and \( F = \{ c, d, s \} \). We now apply a refinement that replaces the characters feature \( c \) by lowercase \( l \) or uppercase \( u \), depending on the presence of the digits feature \( d \). Define \( G = \{ l, u, d \} \) and \( C = \{ d \} \) to be the target features and the context features, respectively. We apply the refinement in context based on \( k \) defined as \( k(\emptyset) = \{ l, u \} \), and \( k(d) = \{ l, u, l, u \} \). Here \( k \) represents how the context influences the replacement of \( f \) in any context: \( f \) can be replaced by the product \( l, u \), and when \( d \) is present \( f \) can also be replaced by \( l \) or \( u \). That is, when there are no digits in a password there must be both lowercase and uppercase letters. The view partition used to calculate the refinement is depicted below.

\[
\begin{align*}
\begin{array}{c}
F \\
\{ s \} \\
\{ c, s \} \\
\end{array} \\
\begin{array}{c}
v = \emptyset \\
\{ l \mapsto c, u \mapsto c, d \mapsto d \} \\
\emptyset \\
\end{array} \\
\begin{array}{c}
G \\
\{ l, u, d \} \\
\{ l, u, l, u \} \\
\end{array}
\end{align*}
\]

The refinement is encoded as follows.

\[
R i c(P w d, c, k) = \{ c, d, s \} \uplus \{ l, u, l, u, d, \emptyset \} = \{ c \uplus l, u, c \uplus l, u, c \uplus l, u, d, c \uplus l, u, s, \emptyset \} = \{ l, u, l, u, d, l, u, l, u, d, s \}
\]

In both this application and the previous one, we had to treat specially the case where a feature could be replaced by the empty set, in effect, that it could be removed from a product line. This hiccough would be placed under the hood of any tool implementing our ideas.

**VIII. RELATED WORK**

Griss [7] briefly mentions the advantage of having different views on a feature model, where views are feature diagrams that display different levels of detail. A more radical insight is that “different stakeholders perceive differently what is variable” [16]. By considering developers and customers as the two main stakeholders, Pohl et al. [16] introduce the concepts of internal and external variability: external variability is visible to the customer while internal variability is hidden. Internal variability often represents finer-grained variation points at lower levels of abstraction.

Höfner et al. [17] formalise software product lines using the feature algebra model, and also describe reconciliation. Product lines are semirings with some extra properties, where features and products differ from product lines only on the properties they obey. The authors use the concrete example of sets and multisets of features to describe products, and sets of products to describe product lines, although other examples also fit their general formalisation. An abstraction of a product line can remove references to features and add new products. For example, the reduction \( \{ t, t 1 b n \} \leadsto \{ t, t i P \} \) from Example 1 is an abstraction only in our setting, while \( \{ t, t 1 b n \} \leadsto \{ t, t 1 b n, t P \} \) and \( \{ t, t 1 b n \} \leadsto \{ \emptyset, P \} \) are abstractions only in feature algebra.

In feature algebra a view is just a product line, and reconciliation of views is achieved by combining all possible products and filtering the result using a given set of requirements. Thus, reconciliation is guided by extra requirements, while in our approach reconciliation is guided by the view partition between features of the reconciled views. We explore compatibility of views, disregarded in Höfner et al.’s approach, and allow developers of different views to refer to simplified versions of each other’s views.

Existing work on views is generally presented at a different level of abstraction than our approach, such as architectural views and views on other software models [18]. Solomon [19] proposes using pushouts to merge architectures when different software systems need to be merged. Other approaches use pushouts and pullbacks and other algebraic techniques for model synchronisation [20] and version control in model-driven engineering [21]. Curiously, our approach uses pullbacks and not pushouts for combining models.

Acher et al. [22] present a technique for composing feature models out of smaller ones. Their work focuses mostly on composing feature diagrams, though some operations at the semantic level (sets of sets of features) are provided. Our work focuses exclusively on the underlying semantics and we provide a much richer theory. Segura et al. [23] use graph transformations for merging feature models, at the diagrammatic level. Their merging is akin to our view reconciliation, but by performing transformations on diagrams, they also lack a clear formal semantics.

Bowman et al. [24] present a framework for viewpoint consistency. Their framework covers many aspects of software models, though not feature models. Recent work in this line [25] considers model transformations across different views, where the views are represented by different kinds of models (state machines, class diagrams, etc). In our setting we work with only one kind of model.

**IX. CONCLUSION AND FUTURE WORK**

This paper presents a semantic perspective on views for feature models. Views enable feature models to be developed in a more modular fashion, where each stakeholder can have independent perspectives on a product line. Our theory provides a means for checking the compatibility of different views and for the reconciliation of compatible views.
For future work we will determine the constraints on view partitions to ensure that our techniques can apply to three or more views. We also plan to apply our techniques to more complex models, such as behavioural ones, underlying software product lines, and to implement our ideas in the developing HATS ABS toolkit [26].

REFERENCES

APPENDIX

Proposition 35. Given a partial function \( f : F \twoheadrightarrow G \). \( f : F \to G \) is total.

Proof: Immediate from definition. ■

Proof of functoriality of \( \overline{\cdot} \) and \( \overset{\sim}{\cdot} \) (III-B). Let \( p \in F \), \( u : F \to G \), and \( v : G \to H \). The functor \( \overline{\cdot} \) preserves identity morphisms and composition, as we show below. The proof that \( \overset{\sim}{\cdot} \) is also a functor follows the same reasoning.

\[
\begin{align*}
\overline{id_F(p)} &= \{ u(v(a)) \mid a \in p \cap \text{def}(id_F) \} \\
 &= \{ a \mid a \in p \} \\
 &= \overline{id_F(p)} \\
\overline{u \circ v(p)} &= \{ u(v(a)) \mid a \in p \cap \text{def}(u \circ v) \} \\
 &= \{ u(v(a)) \mid a \in p, a \in \text{def}(v), v(a) \in \text{def}(u) \} \\
 &= \{ u(v(a)) \mid a \in p, a \in \text{def}(v) \} \\
 &= \{ \overline{\text{def}(v)}(p) \} \\
 &= \{ (\overline{\text{def}} \circ v)(p) \}
\end{align*}
\]

\[\text{Proposition 35.} \]

We also conclude that, because \( R \) is a pullback, our definition of \( l \) is unique.

\[
\begin{align*}
F_r + G_r & \xrightarrow{l} F \\
\overline{\text{w.p.b.}} & \xrightarrow{\text{p.b.}} G \\
\overline{\text{w}} & \xrightarrow{\text{w}} F_u + G_a
\end{align*}
\]

We claim that \( k(R) \) is isomorphic to \( R \), where the \( l \) is the inverse of \( k \).

\[\bullet \ l \circ k = \text{id}_R. \quad \text{Let } (p, q) \in R, \text{ that is, } p \in F, \ q \in \overline{G}, \ \text{and } \overline{w_a(p)} = \overline{w_a(q)}. \text{ Then}
\]

\[k(l(p, q)) = \text{id}_R(p, q) = (p, q) \]

\[= (\overline{w_a(p)}(p), \overline{w_a(q)}(q)) \]

\[= (p, q) \]

\[\text{where the last step follows from Lemma 17, proven below, and because } \overline{w_a(p)} = \overline{w_a(q)}. \]

\[\bullet \ k \circ l = \text{id}_{F_r + G_r}. \quad \text{Let } p \in F_r + G_r \text{ (observe that }
\]

\[\overline{w_a(p)} = \overline{w_a(q)} \text{, that is, } \overline{w_a(p)} \cap \overline{w_a(q)}. \text{ Then}
\]

\[k(l(p)) = \overline{w_a(p)} \cap \overline{w_a(q)} \]

\[\text{where the last step follows from Lemma 17, proven below, and because } \overline{w_a(p)} = \overline{w_a(q)}. \]

\[\text{Lemma 36. For a view partition } \sigma = (F_r, F_a, G_r, G_a, v, w), \text{ the following diagram is a pullback:}
\]

\[
\begin{array}{ccc}
F_r + G_r & \xrightarrow{\overline{w}} & F \\
\overline{\text{w.p.b.}} & \xrightarrow{\text{p.b.}} & \overline{\text{w}} \\
\overline{G} & \xrightarrow{\text{w}} & F_u + G_a
\end{array}
\]

Proof: By definition, the canonical pullback in sets is:

\[R = \{(p, q) \in F \times \overline{G} \mid \overline{w_a(p)} = \overline{w_a(q)}\}.
\]

To prove that the diagram is a pullback we show that \( F_r + G_r \) is isomorphic to \( R \), given by the following functions, where \( \oplus \) is defined in Definition 19:

\[k : R \to F_r + G_r \quad l : F_r + G_r \to R
\]

\[(p, q) \mapsto p \oplus q \quad p \mapsto (\overline{w_a(p)}, \overline{w_a(q)}).
\]

Clearly \( k(R) \subseteq F_r + G_r \), and the reconciliation is defined for all pair in \( R \). We show that the codomain of \( l \) is \( R \) based on the commutativity at the level of features. Observe that \( w_a \circ w_r = w_a \circ v_r \), because when \( a \in F \), \( v_a(w_r(a)) = v_a(a) = v_r(a) = w_a(v_r(a)) \), and similarly when \( a \in G \). Therefore, because \( \overset{\sim}{\cdot} \) is a functor, the outer arrows of the diagram below commute. We define \( \pi_1(a, b) = a \) and \( \pi_2(a, b) = b \). Our definition of \( l \) yields trivially that \( \pi_1 \circ l = \overline{w} \) and \( \pi_2 \circ l = \overline{w} \).

\[\text{Lemma 37. For a view partition } \sigma = (F_r, F_a, G_r, G_a, v, w), \text{ the following diagram is a weak pullback:}
\]

\[
\begin{array}{ccc}
X & \xrightarrow{\exists u} & F \\
\overline{\text{w}} & \xrightarrow{\text{w.p.b.}} & \overline{\text{w}} \\
\overline{G} & \xrightarrow{\text{w}} & F_u + G_a
\end{array}
\]
Proof: Let \( u \) be defined as follows, where \( \oplus \) is defined in Definition 19.
\[
u : X \rightarrow \hat{F}_r + G_r
\]
\[
u(P) = f(P) \oplus g(P)
\]
We show that, for every \( f : X \rightarrow \hat{F} \) and \( g : X \rightarrow \hat{G} \) that makes the outer diagram commute (\( \hat{\nu} \circ f = \hat{w}_a \circ g \)), the function \( u \) exists, i.e., makes the rest of the diagram commute. The proof follows directly from Lemma 20.

\[
(\hat{\nu}_r \circ (f \oplus g))(P) = \hat{\nu}_r(f(P) \oplus g(P)) = f(P) = g(P)
\]

Proof of Lemma 15. We show that, when \( a \in \text{def}(v) \) or \( b \in \text{def}(w) \), then \( \nu_a(v_a(a)) = \nu_a(b) \) implies \( a \in F_r \) or \( b \notin G_r \). It can be easily verified that the codomain of \( \nu_a(v_a(a)) \) and \( \nu_a(b) \) only match when \( a \in F_r \) and \( b \notin G_r \), or when \( a \notin F_r \) and \( b \in G_r \). For example, if \( a \in F_r \) and \( b \in G_r \), then \( \nu_a(v_a(a)) = \nu_a(b) \). Due to partiality, even if the codomains do not match it could also happen that \( \nu_a(a) = \perp = \nu_a(b) \), but this is invalidated by our initial assumption.

Proof of Lemma 17. We show that \( \nu_r(P \oplus Q) = P \), and omit the analog proof for \( \nu_r(P \bigcirc Q) = Q \).
\[
\nu_r(P \oplus Q) = \nu_r((p \mid F_r) \cup (q \mid G_r)) = (p \mid F_r) \cup (q \mid G_r) = (p \mid F_r) \cup (\nu_r(q) \mid F_a) = (p \mid F_r) \cup (\nu_r(p) \mid F_a) = (p \mid F_r) \cup (p \mid F_a) = p
\]
Similarly, for \( \nu_r(p \oplus q) = q \).

Proof of Lemma 20. We show that \( \nu_r(P \oplus Q) = P \), and \( \nu_r(P \bigcirc Q) = Q \).
\[
\nu_r(P \oplus Q) = \nu_r\{p \mid F_r \cup (q \mid G_r) \mid p \in P, q \in Q, p \bigcirc q\}
\]
where the second last step follows from Lemma 17, and the last step follows from the fact that, when \( P \bigcirc Q \), \( \forall p \in P \Rightarrow \exists q \in Q \cdot p \bigcirc q \), as we show below.

\[
P \bigcirc Q \iff \nu_r(P) = \nu_r(Q)
\]

Proof of Lemma 22.
1) \( a \in F_r \Rightarrow \nu_a(v_a(a)) = v_a(a) = w_a(v_a(a)) \)
2) \( a \in G_r \Rightarrow w_a(v_a(a)) = w_a(a) = v_a(w_a(a)) \)
3) Follow from (1) and because \( \hat{\tau} \) and \( \hat{\nu} \) are functors.

Proof of Lemma 23. \( \hat{\nu} \) is a closure operator.
1) \( P \subseteq \hat{\nu}(P) \) by the Lemma 22.
2) Let \( P \subseteq Q \). Then:
\[
\hat{\nu}(Q) = \nu_r(Q) \cup \nu_r(Q) = \{p \mid q \in \nu_r(Q), q \in \nu_r(Q), p \bigcirc q\}
\]
\[
\nu_r(p) \cup \nu_r(q) \mid p \in P, q \in P, \nu_r(p) \bigcirc \nu_r(q) \}
\]
\[
\nu_r(p) \bigcirc \nu_r(q) \mid p \in P, q \in P, \nu_r(p) \bigcirc \nu_r(q) \}
\]
\[
\nu_r(p) \bigcirc \nu_r(q) \mid p \in P, q \in P, \nu_r(p) \bigcirc \nu_r(q) \}
\]

3) Trivially, \( \hat{\nu}(P) \subseteq \hat{\nu}(P) \). For the other direction we start by using the same reasoning as before:
\[
\hat{\nu}(P) = \{p \mid q \in \hat{\nu}(P), q \in \hat{\nu}(P), p \bigcirc q\}
\]
\[
\nu_r(p) \bigcirc \nu_r(q) \mid p, q \in \hat{\nu}(P), \nu_r(p) \bigcirc \nu_r(q) \}
\]
We now show that \( \nu_r(p) \bigcirc \nu_r(q) \subseteq \hat{\nu}(P) \), knowing that (i) \( p \in \hat{\nu}(P) \), and (ii) \( \nu_r(p) \bigcirc \nu_r(q) \).

The condition (i) implies that \( \exists p_1, q_1, q_2 \in P \) such that \( p = \nu_r(p_1) \cup \nu_r(p_2), q = \nu_r(q_1) \bigcirc \nu_r(q_2) \).
\( \varpi_r(p_1) \subset \varpi_r(p_2) \) and \( \varpi_r(q_1) \subset \varpi_r(q_2) \). Hence:

\[
\varpi_r(p) \oplus \varpi_r(q) = \varpi_r((\varpi_r(p_1) \cap F_r) \cup (\varpi_r(p_2) \cap G_r)) \oplus \\
\varpi_r((\varpi_r(q_1) \cap F_r) \cup (\varpi_r(q_2) \cap G_r)) \;
\]

We just need to show that \( \varpi_r(p_1) \subset \varpi_r(q_2) \) to prove that \( \varpi_r(p) \oplus \varpi_r(q) \in \hat{c}(P) \). This last step is shown from condition (ii) and by the Lemma 20:

\[
\begin{align*}
\varpi_r(p) \subset \varpi_r(q) & \iff \varpi_r(\varpi_r(p)) = \varpi_r(\varpi_r(q)) \\
& \iff \varpi_r(\varpi_r(p_1) \oplus \varpi_r(p_2)) = \\
& \iff \varpi_r(\varpi_r(q_1) \oplus \varpi_r(q_2)) \\
& \iff \varpi_r(p_1) \subset \varpi_r(q_2)
\end{align*}
\]

**Proof of Lemma 25.** We show that \( \hat{c}(P) \) is the maximum (wrt set inclusion) of \( [P] \), the equivalence class containing \( P \). That is, if \( P \sim Q \) then (1) \( \hat{c}(P) \sim P \), (2) \( \hat{c}(P) \sim Q \), and (3) \( \hat{c}(P) = \hat{c}(Q) \). Condition (1) is justified by Lemma 20:

\[
\begin{align*}
\hat{w}_r(\hat{c}(P)) = \hat{w}_r(\hat{w}_r(P) \oplus \hat{v}_r(P)) = \hat{w}_r(P) \\
\hat{v}_r(\hat{c}(P)) = \hat{v}_r(\hat{w}_r(P) \oplus \hat{v}_r(P)) = \hat{v}_r(P).
\end{align*}
\]

Condition (2) is a consequence of conditions (1) and (3), and the proof of condition (3) follows the same reasoning as the proof of condition (1), and uses the fact that \( P \sim Q \):

\[
\begin{align*}
\hat{w}_r(\hat{c}(P)) = \hat{w}_r(\hat{w}_r(P) \oplus \hat{v}_r(P)) = \hat{w}_r(\hat{c}(Q)) \\
\hat{v}_r(\hat{c}(P)) = \hat{v}_r(\hat{w}_r(P) \oplus \hat{v}_r(P)) = \hat{v}_r(\hat{c}(Q)).
\end{align*}
\]

**Lemma 38.** Let \( f : E \to F \) and \( g : E \to G \) be view functions such that \( f \supset g \), and let \( R = g \circ f^{-1} \). If \( a_1 \perp b_1 \) and \( a_2 \perp b_2 \), where \( a_1 \neq a_2 \) and \( b_1 \neq b_2 \), then \( (a_1, b_2) \notin R \).

**Proof:** This lemma can be proved by observing that \( f^{-1}(a_i) \neq g^{-1}(b_i) \) can never hold. If \( a_1, a_2, b_1, \) and \( b_2 \) are as defined in the lemma, then \( \exists e_i \in f^{-1}(a_i) \cap g^{-1}(b_i) \) for \( i \in \{1, 2\} \). When \( a_1 \perp b_2 \) the condition \( f^{-1}(a_1) \neq g^{-1}(b_2) \) no longer holds. If \( a_1 \perp b_2 \) then \( \exists e_3 : f(e_3) = a_1 \perp g(e_3) = b_2 \). Therefore \( \{e_1, e_3\} \leq f^{-1}(a_1) \) and \( \{e_2, e_3\} \leq g^{-1}(b_2) \), but \( e_1 \notin g^{-1}(b_2) \) (because \( g(e_1) = b_1 \) and \( e_2 \notin f^{-1}(a_1) \)) (because \( f(e_2) = a_2 \)). We conclude that \( f^{-1}(a_1) \neq g^{-1}(b_2) \) cannot hold.

**Proof of Theorem 27.** Let \( P \in \hat{E} \), and \( R = g \circ f^{-1} \subseteq F \times G \). \( R \) is partitioned into two parts: pairs with more than one image and its complements. For that we define the partitions \( F = F_r \cup F_a \) and \( G = G_r \cup G_a \) as follows.

\[
\begin{align*}
F_a &= \{ x \in F \mid |R(x)| > 1 \} \\
F_r &= F \setminus F_a \\
G_a &= R(F_r) \\
G_r &= G \setminus G_a
\end{align*}
\]

We now define \( v : F_r \to G \) and \( w : G_r \to F \) such that \( v = R \) and \( w = R^{-1} \) restricted to the corresponding domains. To show that \( \sigma = (F_r, F_a, G_r, G_a, v, w) \) is a view partition, we still need to verify that \( v \) and \( w \) are (partial) functions, and that the codomains of \( v \) and \( w \) are \( G_a \) and \( F_a \), respectively.

1. \( v \) is a function – If \( w(y) = x \) and \( w(y) = x' \), then \( x \not\sim x' \). By definition of \( F_a \), \( \exists y' \in G \cdot x \not\sim y' \), which contradicts Lemma 38. Hence \( x = x' \).

2. \( w \) is a function – If \( u(x) = y \), then by the definition of \( F_r \), we know that \( |R(x)| \leq 1 \), hence \( y \) is unique.

3. \( \text{cod}(v) = G_a - \text{By definition of } G_r \).

4. \( \text{cod}(w) = F_a \) – We show that if \( x \in F_a \) and \( x \not\sim y \), then \( y \in G_r \). If \( y \in G_a \), \( \exists x' \in F_r \) because \( v \) is onto, and \( \exists y' \in G \cdot x \not\sim y' \) by the definition of \( F_r \). But (1) and (2) contradict Lemma 38, hence \( y \in G_r \).

By the definition and properties of \( v \) and \( w \) we also conclude that \( v + w^{-1} = R \). As a consequence, we show that

\[
\forall a \in E : f(a) \sim a g(a). \tag{26}
\]

Recall that \( f(a) \equiv f(a) = v_a(f(a)) = w_a(g(a)). \) Observe now that, when \( f(a) \in F_r \), we have \( v_a(f(a)) = v(f(a)) = (g \circ f^{-1})(f(a)) = g(a) = w_a(g(a)), \) and when \( f(a) \in F_a \), we have \( v_a(f(a)) = f(a) = (f \circ g^{-1})(g(a)) = w(a) = w_a(g(a)). \)

Finally, we use Equation (26) to show the final result:

\[
\forall P \in \hat{E} : \hat{f}(P) \sim a \hat{g}(P). \tag{27}
\]

It is enough to verify that the left diagram below always commutes hence, because \( \hat{r} \) is a functor, the right diagram below also commutes.

\[
\begin{array}{ccc}
F_r + G_r & \xrightarrow{f} & F \\
\downarrow v_a & & \downarrow \hat{v} \\
G & \xrightarrow{w_a} & F_a + G_a
\end{array}
\]

\[
\begin{array}{ccc}
F_r + G_r & \xrightarrow{f} & F \\
\downarrow \hat{v} & & \downarrow \hat{v}_a \\
G & \xrightarrow{w_a} & F_a + G_a
\end{array}
\]
Proof of Lemma 30.

\[ P \oplus_{\sigma} Q_{f\emptyset} \]
\[ = \{ p \oplus q \mid p \in P, q \in Q_{f\emptyset}, p \triangleright q \} \]
\[ = \{ p \oplus q \mid p \in P, q \in Q_{f}, q \neq \emptyset, p \triangleright q \} \cup \{ p \oplus \emptyset \mid p \in P, p \triangleright \emptyset \} \]
\[ = \{ p \oplus q \mid p \in P, q \in Q_{f}, q \neq \emptyset, \overline{w_{r}}(p) = \overline{w_{r}}(q) \} \cup \{ p \oplus \emptyset \mid p \in P, \overline{w_{r}}(p) = \emptyset \} \]
\[ \cong \{(p \cap (F \setminus \{f\})) \cup (g \cap G), p \in P, q \neq \emptyset, f \notin p\} \cup \{ p \oplus \emptyset \mid p \in P, f \notin p \} \]

We now reduce the second element of the union.
\[ \{ p \oplus c \mid p \in P, c \in \overline{C}, p \triangleright c \} \]
\[ = \{ p \oplus c \mid p \in P, c \in \overline{C}, p \cap \{f\} \cup C = c \} \]
\[ = \{ p \cap \{f\} \cup C \cup (p \cap C) \mid p \in P, f \notin p, p \cap C = c \} \]
\[ = \{ p \cap \{f\} \cup C \cup (p \cap C) \mid p \in P, f \notin p \} \]

In the forth step, marked with \( * \), we interpret compatibility in our scenario. The condition \( \overline{w_{r}}(p) = \overline{w_{r}}(q) \) holds exactly when \( f \in p \) and \( q \neq \emptyset \), or when \( f \notin p \) and \( q \neq \emptyset \), because \( f \) is the only abstraction.

Proof of Lemma 33.

\[ P \oplus_{\sigma} Q_{k} \]
\[ = P \oplus_{\sigma} \{ (c \cup g \mid (c, g) \in k \} \cup \overline{C} \}
\[ = \{ p \oplus q \mid p \in P, q \in \{ c \cup g \mid (c, g) \in k \}, p \triangleright q \} \]
\[ \cup \{ p \oplus c \mid p \in P, c \in \overline{C}, p \triangleright c \} \]

We start by reducing the first element of the union above.
\[ \{ p \oplus q \mid p \in P, q \in \{ c \cup g \mid (c, g) \in k \}, p \triangleright q \} \]
\[ = \{ p \oplus (c \cup g) \mid p \in P, c \in \overline{C}, g \in k(c), \overline{w_{r}}(p) = \overline{w_{r}}(c \cup g) \} \]
\[ = \{(p \cap (F \setminus \{f\}) \cup C) \cup ((c \cup g) \cap G) \mid p \in P, c \in \overline{C}, g \in k(c), \overline{w_{r}}(p) = \overline{w_{r}}(c \cup g) \} \]
\[ = \{(p \setminus \{f\} \cup C) \cup c \cup g \mid p \in P, c \in \overline{C}, g \in k(c), p \cap (C \setminus \{f\}) = \emptyset \} \]
\[ \cup \{(p \setminus \{f\} \cup C) \cup c \cup g \mid p \in P, c \in \overline{C}, g \in k(c), p \cap (C \setminus \{f\}) = \emptyset \}
\[ = \{(p \setminus \{f\} \cup C) \cup c \cup g \mid p \in P, c \in \overline{C}, g \in k(c), p \cap C = c, f \in p \iff g \neq \emptyset \} \]
\[ = \{ p \setminus \{f\} \cup g \mid p \in P, g \in k(p \cap C), f \in p \iff g \neq \emptyset \} \]
Appendix E

Reconciliation of Feature Models via Pullbacks

The paper “Reconciliation of Feature Models via Pullbacks” [97] follows.
Reconciliation of Feature Models via pullbacks

José Proença and Dave Clarke*

IBBT-DistriNet, Katholieke Universiteit Leuven, Belgium
{dave.clarke,jose.proenca}@cs.kuleuven.be

Abstract. Variability in a Software Product Line (SPL) is expressed in terms of a feature model. As software development efforts involve increasingly larger feature models, scalable and modular techniques are required to manage their complexity. To address these issues of scalability and modularity this paper introduces a theory of views for features models, based on the categorical notion of pullback, encompassing both view compatibility and view reconciliation for plugging together different views of a product line.

1 Introduction

Variability in a Software Product Lines (SPL) is expressed in terms of a feature model. Feature models are usually depicted using feature diagrams [12], and a variety of increasingly expressive notations have been presented [2, 7, 10]. The semantics of feature diagrams are generally understood in terms of sets of (multi)sets of features [3], where at this level of abstraction a feature is simply a name. As software development efforts involve increasingly larger feature models, scalable techniques are required to manage their complexity, and modularity techniques are required to cater for the different interests of the various stakeholders involved in the project.

One approach to both managing complexity and increasing modularity is using views. A view of a feature model shows the decomposition of a concept up to a certain level of detail. With a view, only part of a feature model is visible to each stakeholder, possibly with some details abstracted away. For example, one stakeholder may be interested in the user visible functionality, and thus does not need to see the detailed features required by the programmers who implement that functionality. Programmers in one team need not see the complete variability of the product line from the perspective of other teams. In our approach the stakeholders can check whether their views are compatible and if so compute the global feature model consisting of the combined or reconciled view.

A theory of views for features models is introduced, along with notions of view compatibility and view reconciliation, to address issues of scalability and modularity. Our work is presented quite generally, and uses notions from category theory to ensure the future applicability of our results to more detailed

* This research is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Methods (http://www.hats-project.eu), and the K.U.Leuven BOF-START project STRT1/09/031 DesignerTypeLab.
models of software product lines. Our contributions include: a notion of view for feature models; criteria for determining whether views are compatible for a given SPL; techniques for reconciling compatible views; and the generalisation of these techniques to multiple views. With the exception of the latter, these topics have also been covered in an earlier workshop paper [6].

The paper is organised as follows. §2 provides some motivating examples. §3 introduces some preliminary concepts and defines abstractions and views of feature models. §4 presents our key definitions and results: view compatibility and view reconciliation and their characterisation. §5 takes a different angle, revealing some limitations of our approach. §6 applies our approach to a notion of uniform refinements of feature models. §7 deals with more than two views. §8 discusses related work. §9 presents our conclusions and future work.

2 Motivating Examples

We motivate our work with a collection of small examples, where features are options available in different mobile phones:

\{3g, alarm, wifi, email, maps, gps\}.

When developing software based on product lines, each feature has some associated source code that implements that feature, and a feature model describes valid combinations of these features. In this paper we investigate on how to combine views on product lines. A view consists of a selection of relevant features combined with the abstraction of some other features into more general features. The term reconciliation refers to this combination of views. A simple example will guide the intuition behind the reconciliation process. We start by considering views over the two sets:

\[ F = \{\text{Internet, alarm, email, maps}\} \quad \text{and} \quad G = \{3g, wifi, Apps\}. \]

In \( F \) we abstract 3g and wifi as Internet, and in \( G \) we abstract alarm, email and maps as Apps. We do not use gps yet. A product is a set of features, and a product line is a set of possible products.

Notation. As a convention, we start the name of a feature with an uppercase letter to denote it is an abstraction. For simplicity, we use only the first letter of each feature using true type font, and denote products by the consecutive letters of its features. For example, 3wA denotes the product \{3g, wifi, Apps\}.

Example 1 (Success story). We reconcile the following products lines, over \( F \) and \( G \), respectively.

\[ P = \{I, Ie, Iem\} \quad \text{and} \quad Q = \{3, 3wA\}. \]

The reconciliation of two elements \( x \) and \( y \) is denoted by \( x \oplus y \), and it exists only when \( x \) and \( y \) are compatible, denoted by \( x \succeq b \). Two features \( a \) and \( b \) are
compatible if one feature abstracts the other. In our example, $A \rightarrow I$ but not $A \leftarrow I$. Two products are compatible if every feature from each product has a compatible feature in the other product. In our case, $3wA \leftarrow Ie$ but not $3wA \rightarrow I$.

Finally, two product lines are compatible if every product from each product line has a compatible product in the other product line. The two product lines $P$ and $Q$ are compatible, and the reconciliation $\oplus$ joins every possible combination of compatible products. The reconciliation of these product lines yield:

$$P \oplus Q = \{I \oplus 3, Ie \oplus 3wA, Iem \oplus 3wA\} = \{3, 3we, 3wem\}.$$  

Example 2 (Partiality). We now include also the $gps$ feature, which is neither refined or abstracted by any other feature.

$$G_2 = G \cup \{gps\} \quad Q_2 = \{3, g3wA\}.$$  

When calculating $P \oplus Q_2$, the compatibility relation ignores the $gps$ feature, while the reconciliation operator preserves it. In our example, $g3wA \leftarrow Ie$, and $g3wA \oplus Ie = g3we$. Thus, $P \leftarrow Q_2$ and $P \oplus Q_2 = \{3, g3we, g3wem\}$.

Example 3 (Incompatibility). We now extend $Q$ with $Apps$.

$$Q_3 = \{3, 3wA, A\}.$$  

In this example, $P \nrightarrow Q_3$, because there is no product $p \in P$ for which $p \nleftarrow Apps$.

### 3 Preliminaries

We start by introducing basic mathematical models of concepts such as features, product lines, software product lines, following Schobbens et al. [16], along with other relevant formal definitions.

A **feature** is a unit of variability in a software product line relevant to some stakeholder [18]. Features are represented as elements of a set. **Products** and **product lines** are defined as sets of features and as sets of products, respectively. Typically $F, F', G, G', \ldots$ range over sets of features, $a, b, \ldots$ over features, $p, q, \ldots$ over products, and $P, Q, \ldots$ over product lines. The notation $\mathcal{P}(F)$ denotes the power set of $F$.

**Definition 1 (Product).** Given a set of features $F$. A product is defined as a subset of $F$. The set of all products for features $F$ is denoted $\bar{F}$ and is defined

$$\bar{F} \equiv \mathcal{P}(F).$$  

**Definition 2 (Product Line).** Given a set of features $F$. A product line is defined as a set of products for $F$, that is, a subset of $\bar{F}$. The set of all product lines for features $F$ is denoted $\bar{\bar{F}}$ and defined

$$\bar{\bar{F}} \equiv \mathcal{P}(\bar{F}) = \mathcal{P}(\mathcal{P}(F)).$$
Given a partial function \( f : F \rightarrow G \), the domain of \( f \) is \( \text{dom}(f) = F \), the codomain is \( \text{cod}(f) = G \), the pre-image is \( \text{def}(f) = \{ a \in F \mid f(a) \neq \perp \} \), and the range is \( \text{rng}(f) = \{ f(a) \mid a \in \text{def}(f) \} \). A function \( f \) is onto if \( \text{cod}(f) = \text{rng}(f) \).

The composition of two partial functions \( u \circ v \) is defined as:

\[
(u \circ v)(a) \equiv \begin{cases} 
\perp, & \text{if } v(a) = \perp \\
v(u(a)), & \text{otherwise.}
\end{cases}
\]

A partial function \( f : F \rightarrow G \) from one feature set to another can be lifted to functions over products and product lines, written as \( \overline{f} \) and \( \hat{f} \), respectively.

**Definition 3 (Lifting).** Given a partial function \( f : F \rightarrow G \), define the lifting of \( f \) to products and product lines as \( \overline{f} \) and \( \hat{f} \), respectively, as follows.

\[
\begin{align*}
\overline{f} : F \times G & \rightarrow \overline{G} \\
\hat{f} : \hat{F} & \rightarrow \hat{G}
\end{align*}
\]

\[
\overline{f}(p) \equiv \{ f(a) \mid a \in p \cap \text{def}(f) \} \quad \hat{f}(P) \equiv \{ \overline{f}(p) \mid p \in P \}.
\]

Another useful notion is the restriction of a product or product line to a smaller set of features. Restriction is one of the ingredients of our definition of view.

**Definition 4 (Restriction).** Let \( G \subseteq F \), \( p \in \overline{F} \), and \( P \in \hat{F} \). Define the restriction of \( p \) to \( G \), and the restriction of \( P \) to \( G \), denoted as \( p \upharpoonright G \) and \( P \upharpoonright G \), respectively, as follows:

\[
p \upharpoonright G \equiv p \cap G \quad P \upharpoonright G \equiv \{ p \upharpoonright G \mid p \in P \}.
\]

Abstraction removes distinctions between features, e.g., mapping all different varieties of television to a single feature TV. Views are then defined on top of this notion of abstraction, enabling not only the merging of distinctions, but also the hiding of features. Both concepts can be expressed for features and lifted to products and product lines.

**Definition 5 (Abstraction).** Any onto function \( f : F \rightarrow G \) defines an abstraction \( \hat{f} : \hat{F} \rightarrow \hat{G} \). A product line \( Q : \hat{G} \) can be seen as an abstraction of product line \( P : \hat{F} \) if an \( f : F \rightarrow G \) exists such that \( \hat{f}(P) = Q \). Similarly, an abstraction of products for such an \( f : F \rightarrow G \) is defined as \( \overline{f} : \overline{F} \rightarrow \overline{G} \).

Define a view function as a partial abstraction.

**Definition 6 (View).** For any partial and onto function \( g : F \rightarrow G \), we say that \( g \) is a feature view, \( \overline{g} \) is a product view, and \( \hat{g} \) is a product line view. Furthermore, for any \( P \in \hat{F} \) we say that \( \hat{g}(P) \in \hat{G} \) is a view on \( \hat{G} \). Similarly, a view of products for such an \( g : F \rightarrow G \) is defined as \( \overline{g} : \overline{F} \rightarrow \overline{G} \).

**Example 4.** Consider a product line with features \( F = \{ \text{sonyTV}, \text{philipsTV}, \text{sonyDVD}, \text{philipsDVD}, \ldots \} \). The following function \( v : F \rightarrow G \), where \( G = \{ \text{TV}, \text{sonyDVD}, \text{philipsDVD} \} \), defines a view for a stakeholder interested only in the DVD variability and whether a TV is present or not:

\[
f = \begin{cases} 
\text{sonyTV, philipsTV} & \mapsto \text{TV} \\
\text{sonyDVD} & \mapsto \text{sonyDVD} \\
\text{philipsDVD} & \mapsto \text{philipsDVD}.
\end{cases}
\]
4 View Compatibility and Reconciliation

Consider two stakeholders working with feature sets $F$ and $G$, respectively, who wish to combine their views. If the views are compatible, then it will be the case that some of the features in $F$ are abstractions of features in $G$, and vice versa. It may also be that some features in $F$ do not appear in $G$, and vice versa. This means that we can partition $F$ and $G$ each into two parts and construct a view between pairs of partitions, one going in each direction. The desired structure is called a view partition and is illustrated below. Here $F_r$ and $F_a$ are a partition of $F$. Similarly for $G$. The functions $v$ and $w$ are the view functions, mapping more refined elements in $F_r$ to their abstraction in $G_a$, and from $G_r$ to $F_a$, respectively.\footnote{Subscripts $r$ and $a$ indicate the source set (more refined features) and target set (more abstract features) of view functions of a view partition.}

\[
\sigma = \begin{pmatrix}
F_r & \left\{v\right\} & G_r \\
F_a & \left\{w\right\} & G_a
\end{pmatrix}
\]

**Definition 7 (View partition).** Define a view partition to be a tuple $\sigma = (F_r, F_a, G_r, G_a, v, w)$, where $F_r$, $F_a$, $G_r$, $G_a$ are sets of features such that $F = F_r \cup F_a$, $G = G_r \cup G_a$, $F_r \cap F_a = \emptyset = G_r \cap G_a$, and $v : F_r \rightarrow G_a$ and $w : G_r \rightarrow F_a$ are partial onto functions.

The set $F_r + G_r$ consists of the most refined features of $F$ and $G$ ($+$ is categorical coproduct, i.e., disjoint union). Reconciliation of two views produce a product line in terms of $F_r + G_r$. On the other hand, $F_a + G_a$ is the set of the most abstract features. These are used for checking the compatibility of two views.

Define $v_r : F_r + G_r \rightarrow G$ and $v_a : F \rightarrow F_a + G_a$ to be the partial onto functions that extend $v$ with the identity for elements in $G_r$ and $F_a$, respectively. That is,

\[
v_r(x) = \begin{cases} v(x), & \text{if } x \in F_r \\ x, & \text{if } x \in G_r \end{cases} \quad v_a(x) = \begin{cases} v(x), & \text{if } x \in F_r \\ x, & \text{if } x \in F_a \end{cases}
\]

Define $w_r$ and $w_a$ analogously, swapping $F$ and $G$.

These functions play an important role in what follows. For example, function $v_r : F_r + G_r \rightarrow G$ presents the view of the reconciled product line in terms of $G$. Function $v_a : F \rightarrow F_a + G_a$ recasts a view based on features $F$ in terms of the most abstract features, namely $F_a + G_a$.

Given a view partition, the following definition captures what it means for the underlying views to be compatible for features, products and product lines.

**Definition 8 (Compatibility of feature models).** For any $a \in F$, $p \in \overline{F}$, $P \in \overline{F}$, $b \in G$, $q \in \overline{G}$, and $Q \in \overline{G}$, and for a view partition $\sigma = (F_r, F_a, G_r, G_a, v, w)$, define compatibility of features, products, and product lines as follows:
Observe that \( v_a(a) = w_a(b) \) states that \( b \) is an abstract feature corresponding to \( a \), or vice versa, or both \( a \) and \( b \) only appear in \( F_r \) and \( G_r \). Now, \( p \bowtie q \) states that every feature in \( p \) has a compatible feature in \( q \), and vice versa, and similarly, \( P \bowtie Q \) states that every product in \( P \) has a compatible product in \( Q \), and vice versa. The key role of compatibility is that it provides the conditions for ensuring that two views can be reconciled.

The key machinery to get reconciliation to work is the categorical notion of a pullback [13]. \( A \times_C B \) is the pullback of a pair of arrows \( f : A \to C \) and \( g : B \to C \) if for every function \( h \) and \( k \) that make the outer ‘square’ of the diagram below commute there exists a unique function \( u \) that makes the whole diagram commute. A weak pullback drops the uniqueness criteria.

\[
\begin{array}{ccc}
X & \xrightarrow{\exists u} & A \\
\downarrow h & & \downarrow f \\
A \times_C B & \xrightarrow{\text{p.b.}} & A \\
\downarrow k & & \downarrow g \\
B & \xrightarrow{\text{p.b.}} & C \\
\end{array}
\]

Although pullbacks always exist in \( \textbf{Set} \), we need to select specific objects as the locus of reconciliation, namely, \( F_r + G_r \) and \( \hat{F}_r + \hat{G}_r \), and cannot always work with pullbacks. Theorem 1 describes when these objects are pullbacks (of the appropriate functions) and when we can only work with a weak pullback.

**Theorem 1.** Let \( \sigma = (F_r, F_a, G_r, G_a, v, w) \) be a view partition. The first diagram is a pullback, and the second is a weak pullback.

\[
\begin{array}{ccc}
F_r + G_r & \xrightarrow{\hat{v}_a} & F \\
\downarrow \text{p.b.} & & \downarrow \hat{v}_a \\
\hat{G} & \xrightarrow{\hat{w}_a} & \hat{F}_a + \hat{G}_a \\
\end{array}
\]

We now analyse how to combine two views \( P \in \hat{F} \) and \( Q \in \hat{G} \) for two feature sets \( F \) and \( G \). We start at the level of features and build up to product lines. In the following, let \( \sigma = (F_r, F_a, G_r, G_a, v, w) \) be a view partition.

We reconcile features by splitting into cases.

**Definition 9 (Feature reconciliation).** Given \( a \in F \) and \( b \in G \), define:

\[
a \oplus_{\sigma} b \quad \overset{\text{def}}{=} \begin{cases} 
a & \text{if } a \in F_r \text{ and } b \notin G_r \\
b & \text{if } a \notin F_r \text{ and } b \in G_r \\
\bot & \text{otherwise.}
\end{cases}
\]
Reconciliation of Feature Models via pullbacks

From now on we drop references to the view partition $\sigma$. When $a$ and $b$ are compatible, and at least one is in the domain of $v$ or $w$, then $a$ refines $b$ or vice-versa. Reconciliation choses the most refined feature. The definition of $a \oplus_\sigma b$ is justified by the following lemma.

**Lemma 1.** Let $a \in F$ and $b \in G$, such that $a \in \text{def}(v)$ or $b \in \text{def}(w)$. If $a \bowtie b$ then $a \in F_r \iff b \notin G_r$.

Reconciliation for features can be lifted to products, by selecting the most refined features and discarding the rest.

**Definition 10 (Product reconciliation).** Given a view partition $\sigma = (F_r, F_a, G_r, G_a, v, w)$, and products $p \in F$ and $q \in G$ such that $p \bowtie q$, define:

$$\oplus : F \times G \to F_r + G_r$$

$$p \oplus q \overset{\text{def}}{=} (p \upharpoonright F_r) \cup (q \upharpoonright G_r).$$

The following is used to prove our characterisation property (Corollary 1).

**Lemma 2.** $p \bowtie q \Rightarrow \overline{\varpi}(p \oplus q) = p$ and $\overline{\varpi}(p \oplus q) = q$.

The following corollary is justified by Lemma 2 and from the fact that $F_r + G_r$ is the pullback. This corollary characterises product reconciliation (Definition 10). The ‘outer square’ corresponds precisely to the compatibility condition (Definition 8). The fact that the inner square is a pullback guarantees the uniqueness of the function $1 \rightarrow F_r + G_r$, which can be shown to be $p \oplus q$, the uniquely selected element of $F_r + G_r$.

**Corollary 1.** Given view partition $\sigma = (F_r, F_a, G_r, G_a, v, w)$. If $p \bowtie q$, then $p \oplus q$ is the unique morphism making the following diagram commute:

We now lift the previous results for products and define the reconciliation of product lines as follows.

**Definition 11 (Product line reconciliation).** Given a view partition $\sigma = (F_r, F_a, G_r, G_a, v, w)$, $P \in \hat{F}$, and $Q \in \hat{G}$, define:

$$\oplus : \hat{F} \times \hat{G} \to \hat{F}_r + \hat{G}_r$$

$$P \oplus Q \overset{\text{def}}{=} \{ p \oplus q \mid p \in P, q \in Q, p \bowtie q \}.$$ 

This operation takes all compatible pairs of products from each product line and combines them using the product reconciliation operation. The imposition that $P \bowtie Q$ is a sufficient condition to guarantee that, after reconciling $P$ and $Q$, the original values of $P$ and $Q$ can still be recovered, as stated by Lemma 3.
Lemma 3. $P \bowtie Q \Rightarrow \hat{w}_r(P \oplus Q) = P$ and $\hat{v}_r(P \oplus Q) = Q$.

Furthermore, when the above condition does not hold, it is not possible in general to recover the values of $P$ and $Q$ from $P \oplus Q$, meaning that the $P$ and $Q$ would not be views on $P \oplus Q$. The following corollary is justified by Lemma 3 and that $\hat{F}s + \hat{G}s$ is a weak pullback.

Corollary 2. Given view partition $\sigma = (F_r, F_a, G_r, G_a, v, w)$. If $P \bowtie Q$, then $P \oplus Q$ makes the diagram commute:

$$
\begin{array}{ccc}
F_r + G_r & \xrightarrow{\hat{w}_r} & \hat{F} \\
\downarrow{\nu_r} & \searrow{w.p.b.} & \downarrow{\nu_a} \\
\hat{G} & \xrightarrow{\hat{w}_a} & F_a + G_a \\
\end{array}
$$

This construction is somewhat unsatisfying, as it does not say conclusively that the operation for product line view reconciliation is the best we can hope for. Before presenting an alternative characterisation, we take a step back to consider how our constructions look for views of a feature, product or product line, based on some given view partition.

Define the following functions:

$$
c : F_r + G_r \rightarrow F_r + G_r \quad a \mapsto w_r(a) \oplus v_r(a)
$$

$$
\hat{c} : F_r + G_r \rightarrow F_r + G_r \quad p \mapsto \hat{w}_r(p) \oplus \hat{v}_r(p)
$$

$$
\hat{\hat{c}} : F_r + G_r \rightarrow F_r + G_r \quad P \mapsto \hat{w}_r(P) \oplus \hat{v}_r(P)
$$

The following state the relationship between a feature, product or product line and the views on it for a given view partition. Firstly, the views are compatible and secondly, reconciliation works on the nose for features and products but provides an overapproximation for product lines.

Lemma 4. Given $a \in F_r + G_r$, $p \in F_r + G_r$ and $P \in F_r + G_r$. Then:

1. $w_r(a) \bowtie v_r(a)$
2. $c(a) = a$
3. $\overline{w_r}(p) \bowtie \overline{v_r}(p)$
4. $\overline{\nu}(p) = p$
5. $\hat{w}_r(P) \bowtie \hat{v}_r(P)$
6. $\hat{\nu}(P) \supseteq P$

The following properties capture the nature of $\hat{c}$, and thus provide some intuition about product line reconciliation.

Lemma 5. $\hat{c}$ is a closure operator.$^2$

The equivalence class of product lines that have the same views in $\hat{F}$ and in $\hat{G}$ is captured by the definition below.

$^2$ (1) $P \subseteq \hat{c}(P)$. (2) $P \subseteq Q$ implies $\hat{c}(P) \subseteq \hat{c}(Q)$. (3) $\hat{c}(\hat{c}(P)) = \hat{c}(P)$. 
Definition 12. Define equivalence \( \sim \subseteq \hat{F}^r + G^r \times \hat{F}^r + G^r \) as \( P \sim Q \iff \bar{w}_r(P) = \bar{w}_r(Q) \land \bar{v}_r(P) = \bar{v}_r(Q) \).

Finally, Lemma 6 characterises \( \hat{c} \) as selecting the maximum of all possible equivalent product lines with the same views in \( \hat{F} \) and \( \hat{G} \).

Lemma 6. \( \hat{c}(P) \) is the maximum (wrt \( \subseteq \)) of the equivalence class containing \( P \).

5 Constructing a View Partition

In the previous section, we assumed a view partition was given, though not necessarily the product line. Now we start from the other direction and address whether it is possible to find an appropriate view partition given a product line and two views. Let \( P \in \hat{E} \) be a product line, and \( f : \hat{E} \to \hat{F} \) and \( \hat{g} : \hat{E} \to \hat{G} \) define two views. The following condition guarantees the existence of a view partition.

Definition 13 (View compatibility). Two view functions \( f : E \to F \) and \( g : E \to G \) are compatible, denoted \( f \sim g \), iff for every \( a \in F \) and \( b \in G \),

\[
\quad f^{-1}(a) \not\# g^{-1}(b),
\]

where \( X \not\# Y \) iff \( X \cap Y = \emptyset \), \( X \subseteq Y \), or \( Y \subseteq X \).

Intuitively, Definition 13 states that \( f \) and \( g \) treat the features in \( E \) in terms of compatible abstractions. That is, two feature sets \( E_a = f^{-1}(a) \) and \( E_b = f^{-1}(b) \) are either incompatible, namely \( E_a \cap E_b = \emptyset \), or one is an abstraction of the other, namely \( E_a \subseteq E_b \), or vice versa.

The following theorem states that when two views functions are compatible, then a view partition exists to make the two views of a product line compatible.

Theorem 2. Let \( \hat{f} : \hat{E} \to \hat{F} \) and \( \hat{g} : \hat{E} \to \hat{G} \) be two views. If \( f \sim g \), then there exists a view partition \( \sigma \) such that \( \forall P \in \hat{E} \cdot \hat{f}(P) \sim_{\sigma} \hat{g}(P) \). Specifically, the following gives such \( \sigma = (F_r, F_a, G_r, G_a, v, w) \), where \( R = g \circ f^{-1} \):

\[
\begin{align*}
F_a &\overset{\text{def}}{=} \{ x \in F \mid |R(x)| > 1 \} & G_a &\overset{\text{def}}{=} R(F_r) & v : F_r \to G_a &\overset{\text{def}}{=} \{ R(x) \mid x \in F_r \} \\
F_r &\overset{\text{def}}{=} F \setminus F_a & G_r &\overset{\text{def}}{=} G \setminus G_a & w : G_r \to F_a &\overset{\text{def}}{=} \{ R^{-1}(x) \mid x \in G_r \}.
\end{align*}
\]

The relationship between the original product line \( P \in \hat{E} \) and the reconciled views \( \hat{c}(P) \) has already been described in Lemmas 4, 5 and 6.

Ultimately, one would expect all pairs of views to be compatible. That they are not points to a weakness in our theory. The problem stems from the fact that the two incompatible views have different ‘vocabularies’.

Example 5 (Overlapping views). Given features \{3g, gsm, wifi, gps\}. Consider view functions:

\[
\begin{align*}
f &= \{ 3g, \text{wifi} \mapsto \text{Internet} ; \ gsm, \text{wifi} \mapsto \text{Location} \} \\
g &= \{ \text{gsm}, 3g, \text{wifi} \mapsto \text{Radio} ; \ \text{gps} \mapsto \text{gps} \}
\end{align*}
\]
Observe that \( f \not\dashv g \), because \( f^{-1}(\text{Location}) \neq g^{-1}(\text{Radio}) \) does not hold. Consequently, there is no view partition guaranteeing the compatibility of product lines obtained by \( f \) and \( g \). The cause is the following conflict:

\[
\begin{array}{c}
\text{Location} \quad \text{Internet} \\
\downarrow \quad \quad \quad \downarrow \\
\text{gps} \quad \text{wifi/3g} \\
\downarrow \quad \quad \quad \downarrow \\
\text{gsm} \quad \text{Radio}
\end{array}
\]

\( \text{Location} \) is neither an abstraction of \( \text{Radio} \) nor vice versa, thus the calculations in Theorem 2 fail to yield a view partition.

6 Application: Uniform refinement

We show how to use reconciliation to replace a feature by a collection of products, dubbed uniform refinement. A refinement in context has also been presented in previous work [6], where the replacement is only performed for specific products. In a feature diagram a uniform refinement replaces a leaf in the tree of features by a new tree. Uniform refinement of a feature \( f \) by a set of products \( Q \), consists of taking each product containing \( f \) and replacing \( f \) by product \( p \), for each \( p \in Q \). In feature diagrams this corresponds to replacing a leaf feature \( f \) by a whole feature diagram whose underlying feature model is \( Q \).

**Definition 14 (Uniform Refinement).** Given product line \( P \in \hat{F} \), \( f \in F \), and \( Q \in \hat{G} \), where \( F \cap G = \emptyset \). Define:

\[
\begin{align*}
\text{UniR}(P,f,Q) & \overset{def}{=} \{ p[f \mapsto q] \mid p \in P, q \in Q \} \\
p[f \mapsto q] & \overset{def}{=} \begin{cases} 
  p \{ f \} \cup q, & \text{if } f \in p \\
  p, & \text{otherwise}
\end{cases}
\end{align*}
\]

The resulting refinement is a product line over features \( F \setminus \{ f \} \cup G \). This refinement can be achieved when \( \emptyset \notin Q \) using reconciliation, as follows.

\[
\text{UniR}'(P,f,Q) \overset{def}{=} P \oplus_{\sigma} (Q \cup \{ \emptyset \})
\]

where \( F_r = F \setminus \{ f \} \), \( F_a = \{ f \} \), \( v = \emptyset \), \( G_r = \bigcup Q \), \( G_a = \emptyset \), \( w = \{ g \mapsto f \mid g \in G_r \} \).

The view partition \( \sigma \) is defined such that the to-be-replaced feature \( f \) is in \( F_a \) as the abstraction of the features in \( Q \). The remaining features from \( F \) are kept in \( F_r \) and are not abstracted by \( v \). Consequently, the features in \( F_r \) are preserved during the reconciliation. Correctness is given by the following lemma.

**Lemma 7.** \( \text{UniR}(P,f,Q) = \text{UniR}'(P,f,Q) \).
Example 6. Consider the following product line, where we write $abc$ to represent the product $\{a, b, c\}$:

$$\text{Pwd} = \{c, cd, s\}$$

$\text{Pwd}$ describes the features of a password field. A password can have characters (feature $c$), digits (feature $d$), and symbols ($s$), according to the combinations in $\text{Pwd}$. We apply the uniform refinement

$$(c, \{u, lu\})$$

which refines characters as lowercase and uppercase letters (features $l$ and $u$, respectively), and requires uppercase letters. The view partition used to calculate this refinement is:

$$\begin{align*}
F & = \{d, s\} \\
G & = \{c\} \\
H & = \{\emptyset\}
\end{align*}$$

$$w = \{l \mapsto c, u \mapsto c\}$$

Finally, the refinement is calculated as follows.

$$\text{UniR}(\text{Pwd}, c, \{l, u, lu\}) = \{c, cd, s\} \oplus \{u, lu, \emptyset\}$$

$$= \{c \oplus u, c \oplus lu, cd \oplus u, cd \oplus lu, s \oplus \emptyset\}$$

$$= \{u, lu, du, dlu, s\}.$$  

In this application we had to treat specially the case where a feature could be replaced by the empty set, in effect, that it could be removed from a product line. This hiccup would be placed under the hood of any tool implementing our ideas.

7 Three or more views

We now study how to reconcile views from three feature sets $F$, $G$, and $H$. These results can be easily generalised for more than three feature sets. More specifically, we consider any three products $p \in F$, $q \in G$, and $r \in H$, and show when the operation $\oplus$ is associative, given only three view partitions: $F$-$G$, $G$-$H$, and $H$-$F$, between the respective sets of features, depicted in Fig. 1. Note that every view partition has a natural inverse by swapping the feature sets and the abstraction functions.

To show associativity we start by building new view partitions. For example, when reconciling $p$ with $q \oplus r$ we need to use a view partition between $F$ and the refined elements of $G$ and $H$. These new view partitions can be automatically derived from the three original view partitions. We then show that, using these view partitions, checking the compatibility of $p$, $q$, and $r$ pairwise is enough to guarantee that $p$, $q$, and $r$ can be reconciled in any order.

The domain of the three view partitions $F$-$GH$, $G$-$HF$, and $H$-$FG$ is represented in blue in the diagrams of these views below. For example, $F$-$GH$ can have
elements from either $F$ or from the reconciliation of $G$ and $H$. To guarantee reconciliation is associative we need to ensure that the reconciliation always lies in the intersection of the domains, presented in the rightmost diagram below.

### Notation

(1) We use superscripts to denote the partition under consideration, writing, for example, the elements of $f-g$ as $(F_{fr} G_{fr}, F_{ra} G_{ra}, v_{f-g}, v_{g-f})$.
(2) We write $f-gh$ to denote the view partition between $F$ and $G$ and $H$, represented in blue in Fig. 1.
(3) Finally, we write $F_{gh}^\alpha$ to denote the region $F_h^\alpha \cap F_g^\beta$, where $F_h^\alpha = F_{fr} H_{fr}$ and $F_g^\beta = F_{ra} H_{ra}$.

We now define $f-gh$; the definitions of $g-hf$ and $h-fg$ follow analogously. The view partition $f-gh$ exists iff we can use the abstraction functions $v_{f-g}$ and $v_{f-h}$ to build a unique abstraction function from $F$ to $G^h + H^g$. It is enough to ensure the following condition.

$$(v_{f-g})^{-1}(G_{ra}) \cap (v_{f-h})^{-1}(H_{ar}) = \emptyset$$

(view partition compatibility)

If this condition holds then $F_{gh} = \left\{ \begin{array}{ll} F^g_{fr} & \text{if } v_{f-g}(a) \in G^h_{ra} \\ F^h_{ra} & \text{if } v_{f-h}(a) \in H^g_{ar} \\ \bot & \text{otherwise} \end{array} \right.$$

$$v_{f-gh}(a) = \left\{ \begin{array}{ll} v_{f-g}(a) & \text{if } v_{f-g}(a) \in G^h_{ra} \\ v_{f-h}(a) & \text{if } v_{f-h}(a) \in H^g_{ar} \\ \bot & \text{otherwise} \end{array} \right.$$

The function $v_{f-gh}$ is well-defined only because of the view partition compatibility condition. The refined part $F^g_{fr} + G^h_{ra}$ corresponds exactly to the
intersection of the domains of $F\cap G$, $G\cap H$, and $H\cap F$, and the abstract part
$F_{a_{GH}} + GH_{a_{r}}$ corresponds to the rest of the domain of $F\cap G$.

To show associativity of the reconciliation of three products $p \in F$, $q \in G$, and $r \in H$ we assume that the view partitions $F-G$, $G-H$, and $H-F$ are compatible. We can then conclude the main theorem of this section.

**Theorem 3.** If $p \cap q$, $q \cap r$, and $r \cap p$, then

$$p \oplus (q \oplus r) = q \oplus (p \oplus r) = r \oplus (p \oplus q) = p \upharpoonright F_{a_{GH}} + q \upharpoonright G_{r} + r \upharpoonright H_{r}.$$ 

**Proof.** We only present a sketch for the proof. Observe that $p \cap q$, $q \cap r$, and $r \cap p$ imply that $p \cap (q \oplus r)$, $q \cap (p \oplus r)$, and $r \oplus (p \oplus q)$. The rest of the proof follows from unfolding the rules for reconciliation and restriction of products.

Three stakeholders working over $F$, $G$, and $H$, need to agree initially in the three view partitions, and guarantee that they are compatible. When combining their product lines they must check compatibility pairwise before performing any reconciliation. Alternatively, the stakeholders can define $F$, $G$ and $H$ as views of a common set of features, generate the view partitions according to §5, and check if the view partitions are compatible, without having to check whether their product lines are compatible before reconciling them.

8 Related Work

Griss [10] briefly mentions the advantage of having different views on a feature model, where views are feature diagrams that display different levels of detail. A more radical insight is that “different stakeholders perceive differently what is variable” [14]. By considering developers and customers as the two main stakeholders, Pohl et al. introduce the concepts of external and internal variability: the former is visible to the customer while the latter is hidden. Internal variability often represents finer-grained variation points at lower levels of abstraction.

Höfner et al. [11] formalise software product lines using the feature algebra model, and also describe reconciliation. Product lines are semirings with some extra properties, where features and products differ from product lines only on the properties they obey. The authors use the concrete example of sets and multisets of features to describe products, and sets of products to describe product lines, although other examples also fit their general formalisation. An abstraction of a product line can remove references to features and add new products. For example, the reduction $\{t, \text{tibn}\} \leadsto \{t, \text{tip}\}$ from Example 1 is an abstraction only in our setting, while $\{t, \text{tibn}\} \leadsto \{t, \text{tibn}, \text{tP}\}$ and $\{t, \text{tibn}\} \leadsto \{\emptyset, \text{P}\}$ are abstractions only in feature algebra. In feature algebra a view is just a product line, and reconciliation of views is achieved by combining all possible products and filtering the result using a given set of requirements. Thus, reconciliation is guided by extra requirements, while in our approach reconciliation is guided by the view partition between features of the reconciled views. We explore compatibility of views, disregarded in Höfner et al.’s approach, and allow developers of different views to refer to simplified versions of each other’s views.
Existing work on views is generally presented at a different level of abstraction than our approach, such as architectural views and views on other software models [4]. Solomon [19] proposes using pushouts to merge architectures when different software systems need to be merged. Other approaches use pullbacks and pushouts and other algebraic techniques for model synchronisation [9] and version control in model-driven engineering [15]. Curiously, our approach uses pullbacks and not pushouts for combining models. Segura et al. [17] use graph transformations for merging feature models, at the diagrammatic level. Their merging is akin to our view reconciliation, but by performing transformations on diagrams, they also lack a clear formal semantics.

Bowman et al. [5] present a framework for viewpoint consistency. Their framework covers many aspects of software models, though not feature models. Recent work in this line [8] considers model transformations across different views, where the views are represented by different kinds of models (state machines, class diagrams, etc). In our setting we work with only one kind of model.

9 Conclusion and Future Work

This paper presents a semantic perspective on views for features models. Views enable feature models to be developed in a more modular fashion, where each stakeholder can have independent perspectives on a product line. Our theory provides a means for checking the compatibility of different views and for the reconciliation of compatible views.

For future work we plan to apply our techniques to more complex models, such as behavioural ones, underlying software product lines, and to implement our ideas in the developing HATS ABS toolkit [1]. We also expect to improve the compositionality aspect of reconciliation, by identifying properties that guarantee that the result of reconciling two views is enough to be reconciled with other views.

References


A Proofs

The proofs included in this appendix appear for evaluation purposes.

**Proposition 1.** Given a partial function $f: F \rightarrow G$. \( \overline{f} : \overline{F} \rightarrow \overline{G} \) is total.

**Proof.** Immediate from definition.

**Proof of functoriality of \( \overline{\cdot} \) and \( \overline{\cdot} \) (§???).** Let $p \in \overline{F}$, $u : F \rightarrow G$, and $v : G \rightarrow H$. The functor \( \overline{\cdot} \) preserves identity morphisms and composition, as we show below. The proof that \( \overline{\cdot} \) is also a functor follows the same reasoning.

\[
\begin{align*}
\overline{id_F}(p) & = \{ \overline{id_F}(a) \mid a \in p \} \\
& = \{ a \mid a \in p \} \\
& = id_F(p)
\end{align*}
\]

\[
\begin{align*}
\overline{u \circ v}(p) & = \{ u(v(a)) \mid a \in p \} \\
& = \{ u(v(a)) \mid a \in p, a \in \text{def}(v), v(a) \in \text{def}(u) \} \\
& = \overline{\pi}(\{ v(a) \mid a \in p, a \in \text{def}(v) \}) \\
& = \overline{\pi}(\pi(p)) \\
& = (\overline{\pi} \circ \pi)(p)
\end{align*}
\]

**Proof of Theorem 1.** We split the proof into two parts. We prove in Lemma 8 that the diagram for products is a pullback, and we prove in Lemma 9 that the diagram for product lines is a weak pullback. These two lemmas are presented below.

**Lemma 8.** For a view partition $\sigma = (F_r, F_a, G_r, G_a, v, w)$, the following diagram is a pullback:

\[
\begin{array}{ccc}
F_r + G_r & \xrightarrow{\overline{\pi}} & F \\
\downarrow{\overline{\chi}} & & \downarrow{\overline{\pi}} \\
G & \xrightarrow{\overline{\chi}} & F_a + G_a
\end{array}
\]

**Proof.** By definition, the canonical pullback in sets is:

\[
R = \{ (p, q) \in F \times G \mid \overline{\pi_a}(p) = \overline{\pi_a}(q) \}.
\]

To prove that the diagram is a pullback we show that $F_r + G_r$ is isomorphic to $R$, given by the following functions, where $\oplus$ is defined in **Definition 11**:

\[
\begin{align*}
k : R & \rightarrow F_r + G_r \\
(p, q) & \mapsto p \oplus q
\end{align*}
\]

\[
\begin{align*}
l : F_r + G_r & \rightarrow R \\
p & \mapsto (\overline{\chi_a}(p), \overline{\pi}(p))
\end{align*}
\]
Clearly \( k(R) \subseteq F_r + G_r \), and the reconciliation is defined for all pair in \( R \).

We show that the codomain of \( l \) is \( R \) based on the commutativity at the level of features. Observe that \( v_a \circ w_r = w_a \circ v_r \), because when \( a \in F \), \( v_a(w_r(a)) = v_a(a) = v_r(a) = w_a(v_r(a)) \), and similarly when \( a \in G \). Therefore, because \( \pi \) is a functor, the outer arrows of the diagram below commute. We define \( \pi_1(a,b) = a \) and \( \pi_2(a,b) = b \). Our definition of \( l \) yields trivially that \( \pi_1 \circ l = \overline{w}r \) and \( \pi_2 \circ l = \overline{w}r \).

We also conclude that, because \( R \) is a pullback, our definition of \( l \) is unique.

\[
\begin{array}{c}
F_r + G_r \\
\overline{w}r \\
p.b. \pi
\end{array}
\begin{array}{c}
R \\
\overline{w}r \\
\pi
\end{array}
\begin{array}{c}
F_r + G_r \\
\overline{w}r \\
\pi
\end{array}
\]

We claim that \( k(R) \) is isomorphic to \( R \), where the \( l \) is the inverse of \( k \).

- \( l \circ k = id_R \). Let \((p,q) \in R\), that is, \( p \in F \), \( q \in \overline{w}r \), and \( \overline{w}r(p) = \overline{w}r(q) \). Then
  \[
l(k(p,q)) = l(p \oplus q) = (\overline{w}r(p \oplus q), \overline{w}r(p \oplus q)) = (p,q)
  \]
  where the last step follows from Lemma 2, proven below, and because \( \overline{w}r(p) = \overline{w}r(q) \).

- \( k \circ l = id_{F_r + G_r} \). Let \( p \in F_r + G_r \) (observe that \( \overline{w}r(\overline{w}r(p)) = \overline{w}r(\overline{w}r(p)) \)), that is, \( \overline{w}r(p) = \overline{w}r(p) \). Then
  \[
  k(l(p)) \\
  = k(\overline{w}r(p), \overline{w}r(p)) \\
  = \overline{w}r(p) \oplus \overline{w}r(p) \\
  = (\overline{w}r(p) \cap F_s) \cup (\overline{w}r(p) \cap G_s) \\
  = (\{ w_r(a) \mid a \in p, a \in def(w_r) \} \cap F_s) \cup (\{ w_r(a) \mid a \in p, a \in def(w_r) \} \cap G_s) \\
  = (\{ w_r(a) \mid a \in p, a \in def(w_r), w_r(a) \in F_s \} \cup (\{ w_r(a) \mid a \in p, a \in def(w_r), w_r(a) \in G_s \}) \\
  = (\{ a \mid a \in p, a \in def(w_r), a \in F_s \}) \cup (\{ a \mid a \in p, a \in def(w_r), a \in G_s \}) \\
  = (p \cap F_s) \cup (p \cap G_s) \\
  = p
  \]
Lemma 9. For a view partition \( \sigma = (F_r, F_a, G_r, G_a, v, w) \), the following diagram is a weak pullback:

\[
\begin{array}{ccc}
X & \xrightarrow{f} & \hat{F} \\
\downarrow{g} & & \downarrow{w.r} \\
\hat{G} & \xrightarrow{\tilde{v}_a} & F_a + G_a
\end{array}
\]

Proof. Let \( u \) be defined as follows, where \( \oplus \) is defined in Definition 11.

\[
u : X \to \hat{F}_r + \hat{G}_r
\]

\[
u(P) = f(P) \oplus g(P)
\]

We show that, for every \( f : X \to \hat{F}_r \) and \( g : X \to \hat{G}_a \) that makes the outer diagram commute (\( \tilde{v}_a \circ f = \tilde{w}_a \circ g \)), the function \( u \) exists, i.e., makes the rest of the diagram commute. The proof follows directly from Lemma 3.

\[
(\tilde{w}_r \circ (f \oplus g))(P) = \tilde{w}_r(f(P) \oplus g(P)) = f(P)
\]

\[
(\tilde{v}_r \circ (f \oplus g))(P) = \tilde{v}_r(f(P) \oplus g(P)) = g(P)
\]

Proof of Lemma 1. We show that, when \( a \in \text{def}(v) \) or \( b \in \text{def}(w) \), then \( v_a(a) = w_a(b) \) implies \( a \in F_r \iff b \notin G_r \). It can be easily verified that the codomain of \( v_a(a) \) and \( w_a(b) \) only match when \( a \in F_r \) and \( b \notin G_r \), or when \( a \notin F_r \) and \( b \in G_r \). For example, if \( a \in F_r \) and \( b \in G_r \), then \( v_a(a) = v(a) \in G_a \) and \( w_a(b) \in F_a \), thus \( v_a(a) \neq w_a(b) \). Due to partiality, even if the codomains do not match it could also happen that \( v_a(a) = \bot = w_a(b) \), but this is invalidated by our initial assumption.

Proof of Lemma 2.

\[
\overline{w}_r(p \oplus q) = \overline{w}_r(p \upharpoonright F_r \cup (q \upharpoonright G_r))
\]

\[
= (p \upharpoonright F_r) \cup \overline{w}_r(q \upharpoonright G_r)
\]

\[
= (p \upharpoonright F_r) \cup (\overline{w}_r(q) \upharpoonright F_a)
\]

\[
= (p \upharpoonright F_r) \cup (\overline{w}_a(q) \upharpoonright F_a)
\]

\[
= (p \upharpoonright F_r) \cup \overline{w}_a(p \upharpoonright F_a)
\]

\[
= (p \upharpoonright F_r) \cup (p \upharpoonright F_a)
\]

\[
= p
\]

Similarly, for \( \overline{v}_r(p \oplus q) = q \).

\[\square\]
Proof of Lemma 3. We show that \( \hat{w}_r(P \oplus Q) = P \), and omit the analog proof for \( \hat{v}_r(P \oplus Q) = Q \).

\[
\begin{align*}
\hat{w}_r(P \oplus Q) &= \hat{w}_r((p \mid F_r) \cup (q \mid G_r) \mid p \in P, q \in Q, p \not\sim q) \\
&= \{ \hat{w}_r(p) \mid p \in (p \mid F_r) \cup (q \mid G_r) \mid p \in P, q \in Q, p \not\sim q \} \\
&= \{ \hat{w}_r(p \mid F_r) \cup (q \mid G_r) \mid p \in P, q \in Q, p \not\sim q \} \\
&= \{ p \mid p \in P, q \in Q, p \not\sim q \} \\
&= P
\end{align*}
\]

where the second last step follows from Lemma 2, and the last step follows from the fact that, when \( P \not\sim Q \), \( \forall p \in P \cdot \exists q \in Q \cdot p \not\sim q \), as we show below.

\[
P \not\sim Q \iff \hat{v}_a(P) = \hat{w}_a(Q) \\
\iff \{ \hat{v}_a(p) \mid p \in P \} = \{ \hat{w}_a(q) \mid q \in Q \} \\
\iff \forall p \in P \cdot \exists q \in Q \cdot \hat{w}_a(p) = \hat{w}_a(q) \\
\iff \forall q \in Q \cdot \exists p \in P \cdot \hat{w}_a(p) = \hat{w}_a(q)
\]

Proof of Lemma 5. \( \hat{c} \) is a closure operator.

1) \( P \subseteq \hat{c}(P) \) by the Lemma 4.

3.5) Follow from (1) and because \( \hat{r} \) and \( \hat{a} \) are functors.

2) \( a \in F_r \Rightarrow \hat{w}_r(a) \oplus \hat{v}_r(a) = a \oplus v(a) = a \)

\( a \in G_r \Rightarrow \hat{w}_r(a) \oplus \hat{v}_r(a) = a \oplus v(a) = a \)

The last part follows by the definition of \( \oplus \).

4) \( \hat{w}_r(p) \oplus \hat{v}_r(p) \\
\begin{align*}
&= \{ \hat{w}_r(a) \mid a \in p, a \in \text{def}(w_r), w_r(a) \in F_r \} \\
&\cup \{ \hat{v}_r(a) \mid a \in p, a \in \text{def}(v_r), v_r(a) \in G_r \} \\
&= \{ a \mid a \in p, a \in F_r \} \cup \{ a \mid a \in p, a \in G_r \} \\
&= (p \cap F_r) \cup (p \cap G_r) \\
&= p
\end{align*}
\]

6) \( \hat{w}_r(P) \oplus \hat{w}_r(P) \\
\begin{align*}
&= \{ p \oplus q \mid p \in \hat{w}_r(P), q \in \hat{w}_r(P), p \not\sim q \} \\
&= \{ \hat{w}_r(p) \oplus \hat{w}_r(q) \mid p \in P, q \in P, \hat{w}_r(p) \not\sim \hat{w}_r(q) \} \\
\subseteq \{ \hat{w}_r(p) \oplus \hat{w}_r(p) \mid p \in P, \hat{w}_r(p) \not\sim \hat{w}_r(p) \} \\
&= \{ p \mid p \in P \} \\
&= P
\end{align*}
\]

Proof of Lemma 5. \( \hat{c} \) is a closure operator.

1) \( P \subseteq \hat{c}(P) \) by the Lemma 4.
2) Let \( P \subseteq Q \). Then:
\[
\hat{c}(Q) = \hat{w}_r(Q) \oplus \hat{v}_r(Q)
\]
\[
= \{ p \oplus q \mid p \in \hat{w}_r(Q), q \in \hat{v}_r(Q), p \triangleright q \}
\]
\[
\supseteq \{ \overline{w_r}(p) \oplus \overline{v_r}(q) \mid p \in Q, q \in Q, \overline{w_r}(p) \triangleright \overline{v_r}(q) \}
\]
\[
= \hat{c}(P)
\]

3) Trivially, \( \hat{c}(\hat{c}(P)) \supseteq \hat{c}(P) \). For the other direction we start by using the same reasoning as before:
\[
\hat{c}(\hat{c}(P)) = \{ \overline{w_r}(p) \oplus \overline{v_r}(q) \mid p, q \in \hat{c}(P), \overline{w_r}(p) \triangleright \overline{v_r}(q) \}
\]

We now show that \( \overline{w_r}(p) \oplus \overline{v_r}(q) \in \hat{c}(P) \), knowing that (i) \( p, q \in \hat{c}(P) \) and (ii) \( \overline{w_r}(p) \triangleright \overline{v_r}(q) \).

The condition (i) implies that \( \exists p_1, p_2, q_1, q_2 \in P \) such that \( p = \overline{w_r}(p_1) \oplus \overline{v_r}(p_2), q = \overline{w_r}(q_1) \oplus \overline{v_r}(q_2), \overline{w_r}(p_1) \triangleright \overline{v_r}(p_2) \) and \( \overline{w_r}(q_1) \triangleright \overline{v_r}(q_2) \).

Hence:
\[
\overline{w_r}(p) \oplus \overline{v_r}(q) = \overline{w_r}(\overline{w_r}(p_1) \oplus \overline{v_r}(p_2)) \oplus \overline{v_r}(\overline{w_r}(q_1) \oplus \overline{v_r}(q_2))
\]
\[
= \overline{w_r}((\overline{w_r}(p_1) \cap F_r) \cup (\overline{v_r}(p_2) \cap G_r)) \oplus \overline{v_r}((\overline{w_r}(q_1) \cap F_r) \cup (\overline{v_r}(q_2) \cap G_r))
\]
\[
= (\overline{w_r}(p_1) \cap F_r) \cup (\overline{v_r}(p_2) \cap G_r) \cup (\overline{w_r}(q_1) \cap F_r) \cup (\overline{v_r}(q_2) \cap G_r)
\]
\[
= (\overline{w_r}(p_1) \cap F_r) \cup (\overline{v_r}(q_2) \cap G_r)
\]
\[
= \overline{w_r}(p_1) \cap \overline{v_r}(q_2)
\]

We just need to show that \( \overline{w_r}(p_1) \triangleright \overline{v_r}(q_2) \) to prove that \( \overline{w_r}(p) \oplus \overline{v_r}(q) \in \hat{c}(P) \). This last step is shown from condition (ii) and by the Lemma 3:
\[
\overline{w_r}(p) \triangleright \overline{v_r}(q)
\]
\[
\iff \overline{w_u}(\overline{w_r}(p)) = \overline{w_u}(\overline{v_r}(q))
\]
\[
\iff \overline{w_u}(\overline{w_r}(p_1) \oplus \overline{v_r}(q_2))) = \overline{w_u}(\overline{w_r}(q_1) \oplus \overline{v_r}(q_2))
\]
\[
\iff \overline{w_u}(\overline{w_r}(p_1)) = \overline{w_u}(\overline{v_r}(q_2))
\]
\[
\iff \overline{w_r}(p_1) \triangleright \overline{v_r}(q_2)
\]

\textbf{Proof of Lemma 6.} We show that \( \hat{c}(P) \) is the maximum (wrt set inclusion) of \( [P] \), the equivalence class containing \( P \). That is, if \( P \sim Q \) then (1) \( \hat{c}(P) \sim P \), (2) \( \hat{c}(P) \sim Q \), and (3) \( \hat{c}(P) = \hat{c}(Q) \). Condition (1) is justified by Lemma 3:
\[
\hat{w}_r(\hat{c}(P)) = \hat{w}_r(\hat{w}_r(P) \oplus \hat{v}_r(P)) = \hat{w}_r(P)
\]
\[
\hat{v}_r(\hat{c}(P)) = \hat{v}_r(\hat{w}_r(P) \oplus \hat{v}_r(P)) = \hat{v}_r(P).
\]
Condition (2) is a consequence of conditions (1) and (3), and the proof of condition (3) follows the same reasoning as the proof of condition (1), and uses the fact that \( P \sim Q \):

\[
\begin{align*}
\hat{w}_r(\hat{e}(P)) &= \hat{w}_r(P) = \hat{w}_r(\hat{e}(Q)) \\
\hat{v}_r(\hat{e}(P)) &= \hat{v}_r(P) = \hat{v}_r(\hat{e}(Q)).
\end{align*}
\]

\(\blacksquare\)

**Lemma 10.** Let \( f : E \rightarrow F \) and \( g : E \rightarrow G \) be view functions such that \( f \sim g \), and let \( R = g \circ f^{-1} \). If \( a_1 R b_1 \) and \( a_2 R b_2 \), where \( a_1 \neq a_2 \) and \( b_1 \neq b_2 \), then \( (a_1, b_2) \notin R \).

**Proof.** This lemma can be proved by observing that \( f^{-1}(a_1) \neq g^{-1}(b_2) \) can never hold. If \( a_1, a_2, b_1, b_2 \) are as defined in the lemma, then \( \exists i \in \{1, 2\} \) for \( i \in \{1, 2\} \). When \( a_1 R b_2 \) the condition \( f^{-1}(a_1) \neq g^{-1}(b_2) \) no longer holds. If \( a_1 R b_2 \) then \( \exists e_i \in f^{-1}(a_i) \cap g^{-1}(b_i) \) for \( i \in \{1, 2\} \). Therefore \( \exists e_i \in f^{-1}(a_i) \) and \( \exists e_i, e_j \notin f^{-1}(a_i) \) (because \( g(e_i) = b_1 \)) and \( e_j \notin f^{-1}(a_j) \) (because \( f(e_2) = a_2 \)). We conclude that \( f^{-1}(a_1) \neq g^{-1}(b_2) \) cannot hold. \(\blacksquare\)

**Proof of Theorem 2.** Let \( P \in \hat{E} \), and \( R = g \circ f^{-1} \subseteq F \times G \). \( R \) is partitioned into two parts: pairs with more than one image and its complements. For that we define the partitions \( F = F_r \uplus F_a \) and \( G = G_r \uplus G_a \) as follows.

\[
\begin{align*}
F_a &= \{ x \in F \mid |R(x)| > 1 \} & (1) \\
F_r &= F \setminus F_a & (2) \\
G_a &= R(F_r) & (3) \\
G_r &= G \setminus G_a & (4)
\end{align*}
\]

We now define \( v : F_r \rightarrow G \) and \( w : G_r \rightarrow F \) such that \( v = R \) and \( w = R^{-1} \) restricted to the corresponding domains. To show that \( \sigma = (F_r, F_a, G_r, G_a, v, w) \) is a view partition, we still need to verify that \( v \) and \( w \) are (partial) functions, and that the codomains of \( v \) and \( w \) are \( G_a \) and \( F_a \), respectively.

1. \( w \) is a function - If \( w(y) = x \) and \( w(y') = x' \), then \( x R y \) and \( x' R y \). By definition of \( F_a \), \( \exists y' \in G \cdot x R y' \), which contradicts Lemma 10. Hence \( x = x' \).
2. \( v \) is a function - If \( v(x) = y \), then by the definition of \( F_r \) we know that \( |R(x)| \leq 1 \), hence \( y \) is unique.
3. cod(v) = G_a - By definition of G_a.
4. cod(w) = F_a - We show that if \( x \in F_a \) and \( x R y \), then \( y \in G_r \). If \( y \in G_a \) (\( \equiv y \notin G_r \)), then (1) \( \exists x' \in F_r \cdot x' R y \) because \( v \) is onto \( \), and (2) \( \exists y' \in G \cdot x' R y' \) by the definition of \( F_a \). But (1) and (2) contradict Lemma 10, hence \( y \in G_r \).

By the definition and properties of \( v \) and \( w \) we also conclude that \( v + w^{-1} = R \). As a consequence, we show that

\[
\forall a \in E \cdot f(a) \sim g(a).
\]

(5)
Recall that \( f(a) \bowtie g(a) = v_a(f(a)) = w_a(g(a)) \). Observe now that, when \( f(a) \in F_r \) we have \( v_a(f(a)) = v(f(a)) = (g \circ f^{-1})(f(a)) = g(a) = w_a(g(a)) \), and when \( f(a) \in F_a \) we have \( v_a(g(a)) = f(a) = (f \circ g^{-1})(g(a)) = w(g(a)) = w_a(g(a)) \).

Finally, we use Equation (5) to show the final result:

\[
\forall P \in \hat{E}. \hat{f}(P) \bowtie \hat{g}(P).
\]

It is enough to verify that the left diagram below always commutes hence, because \( \bowtie \) is a functor, the right diagram below also commutes.

\[
\begin{array}{ccc}
F_r + G_r & \xrightarrow{f} & F \\
v_a & \downarrow & \ \\
G & \xrightarrow{w_a} & F_a + G_a
\end{array}
\quad
\begin{array}{ccc}
F_r + G_r & \xrightarrow{\hat{f}} & \hat{F} \\
\bar{g} & \downarrow & \ \\
\hat{G} & \xrightarrow{\bar{w}_a} & \hat{F}_a + \hat{G}_a
\end{array}
\]

**Proof of Lemma 7.**

\[
P \oplus_\sigma Q_{f \emptyset} = \{p \oplus q \mid p \in P, q \in Q_{f \emptyset}, p \bowtie q\} \\
= \{p \oplus q \mid p \in P, q \in Q_f, q \neq \emptyset, p \bowtie q\} \cup \\
\{p \oplus \emptyset \mid p \in P, p \bowtie \emptyset\} \\
= \{p \oplus q \mid p \in P, q \in Q_f, q \neq \emptyset, \ov{\ov{p}}(p) = \ov{\ov{q}}(q)\} \cup \\
\{p \oplus \emptyset \mid p \in P, \ov{\ov{p}(p)} = \emptyset\} \\
= \{(p \cap F_r) \cup (q \cap G_r) \mid p \in P, q \in Q_f, q \neq \emptyset, f \in p\} \cup \\
\{p \oplus \emptyset \mid p \in P, f \notin p\} \\
= \{(p \cap (F \setminus \{f\})) \cup (q \cap \{g \mid g \in q', q' \in Q_f\}) \mid p \in P, q \in Q_f, q \neq \emptyset, f \in p\} \cup \\
\{p \mid p \in P, f \notin p\} \\
= \{(p \setminus \{f\}) \cup q \mid p \in P, q \in Q_f, q \neq \emptyset, f \in p\} \cup \\
\{p \mid p \in P, f \notin p\}
\]

In the forth step, marked with *", we interpret compatibility in our scenario. The condition \( \ov{\ov{p}(p)} = \ov{\ov{q}(q)} \) holds exactly when \( f \in p \) and \( q \neq \emptyset \), or when \( f \notin p \) and \( q \neq \emptyset \), because \( f \) is the only abstraction.

**Proof of Theorem 3.** Given three compatible view partitions \( f-G, g-h, \) and \( h-f \), and three products \( p \in \overline{F}, q \in \overline{G} \) and \( r \in \overline{H} \) such that \( p \bowtie q, q \bowtie r, \) and \( r \bowtie p \), we show that:

\[
p \oplus (q \oplus r) = q \oplus (p \oplus r) = r \oplus (p \oplus q) = p \mid F^{\text{vir}}_r + q \mid G^{\text{vir}}_r + r \mid H^{\text{vir}}_r.
\]

We split this proof into two parts. Lemma 11 shows that after reconciling a pair of products the result is compatible with the third product, and Lemma 12 shows the associativity product based in this fact.
Lemma 11. If \( p \prec q \), \( q \prec r \), and \( r \prec p \), then
\[
p \prec (q \oplus r), \quad q \prec (p \oplus r), \quad \text{and} \quad r \prec (p \oplus q).
\]

Proof. We only show that \( p \prec (q \oplus r) \), since the remaining cases are analogous.

Unfolding the definitions of compatibility and reconciliation, we need to proof that:
\[
\overline{v}^{\text{G-RH}}(p) = \overline{v}^{\text{G-RH}}(q \mid G_r \cup r \upharpoonright H_r^G)
\]

Unfolding the left-hand-side (lhs) and the right-hand-side (rhs), we obtain the following expressions.

\[
\overline{v}^{\text{G-RH}}(p) = \left\{ v^{\text{G-RH}}(x) \mid x \in p \cap \text{def}(v^{\text{G-RH}}) \right\}
\]
\[
\cup \left\{ v^{\text{G-RH}}(x) \mid x \in q \cap \text{def}(v^{\text{G-RH}}) \right\} \cap \text{def}(v^{\text{G-RH}})
\]
\[
\cup \left\{ \text{def}(v^{\text{G-RH}}) \cap G_r \right\}
\]
\[
\cup p \cap (F_{aa} \cup F_{ar}^G)
\]
\[
\cup (p \cap (F_{aa} \cup F_{ar}^G)) (10)
\]

\[
\overline{v}^{\text{G-RH}}(q \mid G_r \cup r \upharpoonright H_r^G) = \left\{ v^{\text{G-RH}}(x) \mid x \in q \mid G_r \cup r \upharpoonright H_r^G \cap \text{def}(v^{\text{G-RH}}) \right\}
\]
\[
\cup \left\{ \text{def}(v^{\text{G-RH}}) \cap G_r \right\}
\]
\[
\cup q \cap G_{rr}^G(r) \cap H_{rr}^G (13)
\]
\[
\cup r \cap H_{rr}^G (14)
\]

It is now enough to show that (7) = (13), (8) = (14), (9) = (11), and (10) = (12), since the domains of these expressions match.

• (7) = (13) \[
\left\{ v^{\text{G-RH}}(x) \mid x \in p \cap \text{def}(v^{\text{G-RH}}) \right\} \cap G_{rr}^G (p) \cap G_{rr}^G = q \cap G_{rr}^G
\]

• (8) = (14) \[
\left\{ v^{\text{G-RH}}(x) \mid x \in p \cap \text{def}(v^{\text{G-RH}}) \right\} \cap H_{rr}^G (p) \cap H_{rr}^G = r \cap H_{rr}^G
\]

• (11) = (9) \[
\left\{ v^{\text{G-RH}}(x) \mid x \in q \cap G_{rr}^G \cap \text{def}(v^{\text{G-RH}}) \right\} = v^{\text{G-RH}}(q \cap G_{rr}^G) = p \cap (F_{aa} \cup F_{ar}^G)
\]
Lemma 12. If $p \vdash (q \oplus r)$, $q \vdash (p \oplus r)$, and $r \vdash (p \oplus q)$, then

\[ p \oplus (q \oplus r) = q \oplus (p \oplus r) = r \oplus (p \oplus q) = p \upharpoonright F_{gr} + q \upharpoonright G_{hf} + r \upharpoonright H_{hf}. \]

Proof. We calculate only $p \oplus (q \oplus r)$, and the other combinations follow analogously.

\begin{align*}
p \oplus (q \oplus r) &= p \uplus ((q \upharpoonright G_{fr}) \cup (r \upharpoonright H_{fr})) \\
&= (p \upharpoonright F_{gr}) \cup (q \upharpoonright G_{fr}) \cup (r \upharpoonright H_{fr}) \\
&= (p \upharpoonright F_{gr}) \cup (q \upharpoonright (G_{fr} \cap G_{hf})) \cup (r \upharpoonright (H_{fr} \cap G_{hf})) \\
&= (p \upharpoonright F_{gr}) \cup (q \upharpoonright G_{fr}) \cup (r \upharpoonright H_{fr})
\end{align*}

\[\blacksquare\]
Appendix F

Feature Petri Nets

The paper “Feature Petri Nets” [89] follows.
Abstract—In software product line (SPL) engineering, formal modelling and verification are critical for managing the inherent complexity of systems with a high degree of variability. The number of products in an SPL can be exponential in the number of features. Therefore, the challenge when modelling SPL lies in analysing and verifying large, complex models efficiently, in order to ensure that all products behave correctly. The choice of a system modelling formalism that is both expressive and well-established is therefore crucial. In this paper we propose two lightweight extensions to Petri nets: Feature Petri Nets provide a framework for modelling and verifying software product lines; and Dynamic Feature Petri Nets provide additional support for modelling dynamic software product lines.

Keywords—software product lines; behavioural models; dynamic variability; Petri nets;

I. INTRODUCTION

The need to tailor software applications to specific requirements, such as specific hardware, markets or customer demands, is growing. If each application variant is maintained individually, the management overhead quickly becomes infeasible [1]. Software Produce Line Engineering (SPL) is seen as a solution.

A Software Product Line (SPL) is a set of software products that share a number of core properties but also differ in certain, well-defined aspects. The products of an SPL are defined and implemented in terms of features, which are subsequently combined in specific ways to obtain the final software products. The key advantage hereby over traditional approaches is that all products can be developed and maintained together. A challenge for the SPL approach is to ensure that all products meet their specifications without having to check each product individually, but rather checking the product line itself. This raises the need for novel SPL-specific formalisms to model SPL and reason about and verify their properties.

Petri nets [2] provide a solid formal basis for system modelling. They have been studied and applied widely, and they come with a wealth of formal analysis and verification techniques.

This research is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Methods (http://www.hats-project.eu), and the K.U.Leuven BOF-START project STRT1/09/031 DesignerTypeLab.

The main contribution of this paper is two Petri net variants suitable for modelling software product lines. These models enable the specification of system behaviour such as resource usage and workflow of the entire product line in one model. We extend Petri nets in two steps. We start by guiding the execution of a Petri net based on the feature selection, and later we introduce a mechanism to update the feature selection based on the execution of the Petri net. We call the first model Feature Petri Nets (FPN), and the second Dynamic Feature Petri Nets (DFPN). Our models provide an elegant separation between behaviour, modelled by the underlying Petri net, and available functionality, modelled by feature sets.

The paper is organised as follows. The next section shows a motivating example. Section III formally introduces the notion of Feature Petri nets. Section IV motivates Dynamic Feature Petri Nets and Section V presents their formal semantics. Section VI discusses the relation of FPN and DFPN to Petri Nets and how they can be used to model SPL. Section VII discusses related work. Section VIII presents our conclusions and future work.

II. FEATURE PETRI NET EXAMPLE

We illustrate the modelling challenge in SPL using an example of a software product line of coffee machines. A manufacturer of coffee machines offers products to match different demands, from the basic black coffee dispenser, to more sophisticated machines, such as ones that can add milk or sugar, or charge a payment for each serving. Each machine variant needs to run software adapted to the set of hardware features. Such a family of different software products that share functionality is typically developed using an SPL approach, that is, as one piece of software structured along distinct features. This has major advantages in terms of code reuse, maintenance overhead and so forth. The challenge is ensuring that the software works appropriately in all product configurations.

Petri nets are used to specify how systems behave. Figure 1 presents an example of a Petri net for a coffee machine. Places, represented by circles, can host tokens, represented by dots or a natural number, and the execution of a Petri net consists of the flow of tokens between places via transitions, depicted as filled rectangles. In our example, the coffee machine has a capacity for \( n \) coffee capsules; it can brew...
and serve coffee, and refill the machine with new coffee capsules. Note that in our model the action ‘refill’ does not mean filling the coffee reservoir completely, but refilling only one unit of coffee.

If we now add the Milk feature, that is, if we assume that the coffee machine can also add milk to the coffee, we need to adapt the Petri net. Furthermore, for every new feature we would need to specify a new Petri net for each possible combination of features, resulting in an explosion of combinations.

We address this problem by annotating transitions with application conditions [3], which are logical formula over features that reflect when the transition is enabled. Our example considers a product line whose products are over the set of features \{Coffee, Milk\}. The new Petri net model is called Feature Petri Nets (FPN). Figure 2 exemplifies an FPN of a coffee machine with a milk reservoir. The conditions on the transitions reflect that the three transitions on the right-hand side can be taken only when both features Coffee and Milk are present, and the three transitions on the left-hand side can be taken when the Coffee feature is present. The restriction of the example net to the transitions that can fire for feature selection \{Coffee\} is exactly the Petri net in Figure 1, after removing unreachable places.

### III. FEATURE PETRI NETS

Feature Petri nets (FPN) are a Petri net variant used to model the behaviour of an entire software product line. For this purpose, FPN have application conditions [3] attached to their transitions. An application condition is a boolean logical formula over a set of features, describing the feature combinations to which the transition applies. It constitutes a necessary (although not sufficient) condition for the transition to fire. In effect, if the application condition is false, it is as if the transition was not present.

We define Feature Petri Nets and give their semantics. We present two semantic accounts of FPN. First, when a set of features is selected, an FPN directly models the behaviour of the product corresponding to the feature selection. Second, by projecting an FPN onto a feature selection, one obtains a Petri net describing the behaviour of the same product. We show that these two notions of semantics coincide.

We start with some necessary preliminaries, first by defining multisets and basic operations over multisets. Then we define Petri nets and their behaviour.

**Definition 1 (Multiset).** A multiset over a set \(S\) is a mapping \(M : S \to \mathbb{N}\).

We view a set \(S\) as a multiset in the natural way, that is, \(S(x) = 1\) if \(x \in S\), and \(S(x) = 0\) otherwise. We also lift arithmetic operators to multisets as follows. For any function \(\circ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}\) and multisets \(M_1, M_2\), we define \(M_1 \circ M_2\) as \((M_1 \circ M_2)(x) = M_1(x) \cdot M_2(x)\).

#### A. Petri Nets

To ground our theory, we recall the terminology and notation surrounding Petri nets [4].

**Definition 2 (Petri Net).** A Petri net is a tuple \((S, T, R, M_0)\), where \(S\) and \(T\) are two disjoint finite sets, \(R\) is a relation on \(S \cup T\) (the flow relation) such that \(R \cap (S \times S) = R \cap (T \times T) = \emptyset\), and \(M_0\) is a multiset over \(S\), called the initial marking. The elements of \(S\) are called places and the elements of \(T\) are called transitions. Places and transitions are called nodes.

Sometimes we omit the initial marking \(M_0\).

**Definition 3 (Marking of a Petri Net).** A marking \(M\) of a Petri net \((S, T, R)\) is a multiset over \(S\). A place \(s \in S\) is marked if \(M(s) > 0\).

**Definition 4 (Pre-sets and post-sets).** Given a node \(x\) of a Petri net, the set \(*x = \{y \mid (y, x) \in R\}\) is the pre-set of \(x\) and the set \(x* = \{y \mid (x, y) \in R\}\) is the post-set of \(x\).

**Definition 5 (Enabling).** A marking \(M\) enables a transition \(t \in T\) if \(M\) marks every place in \(*t\), that is, if \(M \geq *t\).

The behaviour of a Petri net is a sequence of states, where each state is defined by a marking. The change from the current state to a new state occurs by the firing of a transition. A transition \(t\) can fire if it is enabled. Firing transition \(t\) changes the marking of the Petri net by decreasing the marking of each place in the pre-set of \(t\) by one, and increasing the marking of each place in the post-set of \(t\) by one.

**Definition 6 (Transition occurrence rule).** Given a Petri net \(N = (S, T, R)\), a transition \(t \in T\) occurs, leading from a state with marking \(M_t\) to a state with marking \(M_{t+1}\), denoted \(M_t \xrightarrow{t} M_{t+1}\), iff the following two conditions are met:
\[
M_t \geq *t \\
M_{t+1} = (M_t - *t) + t*
\]

(enabling) (computing)
The behaviour defined above is also known as the firing of a transition. Transitions fire sequentially, that is, only one transition occurs at a time.

**Definition 7** (Petri net trace). Given a Petri net \( N = (S, T, R, M_0) \), the behaviour the net exhibits by passing through a sequence of states with markings \( M_0, \ldots, M_n \), where each change of marking is triggered by a transition occurrence \( M_i \xrightarrow{t_i} M_{i+1} \), is called a trace. A trace is written \( M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \cdots \xrightarrow{t_{n-1}} M_n \).

**Definition 8** (Petri net behaviour). The behaviour of a Petri net is given by the set of all traces from a given initial marking.

### B. Feature Petri Nets

**Definition 9** (Application condition). An application condition \( \varphi \) is a logical (boolean) constraint over a set of features \( F \), defined by the following grammar:

\[
  \varphi ::= a \mid \varphi \land \varphi \mid \neg \varphi,
\]

where \( a \in F \). The remaining logical connectives can be encoded as usual. We write \( \Phi_F \) to denote the set of all application conditions over \( F \).

**Definition 10** (Satisfaction of application conditions). Given an application condition \( \varphi \) and a set of features \( FS \), called a feature selection, we say that \( FS \) satisfies \( \varphi \), written as \( FS \models \varphi \), iff

\[
  \begin{align*}
    FS & \models a & \text{iff } a \in FS \\
    FS & \models \varphi_1 \land \varphi_2 & \text{iff } FS \models \varphi_1 \text{ and } FS \models \varphi_2 \\
    FS & \models \neg \varphi & \text{iff } FS \not\models \varphi.
  \end{align*}
\]

We are now in the position to introduce Feature Petri Nets.

**Definition 11** (Feature Petri Net). A Feature Petri net is a tuple \( N = (S, T, R, M_0, F) \), where \( (S, T, R, M_0) \) is a Petri net, \( F \) is set of features, and \( f : T \to \Phi_F \) is a function associating each transition with an application condition from \( \Phi_F \).

For \( f(t) \), the application condition associated with transition \( t \), write \( \varphi_t \). For conciseness, we say that a feature selection \( FS \) satisfies transition \( t \) whenever \( FS \models \varphi_t \).

We now define the behaviour of Feature Petri Nets for a given (static) feature selection.

**Definition 12** (Transition occurrence rule for FPN). Given an FPN \( N = (S, T, R, M_0, F, f) \) and a feature selection \( FS \subseteq F \), a transition \( t \in T \) occurs, leading from a state with marking \( M_i \) to a state with marking \( M_{i+1} \), denoted \( (M_i, FS) \xrightarrow{t} (M_{i+1}, FS) \), iff the following three conditions are met:

\[
  \begin{align*}
    M_i & \geq t & \text{(enabling)} \\
    M_{i+1} & = (M_i - \cdot t) + t^* & \text{(computing)} \\
    FS & \models \varphi_t & \text{(satisfaction)}
  \end{align*}
\]

In the above definition the state of the Petri net is denoted by a tuple consisting of a marking and a feature selection, even though we assume the feature selection is static (constant). Later on, we will look at dynamic feature selections which can change during execution.

The transition rule for FPN is used to define traces that describe the FPN’s behaviour in the same way as Petri nets.

**Definition 13** (FPN Trace). Given an FPN \( N = (S, T, R, M_0, F, f) \) and a feature selection \( FS \subseteq F \), the behaviour the net exhibits by passing through a sequence of markings \( M_0, \ldots, M_n \), where each change of marking is triggered by a transition occurrence \( (M_i, FS) \xrightarrow{t_i} (M_{i+1}, FS) \), is called a trace over \( FS \). A trace is written \( (M_0, FS) \xrightarrow{t_1} (M_1, FS) \xrightarrow{t_2} \cdots \xrightarrow{t_n} (M_n, FS) \).
Given an FPN, there is a set of traces representing the behaviour of the FPN for each feature selection.

**Definition 14 (FPN behaviour for a given feature selection).**
Given an FPN $N = (S, T, R, M_0, F, f)$ and a feature selection $FS \subseteq F$, the behaviour of $N$ for $FS$, denoted $\text{Beh}(N, FS)$, is the set of all traces over $FS$ from the initial marking $M_0$.

If we consider all possible feature selections, we can express the behaviour of the FPN.

**Definition 15 (FPN Behaviour).**
Given an FPN $N = (S, T, R, M_0, F, f)$, we define $\text{Beh}(N)$ to be the combined set of behaviours for all feature selections over $F$:

$$\text{Beh}(N) = \bigcup_{FS \in PF} \text{Beh}(N, FS).$$

**C. Projection-based Semantics of FPN**

We now present an alternative semantics of Feature Petri Nets. Given a feature selection, the semantics of an FPN is a Petri net consisting of just the transitions satisfying the feature selection.

**Definition 16 (Projection).**
Given a Feature Petri Net $N = (S, T, R, M_0, F, f)$ and a feature selection $FS \subseteq F$, the projection of $N$ onto $FS$, denoted $N \downarrow FS$, is a Petri net $(S', T', R', M_0)$, with $T' = \{t \in T \mid FS \models \varphi_t\}$ and the flow relation $R' = R \cap ((S \cup T') \times (S \cup T'))$.

One projects $N$ onto a feature selection $FS$ by evaluating all application conditions $\varphi_t$ with respect to $FS$ for transitions $t \in T$. If $FS$ does not satisfy $\varphi_t$, then transition $t$ is removed from the Petri net. All application conditions are also removed when projecting.

The behaviour of the projection of a Feature Petri net $N$ onto a feature selection $FS$ coincides with the behaviour of $N$ for $FS$, as stated by the following theorem.

**Theorem 1.**
Given a Feature Petri Net $N$ and $FS \subseteq F$, then:

$$\text{Beh}(N, FS) = \text{Beh}(N \downarrow FS).$$

**Proof:** (⊆) We show that every trace $\sigma \in \text{Beh}(N, FS)$ is also a trace in $\text{Beh}(N \downarrow FS)$. Firstly, the initial markings $M_0$ coincide in both petri nets. Secondly, if $(M, FS) \xrightarrow{\sigma} (M', FS)$ then, by Definition 14, $FS \models \varphi_t$, and by Definition 16 it is also a transition of $N \downarrow FS$. Hence, $M \xrightarrow{\sigma} M'$.

(⊇) Following a similar reasoning as before, we show that every trace $\sigma \in \text{Beh}(N \downarrow FS)$ is also a trace in $\text{Beh}(N, FS)$. Observe that, if $M \xrightarrow{\sigma} M'$, then $t$ is a transition of $N \downarrow FS$, and by Definition 16 $FS \models \varphi_t$. Hence, by Definition 14 we conclude that also $(M, FS) \xrightarrow{\sigma} (M', FS)$. \hfill \blacksquare

**IV. DYNAMIC FEATURE RECONFIGURATION EXAMPLE**

Assuming that a product is composed from a static selection of features is sometimes too restrictive. As an example, we can think of a modular appliance, some of whose features can be disabled temporarily. For example, a coffee machine using fresh milk instead of milk powder allows the removal of the milk reservoir, in order to store it in the fridge. That change in the hardware configuration may entail a change in the software configuration. Modelling the presence/absence behaviour of the Milk feature may entail a significant modelling effort.

![Figure 3: DFPN modelling the ability to connect/disconnect a feature at runtime.](image)

To accommodate modelling this kind of dynamic feature reconfiguration, we introduce Dynamic Feature Petri Nets (DFPN). DFPN associate simple update expressions to transitions. Upon firing of a transition, updates affect the feature selection in effect.

In our example, switching the Milk feature on and off can be modelled by the DFPN in Figure 3, as an independent addition to the model in Figure 2. Associated to the DISCONNECT transition is the update expression “Milk off”. By firing the DISCONNECT transition, the current feature selection is updated, dropping the Milk feature. This action globally disables all transitions whose application condition depends on the Milk feature (that is, ADD MILK, REFILL MILK and SERVE COFFEE W/MILK in Figure 2). Conversely, firing the CONNECT transition re-enables all transitions conditioned on the Milk feature.

The feature reconfiguration model can remain disconnected from the “functional” model if the user interaction of removing/reconnecting the Milk feature can occur independently of the state the coffee machine. Alternatively, we can assume that the reconfiguration of features depends on the functional model. Figure 4 shows a model where removing/reconnecting the milk reservoir is only allowed when the machine is in a waiting state, prohibiting, for example, its removal when the machine is in the process of brewing coffee.

**V. DYNAMIC FEATURE PETRI NETS**

Dynamic Software Product Lines (DSPL) is an area of research concerned with runtime variability of systems [5]. DSPL is an umbrella concept that addresses dynamic reconfiguration of products (i.e. features are added and removed
at runtime), but also dynamic evolution of the product line itself (typically referred to as “meta-variability”). Pushing the binding time of features to runtime is often motivated by a changeable operational context, to which a product has to adapt in order to provide context-relevant services or meet quality requirements.

We extend Feature Petri Nets to capture the dynamic reconfiguration of products, resulting in a more general Petri net model. In our approach we associate to each transition an update expression that describes how the feature selection evolves after the transition. The resulting model is called Dynamic Feature Petri Nets (DFPN). DFPN extend Feature Petri nets by adding a variable feature selection to the state of the Petri net, and associating application conditions and update expressions over the feature set to the transitions. This extension enable more concise descriptions of systems based on feature models, without adding expressive power with respect to Petri nets. We now define update expressions before formalising DFPN.

**Definition 17 (Update).** An update is defined by the following grammar:

\[ u ::= \text{noop} \mid a \text{ on} \mid a \text{ off} \mid u; u \]

where \( a \in F \) and \( F \) is a set of features. We write \( U_F \) to denote the set of all updates over \( F \).

Given a feature selection \( FS \in F \), an update expression modifies \( FS \) according to the following rules:

- \( FS \xrightarrow{\text{noop}} FS \)
- \( FS \xrightarrow{a \text{ on}} FS \cup \{a\} \)
- \( FS \xrightarrow{a \text{ off}} FS \setminus \{a\} \)
- \( FS \xrightarrow{u \text{ on}} FS' \xrightarrow{u \text{ off}} FS'' \)

**Definition 18 (Dynamic Feature Petri Net).** A DFPN is a tuple \( N = (S, T, R, M_0, F, f, u) \), where \((S, T, R, M_0, F, f)\) is an FPN and \( u \) is a function \( T \rightarrow U_F \), associating each transition with an update from \( U_F \).

We write \( u(t) \) to denote the update expression \( u(t) \) associated with a transition \( t \).

**Definition 19 (DFPN transition occurrence).** Given a DFPN \( N = (S, T, R, M_0, F, f, u) \) and an initial feature selection \( FS_0 \subseteq F \), a transition \( t \in T \) occurs, leading from a state \((M_i, FS_i)\) to a state \((M_{i+1}, FS_{i+1})\), denoted \((M_i, FS_i) \xrightarrow{t} (M_{i+1}, FS_{i+1})\), if the following four conditions are met:

- \( M_i \geq \bullet t \) (enabling)
- \( M_{i+1} = (M_i - \bullet t) + t^* \) (computing)
- \( FS_i \models \varphi_t \) (satisfaction)
- \( FS_i \xrightarrow{u(t)} FS_{i+1} \) (update)
Definition 20 (DFPN trace). Given a DFPN $N = (S, T, R, M_0, F, f, u)$, the behaviour the net exhibits by assuming a sequence of states $(M_0, FS_0), \ldots, (M_n, FS_n)$, where each change of state is triggered by a transition occurrence $(M_i, FS_i) \xrightarrow{t_i} (M_{i+1}, FS_{i+1})$, is called a trace. A trace is written $(M_0, FS_0) \xrightarrow{t_1} (M_1, FS_1) \xrightarrow{t_2} \ldots \xrightarrow{t_n} (M_n, FS_n)$.

If we consider all possible traces, we obtain the behaviour of the FPN.

Definition 21 (DFPN Behaviour). Given a DFPN $N = (S, T, R, M_0, F, f, u)$, we define $\text{Beh}(N)$ to be the set of all traces starting with the initial state $(M_0, FS_0)$.

VI. DISCUSSION

Petri nets are a general modelling formalism, proposed for a wide variety of applications. FPN and DFPN leverage the power of Petri nets for modelling static and dynamic software product lines. They offer conciseness and convenience when modelling entire software families. Theorem 1 shows that an FPN is equivalent in behaviour to a set of Petri nets, one for each product defined by the SPL. DFPN additionally provision for dynamic SPL, by allowing explicit modelling of feature reconfiguration as part of the behavioural model.

By adding update expressions to Feature Petri Nets, Dynamic Feature Petri Nets do not gain more expressive power than Petri nets, but provide a more elegant separation of concerns. This approach offers orthogonality of the feature reconfiguration from the underlying behaviour, but in a way that enables the reconfiguration to depend upon the underlying behaviour and vice versa. We now justify intuitively this claim.

In our motivating example of the coffee machine, the availability of milk is represented by Petri net tokens, while the capability of doing actions related to the Milk feature is represented by the feature selection. However, the “activation” of transformations based on the available features can also be encoded using more complex Petri nets, where certain markings denote possible products. We illustrate this idea in Figure 5. The place MILK ON is associated to the selection of the feature Milk. When there is a token in this place, the transitions $t_1, \ldots, t_n$ are enabled. They are allowed to occur only when there is a token in MILK OFF, and this token remains in the same place. A similar approach can be used to convert any Dynamic Feature Petri Net into a more complex Feature Petri Net.

We present Feature Petri Nets as a novel SPL modelling formalism, but we do not examine how well this approach fares in practice. If used on a real-world product line, issues of scalability, and the need for a more modular modelling workflow could arise. These are subject to future research, in which we expect to devise a workflow where partial models can be reconciled to a coherent global model. This approach will benefit from previous work on Petri nets, since methods for composing and refining nets are well-studied topics [6]. In addition, many analysis techniques exist to determine the behavioural correctness of a Petri net design [2].

VII. RELATED WORK

Our research relates to a number of areas, specifically Petri net based formalisms, behavioural specification of software product lines and dynamic SPL research. We highlight the most relevant works in these fields.

A range of Petri net extensions based on a modified transition occurrence rule have been proposed, some of which are specifically tailored for representing dynamic, self-modifying behaviour [7, 8, 9, 10]. Unlike our approach, these formalisms generally exceed the expressive power of Petri nets. As a general consequence, properties such as reachability, boundedness and liveness are not decidable for these extensions, and they lack the full range of mathematical tools available to analyse normal Petri nets.

Inhibitor arc Petri nets [7] can test whether a place is empty by conditioning transitions on the absence of tokens. By modelling individual features as places, the presence or absence of tokens could represent whether a feature is on or off. An application condition could be encoded by including feature places in the pre-sets of transitions, thereby conditioning its firing on the presence or absence of features. Compared to our proposed approach, this entails a more complex net, with unclear boundaries between the functional and structural models. Conditional Petri nets [9] associate a transition to a formal language over transitions. Extending the classical occurrence rule, a transition is enabled only if the sequence of transitions that occurred in the past is in that language. An FPN could be encoded as a conditional Petri net by encoding application conditions in a language over the alphabet of transitions.

In self-modifying Petri nets [8], the flow relation changes dynamically according to the number of tokens at certain places in the net. A transition is enabled if it can fire as many tokens as present in the places referenced by its incoming arcs. Dynamic Petri nets [11] are similar, but have an “external control” through which the net’s structure can be changed by adding or removing arcs between nodes. Certain behaviour can thus be enabled or disabled by integrating...
or isolating places and transitions. These Petri net designs, although sporting a mechanism of self-modification, are geared towards dynamic changes in throughput, rather than the discrete activation/deactivation of behaviour offered by DFPN. Using net rewriting systems [10], dynamic changes in the configuration of a Petri net are described using a “rewriting rule”, which relates places and transitions of the two net configurations to each other. It is conceivable to model a dynamic SPL as a sequence of configurations and a set of rewriting rules which relate each configuration to the next. The DFPN approach, however, has the advantage of using a single model, in which each state clearly references a feature selection.

Compared to the surveyed Petri net formalisms, (D)FPN semantics are simpler, being closer to the application domain of variability modelling: through application conditions and update expressions they refer directly to the feature model of the SPL.

Various formalisms have been adopted for specifying the behaviour of software product lines, with the aim of providing a basis for analysis and verification of such models.

UML activity diagrams have been used to model the behaviour of SPL by superimposing several such diagrams in a single model [12]. Attached to the activity diagram’s elements are “presence expression”, which are similar in scope to application conditions. Models of products are obtained using model-to-model transformation by evaluating the presence conditions in the light of a given feature configuration. Compared to activity diagrams, Petri nets have a stronger formal foundation, with a larger spectrum of analysis and verification techniques, although, several studies have expressed the semantics of UML diagram using Petri nets (e.g. [13]).

Gruler et al. extended Milner’s CCS with a product line variant operator that allows an alternative choice between two processes [14, 15]. This calculus, referred to as PL-CCS, includes information about variability: by defining dependencies between features, once can control the set of valid configurations.

Variability is often modelled using transition systems enhanced with product-related information. Modal transition systems (MTS) [16] allow optional transitions, lending themselves as a tool for modelling a set of behaviours at once [17]. Generalised extended MTS [18] introduce cardinality-based variability operators and propose to use temporal logic formulas to associate related variation points. Asirelli et al. encode MTS using propositional deontic logic formulas [19]. Modal I/O automata [20] are a behavioural formalism for describing the variability of components based on MTS and I/O automata. Mechanisms of component composition are provided to support a product line theory. These approaches do not relate behaviour to elements of a structural variability model. Featured transition systems (FTS) [21] are an extension of labeled transition systems. Similar to Feature Petri Nets, transitions are explicitly labeled with respect to a feature model, and a feature selection determines the subset of active transitions. In FTS, transitions are mapped to single features. Transition priorities are used to deal with undesired non-determinism when selecting from transitions associated to different features. Using constraints, priorities are no longer required because we can negate the features in other transitions to obtain the same effect. The authors also envisage an alternative approach in which boolean expressions over features could be used.

The research fields of dynamic software product lines, context-aware and self-adapting systems include works on modelling dynamic systems at the levels of architecture [22, 23] and implementation [24, 25]. Between these, there is a notable gap with respect to formal specification and analysis of dynamic SPL behaviour.

VIII. CONCLUSION AND FUTURE WORK

This paper introduces Feature Petri Nets (FPN) and Dynamic Feature Petri Nets (DFPN), two lightweight Petri net extensions designed for modelling the behaviour of software product lines. The transition firing in an FPN is conditional on the presence of certain features through application conditions. Application conditions explicitly relate behaviour to feature configurations, while keeping that behaviour separate from the SPL structure. FPN capture the behaviour of entire product lines in a single, concise model, opening the way for efficient analysis and verification.

The DFPN model extends FPN with the ability to express dynamic variability. Update expressions associated with DFPN transitions make it easy to model changes in the feature selection of a product based on its execution: firing a transition updates the feature configuration in place. To our knowledge, this is the first model to capture both the variable and dynamic aspects of SPL in a single formalism.

In the future we expect to improve the modularity of our approach. Currently we use a single monolithic net to express the behaviour of all possible products. This issue can be addressed by adopting existing complexity managing techniques for Petri nets, such as abstraction, refinement, and composition [6], to allow the development of partial DFPN for views of a system, which can be later combined into a single coherent model. The main challenge becomes avoiding the development of one model for each possible product, and describing the global model by the combination of key partial DFPN.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their helpful suggestions and comments.
REFERENCES


Appendix G

Modular Modelling with Feature Petri Nets

The paper “Modular Modelling with Feature Petri Nets” [88] follows.
Modular Modelling with Feature Petri Nets

Radu Muschevici, Dave Clarke, and José Proença
DistriNet & IBBT, Dept. Computer Science, Katholieke Universiteit Leuven, Belgium
{radu.muschevici, dave.clarke, jose.proenca}@cs.kuleuven.be

Abstract. Formal modelling and verification are critical for managing the inherent complexity of systems with a high degree of variability, such as systems designed following the software product line (SPL) paradigm. SPL models tend to be large—the number of products in an SPL can be exponential in the number of features. Modelling these systems poses two main challenges. Firstly, a modular workflow that scales well is required. Secondly, the ability to analyse and verify complex models efficiently is key in order to ensure that all products behave correctly. The choice of a system modelling formalism that is both expressive and well-established is therefore crucial. In this paper we show how product lines can be modelled in an incremental, modular fashion using a formal method based on Petri nets. To this end we continue our work on Feature Petri Nets, a lightweight extension to Petri nets, by presenting a framework for modularly constructing Feature Petri Nets to model software product lines.

1 Introduction

The need to tailor software applications to varying requirements, such as specific hardware, markets or customer demands, is growing. If each application variant is maintained individually, the overhead of managing the variants quickly becomes infeasible [17]. Software Produce Line Engineering (SPLE) is seen as a solution to this problem. A Software Product Line (SPL) is a set of software products that share a number of core properties but also differ in certain, well-defined aspects. The products of an SPL are defined and implemented in terms of features, which are subsequently combined to obtain the final software products. The key advantage hereby over traditional approaches is that all products can be developed and maintained together. A challenge for SPLE is to ensure that all products meet their specifications without having to check each product individually, by checking the product line itself. This raises the need for novel SPL-specific formalisms to model SPLs and reason about and verify their properties.

The main contribution of this paper is a modular modelling framework for specifying the behaviour of software product lines. We use Feature Petri Nets, or Feature Nets (FN) for short, as the modelling formalism. Feature Petri Nets

* This research is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Methods (http://www.hats-project.eu), and the K.U.Leuven BOF-START project STRT1/09/031 DesignerTypeLab.
are a Petri net extension that enables the specification of the behaviour of a software product line (a set of systems) in one single model. The execution of an FN is conditional on the feature selection (the features appearing in the product line). The paper makes three contributions. Firstly, it presents Feature Nets. For their in-depth introduction we refer to the corresponding technical report [16]. Secondly, it gives techniques for constructing larger Feature Nets from smaller ones to model the addition of new features to an SPL. Thirdly, it provides two correctness criteria for ensuring that the resulting composition preserves the behaviour of the original model(s).

Organisation: The next section motivates the need for Feature Nets. Section 3 formally introduces the notion of Feature Net. Section 4 details two approaches at modular modelling with FN, and Section 5 discusses behaviour preservation. Section 6 discusses related work. Section 7 presents our conclusions and future work.

2 Software Product Line Modelling Challenge

We illustrate the modelling challenge in SPLE using an example of a software product line of coffee vending machines. A manufacturer of coffee machines offers products to match different demands, from the basic black coffee dispenser, to more sophisticated machines, such as ones that can add milk or sugar, or charge a payment for each serving. Each machine variant needs to run software adapted to the set of hardware features. Such a family of different software products that share functionality is typically developed using an SPLE approach, as one piece of software structured along distinct features. This has major advantages in terms of code reuse, maintenance overhead, and so forth. The challenge is ensuring that the software works appropriately in all product configurations.

Petri nets [11] are used to specify how systems behave. Fig. 1 presents an example of a Petri net for a coffee machine. This has a capacity for $n$ coffee servings; it can brew and dispense coffee, and refill the machine with new coffee.
supplies. Note that in our model the action ‘refill’ means refilling one unit of coffee at a time. If we now add a Milk feature, so that the machine can also add milk to a coffee serving, we need to adapt the Petri net, not only to include the functionality of adding milk, but also to be able to control whether or not this feature is present in the resulting software product.

To address the challenge of modelling a software product line with multiple features, which may or may not be included in any given product, we introduced Feature Petri Nets [16], or Feature Nets (FN) for short. In Feature Nets, transitions are annotated with application conditions [18], which are propositional formula over features that reflect when the transition is enabled. The advantage of Feature Nets is that they enable the superposition of the behaviour of the various products (given by different feature selections) in the same model.

![Feature Net of the product line \{Coffee\}, \{Coffee, Milk\} showing each product in its initial state. Each transition has an application condition attached.](image)

Fig. 2 exemplifies a Feature Net of a coffee machine with a milk reservoir. It considers a product line whose products are over the set of features \{Coffee, Milk\}. The conditions above each transition reflect that the three transitions on the left-hand side can be taken when the Coffee feature is present and the two transitions on the right-hand side can be taken only when the feature Milk is present. Observe that the restriction of this example net to the transitions that can fire for feature selection \{Coffee\} is exactly the Petri net in Fig. 1, after removing unreachable places.

### 3 Feature Nets

A Feature Net (FN) [16] is a Petri net variant used to model the behaviour of an entire software product line. For this purpose Feature Nets have application conditions [18] attached to their transitions. An application condition is a propositional logical formula over a set of features, describing the feature combinations
to which the transition applies. It constitutes a necessary (although not sufficient) condition for the transition to fire. In effect, if the application condition is false, it is as if the transition were not present.

We define Feature Nets and give their semantics. In the corresponding technical report [16] two semantic definitions of Feature Nets are presented. The first semantics directly models the FN for a given feature selection. The second semantics, which we recall here, is given by projecting the FN for a given feature selection onto a Petri net with only the transitions enabled. These two notions have been shown to coincide [16].

We start with some necessary preliminaries, first by defining multisets and basic operations over multisets, then by defining Feature Nets and their behaviour. Our terminology is standard for Petri nets [7].

**Definition 1 (Multiset).** A multiset over a set $S$ is a mapping $M : S \rightarrow \mathbb{N}$.

We view a set $S$ as a multiset in the natural way, that is, $S(x) = 1$ if $x \in S$, and $S(x) = 0$ otherwise. We also lift arithmetic operators to multisets as follows. For any function $\odot : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ and multisets $M_1, M_2$, we define $M_1 \odot M_2$ as $(M_1 \odot M_2)(x) = M_1(x) \odot M_2(x)$.

**Definition 2 (Application condition [18]).** An application condition $\varphi$ is a propositional formula over a set of features $F$, defined by the following grammar:

$$\varphi ::= a \mid \varphi \land \varphi \mid \neg \varphi,$$

where $a \in F$. The remaining logical connectives can be encoded as usual. We write $\Phi_F$ to denote the set of all application conditions over $F$.

**Definition 3 (Satisfaction of application conditions).** Given an application condition $\varphi$ and a set of features $FS$, called a feature selection, we say that $FS$ satisfies $\varphi$, written as $FS \models \varphi$, iff

$$FS \models a \text{ iff } a \in FS$$
$$FS \models \varphi_1 \land \varphi_2 \text{ iff } FS \models \varphi_1 \text{ and } FS \models \varphi_2$$
$$FS \models \neg \varphi \text{ iff } FS \not\models \varphi.$$

**Definition 4 (Feature Net).** A Feature Net is a tuple $N = (S, T, R, M_0, F, f)$, where $S$ and $T$ are two disjoint finite sets, $R$ is a relation on $S \cup T$ (the flow relation) such that $R \cap (S \times S) = R \cap (T \times T) = \emptyset$, and $M_0$ is a multiset over $S$, called the initial marking. The elements of $S$ are called places and the elements of $T$ are called transitions. Places and transitions are called nodes. Finally, $F$ is a set of features and $f : T \rightarrow \Phi_F$ is a function associating each transition with an application condition from $\Phi_F$.

Without $f$ and $F$, a Feature Net is just a Petri net. Sometimes we omit the initial marking $M_0$. We write $\varphi_t$ for $f(t)$, the application condition associated with transition $t$. For conciseness, we say that a feature selection $FS$ satisfies transition $t$ whenever $FS \models \varphi_t$. 
Definition 5 (Marking of a Feature Net). A marking $M$ of a Feature Net $(S, T, R, F, f)$ is a multiset over $S$. A place $s \in S$ is marked iff $M(s) > 0$.

Definition 6 (Pre-sets and post-sets). Given a node $x$ of a Feature Net, the set $\bullet x = \{ y \mid (y, x) \in R \}$ is the pre-set of $x$ and the set $x^* = \{ y \mid (x, y) \in R \}$ is the post-set of $x$.

Definition 7 (Enabling). A marking $M$ enables a transition $t \in T$ if it marks every place in $\bullet t$, that is, if $M \geq \bullet t$.

We now define the behaviour of Feature Nets for a given feature selection.

Definition 8 (Transition occurrence rule for FN). Given a Feature Net $N = (S, T, R, M_0, F, f)$ and a feature selection $FS \subseteq F$, a transition $t \in T$ occurs, leading from a state with marking $M_i$ to a state with marking $M_{i+1}$, denoted $M_i \xrightarrow{t} M_{i+1}$, iff the following three conditions are met:

- $M_i \geq \bullet t$ (enabling)
- $M_{i+1} = (M_i - \bullet t) + t^*$ (computing)
- $FS \models \varphi_t$ (satisfaction)

The transition rule for FN is used to define traces that describe the FN’s behaviour. We now define the semantics of a Feature Net by projecting it onto a Petri net for a given feature selection.

Definition 9 (Projection). Given a Feature Net $N = (S, T, R, M_0, F, f)$ and a feature selection $FS \subseteq F$, the projection of $N$ onto $FS$, denoted $N \downarrow FS$, is a Petri net $(S, T', R', M_0)$, with $T' = \{ t \mid FS \models \varphi_t \}$ and the flow relation $R' = R \cap ((S \cup T') \times (S \cup T'))$.

One projects $N$ onto a feature selection $FS$ by evaluating all application conditions $\varphi_t$ with respect to $FS$ for transitions $t \in T$. If $FS$ does not satisfy $\varphi_t$, then $t$ is removed from the Petri net. All application conditions are also removed when projecting.

The behaviour of a Petri net $N = (S, T, R, M_0)$ is given by the set of all its traces [11], written $\text{Beh}(N) = \{ M_0 \xrightarrow{t_1} \cdots \xrightarrow{t_n} M_n \mid M_i \subseteq S, i \in 1..n, M_{i-1} \xrightarrow{t_i} M_i \}$. We define the behaviour of a FN by considering all possible feature selections.

Definition 10 (FN Behaviour). Given an FN $N = (S, T, R, M_0, F, f)$, we define $\text{Beh}(N)$ to be the union of the behaviours for all feature selections over $F$:

$$\text{Beh}(N) = \bigcup_{FS \subseteq F} \text{Beh}(N \downarrow FS).$$

Lemma 1 (FN behaviour is the set of all traces [16]).

The behaviour $\text{Beh}(N)$ of a Feature Net $N$ coincides with the set of all its traces constructed following the transition occurrence rule for Feature Nets (Definition 8).
4 Modular modelling

For a modelling formalism to be scalable and useful in practice, it needs to facilitate a modular, incremental workflow. This is especially important for modelling software product lines: a single SPL model combines the behaviour of a set of different systems, which are often too complex to develop simultaneously.

Modular approaches include top-down techniques, where initially an abstract model is sketched and more details are added incrementally, and bottom-up approaches, where subsystems are modelled separately and later plugged together to a global model. Petri nets support both approaches [11].

In the following we propose two refactoring techniques that use FNs in a modular way. Both are based on the idea of modelling features as individual subsystems and later combining them to obtain a model of the entire SPL. They differ in how these sub-models are composed together. In the first approach, a single node (place or transition) is refined into a more complex FN. The second approach is based on fusing common nodes from different subsystems.

We show how each of the two techniques can be applied to construct a coffee machine product line with the three features \{Coffee, Payment, Milk\} from separate nets modelling individual features.

4.1 Compositional modelling using refinement

In the first modelling stage, each feature’s behaviour is expressed by a separate FN. The feature’s interaction with the rest of the system is modelled by the flow of tokens to an abstract node. A feature modelled using this technique can be seen as a partially specified model of the entire SPL, where the feature’s behaviour is fully specified, whereas everything else is underspecified. Composition then entails refining the abstract node into the fully specified system.

The three features of our example coffee machine are modelled as separate FN (Fig. 3). Unless a feature’s behaviour is self-contained, it will typically interact with other features that are part of the larger system. To faithfully model such interactions we include an abstract representation of the larger system in the shape of a node. For example, the model of Milk in Fig. 3b reflects the fact that adding milk depends on a state of the system in which a cup of fresh coffee is available. The larger system, which is not part of the feature itself, is represented abstractly by the place COFFEE; a token in this place would denote a state in which a freshly brewed cup of coffee is available. Similarly, Fig. 3c models the fact that after a payment has been accepted, the overall system is able to proceed with performing other actions such as brewing and serving drinks.

Constructing a model of the whole SPL is done by stepwise refining the abstract nodes of each feature. The intuition behind node refinement is that each abstract node is replaced with a more complex Feature Net. In our example, the first step could be to refine Payment’s PROCEED transition by replacing it with the Feature Net for Coffee. In a second step, the feature Milk is refined by
(a) The Coffee feature is at the basis of a coffee machine product line.

(b) Milk feature

(c) Payment feature

Fig. 3: Individual Feature Nets modelling the features Coffee, Payment and Milk.

replacing the place COFFEE with the net obtained in the previous step. We now formally define refinement for places and transitions.

**Refinement of Feature Nets** Notions of refinement of Petri nets are well-known techniques [11] that help manage the complexity of large models. Based on these, we give a definition of refinement of Feature Petri Nets, which will be used to merge separately developed Feature Nets. Let $N_1 = (S_1, T_1, R, M_{01}, F, f_1)$ and $N_2 = (S_2, T_2, R, M_{02}, F, f_2)$ be two Feature Nets with $S_1 \cap S_2 = T_1 \cap T_2 = \emptyset$. Adapting Definition 2.4.1 of [11], we define the refinement of $N_1$ by $N_2$ by replacing a transition or a place.

**Definition 11 (Transition refinement).** Given the Feature Nets $N_1$ and $N_2$, a transition $t \in T_1$, and a relation $B \subseteq (S_1 \times T_2) \cup (T_2 \times S_1)$, the transition
refinement of $N_1$ by $N_2$ is the Feature Net $N = (S, T, R, M_0, F, f)$, where

\[ S = S_1 \cup S_2 \]
\[ R = (R_1 \setminus \{(s, t) \mid s \in S_1 \} \cup \{(t, s) \mid s \in S_1 \}) \cup R_2 \cup B \]
\[ T = (T_1 \cup T_2) \setminus \{t\} \]
\[ f = \{t' \mapsto f_1(t) \land f_2(t') \mid t' \in T_B\} \cup (f_1 \cup f_2) \mid (T \setminus T_B) \]
\[ M_0 = M_{0_1} \cup M_{0_2} \]
\[ T_B = \{t' \mid (t', s) \in B \lor (s, t') \in B\} \]

**Definition 12 (Place refinement).** Given the Feature Nets $N_1$ and $N_2$, a place $s \in S_1$, and a relation $B \subseteq (T_1 \times S_2) \cup (S_2 \times T_1)$, the place refinement of $N_1$ by $N_2$ is the Feature Net $N = (S, T, R, M_0, F, f)$, where

\[ S = (S_1 \cup S_2) \setminus \{s\} \]
\[ R = (R_1 \setminus \{(s, t) \mid t \in T_1\} \cup \{(t, s) \mid t \in T_1\}) \cup R_2 \cup B \]
\[ T = T_1 \cup T_2 \]
\[ M_0 = (M_{0_1} \cup M_{0_2}) \setminus \{s\} \]
\[ \text{and } f = (f_1 \cup f_2) \mid T. \]

In **Definition 11** $B$ relates places from $N_1$ and transitions in $N_2$ while in **Definition 12** $B$ relates transitions from $N_1$ and places in $N_2$. These relations are used to replace the arcs between nodes of $N_1$ and the refined node ($t$ or $s$) with arcs between nodes of $N_1$ and nodes of $N_2$.

Refinement entails replacing an abstract node with the FN model of the overall system. When the node being replaced is a transition, its application condition is used in conjunction with the application conditions of the transitions in $B$. We now show how the refinement operations are applied in the examples above to construct a model of an SPL from separate sub-models. First, a model with the two features Coffee and Payment is composed. We use the following assignment.

- $N_1$ is the FN modelling Payment (Fig. 3c)
- $N_2$ is the FN modelling Coffee (Fig. 3a)
- $t =$ PROCEED
- $B = \{(\text{PAID, BREW COFFEE}), (\text{SERVE, UNPAID})\}.$

Refining the transition $t$ using **Definition 11** produces the FN shown in Fig. 4. Subsequently, we can refine the Feature Net representing Milk to include the behaviour obtained in the previous step by using the assignment below.

- $N_1$ is the FN modelling Milk (Fig. 3b)
- $N_2$ is the FN obtained in the previous step (Fig. 4)
- $s =$ COFFEE
- $B = \{(\text{ADD MILK, READY}), (\text{READY, ADD MILK})\}.$

Following **Definition 12**, refining the place $s$ produces an FN representing the behaviour of a product line over the three features, as shown in Fig. 5.
4.2 Compositional modelling using node fusion

This section introduces an alternative modular modelling technique based on the fusion of nodes from separate Feature Nets. In the first modelling stage, Feature Nets representing individual features are designed such that they also model their interaction with other features. Such interactions are modelled by common nodes, which can be seen as an interface to the net. Feature Nets are then composed by fusing their common nodes.

Unlike the refinement approach presented in the previous section, nets modelling individual subsystems now “know” part of the structure of the nets with which they interact. Previously this knowledge was provided by the $B$ relation upon composition. Now it is included earlier, when modelling the Feature Nets to be composed. This approach makes composition a simple matter of fusing the Feature Nets based on the union between their elements; the composition operation is commutative and associative.

**Definition 13 (Feature Net fusion).** Given two FNs $N = (S, T, R, M_0, F, f)$ and $N' = (S', T', R', M_0', F', f')$ such that $\forall s \in S \cap S': M_0(s) = M_0'(s)$ holds. Define Feature Net fusion $N \oplus N' = (S \cup S', T \cup T', R \cup R', M_0 \cup M_0', F \cup F', f^\dagger)$ where $f^\dagger = \{ t \mapsto f_1(t) \wedge f_2(t) \mid t \in T \cap T' \} \cup f_1 \upharpoonright (T \setminus T') \cup f_2 \upharpoonright (T' \setminus T)$.

Two Feature Nets can be fused by constructing a net based on the union of their elements. Places which appear in both nets must have matching markings; the application conditions of common transitions are joined by conjunction.

To illustrate a modelling technique based on Feature Net fusion, we consider the model for a *Payment* feature shown in Fig. 6. Its interaction with the rest of the system is specified explicitly by modelling the flow of tokens to and from concrete transitions which are part of the *Coffee* feature (Fig. 3a). Fusing these two Feature Nets via the common transitions BREW COFFEE and SERVE leads to the same composition that was reached using the refinement technique (Fig. 4).
To also include the Milk behaviour, we first construct a Feature Net similar to the one in Fig. 3b, only that the place labelled COFFEE has to be renamed to READY. Then, composing \((Coffee \oplus Payment) \oplus Milk\) results in the net shown in Fig. 5.

5 Behaviour preservation

We have shown two approaches to compose a Feature Net \(N\) over features \(F\) with a second net to obtain a new larger net \(N'\) over features \(F'\). A natural question arises: is the behaviour of \(N\) preserved in \(N'\)? We formalise two notions of preservation: (1) by considering the behaviour of \(N\) and \(N'\) when projected onto each feature selection \(FS \subseteq F\) (Definition 14); and (2) by considering traces of \(N\) and \(N'\) projected onto feature selections from \(FS' \subseteq F'\) after filtering out elements of traces not involving a transition from \(N\) (Definition 15).

The major difference between these two notions is that the first one considers only the feature selections relevant for the original Feature Net, ignoring feature selections relevant for the extension. The second notion also takes account of such feature selections, but takes care to ignore the behaviour that does not involve transitions of the original Feature Net.

Recall the two composition examples presented in this paper: adding Payment to the Coffee Feature Net, and adding Milk to the resulting net. In both cases we want the composition to preserve behaviour. That is, if we take the net modelling
Coffee and Payment and block all transitions related to the Payment feature, the result should be a model of a machine which dispenses coffee for free. Similarly, if we take the net that was extended with the Milk option and disregard the transitions conditioned on Milk, then its behaviour should be that of a machine which takes a payment and serves plain black coffee.

We first formalise the preservation of behaviour, and then analyse the composition examples with respect to their ability to preserve behaviour in more detail.

Definition 9 specified how to project a Feature Net over a set of features $FS$. We say that an extension of a Feature Net $N = (S,T,R,F,f)$ to a new net $N' = (S',T',R',F',f')$ preserves projections if projecting onto any feature selection from $F$ before and after the extension yields the same behaviour.

**Definition 14 (Preservation of projection).** The extension of $N$ to $N'$ preserves projections iff $\forall FS \subseteq F : \text{Beh}(N \downarrow FS) = \text{Beh}(N' \downarrow FS)$.

Recall the composition of the nets modelling Coffee and Payment, and the composition of the resulting net with the net modelling Milk. Let $C$, $M$, $P$, $CP$, and $CPM$, be the nets involved in these operations, presented in Fig. 3a, b, c, Fig. 4, and Fig. 5, respectively. Extending $CP$ with $M$ preserves projection, because $\forall FS \subseteq \{\text{Coffee, Milk}\} : \text{Beh}(CP \downarrow FS) = \text{Beh}(CPM \downarrow FS)$. However, the extension of $C$ with the net $P$ does not preserve projection. This is because $\{\text{wait,full}\} \xrightarrow{\text{BREW COFFEE}} \{\text{ready,full,refillable}\}$ is a trace of $C \downarrow \text{Coffee}$, but not of $CP \downarrow \text{Coffee}$, as the latter requires a token in the place paid in order to fire the transition BREW COFFEE. Because projecting does not restore the structure the net had before the extension—the places UNPAID and PAID are not eliminated by the projection—its behaviour is different.

Before formalising preservation based on restricted traces, we define the restriction of traces of a Feature Net to a given set of transitions $Ts$.

**Definition 15 (Trace restriction).** Given a Feature Net $N = (S,T,R,F,f)$ and a set of transition $Ts$, the behaviour of $N$ restricted to $Ts$ is defined as

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig6}
\caption{The highlighted transitions overlap with the transitions of the same name in Fig. 3a. Merging the two nets is achieved by fusing these transitions.}
\end{figure}
Beh($N$)↾$Ts$ = \{tr ↾ Ts | tr ∈ Beh($N$)}}, where the restriction of a single trace to Ts is the sequence defined below.

$$M ↾ Ts = ε, \quad (M \xrightarrow{t} tr) ↾ Ts = \begin{cases} t \cdot (tr ↾ Ts) & \text{if } t ∈ Ts \\ tr ↾ Ts & \text{otherwise.} \end{cases}$$

When restricting a trace to a set of transitions $Ts$ we obtain a sequence of transitions after filtering out the elements not in $Ts$. To check whether the extension of a Feature Net $N = (S, T, R, F, f)$ to a new Feature Net $N’ = (S’, T’, R’, F’, f’)$ preserves its restricted traces, we compare the traces before and after the extension. We consider feature selections that use features from the composed net, and compare only the transitions that $N$ can perform.

**Definition 16 (Preservation of restricted traces).** The extension of $N$ to $N’$ preserves restricted traces iff $\forall FS’ ⊆ F’ : Beh(N ↓ FS’) = Beh(N’ ↓ FS’) ↾ T$.

The extension of the net $C$ modelling Coffee with the net $P$ modelling Payment net from Fig. 4 does not preserve projections, as seen before. When considering the restricted traces for projections over the full set of features $F’$, we reach the same conclusion: the behaviour is still not preserved. The projection $C ↓ \{Coffee, Payment\}$, when restricted to the transitions of $C$, cannot perform the transition BREW COFFEE anymore, because the place UNPAID will never contain a token. However, $CP ↓ \{Coffee, Payment\}$ can brew coffee after accepting the payment. This shows that, when using a transition refinement, extra attention needs to be paid to avoid discarding desired behaviour. Below we show how to adapt the net modelling the Payment feature to preserve the behaviour based on restricted traces after composition. As future work we plan to explore the use of application conditions on places or on arcs between places and transitions to simplify the design of behaviour preserving transformations.

We present in Fig. 7 the composition of the net $C$ modelling the feature Coffee with an alternative net $P’$ modelling Payment. The only difference is the transition SKIP, which overrides the need for paying when the feature Payment is

![Fig. 7: Composition of the Coffee net with a variation of the Payment net.](image-url)
not available. Let $CP'$ be the net from Fig. 7. The projection $CP' \downarrow \text{Coffee}$ now yields the same traces as $C \downarrow \text{Coffee} (= C)$, after filtering all transitions not used by $C$, since tokens can flow from UNPAID to PAID with and without the Payment feature. Observe that the projection is still not preserved, since the transition SKIP is present in the traces of $CP' \downarrow \text{Coffee}$, but not in the traces of $C \downarrow \text{Coffee}$.

6 Related Work

Our research relates to a number of areas, specifically Petri net based formalisms, and the behavioural specification of software product lines. We highlight the most relevant works in these fields.

A range of Petri net extensions based on a modified transition occurrence rule have been proposed [1, 5, 15]. Unlike our approach, these formalisms generally exceed the expressive power of Petri nets. Consequently, properties such as reachability, boundedness and liveness are often not decidable for these extensions, and they generally lack the full range of mathematical tools available to analyse normal Petri nets. Inhibitor arc Petri nets [1] can test whether a place is empty by conditioning transitions on the absence of tokens. By modelling individual features as places, the presence or absence of tokens could represent whether a feature is on or off. An application condition could be encoded by including feature places in the pre-sets of transitions, thereby conditioning its firing on the presence or absence of features. Compared to our approach, this entails a more complex net, with unclear boundaries between the functional and structural aspects. Conditional Petri nets [5] associate a transition with a formal language over transitions. Extending the classical occurrence rule, a transition is enabled only if the sequence of transitions that occurred in the past is in that language. An FN could be encoded as a conditional Petri net by encoding application conditions in a language over the alphabet of transitions. In Open Petri Nets [3], places designated as open represent an interface towards the environment. Open nets are composed by fusing common open places, and the composition operation is shown to preserve behaviour with respect to an inverse decomposition operation. The main difference to our model, apart from the presence of application conditions, is that we also allow transitions to guide the composition.

Various formalisms have been adopted for specifying the behaviour of SPL, with the aim of providing a basis for analysis and verification of such models. UML activity diagrams have been used to model the behaviour of SPL by superimposing several such diagrams in a single model [6]. Attached to the activity diagram’s elements are “presence expressions,” which are similar to application conditions. Compared to activity diagrams, Petri nets have a stronger formal foundation, with a larger spectrum of analysis and verification techniques, although, several studies have expressed the semantics of UML diagram using Petri nets (e.g. [9]). Gruler et al. extended Milner’s CCS with a product line variant operator that allows an alternative choice between two processes [12]. This calculus, referred to as PL-CCS, includes information about variability: by defining dependencies between features, one can control the set of valid configurations.
Variability is often modelled using transition systems enhanced with product-related information. Modal transition systems (MTS) [13] allow optional transitions, lending themselves as a tool for modelling a set of behaviours at once [10]. Generalised extended MTS [8] introduce cardinality-based variability operators and propose to use temporal logic formulas to associate related variation points. Asirelli et al. encode MTS using propositional deontic logic formulas [2]. Modal I/O automata [14] are a behavioural formalism for describing the variability of components based on MTS and I/O automata. Mechanisms for component composition are provided to support a product line theory. These approaches do not relate behaviour to elements of a structural variability model. Featured transition systems (FTS) [4] are an extension of labeled transition systems. Similar to Feature Nets, transitions are explicitly labeled with respect to a feature model, and a feature selection determines the subset of active transitions. In FTS, transitions are mapped to single features. Transition priorities are used to deal with undesired non-determinism when selecting from transitions associated to different features. With application conditions, priorities are no longer required because we can negate the features in other transitions to obtain the same effect.

7 Conclusion and Future Work

This paper introduces a modular framework for modelling systems with a high degree of variability, addressing an important challenge in software product line engineering. The modelling formalism used is Feature Nets, a lightweight Petri net extension in which the transition firing is conditional on the presence of certain features through application conditions. We present two approaches to composing behavioural models from separately engineered models of individual features. Two correctness criteria for such compositions are also presented.

Feature Nets capture the behaviour of entire product lines in a single, concise model, opening the way for efficient analysis and verification. We will follow this direction in future work, applying model checking techniques to our models and studying the question of verification. In addition, sufficient conditions that guarantee the preservation of behaviour through composition will be investigated.

References


Appendix H

A Semantics for Context-oriented Programming with Layers

The paper “A Semantics for Context-oriented Programming with Layers” follows.
A Semantics for Context-oriented Programming with Layers

Dave Clarke Ilya Sergey
Katholieke Universiteit Leuven, Heverlee, Belgium
{dave.clarke,ilya.sergey}@cs.kuleuven.be

Abstract
Context-oriented programming (COP) is a new programming approach whereby the context in which expressions evaluate can be adapted as a program runs. COP provides a degree of flexibility beyond object-oriented programming, while arguably retaining more modularity and structure than aspect-oriented programming. Although many languages exploring the context-oriented approach exist, to our knowledge no formal type-sound dynamic semantics of these languages exists. We address this shortcoming by providing a concise syntax-based formal semantics for context-oriented programming with layers, as witnessed by ContextL, ContextJ*, and other languages. Our language is based on Featherweight Java extended with layers and scoped layer activation and deactivation. As layers may introduce methods not appearing in classes, we also give a static type system that ensures that no program gets stuck (i.e., there exists a binding for each dispatched method call).

1. Introduction
Context-oriented programming is a new programming approach whereby the context in which expressions evaluate can be adapted as a program runs. It can be characterized as possessing the following facets:

Context-dependent evaluation The interpretation of programming language statements (e.g., method dispatch) depends upon the context in which they are evaluated;

Explicit context The notion of context is an explicit concept in the programming language; and

Context manipulation Context may be explicitly manipulated during the execution of the program.

In the approaches to context-oriented programming based on layers, as witnessed by ContextL, ContextJ*, and other languages [6, 9, 16, 2], context is explicitly represented as named layers. During program evaluation these layers can be activated and deactivated using the with() and without() scoping constructs, respectively.

The example in Figure 1 illustrates the ideas. Layer Logging alters the behaviour of the Actor method act(). It prints out a logging message and delegates to the initial act() method via the proceed() statement. The original act() method deactivates the Logging layer to avoid logging of the stealth() method call.

The paper makes the following contributions:

• A concise, syntax-directed dynamic semantics for a context-oriented programming language. The semantics are presented as an extension to Featherweight Java (without inheritance or subtyping), by adding layers, layer activation, layer deactivation, and layer delegation.

• A static type system which ensures that a well-typed binding exists for every dispatched method.

2. ContextFJ
ContextFJ (Figure 2) extends Featherweight Java [7] without inheritance with constructs for expressing layer-based context-oriented programming.
Class Definition
\[ C ::= \text{class } C\{\overline{C}; K; M\} \]

Constructor
\[ K ::= C(\overline{C})\{\text{this.}\overline{f} = \overline{f};\} \]

Method Definition
\[ M ::= C[\Psi] \quad \text{context.} \]

Finally, types of methods provided by a layer have the form \( C \rightarrow C \).

Next we define a collection of named layers. Each layer consists of a name and a map from tuples of a method name and a class name, \((m, C)\), to a method definition. The layer name is used to activate and deactivate the layer.

Terms consist of variables and field lookup, as usual. Method invocation is generalized slightly to the form \( t.m_l(\overline{t}) \), where label \( L \) is a set of so-called excluded layers. When dispatching \( m \), a binding from a layer in set \( L \) cannot be selected. Note that the programmer would write \( t.m(\overline{t}) \), as normal, which corresponds to \( t.m_\emptyset(\overline{t}) \) in the formalism.

Three new terms have been added to the language to deal with layer activation and deactivation, and for delegating to an earlier activated layer. The term \( \text{with}(l)t \) activates layer \( l \) for the duration of the evaluation of expression \( t \). The term \( \text{without}(l)t \) deactivates layer \( l \) for \( t \). The expression \( \text{proceed}(\overline{t}) \) delegates the current call to a previously activated layer containing the current method, ignoring all activations of the layer, which it was invoked from. proceed may only appear in the body of methods appearing in a layer.

The most general form of the method type is as follows, though this form is not always used in its full generality:

\[ MT ::= (m, C_0) \rightarrow [\Psi]\overline{C} \rightarrow C \bullet L \]

where

- \( m \) is a method’s name; \( C_0 \) is a receiver class type; \( \overline{C} \) are parameter types; and \( C \) is the result type;
- \( L \) is the set of excluded layers, namely the layers in which this method cannot be dispatched to. That is, a binding for method \((m, C_0)\) cannot be taken from some layer in \( L \). These layers are the ones excluded by \( \text{without}(\_\_\_) \) or \( \text{without}(\_\_\_\_\_) \) already visited when doing delegation via proceed.
- \( \Psi \) is a set of method assumptions, namely the methods used by this method, excluding methods appearing in classes. These ‘missing’ methods must be provided by some active layer when the method is invoked. These method assumptions have the form: \((m, C_0) \rightarrow \overline{C} \rightarrow C \bullet L\), which states that a method \( m \) of type \( \overline{C} \rightarrow C \) from class \( C_0 \) is called within the given method. \( L \) indicates the layers excluded for the \( m \) method lookup. In other words, all with(l) statements, such that method \( m \) is defined for class \( C_0 \) in \( l \) will be ignored for \( l \in L \) during such a lookup.

Classes and constructors are as in Featherweight Java, although all inheritance related constructions have been removed. Methods have an additional annotation indicating which methods are invoked within the method and not defined in some appropriate class, and thus need to be satisfied by some layer. These will be explained in detail in a moment.
3. Dynamic Semantics

We introduce the necessary auxiliary notions to define the dynamic semantics. The standard notion of evaluation context [14], which is an expression with a single hole in it, is used to define the reduction rules by capturing the notion of the 'next subterm to be reduced.' We write $E[l]$ for the ordinary term obtained by filling the hole in $E$ with $t$.

The following collects the methods, $(m, C)$ pairs, defined in a layer:

**Definition 1** (Layer domain). Given layer $l \{ \overline{B} \}$. Define:

$$\text{dom}(l) = \{(m, C_0) \mid (m, C_0) \mapsto M \in \overline{B}\}.$$

The following function computes the methods available in an evaluation context (at the hole), accounting for methods excluded by without($l$) statements.

**Definition 2** (Bound Methods).

$$BM_L(\{\}) = \emptyset$$

$$BM_L(E[[.,f]])$$

$$BM_L(E[[.,m(T)])$$

$$BM_L(E[v.m(\overline{\pi}),T)])$$

$$BM_L(E[\text{new}\ C(\overline{\pi}),T)])$$

$$BM_L(E[\text{with}(l)[]]) = \begin{cases} BM_L(E), & \text{if } l \in L \\ BM_L(E) \cup \text{dom}(l), & \text{otherwise} \end{cases}$$

$$BM_L(E[\text{without}(l)[]]) = BM_{L \setminus (l)}(E)$$

The following function computes the layers excluded by without(-) in some evaluation context:

**Definition 3** (Excluded Layers).

$$XL(\{\}) = \emptyset$$

$$XL(E[[.,f]])$$

$$XL(E[[.,m(T)])$$

$$XL(E[v.m(\overline{\pi}),T)])$$

$$XL(E[\text{new}\ C(\overline{\pi}),T)])$$

$$XL(E[\text{with}(l)[]]) = \{l\} \cup XL(E)$$

$$XL(E[\text{without}(l)[]]) = \{l\} \cup XL(E)$$

It’s important to note, that the evaluation semantics we present is syntax-directed, i.e., the order of layer activations is defined by the structure of the reduction context surrounding term to be reduced. The innermost with(-) or without(-) statement takes effect before outer ones. Any with(-) or without(-) statements occurring in method bodies are analyzed only after the body is substituted. So, at the moment of method invocation, the set of activated layers of the appropriate context is fixed. The lookup functions for methods in both layers and classes and for fields are given in Figure 3.

Figure 4 contains the reduction rules for ContextFJ. Rules (E-With) and (E-Without) discard a layer activation or deactivation when the wrapped expression finishes evaluating. Rule (E-ProjNew) is field access. Both rules (E-InvkLayer) and (E-InvkClass) handle method calls. Rule (E-InvkLayer) finds the first available layer, $l$, containing a binding for method $m$ of class $C$, from the inside out, which is not excluded by a without(-) clause or the set $L$ of already visited layers. In addition to substituting the arguments and this, the call to proceed within the method body $t$ is replaced with a call to the same method, except that dispatching the method will skip the present layer $l$ as well as layers in $L$. This is how proceed is modelled: it invokes the first layer not excluded; this layer is then excluded by subsequent calls to proceed. Rule (E-InvkClass) states that if the method is not found in a layer, then it is selected from the appropriate class body.

4. Static Semantics

Before presenting the typing rules, we present some auxiliary definitions in Figure 5. These definitions rely on the following notion of assumption dominance, which captures that if we are able to satisfy a ‘greater’ assumption set, then we are able to satisfy ‘lesser’ one also.

**Definition 4** (Assumptions dominance (≤)).

$$\Phi \preceq \Psi \iff \forall (m, C_0) \mapsto C \cdot L \in \Phi.$$  

$$\exists (m, C_0) \mapsto C \cdot L' \in \Psi \text{ s.t } L \subseteq L'$$

The key novelty here is in the definitions of mtype. First lookup is performed in the class of the receiver of a method, and then in the assumption set.

In Figure 6 we describe typing rules for methods, layers and classes. All other typing rules, such as (T-Var), (T-Field) and (T-New) are straightforward adaptions of Featherweight Java [7], and thus omitted.

Rules (T-Class) and (M OK in C), for methods appearing in classes, are standard. The rule (T-Layer) en-
Exercise typing is based on the standard typing environment needs to work harder to find a suitable method (this included layer set for this method to grow, meaning the environment needs to work harder to find a suitable method (this is sound).

The rule for with(−) requires a few auxiliary notions. provides(l) gives the types of the methods defined in layer l and requires(l) gives the assumptions of all methods defined in l.

**Definition 5.** provides(l) and requires(l) Given layer l \( \{B\} \).

Define:

<table>
<thead>
<tr>
<th>provides(l)</th>
<th>requires(l)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (m, C_0) \mapsto C \mapsto D ) { ( m, C_0 \mapsto D { m, C } \mapsto C { \varphi } { x } \in B } )</td>
<td>( \cup { \varphi \mid (m, C_0) \mapsto D { m, C_0 } } { m, C } \mapsto C { \varphi } { x } \in B } )</td>
</tr>
</tbody>
</table>

**Figure 4.** Reduction rules

| (overlaid(m, C_0, [Φ | C \mapsto C)) | (mtype-class) | (mtype-assumptions) |
|----------------|--------------|-------------------|
| \( \text{mtype}(m, C_0, \emptyset) = [\varphi | C \mapsto C) \) \ implies \( C = D, C = D \) and \( \Phi \times \Psi \) \| \( CT(C) = \text{class C \{ C \mapsto K \{ D \}} \) \| \( m \text{ is not defined in C} \) \| \( m \mapsto E \mapsto E \mapsto E \mapsto E \mapsto L \in \Psi \) \| \( \text{mtype}(m, C, \Psi) = E \mapsto E \mapsto L \) \| \( (m, C) \mapsto [\Psi | C \mapsto C \mapsto L] \mapsto (m, C_0) \mapsto C \mapsto C \) \|

The auxiliary function \( \parallel \cdot \parallel \) erases all assumptions and excluded layers from a method type:

\( \parallel (m, C_0) \mapsto \{ m, C \} \mapsto C \mapsto C \mapsto L \parallel := (m, C_0) \mapsto C \mapsto C \)

Now in the rule (T-\text{With}) the wrapped expression \( t \) is typed in an environment where the provided methods of the layer, provides(l), are available (Φ). In addition, the required methods must appear in the context surrounding the entire expression (Ψ) which provides all of the required methods used by the layer and the unsatisfied assumptions of expression \( t \).

The rule (T-\text{Without}) modifies the assumption set to exclude all methods occurring in the excluded layer \( l \) from \( C \). This is achieved using the following definition:

**Definition 6.** Layer exclusion from assumptions

\[
\begin{align*}
\epsilon|l & = \epsilon \\
((m, C) \mapsto X \mapsto L, \Psi)|l & = \\
& \begin{cases}
(m, C) \mapsto X \mapsto L \cup \{ l \}, (\Psi)(l) & \text{if } (m, C) \in \text{dom}(l) \\
(m, C) \mapsto X \mapsto L, (\Psi)(l) & \text{otherwise}
\end{cases}
\end{align*}
\]
5. Discussion

The present approach groups the assumptions/requirements of a layer at the granularity of a layer. This means that when a method from a layer is used, all assumptions of the layer need to be satisfied for the program to type check. Indeed, simply activating a layer without even using it imposes such a requirement. A more fine-grained approach would involve collecting the method calls which are actually made within an expression and recording only those per method and only requiring that the methods actually used from an active layer need to have an appropriate binding.

The course-grained approach is more modular, in the sense that small changes to code do not necessitate changes to the interface of layers. The fine-grained approach is more precise and does not require activating layers which will not be used just to satisfy the type checker.

We expect that much of our annotations can be inferred, resulting in a significantly simpler system for a programmer to use.

6. Related Work

Our dynamic semantics draws heavily from the semantics of dynamic binding [8, 11]. These models do not have without(−). Furthermore, their type system does not exclude failure of dynamic binding due to a missing binding. The semantics of proceed adapts ideas from Schippers et al [15], although our approach does not use the notions of heap and stack, and we provide a type system.

Our type system heavily drew inspiration from Contextual Modal Type Theory [12], which had neither proceed nor without(−). In particular, our method types \([\Psi][C] \rightarrow C\) resemble contextual modal types.

The calculus of evolving objects [3] gives a foundational account of highly dynamic systems. Their setting is quite different from ours and our dynamic semantics is significantly simpler. Quite a few programming languages have concepts similar to layers and dynamic binding. These include Haskell’s implicits [10], Clojure [5], Scala’s implicits [13], and Groovy’s [9] and Objective C’s [4] categories. Due to space limitations we cannot give a detailed comparison.

7. Conclusion and Future Work

This paper presented the first type-sound semantic for a context-oriented language with layers. The semantics is based on Featherweight Java and the type system ensures that all dispatched methods find an appropriate binding either in some layer or in the original collection of classes.
Directions for future work include extending the core language to incorporate inheritance and subtyping, including inheritance between layers, and considering adding dependencies between the layers, such as that one layer requires another (kind of) layer to be present, or that two layers can never be active at the same time.

One of the main problems to address is ambiguity, which occurs because there may be multiple candidate methods available: does a less specific method in a closer layer take precedence over a more specific method?

A. Contextλ

For reference and comparison, Figure 8 presents the dynamic semantics for context-oriented λ-calculus without proceed. The combination of proceed and closures causes problems, and is addressed by Clarke et al. [1].

References

Figure 8. Contextλ
Appendix I

How should context-escaping closures proceed?

The paper “How should context-escaping closures proceed?” [29] follows.
How should context-escaping closures proceed?

Dave Clarke∗
DistriNet
Katholieke Universiteit Leuven
B-3001 Heverlee, Belgium
dave.clarke@cs.kuleuven.be

Pascal Costanza†
Programming Technology Lab
Vrije Universiteit Brussel
B-1050 Brussels, Belgium
pascal.costanza@vub.ac.be

Éric Tanter‡
PLEIAD Laboratory
Computer Science Dept (DCC)
University of Chile, Santiago, Chile
etanter@dcc.uchile.cl

Abstract
Context-oriented programming treats execution context explicitly and provides means for context-dependent adaptation at runtime. This is achieved, for example, using dynamic layer activation and contextual dispatch, where the context consists of a layer environment of a stack of active layers. Layers can adapt existing behaviour using proceed to access earlier activated layers. A problem arises when a call to proceed is made from within a closure that escapes the layer environment in which it was defined. It is not clear how to interpret proceed when the closure is subsequently applied in a different environment, because the layers it implicitly refers to (such as the original layer and/or the remaining layers) may no longer be active. In this paper, we describe this problem in detail and present some approaches for dealing with it, though ultimately we leave the question open.

1. Introduction
An approach to context-oriented programming as embodied by ContextL and others [4] enables programmers to dynamically adapt or replace code at run-time by activating layers which intercept dispatched methods. Code is organised into modules called layers which cross-cut existing classes. Previously activated methods can be invoked using a command proceed, and thus layers can be used to implement a variant of before, after, and around advice. proceed provides delegation within the stack of layers, analogous to super calls in object-oriented languages, ‘proceed’ in aspect languages, or delegation in general, in that proceed goes through the remaining layers, though new method invocations go to the most recently activated layer.

The structure of the layer environment changes dynamically as layers are activated and deactivated. This can cause problems in higher-order languages, because a closure containing a call to proceed can escape the layer environment in which it originates (and makes sense) and be applied in a new layer environment. It is unclear what proceed means, nor what it should mean, in the new environment.

1.1 The proceed Problem
So far, there seem to be two basic approaches for providing semantics for proceed, and in the following we use Context* and ContextL as two representative context-oriented languages to illustrate the two approaches.

Context* [4], which is based on Java, implements its extension of method dispatch as a search through the current stack of active layers. This suggests that proceed is the continuation of a search. Currently, this is not a problem because like Java, Context* does not have first-class closures. Therefore, this search can always continue in the current layer environment, at the position in the stack of active layers where it found the currently executing method. However, if Context* had closures, and invocations of proceed could be captured in such closures, these semantics suggest that the search has to continue in an as-yet unspecified stack of layers, and it is unclear which layers to use: the original layers, the currently active layers, or some combination thereof.

ContextL [3] is based on Common Lisp, which provides lexical closures. However, in contrast to Context*, an invocation of proceed is not the continuation of a search, due to the semantics of generic functions in the Common Lisp Object System (CLOS, [1]), on which ContextL is based. Generic function invocation, CLOS’s equivalent to object-oriented message sending, is performed in three steps.

Java’s recent incarnation of closure-like constructs are still not closures.

In CLOS, proceed is actually call-next-method.

We have simplified the actual semantics somewhat for clarity here.
1. A set of applicable methods is determined, based on
the classes the candidate methods are defined for. (In
ContextL, the current stack of active layers is also taken
into account in this step.)

2. The set of applicable methods is sorted according to
specificity. (In ContextL, more recently activated layers
render their methods more specific.)

3. The most specific method is invoked, and each method
can invoke proceed to call the next most specific method.

The intuition here is that proceed merely navigates through
a precomputed, ordered set of applicable methods, just like
super calls in other object-oriented languages merely invoke
methods in the statically determined direct superclass. A
change in the current stack of active layers does not affect
this set of applicable methods anymore: Once it is deter-
mined, it will remain fixed. This gives a reasonable seman-
tics for proceed, even when captured in closures.

However, the fact that ContextL seems to have reasonable
semantics here is actually an accident, due to the reuse of
CLOS semantics. In fact, ContextL still fails to reinstate the
original stack of active layers when proceed is invoked. So
neither the layers that invoked the method in which proceed
was captured, nor the layers that provide the definitions of
the methods that will be invoked by proceed, may actually be
active when that proceed is eventually executed. However,
especially the already executed definitions may rely on the
presence of their own layers in such a situation.

1.2 How to proceed?

To summarize, this paper tries to pose, and partially answer,
the following questions:

- Should the invocation of a proceed find a method either
  in the stack of layers that was active when it was captured
  in a closure, in the stack of currently active layers, or in a
  combination thereof?

- Should the method found by proceed then be executed in
  a dynamic environment with the original stack of layers,
  with the current stack of layers, or a combination thereof?

- If the layer environments should be combined in these
cases, what should the composition look like?

We also briefly discuss ideas for advanced language con-
structs to influence the composition of layer environments.

2. A Small COP Language

In order to precisely describe the problem and potential
solutions, we adopt the formal calculus Contextλ developed
by Clarke and Sergey [2], which derives from the semantics
of dynamic binding [5, 6]. Contextλ extends the lambda
calculus with layer definitions, layer activation (with(l)e)
and deactivation (without(l)e), and contextual dispatch. As
layer deactivation is not required for our story, we remove
without, resulting in a considerably simpler calculus, which
we call Contextλ−. Its syntax is as follows:

\[
\begin{align*}
P & ::= \Delta; e \\
\Delta & ::= \Gamma = \bar{B} \\
B & ::= \bar{p} = \bar{e} \\
v & ::= \lambda x.e \mid x \\
e & ::= v \mid p \mid e \ e \mid \text{with}(l)e \\
E[] & ::= [ ] \mid E[\ ] e \mid E[v \ ] \mid E[\text{with}(l)[ ]]
\end{align*}
\]

A program \(P\) consists of a collection of layers \(\Delta\) followed by
a single expression \(e\). Each layer is a mapping from the layer
name \(l\) to a set of bindings of parameters \(p\) to expressions
\(e\). Values \(v\)—the results of successful reductions—consist
of lambda abstractions and variables. In closed expressions,
the only values are lambda abstractions \(\lambda x.e\), though for
examples we also use integers.

Expressions \(e\) consist of values \(v\), parameters \(p\), function
application \(e \ e\), and context activation \(\text{with}(l)e\). Parameters
\(p\) are dynamically scoped and dynamically bound to a
definition in some layer—called contextual dispatch or just
dispatch. Variables, however, are lexically scoped. The con-
struct \(\text{with}(l)e\) activates layer \(l\) for the (dynamic) duration of
the reduction of expression \(e\), i.e., until \(e\) reduces to a value.

Finally, \(E[\ ]\) gives the syntax of evaluation contexts,
which are expressions with a single hole. \(E\) can be seen as
holding the context surrounding an expression, in particular,
it contains a stack of \(\text{with}(l)e\) expressions denoting the acti-
vated layers. For example, \(E[\ ] = \text{with}(l_1)(\text{with}(l_2)([ [ ] ]))\)
can be seen as the stack of layers \(l_1 : l_2\), where \(l_2\) is the in-
nernost layer. Dispatch operates by searching for a binding
of a parameter moving outwards from the innermost layer.
This stack of layers is called the layer environment.

The reduction rules use the following function to deter-
mine the set of parameters active in the layer environment,
so which layer to dispatch to for a particular parameter:

\[
\text{Bound parameters.}
\]

\[
\begin{align*}
\text{BP}([ ]) & = \emptyset \\
\text{BP}(E[\ ] e) & = \text{BP}(E) \\
\text{BP}(E[v \ ] \ ] ] & = \text{BP}(E) \\
\text{BP}(E[\text{with}(l)[ ]]) & = \text{BP}(E) \cup \text{dom}(\Delta(l))
\end{align*}
\]

The reduction rules for Contextλ− are as follows:

\[
\begin{align*}
E[(\lambda x.e) \ v] & \rightarrow E[e[v / x]] \quad (\beta) \\
E[\text{with}(l) v] & \rightarrow E[v] \quad (\text{Esc}) \\
E[\text{with}(l)E'[p]] & \rightarrow E'[\text{with}(l)e'[e]] \quad (\text{DISP}) \\
& \quad \text{if } p \notin \text{BP}(E') \\
& \quad \text{and } e = \Delta(l)(p)
\end{align*}
\]

Evaluation is call-by-value. The first rule is the standard \(\beta-
\)reduction rule. We use \(e[v / x]\) to denote the substitution of
\(x\) for \(e'\) in \(e\). The second rule states that when an expression
finishes evaluating, the surrounding layer activation has no further effect and is removed. The third rule covers the case of looking up a parameter in some surrounding layer. Here $E'[ ]$ is the (inner) part of the evaluation context which does not contain a binding for parameter $p$, denoted by $p \notin \text{BP}(E')$. Thus, $l$ is the first layer, from inside to outside, containing a binding for $p$.

A program $P = \Delta; e$ evaluates by evaluating $e$, where $\Delta$ provides the layers.

3. The Problem

The question we wish to raise and study here is what happens when proceed is added to Context$\land$. The intention of proceed is to delegate the parameter dispatch to the next surrounding layer. When combined with closures, proceed may be invoked in the body of a closure that has escaped from its definition context and is applied in another layer environment. It is then not clear which layer proceed refers to.

Following is an informal account of how proceed behaves, firstly when no closures are present, and then we indicate what the potential problem is when closures are present.

Let

\[
\Delta_0 = \begin{cases} 
    l_1 = \{ p = 10 \}, \\
    l_2 = \{ p = \text{proceed} + 5 \}
\end{cases}
\]

The next example illustrates how proceed works, in this case by marking proceed to indicate which parameter to dispatch to, and which layer to begin the search (we use $\text{with}(l_1, l_2, l_3)[ ]$ to denote $\text{with}(l_1)\text{with}(l_2)\text{with}(l_3)[ ]$):

\[
\text{with}(l_1, l_2)(p + 3) \rightarrow \text{with}(l_1, l_2)((\text{proceed}_{l_1} + 5) + 3) \rightarrow \text{with}(l_1, l_2)((10 + 5) + 3) \rightarrow^* 18
\]

This naive approach falls over when a closure containing a call to proceed escapes the layer environment in which it was created. In this example we use:

\[
\Delta_1 = \begin{cases} 
    l_1 = \{ p = \lambda x.1 \}, \\
    l_2 = \{ p = \lambda x.\text{proceed} x \}, \\
    l_3 = \{ p = \lambda x.3 \}
\end{cases}
\]

\[\lambda f.\text{with}(l_3)(f\ 10)\ (\text{with}(l_1, l_2)p) \rightarrow (\lambda f.\text{with}(l_3)(f\ 10)) \ (\text{with}(l_1, l_2)p) \]

\[\rightarrow (\lambda f.\text{with}(l_3)(f\ 10)) \ (\text{with}(l_1, l_2)\lambda x.\text{proceed}_{l_1} p\ x) \rightarrow^* (\lambda f.\text{with}(l_3)(f\ 10)) \ (\lambda x.\text{proceed}_{l_1} p\ x) \]

\[\rightarrow \text{with}(l_3)((\lambda x.\text{proceed}_{l_1} p\ x)\ 10) \rightarrow \text{with}(l_3)(\lambda x.\text{proceed}_{l_1} p\ x)\ 10 \rightarrow \text{with}(l_3)(\lambda x.3)\ 10 \rightarrow \text{with}(l_3)1 \rightarrow 1\]

Note that although Clarke & Sergey present two languages—Context$\land$ with layers and closures, but not proceed; and Context$\land$ with layers, classes and proceed, but no closures [5]—they do not consider this issue.

4. Semantics of proceed

We now present a few approaches to giving semantics to proceed. Without closures, they are all equivalent. For our example with $\Delta_0$, the same program always reduces to 18. They do however differ when involving escaping closures.

4.1 Capture and Reinstate Layer Environment

In this semantics, whenever a parameter is dispatched the layer environment above the layer in which the binding is found is captured; proceed is replaced by a call to $p$ in the captured layer environment. The reduction rule is:

\[E[\text{with}(l)E'[p]] \rightarrow E[\text{with}(l)E'[e[\text{with}(l)p / \text{proceed}] \text{if } p \notin \text{BP}(E') \text{ and } e = \Delta(l)(p)] (\text{DISP2})\]

where $\text{with}(l)[ ]$, which extracts the surrounding layer environment from an evaluation context, is defined as:

\[\text{with}(l)[ ] = \begin{cases} 
    [] & \text{if } l = 0 \\
    \text{with}(l)[ ] & \text{if } l = \text{proceed} + 5 \\
    \text{with}(l_1)[ ] & \text{if } l = \lambda x.1 \\
    \text{with}(l_2)[ ] & \text{if } l = \lambda x.\text{proceed} x \\
    \text{with}(l_3)[ ] & \text{if } l = \lambda x.3
\end{cases}\]

The following example reduction is in the context of $\Delta_1$:

\[
(\lambda f.\text{with}(l_3)(f\ 10)) \ (\text{with}(l_1, l_2)p) \rightarrow (\lambda f.\text{with}(l_3)(f\ 10)) \ (\text{with}(l_1, l_2)\lambda x.\text{proceed}_{l_1} p\ x) \rightarrow (\lambda f.\text{with}(l_3)(f\ 10)) \ (\lambda x.\text{proceed}_{l_1} p\ x) \rightarrow \text{with}(l_3)((\lambda x.\text{proceed}_{l_1} p\ x)\ 10) \rightarrow \text{with}(l_3)(\lambda x.\text{proceed}_{l_1} p\ x)\ 10 \rightarrow \text{with}(l_3)(\lambda x.3)\ 10 \rightarrow \text{with}(l_3)1 \rightarrow 1
\]
Here, whenever proceed dispatches to another parameter, the context in which the evaluation occurs is (part of) the original, not the new context. So the parameter is not dynamically bound in the new context. For example, given layers:

\[
\Delta_2 = \begin{cases} 
  l_1 & = \{ p = q, q = \lambda x.4 \} \\
  l_2 & = \{ p = \lambda x.\text{proceed} \} \\
  l_3 & = \{ p = \lambda x.3, q = \lambda x.5 \} 
\end{cases}.
\]

the following reduction sequence illustrates that the environment associated with proceed remains longer than expected:

\[
\begin{align*}
(\lambda f.\text{with}(l_3)(f\ 10)) & \text{ with } (l_1, l_2)p \\
\longrightarrow & (\lambda f.\text{with}(l_3)(f\ 10)) \\
& \text{ with } (l_1, l_2)((\lambda x.\text{proceed} x)\{\text{with}(l_1)p/\text{proceed}\}) \\
\longrightarrow & (\lambda f.\text{with}(l_3)(f\ 10)) \text{ with } (l_1, l_2)(\lambda x.\text{with}(l_1)p\ x) \\
\longrightarrow & \ast (\lambda f.\text{with}(l_3)(f\ 10)) \text{ with } (l_1, l_2)(\lambda x.\text{with}(l_1)p\ x) \\
\longrightarrow & \ast \ast (\lambda f.\text{with}(l_3)(f\ 10)) \text{ with } (l_1, l_2)(\lambda x.\text{with}(l_1)p\ x) \\
\longrightarrow & \ast \ast \ast (\lambda f.\text{with}(l_3)(f\ 10)) \text{ with } (l_1, l_2)(\lambda x.\text{with}(l_1)p\ x) \\
\longrightarrow & \ast \ast \ast \ast (\lambda f.\text{with}(l_3)(f\ 10)) \text{ with } (l_1, l_2)(\lambda x.\text{with}(l_1)p\ x) \\
\end{align*}
\]

This may be considered wrong, because parameters are by definition dynamically scoped, so one should always use the most recent binding. So \(q\) should dispatch to layer \(l_3\). Alternatively, as the layer environment of the original dispatch was \((l_1, l_2)\), perhaps \(q\) should dispatch to \(l_1\).

### 4.2 Capture Layer Environment and Use It to Interpret proceed: Variant I

In this semantics, we capture the layer environment to interpret proceed, but do not reinstate the environment. In this semantics, when a parameter \(p\) is called in environment \(E_0\), proceed is replaced by a term \(p_{E_0}\) recording this information. Term \(p_{E_0}\) is evaluated in environment \(E\) by finding an appropriate binding in environment \(E[E_0][]\), and then continuing evaluation in \(E[\ ]\). (In comparison, the previous semantics would continue in environment \(E[\ E_0][]\).)

To model this, we add the following term to the language, and note that an ordinary parameter is modelled by \(p_E\):

\[
e := \cdots | p_E
\]

The modified rule for parameter dispatch is:

\[
E[p_{E_0}] \rightarrow E[e^{\{p_{E_1}/\text{proceed}\}}] \quad \text{if } E[E_0][] \equiv E_1[\text{with}(l)E_2][] \quad \text{and } p \notin \text{BP}(E_2) \quad \text{and } e = \Delta(l)(p) \\
\text{(Disp3)}
\]

Firstly, the environment \(E[\ ]\) is extended with \(E_0[\ ]\). An appropriate layer \(l\) is found containing parameter \(p\). In the binding of \(p\) in \(l\), proceed is interpreted as \(p_{E_0}\), where \(E_2\) is the remaining layer environment, above layer \(l\).

We now redo the previous examples. Firstly with \(\Delta_1\):

\[
\begin{align*}
(\lambda f.\text{with}(l_3)(f\ 10)) & \text{ with } (l_1, l_2)p \\
\longrightarrow & (\lambda f.\text{with}(l_3)(f\ 10)) \\
& \text{ with } (l_1, l_2)((\lambda x.\text{proceed} x)\{\text{with}(l_1)p/\text{proceed}\}) \\
\longrightarrow & (\lambda f.\text{with}(l_3)(f\ 10)) \text{ with } (l_1, l_2)(\lambda x.\text{with}(l_1)p\ x) \\
\longrightarrow & \ast (\lambda f.\text{with}(l_3)(f\ 10)) \text{ with } (l_1, l_2)(\lambda x.\text{with}(l_1)p\ x) \\
\longrightarrow & \ast \ast (\lambda f.\text{with}(l_3)(f\ 10)) \text{ with } (l_1, l_2)(\lambda x.\text{with}(l_1)p\ x) \\
\longrightarrow & \ast \ast \ast \ast (\lambda f.\text{with}(l_3)(f\ 10)) \text{ with } (l_1, l_2)(\lambda x.\text{with}(l_1)p\ x) \\
\end{align*}
\]

The following example, with layers \(\Delta_2\), shows that this approach differs from the previous one:

\[
\begin{align*}
(\lambda f.\text{with}(l_3)(f\ 10)) & \text{ with } (l_1, l_2)p \\
\longrightarrow & (\lambda f.\text{with}(l_3)(f\ 10)) \\
& \text{ with } (l_1, l_2)((\lambda x.\text{proceed} x)\{\text{with}(l_1)p/\text{proceed}\}) \\
\longrightarrow & (\lambda f.\text{with}(l_3)(f\ 10)) \text{ with } (l_1, l_2)(\lambda x.\text{with}(l_1)p\ x) \\
\longrightarrow & \ast (\lambda f.\text{with}(l_3)(f\ 10)) \text{ with } (l_1, l_2)(\lambda x.\text{with}(l_1)p\ x) \\
\longrightarrow & \ast \ast (\lambda f.\text{with}(l_3)(f\ 10)) \text{ with } (l_1, l_2)(\lambda x.\text{with}(l_1)p\ x) \\
\longrightarrow & \ast \ast \ast (\lambda f.\text{with}(l_3)(f\ 10)) \text{ with } (l_1, l_2)(\lambda x.\text{with}(l_1)p\ x) \\
\longrightarrow & \ast \ast \ast \ast (\lambda f.\text{with}(l_3)(f\ 10)) \text{ with } (l_1, l_2)(\lambda x.\text{with}(l_1)p\ x) \\
\begin{align*}
\end{align*}
\]

This shows that with this approach, parameters are now (correctly) dynamically scoped.

### 4.3 Capture Layer Environment and Use It to Interpret proceed: Variant II

The approach above performs the search in the current layer environment augmented with the captured layers. Another variant performs the search exclusively in the original layer environment. Thus we would have the two rules:

\[
\begin{align*}
E[\text{with}(l)E'[p]] & \rightarrow E[\text{with}(l)E'[e^{\{p_{E_2}/\text{proceed}\}}]] \\
& \text{if } p \notin \text{BP}(E') \quad \text{and } e = \Delta(l)(p) \\
& \text{(Disp4A)} \\
E[p_{E_0}] & \rightarrow E[e^{\{p_{E_2}/\text{proceed}\}}] \\
& \text{if } E_0[] \equiv E_2[\text{with}(l)E_3][] \quad \text{and } p \notin \text{BP}(E_3) \quad \text{and } e = \Delta(l)(p) \\
& \text{(Disp4B)}
\end{align*}
\]

The first rule describes the initial dispatch to a parameter. The second rule describes dispatch corresponding to proceed. The environment in which the search is performed is \(E_0[\ ]\), rather than \(E[\ ]\). For our example, the results are the same as above.
4.4 Build Composite Expression
In this approach, when a parameter is dispatched, the entire expression to which it corresponds, with all the proceed calls expanded in place, is computed:

\[ E[p] \rightarrow E[\text{build}(p, E)] \]  (DISP5)

\[ \text{build}(p, []) = \text{error} \]
\[ \text{build}(p, [l]) = \begin{cases} e^{\text{build}(p, \text{with}(l))}/\text{proceed} & \text{if } e = \Delta(l_0)(p) \\ \text{build}(p, \text{with}(l)) & \text{otherwise} \end{cases} \]

This approach is illustrated using layers \( \Delta_2 \):

\[
(\lambda f. \text{with}(l_2)(f 10)) \; \text{with}(l_1, l_2) \; p
\]
\[
\text{build}(p, \text{with}(l_2))
\]
\[
\lambda x. \text{proceed } x
\]
\[
\text{build}(p, \text{with}(l_1)) / \text{proceed}
\]
\[
\lambda x. q \; x
\]
\[
\text{build}(p, \text{with}(l_0))
\]
\[
\rightarrow (\lambda f. \text{with}(l_2)(f 10)) \; \text{with}(l_1, l_2)(\lambda x. q \; x)
\]
\[
\rightarrow (\lambda f. \text{with}(l_2)(f 10)) (\lambda x. q \; x)
\]
\[
\rightarrow \text{with}(l_2)(\lambda x. q \; x) 10
\]
\[
\rightarrow \text{with}(l_3)(q 10)
\]
\[
\rightarrow \text{with}(l_3)\{ (\lambda x. 5) 10 \}
\]
\[
\rightarrow^* 5
\]

This approach is actually semantically the same as the approach in Section 4.3 (we conjecture). They differ in the way they perform the computation. One approach (§ 4.3) performs the computation of the ‘generic function’ on-the-fly, by reusing the previously saved layer to perform dispatch. The other (§ 4.4) determines the complete generic function whenever dispatching to a parameter. This second approach corresponds to the precomputation of applicable methods in generic functions in CLOS, and hence ContextL.

4.5 Discussion
Table 1 summarizes the different approaches described until now. For each proposal, the layer environment in which the lookup of a binding is performed is specified (\( E_{\text{lookup}} \)), as well as the layer environment in which execution continues (\( E_{\text{eval}} \)). As before, \( E_0 \) refers to the remaining definition-time layer environment above the layer where proceed was captured, and \( E \) is the current layer environment. Also, \( E_{0, \text{full}} \) is the complete definition-time layer environment, including the layers below and the layer where proceed was captured.

From this table, we can rule out several approaches that have clear drawbacks. Looking up the binding in the current environment \( E[\; ] \) is unsatisfactory because it may fail to find an appropriate layer. Also, continuing evaluation in an environment where the definition-time layer environment is at the bottom, such as \( E[\Delta E_0[\; ]]], \) (4.1), destroys dynamic binding of parameters, by allowing definition-time bindings to shadow the currently-active ones. Using simply \( E_0[\; ] \) or \( E_{0, \text{full}}[\; ] \) to continue in would be similarly wrong, since current dynamic bindings would not be available at all. However, \( E_{0, \text{full}}[E[\; ]]) \) seems to be a reasonable choice for continuing evaluation, since both dynamic binding of parameters is respected and the definition-time layer environment is present to fulfill the expectations of original definitions.

For lookup, we have three remaining possibilities:

- \( E_0[\; ] \) lookup can fail even though a parameter binding is available in the current layer environment. This seems to contradict the purpose of dynamically-activated layers.
- \( E_{0, \text{full}}[E[\; ]]) \) A binding in the current layer environment can shadow a binding in the definition-time environment.
- \( E_{0, \text{full}}[E[\; ]]) \) A binding in the definition-time environment shadows bindings in the current environment.

It is obvious that \( E_0[\; ] \) should be present, but we currently do not agree whether or how \( E[\; ] \) should be included.

5. Obtaining More Control
The previous strategies are meant to be possible defaults: the same semantics for all lambdas and for all layers. First the appropriate layer is found in an environment \( E_{\text{lookup}} \), then the evaluation is continued in environment \( E_{\text{eval}} \).

It is also possible to give programmers more fine-grained control. For instance, at the lambda level, we can distinguish between a lambda that captures \( E_0 \) and one which does not (resulting in \( E_0[\; ] \)). We explore this in Section 5.1. In Section 5.2 we explore the dual approach, where fine-grained control is given at the layer level: we can specify that some layers stick to an escaping closure (and are therefore part of \( E_0 \)) so that they are reinstated when the closure is applied, whereas others do not.

5.1 Control at the Lambda Level
Here, we annotate \( \lambda \)-abstractions to indicate whether or not they capture the layer environment they escape. Assume that \( \lambda^0 \) denotes the \( \lambda \) which does not capture its context and \( \lambda^* \) be the \( \lambda \) which does. We have the following rules:

\[ E[\text{with}(l) \lambda^0 x. e] \rightarrow E[\lambda^0 x. e] \] (Esc-\( \lambda^0 \))
\[ E[\text{with}(l) \lambda^* x. e] \rightarrow E[\lambda^* x. \text{with}(l) e] \] (Esc-\( \lambda^* \))

The same \( \beta \)-reduction rule applies to both \( \lambda \)s.

\( \lambda^0 \) behaves as before, discarding the surrounding layer activations. \( \lambda^* \) behaves differently, wrapping the body of the \( \lambda \)-abstraction with the layer environment, so that when the \( \lambda \)-abstraction is applied, the captured layers will be reactivated.

\( ^5 \)This is similar to linearization of class hierarchies in, for example, Scala.

\( ^6 \)Both not described in this paper.

\( ^7 \)Not described in this paper.
Tables 1. Comparison of Approaches: $E_{\text{look}}$ is where the lookup of a binding is performed, and $E_{\text{eval}}$ is where the execution continues. $E_0$ is the environment in which the closure containing proceed was defined and $E$ is the current layer environment.

6. Conclusion
We argued that it is not clear how to give semantics to a closure containing a call to proceed which subsequently escapes its defining layer environment. We offer a number of possible solutions to this problem, along with mechanisms for better handling the layer environment associated with an escaping closure. By exploring various possibilities, we want to offer programmers both a predictable semantics so that they can reason about their code, and flexibility so that they can design the code to do exactly what they want it to. We offer no definitive answer. Furthermore, we conjecture that delimited dynamic bindings [5] will provide even more flexibility.

References
Appendix J

Multiple Dispatch in Practice

The paper “Multiple Dispatch in Practice” [90] follows.
Multiple Dispatch in Practice
Radu Muschevici, Alex Potanin, Dave Clarke and James Noble

Abstract—Multiple dispatch uses the run time types of more than one argument to a method call to determine which method body to run. While several languages over the last 20 years have provided multiple dispatch, most object-oriented languages still support only single dispatch — forcing programmers to implement multiple dispatch manually when required. This paper presents an empirical study of the use of multiple dispatch in practice, considering six languages that support multiple dispatch. We hope that this study will help programmers understand the uses and abuses of multiple dispatch; virtual machine implementors optimise multiple dispatch; and language designers to evaluate the choice of providing multiple dispatch in new programming languages.

Index Terms—empirical software engineering, multiple dispatch, double dispatch, multimethods.

I. INTRODUCTION

All object-oriented languages provide single dispatch: when a method is called on an object, the actual method executed is chosen based on the dynamic type of the first argument to the method (the method receiver, generally self, or this). Some object-oriented languages provide multiple dispatch, where methods can be chosen based on the dynamic types of more than one argument.

The goal of this paper is to understand how programmers write programs that use multiple dispatch when it is available. We ask the question: how often is multiple dispatch used? — what proportion of method declarations dispatch on more than one argument.

To this end, we describe a corpus analysis of programs written in six languages that provide multiple dispatch (CLOS, Dylan, Cecil, Diesel, Nice and MultiJava). While there are a range of other multiple dispatch languages (e.g. Slate [1], Groovy [2], Clojure [3]), we focus on these six languages here because we were able to obtain a corpus for each of these languages.

Contributions

This paper makes the following contributions: a language independent model of multiple dispatch; a suite of language independent metrics measuring the use of multiple dispatch; a corpus analysis study using those metrics on a collection of programs in six multiple dispatch languages.

Outline

This paper is organised as follows: Section II presents an overview of multiple dispatch and related work. Section III describes a language-independent model of multiple dispatch and defines a set of eight metrics in terms of that model. Section IV presents the corpus we analyse and describes the methodology we apply to measure multiple dispatch across this corpus. Section V presents the results of our corpus analysis study of multiple dispatch languages. Section VI puts our results in perspective, shows directions for future work and concludes.

II. BACKGROUND

In this section we introduce the concept of multiple dispatch and contrast it with techniques used to simulate multiple dispatch in languages that support only single dispatch. We also present an overview of the research surrounding multiple dispatch by surveying programming languages that include multiple dispatch, efforts targeted at optimising multiple dispatch, and studies related to the use of multiple dispatch.

A. Multiple Dispatch

Multiple dispatch uses the run time types of all arguments in a method call to determine which method body to run. In contrast, in single dispatch languages, such as SIMULA, Smalltalk, C++, Java, and C#, only the first argument of a method call is used in dynamic method lookup. In Java, for example, the first argument of a method call is called the receiver object, is written “before the dot” in a method call: receiver.method(arguments), and is called “this” inside a method body. The class of the receiver object designates the method body to be executed. We will refer to a method body as being specialised on the class where it is defined, and to the class of that first formal parameter as the parameter’s specialiser.

In Java, as in most single dispatch languages, a method’s specialiser is implicitly defined by the class enclosing the method definition, for example:

```java
class Car extends Vehicle {
    void drive () { print("Driving a car"); }
    void collide (Vehicle v) { print("Car crash"); }
}
```

In single dispatch languages, every dynamically dispatched method is specialised on precisely one class so it is easy to think of methods as operations on classes. Given this definition of the Vehicle class:

```java
abstract class Vehicle {
    void drive () { print("Brmmm!"); }
    void collide (Vehicle v) { print("Unspecified vehicle collision"); }
}
```

the following code will involve the Car class’s collide(Vehicle) method shown above, and print “Car crash”.

```java
Vehicle car = new Car();
Vehicle bike = new Bike();
car.collide(bike);
```
The method defined in Car is called instead of the method defined in Vehicle, because of the dynamic dispatch on the receiver of the message, which is an instance of class Car.

Consider now the Car class overloaded the collide method with a different argument:

```java
class Car extends Vehicle {
    // ... as above
    void collide (Bike b) { print("Car hits bike"); }
}
```

In a language that supports only single dispatch semantics, the method that prints “Car crash” will still be invoked when calling car.collide(bike), even though there exists a method that specifically handles the case of a car colliding with a bike. In a multiple dispatch language, however, the “Car hits bike” message would be printed.

Getting to the Car.collide(Bike) method from a call of Vehicle.collide(Vehicle) requires two dynamic choices: one on the type of the first “this” argument and the other on the type of the second (Vehicle or Bike) argument — this is why these semantics are called multiple dispatch. A method that uses multiple dispatch is often called a multimethod.

Of course, some languages may also have non-dispatched methods (such as Java static methods) that are not dynamically dispatched at all. Following C++, Java and C# also support method overloading, where methods may be declared with different formal parameter types, but only the receiver is dynamically dispatched.

1) Classes and Multiple Dispatch: Methods in single dispatch languages are usually defined in classes, and the `receiver.method(arguments)` syntax for method calls supports the idea that methods are called on objects (or that “messages are sent to objects” as Smalltalk would put it). This does not apply to multiple dispatch languages, however, where a concrete method body can be specialised on a combination of classes, and so methods are not necessarily associated with a single class. Some multiple dispatch languages declare methods separately, outside the class hierarchy, while others consider them part of none, one or several classes, depending on the number of specialised parameters. Since method bodies no longer have a one-to-one association with classes, all parameter specialisers have to be stated explicitly in method body definitions, as this example in the Nice programming language [4] shows:

```java
abstract Class Vehicle:
class Car extends Vehicle {}
class Bike extends Vehicle {}

void drive (Car c) {
    // a method specialised on the class Car */
    print("Driving a car");
}

void collide (Car c, Bike b) {
    // a method specialised on two classes */
    print("Car hits bike");
}
```

Similarly, while Java method call syntax follows Smalltalk by highlighting the receiver object and placing it before the method name: myCar.drive(), multiple dispatch languages generally adopt a more symmetrical syntax for calls to functions: `collide(myCar, yourBike)` or `drive(myCar)`, often while also supporting Java-style receiver syntax.

2) Simulating Multiple Dispatch: Multiple dispatch is more powerful and flexible than single dispatch. Any single dispatch idiom can be used in a multiple dispatch language — multiple dispatch semantics encompass single dispatch semantics. On the other hand, implementing multiple dispatch idioms will require specialised hand-coding in a single dispatch language.

Binary methods [5], for example, operate on two objects of related types. The Vehicle.collide(Vehicle) method above is one example of a binary method. Object equality: `Object.equals(Object)`, object comparisons, and arithmetic operations are other common examples.

In a single dispatch language, overriding a binary method in a subclass is not considered safe because it violates the contravariant type checking rule for functions [6]. In the example above it is unsafe to override the `collide(Vehicle)` method in the Vehicle class with `collide(Bike)` in the Car class. Because a single dispatch language only considers the runtime type of the first argument in a method call, nothing would prevent dispatching a call to Car.collide(Bike) when the second argument supplied in the method call is in fact not of type Bike (but of a supertype, such as Vehicle). This would cause a type error at runtime. Note that Java avoids such runtime errors by statically overloading the Vehicle.collide(Vehicle) method: this can cause the undesired program behaviour shown in Section II-A.

To avoid violating the contravariant type checking rule for functions, single dispatch languages like Smalltalk generally use the Double Dispatch pattern to implement binary methods, encoding multiple dispatch into a series of single dispatches [7]. Double Dispatch is also at the core of the Visitor pattern [8] which decouples operations from data structures.

For example, we could rewrite the collision example to use the Double Dispatch pattern in Java as follows:

```java
class Car {
    void collide(Vehicle v) { v.collideWithCar(this); }
    void collideWithCar(Car c) { print("Car hits car"); }
    void collideWithBike(Bike b) { print("Car hits bike"); }
}

class Bike {
    void collide(Vehicle v) { v.collideWithBike(this); }
    void collideWithCar(Car c) { print("Bike hits car"); }
    void collideWithBike(Bike b) { print("Bike hits bike"); }
}
```

Calling a `collide` method provides the first dispatch, while the second call to a `collideWithXXX` method provides the second dispatch. The arguments are swapped around so that each argument gets a chance to go first and be dispatched upon. External clients of these classes should only call the `collide` method, while actual implementations must be placed in the `collideWithXXX` methods.

The double dispatch idiom is common in languages like Smalltalk where single dispatch is the preferred control structure. Java’s `instanceof` type test provides an alternative technique for implementing multiple dispatch. The idiom here is a cascade of `if` statements, each testing an argument’s class, and the body of each `if` corresponding to a multimethod body. To return to the Car and Bike classes:

```java
class Car {
    void collide(Vehicle v) {
        if (v instanceof Car) { print("Car hits car"); return; }
```
if (v instanceof Bike) { print("Car hits bike"); return; } throw Error("missing case: should not happen"); }

class Bike {
  void collide(Vehicle v) {
    if (v instanceof Car) { print("Bike hits car"); return; }
    if (v instanceof Bike) { print("Bike hits bike"); return; }
    throw Error("missing case: should not happen");
  }
}

Compared with directly declaring multimethods, both idioms for double dispatching code are tedious to write and error-prone. Code to dispatch on three or more arguments is particularly unwieldy. Modularity is compromised, since all participating classes have to be modified upon introducing a new class, either by writing new dispatching methods or new cascaded if branches. The cascaded if idiom has the advantage that it doesn’t pollute interfaces with dispatching methods, but the methods with the cascades become increasingly complex, and it is particularly easy to overlook missing cases.

B. Multiple Dispatch Research

Despite its advantages over single dispatch (§ II-A), multiple dispatch is not present in current mainstream object-oriented languages such as Smalltalk, C++, C♯, and Java.

These languages follow an object model that views methods as operations on particular (classes of) objects. Such an operation always depends on the type of one single object: it is a property of that type, and it is encapsulated inside the object. The single dispatch idiom, where functions dispatch on a distinct receiver argument, consequently models this approach to object-orientation. In contrast, multiple dispatch allows operations to depend on multiple different types of objects, thus defying object-based encapsulation. This partly explains why traditional object-oriented languages do not support multiple dispatch.

Other reasons might be related to the early state of multiple dispatch research at the time when the above languages were designed. For example, the issue of independent static type checking of separate code modules was tackled only during the late 1990s. Multiple dispatch is less time-efficient than single dispatch, due to the more complex lookup mechanism which involves evaluating the types of several arguments instead of just a single one. Therefore, the additional run time cost of multiple dispatch is acceptable only when multimethods are actually invoked (following the principle “you don’t pay for what you don’t use”), and that expense should not exceed the cost of hand-coded double dispatch (simulated multiple dispatch). Finally, the space efficiency of virtual dispatch tables has only been the subject of more recent research (§ II-B.3).

Bjarne Stroustrup, the designer of C++, offers a first-hand account of the difficulties encountered when evaluating multiple dispatch for C++ [9, Section 13.8]. The two main obstacles which, he regrets, kept him from including multimethods in C++ were that, for one, he could not come up with a “calling mechanism that was simple and efficient as the table lookup used for virtual functions [, the C++ equivalent of single dispatch methods]” and second, difficulties in resolving method ambiguity at compile time. Further he offers the following thought, which we cite here as it reflects the driving force behind our research, almost 30 years later: “Multi-methods is one of the interesting what-ifs of C++. Could I have designed and implemented them well enough at that time? Would their application have been important enough to warrant the effort?”

1) Multiple Dispatch Languages: Multiple dispatch was pioneered by CommonLoops [10], [11] and the Common Lisp Object system (CLOS) [12], both aimed at extending Lisp with an object-oriented programming interface. The extensions were meant to integrate “smoothly and tightly with the procedure-oriented design of Lisp” [11] and facilitate the incremental transition of code from the procedural to the object-oriented programming style.

The basic idea is that a CLOS generic function is made up of one or more methods. A CLOS method can have specialisers on its formal parameters, describing types (or individual objects) it can accept. At run time, CLOS will dispatch a generic function call on any or all of its arguments to choose the method(s) to invoke — the particular methods chosen generally depend on a complex resolution algorithm to handle any ambiguities.

Several more recent programming languages aim to provide multimethods in more object-oriented settings. Dylan [13] is based on CLOS. Dylan’s dispatch design differs from CLOS in that it features optional static type declarations which can be used to type generic functions, that is, to constrain their parameters to something more specific than <object>, the root of all classes in Dylan. Dylan also omits much of the CLOS’s configurability, treating all arguments identically when determining if a generic function call is ambiguous.

Cecil [14], [15] is a prototype-based programming language that was the first to implement a modularly checked static type system for multimethods. Cecil treats each method as encapsulated within every class upon which it dispatches. This way a method is given privileged access to all objects of which it is a part. This is different from, for example, Java, where methods are part of precisely one class and also unlike CLOS or Dylan in which methods are not part of any class.

Diesel [16] is a descendant of Cecil that shares many of its multiple dispatch concepts. The main differences with Cecil are Diesel’s module system (unlike Cecil, Diesel method bodies are separate from the class hierarchy and encapsulated in modules) and explicit generic function definitions (which bring it closer to CLOS). As in Dylan and Cecil, message passing is the only way to access an object’s state. Diesel uses a modular type system initially designed by Millstein and Chambers [17] for the Dubious language.

The Nice programming language [4] strives to offer an alternative to Java, enhancing it with multimethods and open classes. In Nice, operations and state can be encapsulated inside modules, as opposed to classes. Message dispatching is based on the first argument and optionally on any other arguments.

MultiJava [18] extends Java with multimethods and open classes. MultiJava retains the concept of a privileged receiver object to associate methods with a single class for encapsulation purposes, however, the runtime selection of a method body is no longer based on the receiver’s type alone. Rather, any parameter in addition to the receiver can be specialised by specifying a true subtype of the corresponding static type or a constant value. The MultiJava compiler mjc, translates MultiJava source code into standard Java bytecode. For methods that specialise additional parameters, it introduces cascaded sequences of instanceof tests (or equality
comparisons, for value dispatch).

There are of course other multiple dispatch languages. Kea [19] was the first statically typed language with multiple dispatch. Slate [1] integrates Self-like prototype-based programming with multiple dispatch to propose a new object model. Some more recent programming languages are designed with multiple dispatch already on-board, among them Perl 6 [20, Section 8.6.2], Clojure [3] and Groovy [2, Section 4.7]. Scala [21] supports a form of pattern matching that can be used to dispatch on arbitrary predicates.

2) Multiple Dispatch Extensions: Several popular single dispatch languages (Perl [22], Python [23], Ruby [24], C++ [25]) have been extended to support multiple dispatch, often by means of libraries. Smalltalk has been extended with multiple dispatch [26] using its reflective facility. Fickle [27], a statically typed, class-based object-oriented language with support for object reclassification has been extended with multiple dispatch and first class relationships [28].

Java has been augmented with multiple-dispatch and similar facilities using several different approaches. Parasitic Multimethods [29] is such an extension: methods which are defined using the parasitic keyword override less specific methods. A modified compiler translates code that uses the extended semantics into standard bytecode by introducing type testing statements (instanceof) to determine the runtime types of all arguments in a method call, thereby dispatching to the most specific parasite.

The Walkabout [30] uses the reflection interface of Java 1.1 and later to simplify the implementation of the Visitor pattern. Walkabouts greatly improve extensibility by eliminating the need for visitable classes to implement a visit() method and allowing the addition of visitable classes without modification of existing visitors. However, the use of reflection to invoke the appropriate visit method makes this approach impractically slow.

The Runabout [31] improves upon the Walkabout approach in terms of performance: Where the Walkabout uses reflection to invoke visit methods, the Runabout dynamically generates (and caches) bytecode that will invoke the appropriate visitor. This makes the dispatch performance of the Runabout comparable to that of the classic visitor pattern and typically exceeds that of instanceof tests.

Dutchyn et al. [32] modified the Java virtual machine to treat static overloading as dynamic dispatch in classes that implement the provided MultiDispatchable marker interface.

Millstein et al. [33] have evolved MultiJava into Relaxed MultiJava (RMJ) which essentially allows the programmer to write code in a more flexible style, without sacrificing static type checking.

Predicate dispatching generalises multiple dispatch to include field values and pattern matching [34], while aspect-oriented programming [35], [36] is based around pointcuts that can dispatch on almost any combination of events and properties in a program’s execution.

3) Multiple Dispatch Efficiency: Most language implementations summarised in the previous section include efficiency evaluations of the respective implementation. Additionally, space and time efficiency of method dispatch has been the subject of a large body of research [34], [37]–[40]. Cunei and Vitek [41] include a recent comparison of the efficiency of a range of multiple dispatch implementations such as the Visitor pattern, the Runabout and MultiJava.

4) Empirical Multiple Dispatch Studies: Studies investigating the practical use of multiple dispatch are less widespread than multiple dispatch implementations — Kempf at al.’s early 1987 study [42] of the CommonLoops language (a CLOS predecessor) is one notable exception. One of that study’s goals was to assess how useful generic functions and multimethods are for developers, by measuring how often these constructs are used in the implementations of CommonLoops itself and a window library called BeatriX.

Finally, this present study is part of a project that compares the use of multiple dispatch to the use of techniques that simulate multiple dispatch (such as those described in § II-A.2) in Java. In an earlier paper [43], we asked a complementary question: how much could multiple dispatch be used? — that is, what proportion of methods that simulate multiple dispatch could be refactored to use multiple dispatch if it was provided by the language. That paper also includes some preliminary results from this study.

The current paper delivers the final results from our multiple dispatch study. Compared to the results we reported on earlier [43], we now apply a more comprehensive set of metrics (eight instead of six) to the software corpus. The results are presented in more detail and discussed in greater depth. We also corrected a measurement error that led to misleading figures in the case of two applications with respect to one metric. This paper does not include the results from the study on simulated multiple dispatch in Java. For these we refer to the original paper [43].

III. MODEL

In this section we introduce a language-independent model for dynamic dispatch. We then describe six multiple dispatch languages in terms of that model. Finally, we use the model to define metrics for multiple dispatch. The model and metrics we designed allow us to reason about the application of the multiple dispatch paradigm in a language-independent way.

A. Modelling Dynamic Dispatch

The model shown in Fig. 1, is designed to allow us to compare multiple dispatch consistently across different programming languages. The model’s terminology has been chosen to match general usage, rather than following any particular programming language. Section III-C will use the model to define the metrics that can be used across a range of programming languages. We now present the main entities of the model in turn.

![Fig. 1](image-url)

A Model for Multiple Dispatch. GF refers to generic function, CM refers to concrete method, and SPEC refers to specialiser.
a) Generic function: A generic function is a function that may be dynamically dispatched, such as a CLOS generic function, a Smalltalk message, a MultiJava method family, or Java method call. Each generic function will have one or more concrete methods associated with it: calling a generic function will invoke one (or more) of the concrete methods that belong to that function. Generic functions are identified by a name and a signature. Some languages allow a generic function to be defined explicitly (e.g. CLOS’s defgeneric, Dylan’s define generic and Diesel’s fun), whereas in other languages (such as MultiJava or Cecil) they are implicit and must be inferred from method definitions.

Some languages also automatically generate generic functions as accessors to all field declarations. Because we wish to focus on programmer specified multiple dispatch methods, we omit automatically generated accessors from our analysis.

b) Name: Generic functions and concrete methods are referred to by their names. In our model, a name is always “fully-qualified”, that is, if a namespace is involved then that information is part of the name. To avoid ambiguity, our analyses always compute fully-qualified names where necessary.

c) Signature: The permissible arguments to a generic function are defined by that function’s signature, and all the concrete methods belonging to a generic function must be compatible with that signature. In languages with only dynamic typing, a generic function’s signature may be simply the number of arguments required by the function: some language’s signatures additionally support refinements such as variable length argument lists or keyword arguments. In languages with (optional or mandatory) static type systems, a generic function’s signature may also define static types for each formal argument of the function.

Some languages have implicit parameters (such as the “receiver” or “this” parameter in traditional object-oriented languages such as SIMULA, Smalltalk, Java, C++, C#). In our model, these parameters are made explicit in the signature (hence our use of the term “function”). In the case of traditional object-oriented languages, the receiver is the first formal parameter position.

d) Concrete method: A concrete method gives one code body for a generic function — roughly corresponding to a function in Pascal or C, a method in Java or Smalltalk, or CLOS method. As well as this code, a concrete method will have a name and an argument list — the argument list must be compatible with the signature of its generic function (as always depending on the signature). A concrete method may also have a specialiser for each formal argument position. The rules of each language determine the generic function(s) to which a given concrete method belongs.

e) Specialiser: Formal parameters of a concrete method can have specialisers. Specialisers are used to select which concrete method to run when a generic function is called. When a generic function is called, the actual arguments to the call are inspected, and only those concrete methods whose formal specialisers match those arguments can be invoked in response to the call. Specialisers can describe types, singleton objects, or sets of objects and types (details depend on the language in question).

Some concrete method parameters may have no specialiser (they are unspecialised) — the method is applicable for any argument values supplied to those parameters. In contrast, in a class-based object-oriented language, every instance method will belong to a class, and its distinguished receiver argument will be specialised to that class. For example, this is true for every non-static, non-constructor method in Java; Java statics and constructors are not specialised.

Dynamic specialisers are closely related to generic function signatures in statically typed languages: whenever a generic function is called, its actual arguments must conform to the types described by its signature. Depending on the language, specialisers may or may not be tied into a static type system.

f) Dispatch: When a generic function is called at run time, it must select the concrete method(s) to run. In our model, this is a dynamic dispatch from the generic function to its concrete methods. If this dispatch is based on the type of one argument, we call it single dispatch; if on the type of more than one argument, multiple dispatch. If a generic function has only a single concrete method, then no dynamic dispatch is required: we say the function is monomorphic or statically dispatched.

B. Modelling Programming Languages

To ground our study, we now describe how the features of each of the languages we analyse are captured by the model. The crucial differences between the languages can be seen as whether they offer static typing, dynamic typing, or optional (static) typing; the number of generic functions per method name; and whether a concrete method can be in more than one generic function. These details are summarised in Table 1, which also gives an overview of terminology used by each language, with Java and Smalltalk for comparison.

1) CLOS: CLOS [12] fits quite directly into our model. CLOS generic functions are declared explicitly, and then (concrete) methods are declared separately; both generic functions and methods lie outside classes. Each generic function is identified by its name (within a namespace), so all methods of the same name belong to the same generic function. CLOS requires "lambda list congruence": all methods must agree on the number of required and optional parameters, and the presence and names of keyword parameters [44].

2) Dylan: Dylan’s dispatch design [13] is similar to CLOS in most respects, including concrete methods being combined via explicit generic function definitions, and similar parameter list congruency conditions. Dylan supports optional static type checking, and specialisers and static type declarations are expressed using the same syntax. When defining a concrete method, the type declarations serve as dynamic specialisers if they are more specific than the types declared by the generic function.

3) Cecil: Cecil [14] generic functions (multimethods) are declared implicitly, based on concrete method definitions, and each concrete method is contained within one generic function. Unlike CLOS, a generic function comprises concrete methods of the same name and number of arguments: generic functions with the same name but different parameter counts are independent. Like Dylan, Cecil supports optional static type declarations, but unlike Dylan, different syntactic constructs are used to define static type declarations and dynamic specialisers. A parameter can incur a static type definition, specialisation, or both.

4) Diesel: Diesel [16] is a descendant of Cecil, however generic functions are declared explicitly (called functions). Each Diesel function can have a default implementation, which in our model corresponds to a concrete method with no specialised parameters. Additional concrete methods (simply called methods) can augment a function by specialising any subset of its parameters.
<table>
<thead>
<tr>
<th>Language</th>
<th>Typinga</th>
<th>GF termb</th>
<th>GF def.c</th>
<th>CM termd</th>
<th>CM grouped in GF bye</th>
<th>Multif</th>
<th>Auto acc.g</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOS</td>
<td>dyn</td>
<td>generic function</td>
<td>explicit</td>
<td>method</td>
<td>namec</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Dylan</td>
<td>opt</td>
<td>generic function</td>
<td>explicit</td>
<td>method</td>
<td>namec</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Cecil</td>
<td>opt</td>
<td>method</td>
<td>implicit</td>
<td>method body</td>
<td>name + #args</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Diesel</td>
<td>opt</td>
<td>function</td>
<td>explicit</td>
<td>method</td>
<td>name + #args</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Nice</td>
<td>static</td>
<td>method declaration</td>
<td>implicit</td>
<td>method implementation</td>
<td>name + #args + types</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>MultiJava</td>
<td>static</td>
<td>method family</td>
<td>implicit</td>
<td>method</td>
<td>name + #args + types</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Java</td>
<td>static</td>
<td>message</td>
<td>implicit</td>
<td>method body</td>
<td>name + #args + types</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Smalltalk</td>
<td>dyn</td>
<td>method call</td>
<td>implicit</td>
<td>method</td>
<td>name( + #args)</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

a static, optional static, or dynamic typing  
b the terminology used for “generic function” (GF)  
c whether generic function definitions are explicit or implicit  
d the term used for “concrete method” (CM)  
e how concrete methods are grouped into generic functions (i.e. how a generic function signature is defined)  
f whether one concrete function can be part of multiple generic functions  
g whether the language automatically generates accessor messages (which we elide from our analysis)h All argument lists (lambda lists) must be congruent.  
i as outlined by a study of dynamic dispatch in Java [43]  
j Smalltalk message selectors encode the number of arguments to the message.

5) Nice: Nice [4] is a more recent multiple dispatch language design based on Java. A Nice generic function (method declaration) supplies a name, a return type and a static signature. Different concrete methods (method implementations) can exist for a declaration. When defining a concrete method, the parameter type declarations serve as dynamic specialisers if they are different to (that is more specific than) the types stated in the method declaration.

6) MultiJava: MultiJava [45] is an extension of Java that adds the capability to dynamically dispatch on other arguments in addition to the receiver object. A generic function (also called method family) consists of a top method, which overrides no other methods, and any number of methods that override the top method. Any method parameter can be specialised by specifying a true subtype of the corresponding static type or a constant value.

Multiple dispatch subsumes single dispatch: where in multiple dispatch languages, any argument to a function can be used to dispatch, in single dispatch languages it is only the first parameter that can be dispatched on. To compare the multiple dispatch languages described above to mainstream languages, we use our model to additionally describe Smalltalk and Java.

7) Smalltalk: Smalltalk is a single dispatch language which introduced the terms message, roughly corresponding to implicitly defined generic function, and method for concrete method. Smalltalk is dynamically typed, and every message is single dispatched (even the equivalent of constructors and static messages, which are sent dynamically to classes). Every method name (or selector) defines a new generic function, and the names encode the number of arguments to the message.

8) Java: Java is a single dispatch statically typed class-based language; it uses the term “method” for both generic functions (method call) and concrete methods (method bodies). Generic functions are defined implicitly, and depend on the names and the static types of their arguments.

C. Metrics

Our study approaches multimethods and multiple dispatch from a programmer’s point of view by analysing source code available publicly, mostly under open-source licenses. We focus on method definitions which we examine statically.

To study multiple dispatch across languages we define metrics based on our language independent model. Table II summarises the metrics we define in this section.

1) Dispatch Ratio (DR): We are most interested in measuring the relationships between generic functions and concrete methods. We define $CM(g)$ as the set of concrete methods belonging to a given generic function $g$. The number of concrete methods that belong to a generic function $g$ gives the basic metric Dispatch Ratio $DR(g) = |CM(g)|$. DR measures, in some sense, the amount of choice offered by a generic function: monomorphic functions will have $DR(g) = 1$, while polymorphic functions will have $DR(g) > 1$.

We are usually not interested in the measurements from the above metrics (Table II) for individual generic functions or concrete methods, but rather we want to know about their distribution over a given application, or even collection of applications. We can report the measurements as a frequency distribution, that is, for a value $dr$, what proportion of generic functions $g$ have $DR(g) = dr$. Frequency distributions provide information such as: what proportion of generic functions have exactly one concrete method.

We use the basic DR metric to define an average Dispatch Ratio across each application. The average Dispatch Ratio $DR_{ave}$, that is, the average number of concrete methods that a generic function would need to choose between is:

$$DR_{ave} = \frac{\sum_{g \in G} DR(g)}{|G|}$$

where $G$ is the set of all generic functions. The intuition behind $DR_{ave}$ is that if you select a generic function from a program at random, to how many concrete methods could it dispatch?
2) Choice Ratio (CR): Because a generic function with a DR > 1 necessarily contains more methods than a monomorphic generic function, we were concerned that DR_{ave} can give a misleading low figure for programs where some generic functions have many more concrete methods than others.

For example, consider a program with one generic function with 100 concrete methods, DR(g_1) = 100, and another 100 monomorphic methods DR(g_{92..101}) = 1. For this program, DR_{ave} = 1.98, even though half the concrete methods can only be reached by a 100-way dispatch.

To catch these cases, we define the Choice Ratio of a concrete method m to be the total number of concrete methods belonging to all of the generic functions to which m belongs:

$$\text{CR}(m) = \bigcup_{g \in GF(m)} CM(g)$$

Note that this counts each concrete method only once, even if it belongs to multiple generic functions. An application-wide average, CR_{ave} can be defined similarly:

$$\text{CR}_{ave} = \frac{\sum_{m \in M} \text{CR}(m)}{|M|}$$

where M is the set of concrete methods. The intuition behind CR_{ave} is that if you select a concrete method from a program at random, then how many other concrete methods could have been dispatched instead of this one? For the example above CR_{ave} = 50.5.

3) Degree of Specialisation of a Concrete Method (DoS): The Degree of Specialisation of a concrete method simply counts the number of specialised parameters:

$$\text{DoS}(m) = |\text{spec}(m)|$$

where spec(m) is the set of argument positions of all specialisers of the method m (we will later write spec_i(m) for the i‘th specialiser). DoS can also be extended to an average, DoS_{ave} in the obvious manner, over all concrete methods.

Dynamically specialised multiple method parameters is a key feature of multiple dispatch: DoS measures this directly. Functions without dynamic dispatch, such as Java static methods, C functions, or C++ non-virtual functions, will have DoS = 0. Singly dispatched methods like Java instance methods, C++ virtual functions, and Smalltalk methods will have DoS = 1. Methods that are actually specialised on more than one argument will have DoS > 1.

4) Degree of Specialisation of a Generic Function (DoS_G): The Degree of Specialisation of a generic function counts the specialisers of that generic function.

$$\text{DoS_G}(g) = |\{i \in \text{spec}(m) | m \in CM(g)\}|$$

Specialisers for formal parameters of a generic function are inferred from the concrete methods that belong to it: if at least one concrete method in the generic function has a dynamic specialiser at a certain position, then the same parameter position of the generic function is considered specialised.

We define this metric in addition to the per concrete method DoS metric for accuracy: we compare DoS_G measurements with those furnished by the Degree of Dispatch (DoD) metric (§ III-C.7), which is also defined in terms of generic functions.

DoS_G can also be extended to an average, DoS_{G,ave} over all generic functions.

5) Rightmost Specialiser of a Concrete Method (RS): Programmers read method parameter lists from left to right. This means that a method with a single specialiser on the last (rightmost) argument may be qualitatively different to a method with one specialiser on the first argument. To measure this we define the Rightmost Specialiser:

$$\text{RS}(m) = \max(\text{spec}(m))$$

If a method has some number of specialised parameters (perhaps none) followed by a number of unspecialised parameters, then RS = DoS; where a method has some unspecialised parameters early in the list, and then some specialised parameters, RS > DoS. The capability to specialise a parameter other than the first distinguishes multiple dispatch languages from single dispatch languages. RS can, for example, identify methods that use single dispatching (DoS=1) but where that dispatch is not the first method argument. Once again, we can define a summary metric RS_{ave} by averaging RS over all concrete methods.

6) Rightmost Specialiser of a Generic Function (RS_G): Similarly to the Rightmost Specialiser of a concrete method, we define the Rightmost Specialiser of a generic function as the rightmost specialised parameter position of that function:

$$\text{RS_G}(g) = \{\max(\text{RS}(m) | m \in CM(g))\}$$
By averaging $RS_G$ over all generic functions, we can define $RS_{G,ave}$.

7) Degree of Dispatch (DoD): The Degree of Dispatch is the number of parameter positions required for a generic function to select a concrete method. The key point here is that specialising concrete method parameters does not by itself determine whether that parameter position will be required to dispatch the generic function. This is because all the concrete methods in the generic function could specialise the same parameter position in the same way. Similarly, if only one concrete method specialises a parameter position, that position could still participate in the method dispatch even if no other concrete method specialises that parameter — the other concrete methods acting as defaults.

The DoD metric counts the number of parameter positions where two (or more) concrete methods in a generic function have different dynamic specialisers. In general, these are the positions that must be considered by the dispatch algorithm.

\[
DoD(g) = |P|, \quad P = \{i \mid \exists m_1, m_2 \in CM(g) : \text{spec}_i(m_1) \neq \text{spec}_i(m_2)\}
\]

Where the Degree of Specialisation (DoSC) measures the number of parameters of a generic function which the programmer has designed dispatch-able (by specialising them), the Degree of Dispatch counts only those parameters on which the generic function actually has the potential to dispatch upon.

We can once again define a summary metric $DoD_{ave}$ as the average over all generic functions. If $DR_{ave}$ and $CR_{ave}$ measure the amount of choice involved in dispatch, then $DoD_{ave}$ measures the complexity of that choice.

8) Rightmost Dispatch (RD): Finally, by analogy to RS, we can define RD as the rightmost parameter a generic function actually dispatches upon.

\[
RD(g) = \max(P)
\]

RD is to RS as DoD is to DoS: the “Do” versions count specialisers of methods, or dispatching positions of generic functions, while the “R” versions consider only the rightmost position. RD for a generic function will usually be the maximum RS of that function’s methods, unless every concrete method in the generic function specialises the rightmost parameter in the same way. For a whole application, we can report $RD_{ave}$ as the average RD across all generic functions.

D. Example

To illustrate the metrics, consider the following simple multiple dispatch example written in Dylan, which defines a range of binary methods for the type hierarchy `<sports-car>` extends `<car>` extends `<vehicle>`.

This example defines two generic functions (collide and pileup) with two and four concrete methods respectively. The values for the metrics relevant to each declaration are in the comments above them.

```dylan
define class <vehicle> (<object>) ... ;
define class <car> (<vehicle>) ... ;
define class <sports-car> (<car>) ... ;

// DR = 2, DoD = 1, RD = 2, DoS(g) = 2, RS(g) = 2
define generic collide(v1 :: <vehicle>, v2 :: <vehicle>);
// CR = 2, DoS = 1, RS = 1

// DR = 4, DoD = 3, RD = 3, DoS(g) = 3, RS(g) = 3
define generic pileup(v1 :: <vehicle>, v2 :: <vehicle>, v3 :: <vehicle>);
// CR = 4, DoS = 2, RS = 3

// DR = 4, DoD = 3, RD = 3
define method collide(sc :: <sports-car>, v :: <vehicle>) ... ;
define method collide(sc :: <sports-car>, c :: <car>) ... ;

// DR = 4, DoD = 3, RD = 3
define method pileup(sc :: <sports-car>, v :: <vehicle>, c :: <car>) ... ;
define method pileup(sc :: <sports-car>, c :: <car>, v :: <vehicle>) ... ;
// CR = 4, DoS = 2, RS = 3

// DR = 2
define method collide(sc :: <sports-car>, v :: <vehicle>) ... ;
define method collide(sc :: <sports-car>, c :: <car>) ... ;
define method collide(sc :: <sports-car>, v :: <vehicle>, c :: <car>) ... ;
// CR = 2, DoS = 1, RS = 2

// DR = 4
define method pileup(sc :: <sports-car>, v :: <vehicle>, c :: <car>) ... ;
define method pileup(sc :: <sports-car>, c :: <car>, v :: <vehicle>) ... ;
// CR = 4, DoS = 2, RS = 4

// DR = 4
define method pileup(c :: <car>, v :: <vehicle>, c :: <car>) ... ;
define method pileup(c :: <car>, c :: <car>, v :: <vehicle>) ... ;
// CR = 4, DoS = 2, RS = 0
```

IV. Methodology

In this Section we first motivate Corpus Analysis, the research methodology underlying our study (§ IV-A). We then present our corpus, a collection of applications written in multiple dispatch languages (§ IV-B).

A. Corpus Analysis

The Software Corpus Analysis approach uses an automated process to measure certain artifacts in a body of programs (in the form of their source code or any compilation stage). Assuming that the studied corpus is representative of how the language is used in general, one can derive a set of rules or patterns governing that language.

A large amount of research in software engineering has focused on developing models and methodologies of how software should
be written to meet certain quality criteria such as re-usability, maintainability and cost-effectiveness. Corpus Analysis is an approach that measures attributes of software as it actually is. By understanding the shape of existing software, we can learn about the characteristics of good software.

Software Corpus Analysis is a widely used empirical research method. There are many recent examples addressing program topology [46], [47], mining patterns [48], [49], object initialisation [50], aliasing [51], [52], dependency cycles [53], exception handling [54], inheritance [55], non-nullity [56] and visualisation [57].

We adopt the corpus analysis approach to study the use of multiple dispatch. Given that, using a multiple dispatch language, programmers can freely choose whether to use either multiple dispatch or another technique, we want to know how much multiple dispatch is used in practice in a set of languages. Corpus analysis can answer this question by measuring — across a given corpus — the proportion of method declarations that dispatch on more than one argument.

There are of course alternative approaches to studying the use of multiple dispatch. They range from manually analysing selected source code artifacts, surveying practitioners about their perceived usage of multiple dispatch to using multiple dispatch languages in teaching and as part of class projects. These approaches are valuable and we acknowledge their potential to complement corpus analysis: they can help verify our results and answer further questions, such as when and why is multiple dispatch used. For some examples, corpus analysis can draw attention to interesting code portions in a program, which can then be examined manually in order to understand the kind of problems they solve. A study involving in-field interviews could reveal the level of awareness practitioners have of multiple dispatch and its potential. Any discrepancies between the findings from our study and the subjective value attributed to multiple dispatch by practitioners could help improve software engineering pedagogy. Class projects involving a multiple dispatch language and a single dispatch language as a control could provide further insight into the relative value of multiple dispatch in practice of students.

Our Corpus Analysis of multiple dispatch is a first step in a line of research aimed at understanding how much value is gained from the use of multiple dispatch in practice.

B. Corpus

For this study we have gathered a corpus of nine applications written in six languages that offer multiple dispatch (Table III).

Most applications in this corpus are compilers for the respective language — they are all too often the only applications of significant size that we could obtain. This fact certainly introduces a degree of bias in our analysis. An additional reason that limits the generalisability of our results comes from the nature of these compilers, being mostly academic projects: the development processes undertaken at universities cannot be regarded as representative for software development at large.

CLOS is notably distinct with respect to the availability of large projects and the corpus could be expanded by several CLOS projects. We opted to cover a broad spectrum of languages rather than weighting this study towards one language because we are interested in measuring multiple dispatch across languages. We note that this corpus contains a single application per language (with exceptions of CLOS with three and Dylan with two applications): this largely reflects the fact that multiple dispatch languages are not in wide use today.

All of the languages studied here come with more or less extensive standard libraries. Our measurements of each application include the contribution due to the libraries.

As is often the case when measuring real code, we have to make assumptions about exactly what to measure. One assumption is with respect to the auto-generated field accessors some languages provide (see Table I). As our interest is in how programmers interact with language features, we do not measure these accessors, nor do we measure other compiler-generated artefacts.

The following paragraphs briefly present each application in our corpus, giving a rough idea about its scope, development history and size. Fig. 2 shows the relative sizes of applications at a glance, measured in terms of generic functions and concrete methods.

1) CMUCL (CLOS): The first of three Common Lisp applications in our corpus, CMUCL is a free implementation of the Common Lisp programming language as defined by ANSI [58]. It has been in development since the early 1980’s at the Computer Science Department of Carnegie Mellon University. The package includes a compiler and an extensive library [59]. CMUCL auto-generates a large number of generic functions, which we exclude from our analysis [60, Section 4.4.1]. We examined version 19d and found it to define 271 generic functions and 550 concrete methods.

2) SBCL (CLOS): Steel Bank Common Lisp (SBCL) is an open source/free compiler and runtime system for ANSI Common Lisp. SBCL was forked from CMUCL in 1999 and has evolved independently since then. The system is described to be simpler than CMUCL, partly because some extensions were removed, partly due to ample refactoring [61]. We found this fact reflected in SBCL’s smaller size, compared to CMUCL (before excluding auto-generated class predicates): the version we examined (0.9.16) has 363 generic functions and 861 concrete methods.

3) McCLIM (CLOS): McCLIM is an open source implementation of the Common Lisp Interface Manager specification [62], a toolkit for writing graphical user interfaces in Common Lisp. It runs on top of several Lisp implementations, including CMUCL and SBCL. The project initiated in 2000 by merging several
developers' individual efforts to write a free implementation of CLIM; by the end of 2002 it was considered “almost feature complete” by its developers [63]. McClim is the largest Common Lisp project in our corpus, containing 1906 generic functions and 4419 concrete methods.

4) Open Dylan (Dylan): OpenDylan, also known as Functional Developer, is a fully-featured development environment originally created in the early 1990’s at Apple with the objective of combining the best qualities of Common Lisp with the advantages offered by static languages like C++ [13]. Only the command line version of the compiler, known as the minimal console compiler currently runs on GNU/Linux, the platform we used for our analysis. The version we inspected (1.0 beta5; obtained from the SVN repository on 27. April 2008) has 2143 generic functions and 5389 concrete methods.

5) Gwydion (Dylan): Gwydion Dylan includes a Dylan-to-C (d2c) compiler originally created at Carnegie Mellon University, mostly by developers who had previously worked on the CMU Common Lisp (CMUCL) project. d2c uses gcc as the back-end for generating native code. The current release 2.4.0 is considered a technology preview by its maintainers [64] due to limitations that make it hard to use for production development, such as the lack of incremental compilation (“d2c generates fast code slowly”), uncomfortable debugging and slightly incomplete support for the Dylan language specification. We analysed a pre-release of version 2.5 which we obtained from the SVN repository on 12. March 2008. Including contributions of various libraries, this version has 3799 generic functions and 6621 concrete methods.

6) Vortex (Cecil) and Whirlwind (Diesel): Vortex and Whirlwind are compiler front-ends developed by the Cecil Group [65] at the University of Washington’s Department of Computer Science and Engineering. They are written in Cecil and Diesel respectively and are designed to plug into the Vortex compiler back-end. We use release 3.3 of the Vortex compiler infrastructure and count 6541 generic functions and 15212 concrete methods in the Cecil code, versus 5737 generic functions and 11871 concrete methods in Diesel source.

7) The Nice Compiler (Nice): The Nice language and compiler were developed as part of an academic project on object-orientation at Institut National de Recherche en Informatique et en Automatique (INRIA) in Rocquencourt, France[4]. Presently (version 0.9.13), it is not considered feature complete. The compiler and Nice standard library itself are written partly in Java (some 35,000 lines of code) and partly in Nice (23,000 lines of code). For our analysis we only consider the contribution made by native Nice code, which we found to define 1184 generic functions and 1615 concrete methods.

8) The Location Stack (MultiJava): The MultiJava-based Location Stack [66] is a framework for processing measurements from a heterogenous network of geographical location sensors. Its development began as part of the Portolano project on ubiquitous computing at the University of Washington; the efforts have been since carried forward into the PlaceLab project at Intel Research Seattle, though PlaceLab has abandoned MultiJava in favour of more mainstream Java. We base our analysis on the latest release 0.8, counting 491 generic functions and 735 concrete methods.

V. Results

This section presents the results obtained from the metrics introduced in Section III applied to a corpus of nine applications, as described in Section IV.

The presentation is structured into three sections: Ratios (V-A) summarises the results from the basic Dispatch Ratio (DR) and Choice Ratio (CR) metrics; Specialisation (V-B), presents the results we obtain from the Degree of Specialisation (DoS, DoS_C) and Rightmost Specialiser (RS, RS_C) metrics; and Dispatch (V-C) covers the Degree of Dispatch (DoD) and Rightmost Dispatch (RD) metrics.

The result averages for the metrics we use are summarised in Table IV. Table V of the appendix presents all values we measured across the corpus in a single table.

The final section (V-D) explores the hypothesis that some of our results might be distributed according to power laws, thus making these distributions scale free. We investigate the possibility and explain the implications.

A. Ratios

In this section we present the results from the basic Dispatch Ratio (DR) and Choice Ratio (CR) metrics, which measure basic relations between generic functions and concrete methods.
TABLE IV
METRICS: AVERAGES ACROSS APPLICATIONS

<table>
<thead>
<tr>
<th></th>
<th>Gwydion</th>
<th>OpenDylan</th>
<th>CMUCL</th>
<th>SBCL</th>
<th>McCLIM</th>
<th>Vortex</th>
<th>Whirlwind</th>
<th>NiceC</th>
<th>LocStack</th>
</tr>
</thead>
<tbody>
<tr>
<td>DR_ave</td>
<td>1.74</td>
<td>2.51</td>
<td>2.03</td>
<td>2.37</td>
<td>2.32</td>
<td>2.33</td>
<td>2.07</td>
<td>1.36</td>
<td>1.50</td>
</tr>
<tr>
<td>CR_ave</td>
<td>18.27</td>
<td>43.84</td>
<td>6.34</td>
<td>26.57</td>
<td>15.43</td>
<td>63.30</td>
<td>31.65</td>
<td>3.46</td>
<td>8.92</td>
</tr>
<tr>
<td>DoS_ave</td>
<td>2.14</td>
<td>1.23</td>
<td>1.17</td>
<td>1.11</td>
<td>1.17</td>
<td>1.06</td>
<td>0.71</td>
<td>0.33</td>
<td>1.02</td>
</tr>
<tr>
<td>DoS_G_ave</td>
<td>2.23</td>
<td>1.19</td>
<td>1.11</td>
<td>1.10</td>
<td>1.05</td>
<td>0.85</td>
<td>0.35</td>
<td>0.16</td>
<td>0.85</td>
</tr>
<tr>
<td>RS_ave</td>
<td>2.24</td>
<td>1.34</td>
<td>1.24</td>
<td>1.23</td>
<td>1.39</td>
<td>1.10</td>
<td>0.78</td>
<td>0.34</td>
<td>1.08</td>
</tr>
<tr>
<td>RS_G_ave</td>
<td>2.37</td>
<td>1.29</td>
<td>1.17</td>
<td>1.29</td>
<td>1.26</td>
<td>0.88</td>
<td>0.40</td>
<td>0.16</td>
<td>0.85</td>
</tr>
<tr>
<td>DoD_ave</td>
<td>0.20</td>
<td>0.39</td>
<td>0.42</td>
<td>0.42</td>
<td>0.45</td>
<td>0.36</td>
<td>0.32</td>
<td>0.15</td>
<td>0.08</td>
</tr>
<tr>
<td>RD_ave</td>
<td>0.24</td>
<td>0.48</td>
<td>0.45</td>
<td>0.45</td>
<td>0.54</td>
<td>0.41</td>
<td>0.37</td>
<td>0.15</td>
<td>0.11</td>
</tr>
</tbody>
</table>

1) Dispatch Ratio: The Dispatch Ratio distribution for all applications is shown in Fig. 3. All applications follow a similar distribution with 58–93% of generic functions having a single concrete method. The shares for generic functions with two (2–24%), three (1–8%), and more methods decrease rapidly.

The generic function with the most concrete methods (print_string(a)) has 825 definitions in Vortex. This concrete method shows up among the top polymorphic functions in several other applications (e.g. Whirlwind 462, SBCL 136, OpenDylan 128, LocStack 61). Other generic functions with a high degree of polymorphism include binary operators such as the equality test for two objects (Vortex 257 definitions, Whirlwind 213, OpenDylan 58, Gwydion 42), ‘less than’ and addition operators.

The DR_ave values for the applications in our corpus are shown in Table IV. Six of the applications have a DR_ave measurement of at least 2, indicating that for every generic function, on average a dispatch decision must be made between two concrete functions.

2) Choice Ratio: The results for CR_ave in Table IV show considerable variance. On average, any concrete method in Vortex is part of a dispatch decision with 60 or so other methods, whereas for the NiceC it would be only with 3.5 other methods. This is somewhat surprising for NiceC, which is the only language model we study with a concept of overridable method declarations that allows the same concrete method to be part of multiple generic functions. In reality we found only two NiceC generic functions that take advantage of this feature.

Fig. 4 shows the Choice Ratio distribution for each application separately. The first thing to note is that most concrete methods (25 – 63%) have a Choice Ratio of 1, indicating that each is the sole method in a monomorphic generic function.
We note further that a relatively high proportion of concrete methods have choice ratios of nine and higher. This factor can be attributed to some generic functions with a high degree of polymorphism, that is, a large number of concrete methods. However, rare these generic functions are, the concrete methods they contain are not as rare, as they occur in large numbers. For example, if a single generic function has a DR of 100, then there will be at least 100 concrete methods with CR ≥ 100. The strength of the Choice Ratio metric is the ability to expose this correlation.

B. Specialisation

This section presents the results from the Degree of Specialisation metrics, measured per concrete method (DoS) and per generic function (DoSG).

1) Specialisation of Concrete Methods: Fig. 5a shows, for each application, what proportion of concrete methods have a given DoS measurement. At the top are the highest DoS values measured for the respective application. While it is quite common for methods to specialise up to three parameters, we found generic functions that specialise seven (OpenDylan, make-source-location), eight (Whirlwind, resolve8) and 20 (Gwydion, parser:production_113) parameters. The range of proportions of generic functions with no specialisation varies considerably across the applications.

The results for the RS metric are shown in Fig. 5b. Comparing RS to DoS in Fig. 6, we see that some applications (OpenDylan, SBCL, McCLIM) have significantly higher values for Rightmost Specialiser RS than they do for Degree of Specialisation DoS. This means that some methods’ parameter list must have some non-specialised parameters “to the left of” specialised parameters. Since these methods’ parameters are not ordered based on their specialisation, we hypothesise that programmers make use of the greater flexibility in ordering parameters to enhance code readability. In other applications (Gwydion, CMUCL, Vortex, NiceC), RS and DoS values seem to mostly equal each other. This reflects the fact that for most methods, specialised parameters come first and are then followed by un specials ed parameters. The RS distribution for McCLIM shows the most pronounced deviation from the DoS distribution, as we measure 15% less methods with RS = 1 than with DoS = 1. For RS = 2 the relation is inverted, 12% more methods having RS = 2 than DoS = 2.

Fig. 7 shows scatter plots that account for the relation between DoS and RS for each concrete method. It reveals certain differences between applications: While the values for RS in CMUCL, SBCL and LocStack only exceed DoS values by 1, at least one concrete method in McCLIM has DoS = 1 and RS = 6, meaning that the method only specialises its sixth parameter. We can find a similar case for Whirlwind, with a method that specialises its seventh parameter only.

2) Specialisation of Generic Functions: The Degree of Specialisation and Rightmost Specialiser distributions of generic functions (DoSG and RSg) are similar to their per concrete method siblings (Fig. 8).

C. Dispatch

In this section we first present the results from the Degree of Dispatch (DoD) and Rightmost Dispatch (RD) metrics. We then compare DoD with Dispatch Ratio and Degree of Specialisation values and demonstrate their relationships.

1) Degree of Dispatch: Fig. 9a shows the Degree of Dispatch (DoD). Excluding NiceC and LocStack, most applications have similar levels (2.7–6.5%) of multiple dispatch (DoD > 1), and single dispatch (14–32%). NiceC has the lowest proportion of multiple dispatch (1.0%) among the analysed applications, even though we have excluded that part of the source written in Java. LocStack closely follows with 1.4%.

Across the entire corpus, the share of generic functions that are not required to dispatch dynamically ranges from 62% to 93%; this corresponds nicely with the proportions of generic functions having a single concrete method and thus a Dispatch Ratio of 1.

On average, across all measured applications, we found that around 4% of generic functions utilise multiple dispatch (DoD > 1) and around 22% utilise single dispatch (DoD = 1).

Fig. 9b shows the Rightmost Dispatched parameter (RD). This generally follows DoD, although the proportions are often a little higher for RD ≥ 2 (see Fig. 10). This shows that a significant
The scatter plots in Fig. 11 show the relation between DoD and RD measurements for all generic functions. These plots clearly demonstrate the value of the RD metric, as it is able to capture conditions when functions dispatch on a single parameter found as far right as the sixth (McCLIM) or seventh (Vortex, Whirlwind) number of single-dispatched generic functions have their dispatch decision made on the second or beyond argument supplied in the call.

We repeat a portion of Table IV below to show the averages of the Degree of Dispatch and Rightmost Dispatch metrics. As can be seen, RD is generally a little larger than the Degree of Dispatch (DoD): $RD \geq DoD$ by definition (because dispatch must occur on the RD'th argument, but there could be arguments to the left of it that do not dispatch). RS is higher than DoS for the same reason.

<table>
<thead>
<tr>
<th>Application</th>
<th>DoS</th>
<th>RD</th>
<th>RD/DoS</th>
<th>RS</th>
<th>RD/RS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gwydion</td>
<td>0.42</td>
<td>0.50</td>
<td>0.44</td>
<td>0.40</td>
<td>0.54</td>
</tr>
<tr>
<td>OpenDylan</td>
<td>0.24</td>
<td>0.26</td>
<td>0.24</td>
<td>0.24</td>
<td>0.24</td>
</tr>
<tr>
<td>CMUCL</td>
<td>0.24</td>
<td>0.26</td>
<td>0.24</td>
<td>0.24</td>
<td>0.24</td>
</tr>
<tr>
<td>SBCL</td>
<td>0.24</td>
<td>0.26</td>
<td>0.24</td>
<td>0.24</td>
<td>0.24</td>
</tr>
<tr>
<td>Vortex</td>
<td>0.24</td>
<td>0.26</td>
<td>0.24</td>
<td>0.24</td>
<td>0.24</td>
</tr>
<tr>
<td>Whirlwind</td>
<td>0.24</td>
<td>0.26</td>
<td>0.24</td>
<td>0.24</td>
<td>0.24</td>
</tr>
<tr>
<td>NiceC</td>
<td>0.24</td>
<td>0.26</td>
<td>0.24</td>
<td>0.24</td>
<td>0.24</td>
</tr>
<tr>
<td>LocStack</td>
<td>0.24</td>
<td>0.26</td>
<td>0.24</td>
<td>0.24</td>
<td>0.24</td>
</tr>
</tbody>
</table>

The scatter plots in Fig. 11 show the relation between DoD and RD measurements for all generic functions.
position in the functions formal parameter list. Such generic functions, although they are technically single dispatch, cannot be implemented as is in a single dispatch language, because in single dispatch languages functions only dispatch upon the first argument in a call.

2) Dispatch Ratio and Degree of Dispatch: The proportion of generic functions with DoD = 0 exactly matches the proportion of generic functions with DR = 1 for seven out of the nine applications (see leftmost data points in Fig. 12). A monomorphic function will never need to dispatch dynamically (its DoD is always 0) because there are no alternative implementations to choose from. We infer that the set of monomorphic generic functions is contained in the set of non-dispatched generic functions (these have DR = 1).

Conversely, all polymorphic generic functions will have a Degree of Dispatch of at least 1. This seems logical, since most of the languages we analyse will not allow two concrete methods of the same generic function to have the same set of specialisers; in other words, two concrete methods have to specialise at least one parameter position differently, to allow the dispatch mechanism of the language to choose the single, most applicable method from the two given definitions.
The exception here is Common Lisp, where it is perfectly legal to have several most applicable methods, given that these have different qualifiers (none, one or more of BEFORE, AFTER, AROUND). Common Lisp qualifiers prescribe how methods can be combined together to one runnable entity. CMUCL and McCLIM make some use of this method combination capability, which explains why the proportion of functions with DoD = 0 is slightly lower than the proportion of monomorphic generic functions (DR = 1): a relatively small number of generic functions have DoD = 0 even though they have multiple concrete methods. These methods’ specialiser vectors are identical, but their qualifiers differ — which technically means that they are not designed as alternatives to each other. Rather, these methods are designed to be combined to a single runnable entity — meaning that there is no need for the dispatch mechanism to select one most-applicable code body from a set of alternatives. CMUCL has one generic function with this property (representing 0.2% of total), and McCLIM has 72 (3.2%).

For Dispatch Ratios higher than 1 and Degrees of Dispatch higher than 2, the correlation is less obvious, with the share for DR = 2 being consistently and considerably lower than the share for DoD = 1 (as can be seen from Table V; for example, Gwydion has roughly 7% of generic functions with a DR of 2 while double that proportion have DoD = 1).

The above relationship is reversed for DR > 3 and DoD > 2, from which point on the Degree of Dispatch stays consistently below the Dispatch Ratio (see Fig. 12). In other words, the Degree of Dispatch distributions consistently lean to the left of the Dispatch Ratio distributions; also, the “tail” of the Dispatch Ratio distribution curve is much longer (which Fig. 12 does not show). The explanation for this phenomenon is simple: although we found generic functions with a large number of concrete methods (i.e., a DR as large as many hundreds; see “Max” column in Table V), the corresponding DoD is much lower (the highest DoD measured across the whole corpus is 8). This shows to some extent that two guidelines for good program design are being followed by a) using polymorphic functions to confine the complexity of a program and b) keeping a function’s list of parameters short to prevent losing their count\(^1\).

Fig. 13 shows the relation between DR and DoD metrics from a different perspective. The dots represent the intersection between DR and DoD values for each generic function across the corpus, on a per-application basis. The resulting plots reflects the fact that, although both of these metrics measure generic functions, they do so along two different dimensions: DR measures the degree of polymorphism, expressed by the number concrete methods in a generic function, while the DoD’s upper boundary is set by the length of the list of parameters to that function.

3) Specialisation and Dispatch: Fig. 14 compares DoS\(_G\) and DoD distributions. It shows that the dispatch metrics DoD and RD are generally below the specialiser metrics DoS\(_G\) and RS\(_G\), because generic functions dispatch on specialised positions, but

\(^1\)Alan Perlis pointedly addressed both these concerns in his *Epigrams on Programming*: “6. Symmetry is a complexity-reducing concept (co-routines include sub-routines); seek it everywhere.” and “11. If you have a procedure with ten parameters, you probably missed some.” [67]
among related studies of corpus analysis, some have sought to provide evidence that many important relationships between software artifacts follow a power law distribution ([46], [47], [68]).

The lines shown in Fig. 3 for the Dispatch Ratio distribution are reminiscent of logarithmic curves. This observation prompts us to investigate the possibility of the Dispatch Ratio following a power law distribution.

As the curves seem fairly close to each other, we first plot all values on the same log-log scale (Fig. 15). To reduce the noise caused by high DR values, which are, by their nature, very rare, the array of measured DR values is first transformed into a vector of consecutive ranks. The strong indication of a straight line is further evidence of the possibility that power laws are being followed.

Fig. 16 shows separate logarithmic plots for each application’s Dispatch Ratio distribution. While most distributions are indeed reminiscent of straight lines, there is considerable spread between how close the applications follow a power law: Gwydion, Vortex and Whirlwind fit the power law distribution quite closely, while NiceC and LocStack hardly follow a straight line. Baxter et al. [47] came to similar results in their large-scale analysis of Java code. They surmise that certain attributes of the application’s code (such as its design or domain) may affect the resulting distribution.

All samples show a considerable spread for higher ranks, which reflects the fact that many generic functions with a high Dispatch Ratio have a low frequency of one, two or three. In other words, generic functions with a certain very high number of concrete methods (i.e. several hundred) rarely occur more than once in an application.

VI. CONCLUSION

In this Section we summarise the contributions of our research, discuss our findings and look at potential directions for future work.

A. Contributions

This paper presents an empirical study of multiple dispatch in object-oriented languages. To our knowledge it is the first cross-language corpus analysis of multiple dispatch. The main contributions of this study are a language-independent model for multiple dispatch; a suite of metrics to measure the use of multiple dispatch; and the results of measuring a corpus in six multiple dispatch languages, which together provide a quantitative
1. We discussed parameter specialisation practices with programmers in person and on the Lisp and Dylan IRC channels.

We presented the values measured according to each metric for a dispatch to a concrete method.

The Template Method pattern [8], for example, will contribute to this effect, as only “hook methods” should be overridden in subclasses, while methods providing abstract, concrete, and primitive operations will not be overridden.

Our metrics cannot say anything about how important multiple dispatch (or even single dispatch) is to program design: simply that many methods are monomorphic, and most of the remainder use single dispatch. Those dispatching methods may be crucial to the functioning of a particular program — as well as Template method, many other patterns (Visitor, Observer, Strategy, State, Composite) are about scaffolding a well-chosen dynamic dispatch with lots of relatively straightforward non-dispatching code.

Another point here is that a language specification does not dictate a programming style. Just supporting multiple (or even single) dispatch in a programming language doesn’t mean it will be used in programs, the Nice compiler being a prime example.

2. **A Language-independent Model for Multiple Dispatch:** We have developed a model of multiple dispatch that unifies language-specific concepts to support reasoning about multiple dispatch in a language-independent way. The model (represented in Fig. 1) defines five main entities: A generic function is a function that may be dynamically dispatched. Each generic function dispatches to one or more concrete methods. Concrete methods specify their applicability by defining specialisers for any of the function’s formal parameters. A generic function and its associated concrete methods are referred to by a name and share a signature defining the number of permissible arguments and, if applicable, the static types for each formal argument of that function.

3. **A Metrics Suite for Measuring Multiple Dispatch:** We defined six metrics to measure multiple dispatch: the Dispatch Ratio and Choice Ratio measure basic relations between generic functions and concrete methods. The Degree of Specialisation and Rightmost Specialiser both capture the use of formal parameter specialisers in methods and generic functions. The Degree of Dispatch and Rightmost Dispatch determine the number and position of formal parameters needed for a generic function to dispatch to a concrete method.

4. **An Evaluation of the Use of Multiple Dispatch in Practice:** We presented the values measured according to each metric for a corpus of nine programs written in six multiple dispatch languages: CLOS, Dylan, Cecil, Diesel, Nice and MultiJava, thus providing quantitative evidence of the use of multiple dispatch in practice.

In answer to our introductory question how much is multiple dispatch used?, we found that around 4% of generic functions utilise multiple dispatch and around 22% utilise single dispatch. Determining how much these results generalise — i.e., how well these measurements represent the use of multiple dispatch in other applications and languages — necessarily requires further study, but we expect these results to provide a benchmark for comparison.

**B. Discussion**

There are a number of inferences which can be drawn from the results presented in Section V. Perhaps the most obvious is that many of the metric values are low. Every language we measured had more than 55% monomorphic generic functions; less than 7% of functions dispatch on two or more arguments (Fig. 9). This is reflected in Dispatch Ratio $DR_{ave}$ values: no language had more than an average of 2.5 concrete methods for each generic function. On average, a function dispatches only on 0.1 to 0.7 arguments (Table IV, $DoD_{ave}$ values).

1. **Monomorphic vs Polymorphic Functions:** It seems that in all of our studied languages, there will be many generic functions that do not dispatch: static methods, constructors, but also auxiliary methods, methods that provide default argument values in languages without variable argument lists or keyword arguments. On the other hand, there are a significant number of generic functions that do dispatch to three or more different concrete methods — and the methods belonging to those functions make up a substantial fraction of the program’s methods.

The Template Method pattern [8], for example, will contribute to this effect, as only “hook methods” should be overridden in subclasses, while methods providing abstract, concrete, and primitive operations will not be overridden.

In answer to our introductory question how much is multiple dispatch used?, we found that around 4% of generic functions utilise multiple dispatch and around 22% utilise single dispatch. Determining how much these results generalise — i.e., how well these measurements represent the use of multiple dispatch in other applications and languages — necessarily requires further study, but we expect these results to provide a benchmark for comparison.

2. We discussed parameter specialisation practices with programmers in person and on the Lisp and Dylan IRC channels.
scope of a method by stating the types of objects it is designed to handle. How and which specialisers are used to dispatch is not the programmer’s focus of attention. This is regarded as the key advantage of multiple dispatch over hand-coded alternatives (instanceof-tests, double dispatch pattern). Those alternatives are low-level techniques, as they force developers to focus on how to simulate multiple dispatch. Multiple dispatch is a high-level language feature because it does not reflect the programmer’s attention from the actual programming problem.

b) Specialisation Patterns: Comparing RS and DoS distributions in Fig. 6 we find that for some applications (mainly Gwydion, CMUCL and Vortex) these distribution closely follow each other, while for OpenDylan, SBCL and McCLIM there is a larger proportion of methods with RS > DoS. From these observations we can tell that Gwydion, CMUCL and Vortex tend to specialise parameters left-to-right, then following with unspecialised parameters; the applications that fall into the second group do not order parameters based on specialisation.

c) Dispatch: Comparing RD and DoD metrics in Table IV (averages), Fig. 10 (distributions) and Fig. 11 (values) we see that some applications (primarily McCLIM, and OpenDylan, but also Gwydion, Vortex and Whirlwind) have significantly higher values for Rightmost Dispatched parameter RD than they do for Degree of Dispatch DoD. This means that some generic functions’ argument lists must have some non-dispatching parameters “to the left of” the dispatching parameters — in contrast to single-dispatch languages where the dispatch is always on the single leftmost parameter. For example, programs contain two-argument generic functions which dispatch on the second argument but not on the first.

In the case of McCLIM, this must partly be explained by the fact that the CLIM Standard [62] explicitly requires some types of generic functions to dispatch on their second arguments (setf and mapping functions). More generally, multiple dispatch gives more options to API designers, who can choose argument order to reflect application semantics rather than be restricted by having to place a dispatching argument first. In single dispatch languages, code can fall into a “Object Verb Subject” order: rectangle.drawOn(window). Here, Rectangle must come first, purely because the code needs to dispatch on Rectangle to draw different kinds of figures. In multiple dispatch languages, this could equally be written window.draw(rectangle) matching the “Subject Verb Object” word order commonly used in English, or perhaps “Verb Subject Object” draw(window,rectangle). Multiple dispatch languages offer this flexibility, even where only single dispatch is required, and our metrics demonstrate that programmers take advantage of this flexibility.

3) A Critical Perspective: The corpus we use in this study may appear relatively small when compared to corpora used in studies taking similar methodological approaches ([47], [55], the Java study of [43]). This largely reflects the fact that multiple dispatch languages are not in wide use today. For most languages covered, we only analyse a single application (the exceptions are CLOS with three and Dylan with two applications). Furthermore, most applications are compilers and their standard libraries. Therefore we cannot claim that our results are representative for the use of multiple dispatch in any of these languages.

Our study provides “hard evidence” in terms of concrete numbers that account for the use of multiple dispatch in existing programs. Still, these numbers only express measurements based on the information furnished by compilers through their respective interfaces.

Finally, the strength of our approach comes from its diversity. Our study analyses a broad range of six languages and eight compilers, yet when comparing the results among each other, their similarity is striking. Function polymorphism, as measured by the Dispatch Ratio metric, is distributed in all applications according to the same logarithmic curve (Fig. 3); methods generally specialise one single parameter (Fig. 5); and the proportions of static (non-dispatched), single-dispatch and multiple dispatch functions uniformly follow the same distribution across the whole corpus (Fig. 9). This common result across a range of languages, libraries, applications, implementations and analysis instruments increases our confidence in the large-scale findings of our study.

C. Evidence Based Language Design

I have always remark’d, that the author proceeds for some time in the ordinary ways of reasoning…when all of a sudden I am surpriz’d to find, that instead of the usual copulations of propositions, is, and is not, I meet with no proposition that is not connected with an ought, or an ought not.


Hume’s Law states that normative (prescriptive) statements — in this case, statements about how programs ought to be written — cannot be justified exclusively by descriptive statements. Our corpus analysis is descriptive: it tells us about how programs are written, but cannot (on its own) tell us about whether that is a “good” way to write programs, or whether language designers should consider multiple dispatch (or even single dispatch) as a language feature worth retaining. In this paper, we do not try to make any of these claims — we do not even claim whether high or low values for metrics are desirable: our metrics characterise program structures: they do not attempt to measure program quality.

Nonetheless, there seem to be clear advantages to informing the design of future languages with evidence drawn by something other than anecdote, personal experience or small-scale observational studies. Similarly, maintenance and debugging tasks — and even teaching about programming paradigms — would surely benefit from being based in evidence about the world as it is, as well as the world as we would like it to be!

D. Future Work

Our study is but a beginning in a line of research aiming at the question how much value is gained in practice by the use of multiple dispatch?, and we hope our work will inspire more studies. This section lists possible directions for future research in this area.

1) Method Calls and Dynamic Aspects of Multimethods: Our quantitative study of multiple dispatch approaches the use of multimethods from a static point of view. We focus on method definitions which we examine at compile-time. We do not examine method calls or dynamic aspects of a program, such as the frequency of method calls through a call site or the frequency of invocations per method. Consequently, we would like to see these aspects covered in future studies.
2) Object-Orientation and Beyond: While single dispatch naturally fits the class- or object-based method encapsulation approach of mainstream programming languages, multiple dispatch seems to be the natural solution for a different category of problems. These problems occur whenever behaviour depends on the types of multiple objects being used together (combined), and cannot be regarded as an invariant property of a single type of object, as are methods according to the object-oriented paradigm [69].

Clifton et al. [45, Section 5.1] present anecdotal evidence for some programming scenarios where multiple dispatch is used in MultiJava projects. These include binary methods, event handling, tree traversals, and finite state machines. Our study complements their findings by providing quantified evidence that multiple dispatch is used for solving existing programming problems. However, further qualitative studies are needed to identify and classify the real-world problems for which multiple dispatch provides an adequate solution — and for which traditional object-oriented languages do not.

3) Java Method Overloading: One aspect to consider about this study is that most languages examined here are primarily dynamic, in the sense that even if they support static type declarations, these are purely optional. Java, C++ and C#, on the other hand, have a mandatory prescriptive static type system which can be used to overload method definitions. Dispatch to these overloaded methods occurs at compile time, which, as the examples in Section II-A show, can lead to incorrect behaviour. Due to its error potential, method overloading is generally considered bad programming style and discouraged [70]–[72].

Because the multiple dispatch facility of the languages covered in this study subsumes static overloading of methods, we cannot tell how many programming problems that are solved using multiple dispatch in these languages are customarily solved for example in Java using static method overloading. We note however that Nice and MultiJava, the two languages which support both Java-like static overloading and dynamically dispatched multimethods, exhibit a significantly lower proportion of multiple dispatch (1.0–1.4%) than applications in other languages (on average 4.9%), as measured by the Degree of Dispatch metric.

We hope to see studies of method overloading in Java. Such studies could reveal the extent to which method overloading is used in existing programs and strengthen the case for multiple dispatch as a safe, upward compatible alternative.

APPENDIX I

RESULTS

Table V contains the frequency distributions measured by each metric. The “Max.” column gives the highest value measured for the given application. Subsequent columns show the absolute number and proportion (in percent) of measured subjects (generic functions or concrete methods) with the measurement given in the corresponding column header. The last column shows the sum for subjects with a metric value of 9 or higher.

REFERENCES


TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. X, NO. X, AUGUST XXXX

20

TABLE V
M ETRICS : FREQUENCY DISTRIBUTIONS
Application
Gwydion
Gwydion
Gwydion
Gwydion
Gwydion
Gwydion
Gwydion
Gwydion
OpenDylan
OpenDylan
OpenDylan
OpenDylan
OpenDylan
OpenDylan
OpenDylan
OpenDylan
CMUCL
CMUCL
CMUCL
CMUCL
CMUCL
CMUCL
CMUCL
CMUCL
SBCL
SBCL
SBCL
SBCL
SBCL
SBCL
SBCL
SBCL
McCLIM
McCLIM
McCLIM
McCLIM
McCLIM
McCLIM
McCLIM
McCLIM
Vortex
Vortex
Vortex
Vortex
Vortex
Vortex
Vortex
Vortex
Whirlwind
Whirlwind
Whirlwind
Whirlwind
Whirlwind
Whirlwind
Whirlwind
Whirlwind
NiceC
NiceC
NiceC
NiceC
NiceC
NiceC
NiceC
NiceC
LocStack
LocStack
LocStack
LocStack
LocStack
LocStack
LocStack
LocStack

Metric
DR
CR
DOS
DOSG
RS
RS G
DOD
RD
DR
CR
DOS
DOSG
RS
RS G
DOD
RD
DR
CR
DOS
DOSG
RS
RS G
DOD
RD
DR
CR
DOS
DOSG
RS
RS G
DOD
RD
DR
CR
DOS
DOSG
RS
RS G
DOD
RD
DR
CR
DOS
DOSG
RS
RS G
DOD
RD
DR
CR
DOS
DOSG
RS
RS G
DOD
RD
DR
CR
DOS
DOSG
RS
RS G
DOD
RD
DR
CR
DOS
DOSG
RS
RS G
DOD
RD

Max.
204
204
20
20
20
20
4
6
362
362
7
7
7
7
4
5
29
29
3
3
4
4
3
4
136
136
3
3
4
4
3
4
160
160
5
6
6
6
5
6
825
825
6
6
8
8
4
7
462
462
8
8
8
8
8
8
24
24
3
3
4
4
3
4
61
61
3
3
3
3
2
3

0
0
0
380
355
380
355
3167
3167
0
0
349
130
349
130
1459
1459
0
0
49
11
49
11
177
177
0
0
63
10
63
10
231
231
0
0
343
121
343
121
1175
1175
0
0
1839
1737
1839
1737
4441
4441
0
0
5065
4055
5065
4055
4180
4180
0
0
1131
1016
1131
1016
1025
1025
0
0
92
92
92
92
458
458

%0
0.00
0.00
5.74
9.34
5.74
9.34
83.36
83.36
0.00
0.00
6.48
6.07
6.48
6.07
68.08
68.08
0.00
0.00
8.91
4.06
8.91
4.06
65.31
65.31
0.00
0.00
7.32
2.75
7.32
2.75
63.64
63.64
0.00
0.00
7.76
6.35
7.76
6.35
61.65
61.65
0.00
0.00
12.09
26.56
12.09
26.56
67.89
67.89
0.00
0.00
42.67
70.68
42.67
70.68
72.86
72.86
0.00
0.00
70.03
85.81
70.03
85.81
86.57
86.57
0.00
0.00
12.52
18.74
12.52
18.74
93.28
93.28

1
3167
3167
2404
1536
2331
1499
528
405
1459
1459
3684
1538
3231
1360
545
401
176
176
372
222
351
210
77
72
231
231
647
311
570
245
115
106
1103
1103
3153
1596
2499
1253
614
491
4441
4441
10900
4140
10668
4060
1843
1629
4180
4180
5354
1432
4871
1244
1331
1141
1025
1025
440
153
420
148
147
143
458
458
553
385
553
385
26
19

%1
83.36
47.83
36.31
40.43
35.21
39.46
13.90
10.66
68.08
27.07
68.36
71.77
59.96
63.46
25.43
18.71
64.94
32.00
67.64
81.92
63.82
77.49
28.41
26.57
63.64
26.83
75.15
85.67
66.20
67.49
31.68
29.20
57.87
24.96
71.35
83.74
56.55
65.74
32.21
25.76
67.89
29.19
71.65
63.29
70.13
62.07
28.18
24.90
72.86
35.21
45.10
24.96
41.03
21.68
23.20
19.89
86.57
63.47
27.24
12.92
26.01
12.50
12.42
12.08
93.28
62.31
75.24
78.41
75.24
78.41
5.30
3.87

2
261
522
2071
952
2072
970
99
176
312
624
1172
425
1503
568
126
239
55
110
118
34
123
44
13
16
77
154
140
38
194
102
13
20
448
896
747
167
1266
469
108
196
1038
2076
2255
584
2271
592
232
367
823
1646
1288
213
1499
319
196
305
91
182
42
13
58
15
10
13
9
18
73
12
30
8
7
8

%2
6.87
7.88
31.28
25.06
31.29
25.53
2.61
4.63
14.56
11.58
21.75
19.83
27.89
26.50
5.88
11.15
20.30
20.00
21.45
12.55
22.36
16.24
4.80
5.90
21.21
17.89
16.26
10.47
22.53
28.10
3.58
5.51
23.50
20.28
16.90
8.76
28.65
24.61
5.67
10.28
15.87
13.65
14.82
8.93
14.93
9.05
3.55
5.61
14.35
13.87
10.85
3.71
12.63
5.56
3.42
5.32
7.69
11.27
2.60
1.10
3.59
1.27
0.84
1.10
1.83
2.45
9.93
2.44
4.08
1.63
1.43
1.63

3
135
405
912
216
877
129
4
36
125
375
153
39
255
68
11
37
13
39
11
4
24
5
4
5
23
69
11
4
31
5
4
5
148
444
166
18
207
47
8
33
402
1206
200
68
285
114
24
81
282
846
149
26
347
79
22
74
33
99
2
2
4
4
2
2
6
18
17
2
60
6
0
6

%3
3.55
6.12
13.77
5.69
13.25
3.40
0.11
0.95
5.83
6.96
2.84
1.82
4.73
3.17
0.51
1.73
4.80
7.09
2.00
1.48
4.36
1.85
1.48
1.85
6.34
8.01
1.28
1.10
3.60
1.38
1.10
1.38
7.76
10.05
3.76
0.94
4.68
2.47
0.42
1.73
6.15
7.93
1.31
1.04
1.87
1.74
0.37
1.24
4.92
7.13
1.26
0.45
2.92
1.38
0.38
1.29
2.79
6.13
0.12
0.17
0.25
0.34
0.17
0.17
1.22
2.45
2.31
0.41
8.16
1.22
0.00
1.22

4
66
264
365
301
450
385
1
12
78
312
11
8
23
13
2
6
10
40
0
0
3
1
0
1
10
40
0
0
3
1
0
1
78
312
9
3
58
8
0
5
221
884
15
10
79
28
1
18
158
632
13
9
52
24
7
22
10
40
0
0
2
1
0
1
1
4
0
0
0
0
0
0

%4
1.74
3.99
5.51
7.92
6.80
10.13
0.03
0.32
3.64
5.79
0.20
0.37
0.43
0.61
0.09
0.28
3.69
7.27
0.00
0.00
0.55
0.37
0.00
0.37
2.75
4.65
0.00
0.00
0.35
0.28
0.00
0.28
4.09
7.06
0.20
0.16
1.31
0.42
0.00
0.26
3.38
5.81
0.10
0.15
0.52
0.43
0.02
0.28
2.75
5.32
0.11
0.16
0.44
0.42
0.12
0.38
0.84
2.48
0.00
0.00
0.12
0.08
0.00
0.08
0.20
0.54
0.00
0.00
0.00
0.00
0.00
0.00

5
38
190
95
54
64
23
0
2
56
280
19
2
27
3
0
1
6
30
0
0
0
0
0
0
6
30
0
0
0
0
0
0
33
165
1
0
36
4
1
2
113
565
2
1
64
5
0
2
55
275
0
0
30
11
0
11
5
25
0
0
0
0
0
0
3
15
0
0
0
0
0
0

%5
1.00
2.87
1.43
1.42
0.97
0.61
0.00
0.05
2.61
5.20
0.35
0.09
0.50
0.14
0.00
0.05
2.21
5.45
0.00
0.00
0.00
0.00
0.00
0.00
1.65
3.48
0.00
0.00
0.00
0.00
0.00
0.00
1.73
3.73
0.02
0.00
0.81
0.21
0.05
0.10
1.73
3.71
0.01
0.02
0.42
0.08
0.00
0.03
0.96
2.32
0.00
0.00
0.25
0.19
0.00
0.19
0.42
1.55
0.00
0.00
0.00
0.00
0.00
0.00
0.61
2.04
0.00
0.00
0.00
0.00
0.00
0.00

6
29
174
180
171
162
153
0
1
26
156
0
0
0
0
0
0
1
6
0
0
0
0
0
0
0
0
0
0
0
0
0
0
32
192
0
1
10
4
0
4
68
408
1
1
4
3
0
2
45
270
1
1
4
2
0
2
3
18
0
0
0
0
0
0
1
6
0
0
0
0
0
0

%6
0.76
2.63
2.72
4.50
2.45
4.03
0.00
0.03
1.21
2.89
0.00
0.00
0.00
0.00
0.00
0.00
0.37
1.09
0.00
0.00
0.00
0.00
0.00
0.00
0.00
0.00
0.00
0.00
0.00
0.00
0.00
0.00
1.68
4.34
0.00
0.05
0.23
0.21
0.00
0.21
1.04
2.68
0.01
0.02
0.03
0.05
0.00
0.03
0.78
2.27
0.01
0.02
0.03
0.03
0.00
0.03
0.25
1.11
0.00
0.00
0.00
0.00
0.00
0.00
0.20
0.82
0.00
0.00
0.00
0.00
0.00
0.00

7
28
196
54
54
4
4
0
0
5
35
1
1
1
1
0
0
0
0
0
0
0
0
0
0
2
14
0
0
0
0
0
0
7
49
0
0
0
0
0
0
58
406
0
0
1
1
0
1
39
273
0
0
1
1
0
1
3
21
0
0
0
0
0
0
2
14
0
0
0
0
0
0

%7
0.74
2.96
0.82
1.42
0.06
0.11
0.00
0.00
0.23
0.65
0.02
0.05
0.02
0.05
0.00
0.00
0.00
0.00
0.00
0.00
0.00
0.00
0.00
0.00
0.55
1.63
0.00
0.00
0.00
0.00
0.00
0.00
0.37
1.11
0.00
0.00
0.00
0.00
0.00
0.00
0.89
2.67
0.00
0.00
0.01
0.02
0.00
0.02
0.68
2.30
0.00
0.00
0.01
0.02
0.00
0.02
0.25
1.30
0.00
0.00
0.00
0.00
0.00
0.00
0.41
1.90
0.00
0.00
0.00
0.00
0.00
0.00

8
14
112
84
84
185
185
0
0
18
144
0
0
0
0
0
0
1
8
0
0
0
0
0
0
3
24
0
0
0
0
0
0
9
72
0
0
0
0
0
0
28
224
0
0
1
1
0
0
25
200
1
1
2
2
1
1
2
16
0
0
0
0
0
0
0
0
0
0
0
0
0
0

%8
0.37
1.69
1.27
2.21
2.79
4.87
0.00
0.00
0.84
2.67
0.00
0.00
0.00
0.00
0.00
0.00
0.37
1.45
0.00
0.00
0.00
0.00
0.00
0.00
0.83
2.79
0.00
0.00
0.00
0.00
0.00
0.00
0.47
1.63
0.00
0.00
0.00
0.00
0.00
0.00
0.43
1.47
0.00
0.00
0.01
0.02
0.00
0.00
0.44
1.68
0.01
0.02
0.02
0.03
0.02
0.02
0.17
0.99
0.00
0.00
0.00
0.00
0.00
0.00
0.00
0.00
0.00
0.00
0.00
0.00
0.00
0.00

9+
61
1591
76
76
96
96
0
0
64
2004
0
0
0
0
0
0
9
141
0
0
0
0
0
0
11
299
0
0
0
0
0
0
48
1186
0
0
0
0
0
0
172
5002
0
0
0
0
0
0
130
3549
0
0
0
0
0
0
12
189
0
0
0
0
0
0
11
202
0
0
0
0
0
0

%9+
1.61
24.03
1.15
2.00
1.45
2.53
0.00
0.00
2.99
37.19
0.00
0.00
0.00
0.00
0.00
0.00
3.32
25.64
0.00
0.00
0.00
0.00
0.00
0.00
3.03
34.73
0.00
0.00
0.00
0.00
0.00
0.00
2.52
26.84
0.00
0.00
0.00
0.00
0.00
0.00
2.63
32.88
0.00
0.00
0.00
0.00
0.00
0.00
2.27
29.90
0.00
0.00
0.00
0.00
0.00
0.00
1.01
11.70
0.00
0.00
0.00
0.00
0.00
0.00
2.24
27.48
0.00
0.00
0.00
0.00
0.00
0.00




Radu Muschevici Radu Muschevici is a PhD student in Computer Science at Katholieke Universiteit Leuven, Belgium. He received a BSc (Hons.) degree from the University of Applied Sciences, Munich, Germany and a MSc degree from Victoria University of Wellington, New Zealand. Radu’s current research involvement is in component-oriented programming and software product line engineering. His particular interests lie in exploring ways to express feature and platform models, and integrate them with behavioural components of programming languages.

Alex Potanin Alex Potanin is originally from Moscow, Russia with a background in Mathematics. Alex has completed his PhD in programming languages in 2006 and took up a job as a Lecturer in Software Engineering at Victoria University of Wellington. During his studies he took short breaks to work as a Visiting Researcher at Purdue University, and Software Engineer at two Wellington start-ups.

Dave Clarke Dave Clarke is a professor in the Department of Computer Science at the Katholieke Universiteit Leuven, Belgium. He received his PhD from the University of New South Wales in Sydney, Australia, and he continued his scientific career as a postdoctoral researcher in Utrecht University, and as a scientific staff member at the CWI in Amsterdam. In his PhD thesis, and in several influential ECOOP and OOPSLA publications, Dave has pioneered and further developed the notion of ownership types to control aliasing in object-oriented languages. His research covers programming languages, type systems, automata and logics for improving the quality of software, in particular, with applications in coordination, software product line engineering, and secure software.
James Noble is Professor of Computer Science and Software Engineering at Victoria University of Wellington, New Zealand. His research centres around software design. This includes the design of the users’ interface, the parts of software that users have to deal with every day, and the programmers’ interface, the internal structures and organisations of software that programmers see only when they are designing, building, or modifying software. James’s research in both of these areas is coloured by his longstanding interest in object-oriented approaches to design, and topics he has studied range from aliasing and object ownership, design patterns, agile methodology, via usability, visualisation and computer music, to postmodernism and the semiotics of programming.
Appendix K

Implementing Software Product Lines using Traits

Implementing Software Product Lines using Traits

Lorenzo Bettini Ferruccio Damiani
Dipartimento di Informatica, Università di Torino,
C.so Svizzera, 185 - 10149 Torino, Italy
{bettini,damiani}@di.unito.it

Ina Schaefer †
Chalmers University of Technology,
421 96 Gothenburg, Sweden
schaef@chalmers.se

ABSTRACT
A software product line (SPL) is a set of software systems with well-defined commonalities and variabilities that are developed by managed reuse of common artifacts. In this paper, we present a novel approach to implement SPL by fine-grained reuse mechanisms which are orthogonal to class-based inheritance. We introduce the Featherweight Record-Trait Java (FRTJ) calculus where units of product functionality are modeled by traits, a construct that was already shown useful with respect to code reuse, and by records, a construct that complements traits to model the variability of the state part of products explicitly. Records and traits are assembled in classes that are used to build products. This composition of product functionalities is realized by explicit operators of the calculus, allowing code manipulations for modeling product variability. The FRTJ type system ensures that the products in the SPL are type-safe by type-checking only once the records, traits and classes shared by different products. Moreover, type-safety of an extension of a (type-safe) SPL can be guaranteed by checking only the newly added parts.

Categories and Subject Descriptors
D.3.1 [Programming Languages]: Formal Definitions and Theory; D.3.3 [Programming Languages]: Language Constructs and Features; F.3.3 [Studies of Program Constructs]: Type Structure

General Terms
Design, Languages, Theory

Keywords
Featherweight Java, Feature Model, Software Product Line, Trait

†This author has been partially supported by the Deutsche Forschungsgemeinschaft (DFG).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC’10 March 22-26, 2010, Sierre, Switzerland.
Copyright 2010 ACM 978-1-60558-638-0/10/03 ...$10.00.

1. INTRODUCTION
A software product line (SPL) is a set of software systems with well-defined commonalities and variabilities [12]. SPL engineering aims at developing these systems by managed reuse. Products are implemented by artifacts from a common product line artifact base. The reuse mechanisms for the implementation have to be flexible enough to statically express SPL variability appropriately. Additionally, they should provide static guarantees that the resulting products are type-safe. In order to be of effective use, the type-checking has to facilitate the analysis of newly added parts without re-checking already existing products.

Products of a SPL are commonly described in terms of features, where a feature is a unit of product functionality. In feature-oriented programming (FOP) [5], the artifacts of a SPL are organized in feature modules. A product is the result from a composition of features represented as feature modules. FOP approaches are typically based on the OO paradigm (see, e.g., [3, 13]). The rigid structure of class-based inheritance puts limitations on the effective modeling of product variability and on the reuse of code (in particular, code reuse can be exploited only from within a class hierarchy) [23, 14]. FOP approaches overcome the limitations of class-based inheritance by representing a feature module by a collection of class definitions and class refinements. A class refinement can modify an existing class by adding new fields/methods, by wrapping code around existing methods or by changing its parent. The composition of a feature module with an existing product introduces new classes and/or alters existing ones. Therefore, in different products the same class name can refer to different class definitions.

In this paper, we explore a novel approach to the development of SPL that provides flexible code reuse with static guarantees. The main idea is to overcome the limitations of class-based inheritance with respect to code reuse by replacing it with trait composition. A trait is a set of methods, completely independent from any class hierarchy. In the original proposals of traits [27, 14] (and in most of the subsequent formulation of traits within a Java-like nominal type system [29, 24, 26, 21]) trait composition and class-based inheritance live together as the main reuse construct and as a conceptual design tool, respectively. However, a taxonomy represented by a class hierarchy is an obstacle in implementing SPL. It limits the possibilities for composing products from building blocks in an arbitrary way. Therefore, in our approach, class-based inheritance is ruled out and classes are built only by composition of traits, interfaces and records (a construct, introduced in [9], representing the counterpart of traits with respect to the state). The presented approach separates the concepts of types, state, behavior into different and orthogonal linguistic concepts (interfaces, records and traits, respectively). These are the reusable building blocks that can be assembled into classes that are re-used in several products.
We formalize our approach in \textsc{Featherweight Record-Trait} Java (FRTJ), a minimal core calculus (in the spirit of FJ [15]) for interfaces, records, traits and classes. SPL are implemented in three layers. First, the FRTJ language is used for programming records, traits and interfaces which are assembled into classes. Second, a product is specified by the classes it uses. Third, a SPL is described by its products and its \textit{artifact base}, consisting of the records, traits, interfaces and classes used to build the products of the SPL. The type system of FRTJ provides static guarantees on safe and consistent class assembly from records, traits and interfaces. Furthermore, it ensures that a SPL is type-safe (i.e., that all the products in the product line are type-safe). The approach easily allows for the safe extension of product lines by products using new classes, interfaces, traits, records and by providing means to efficiently type-check only the new parts.

The main differences of the trait-oriented approach presented in this paper with respect to the approaches based on FOP and class-based inheritance, such as [17, 28, 5, 3, 13], are as follows:

- The efficient modeling of SPL variability and the associated code reuse are only achieved by trait and record composition operations (for creating new traits/records by removing, aliasing, and renaming members from already defined traits/records), without introducing, e.g., feature modules and class refinements.
- The classes, interfaces, records and traits of all the products coexist in the artifact base. Generation of a single product just amounts to selecting a subset of these artifacts. Therefore, a class/interface/record name refers to the same definition entry in all the products.
- The type system of FRTJ ensures that a SPL is type-safe by type-checking the artifacts in the artifact base only once. Type-safety of an extension of a (type-safe) product line can be guaranteed by considering only the newly added parts.

FRTJ programs may look more verbose than standard class-based programs; however, the degree of re-use provided by records and traits is higher than the re-use potential of standard static class-based hierarchies. The intent of this paper is not to present the calculus FRTJ in itself, but to formalize the implementation of SPL based hierarchies. The organization of the paper is as follows:

- A basic trait defines a set of methods and declarations of required fields, that parametrize the behavior itself, and of required methods, that can be directly accessed in the body of the provided methods. Traits are building blocks to compose classes or other, more complex, traits. A suite of trait composition operations allows the programmer to build classes and composite traits. A distinguished characteristic of traits is that the composite unit (class or trait) has complete control over conflicts that may arise during composition and must solve them explicitly. Traits do not specify any state, therefore a class composed by using traits has to provide the required fields. The trait composition operations considered in this paper are as follows:
  - A basic trait defines a set of methods and declarations of required fields and required methods.
  - The symmetric sum operation, \(+\), merges two traits to form a new trait. It requires that the summed traits must be disjoint (that is, they must not provide identically named methods).
  - The operation \textit{exclude} forms a new trait by removing a method from an existing trait.
  - The operation \textit{aliasAs} forms a new trait by giving a new name to an existing method.
  - The operation \textit{renameTo} creates a new trait by renaming all the occurrences of a required field name or of a required/provided method name from an existing trait.

Note that the actual names of the methods defined in a trait (and also the names of the required methods and fields) are irrelevant, since they can be changed by the renameTo operation.

A record is a set of fields, completely independent from any class hierarchy. Records have been recently proposed in [9] as the counterpart of traits with respect to state to play the role of \textit{units for state fine-grained reuse}. The common state (i.e., the common fields) of a set of classes can be factored into a record. Records are building blocks to compose classes or other, more complex, records by means of operations analogous to the ones described above for traits. The record construct considered in this paper enhances the original one [9] by providing a richer set of composition operations.

In the following, we illustrate the trait and record constructs by an example implementation of bank accounts (cf. [13]). We use a \textsc{Java}-like notation and a more general syntax (including, e.g., the types \text{void} and \text{boolean}, the assignment operator, etc.) than the one of the FRTJ calculus presented in Section 4. We omit the class constructors in the examples. All constructors are assumed to be of the form

\[
\mathbb{C}(f_1,..., f_n) \{ \text{this. } f_1 = f_1;...;\text{this. } f_n = f_n; \}
\]

where \(f_1,..., f_n\) are all the fields of the class \(C\). We consider the implementation of a class \texttt{IAccount} providing the basic functionality to update the balance of an account with the interface:

\begin{verbatim}
interface IAccount {
    void update();
}
\end{verbatim}

In a language with traits and records, the fields and the methods of the class can be defined independently from the class itself, as illustrated by the code at the top of Listing 1. The class \texttt{IAccount} is composed as shown at the bottom of Listing 1.

A class \texttt{SyncAccount} implementing a variant of the basic bank account that guarantees synchronized access can be developed by introducing a record \texttt{RSync} that provides a field for a lock and a trait \texttt{TSync} that provides a method that wraps the code for synchronization around a non-synchronized method. Based on these and the record \texttt{RAccount} and trait \texttt{TAccount} for the basic account, a record \texttt{RSyncAccount} and a trait \texttt{TSyncAccount} can be defined providing the fields and methods of the class \texttt{SyncAccount}. The corresponding code is shown in Listing 2.
interface IAccount { void update(int x); }
record RAccount is { int balance; /* provided field */ }
trait TAccount is ...

Figure 1: Feature Model for Bank Account Product Line

The record RSync and the trait TSync are completely independent from the code for the basic account. Because of the trait and record operations to rename methods and fields, they can be re-used for synchronizing any method (provided the signature is the same as in TSync) or several methods on the same lock (as we will see in Section 3). FRTJ extends method re-usability of traits to state re-usability of records, and fosters a programming style relying on small components that are easy to re-use.

Traits/records satisfy the so called flattening principle [24] (see also [20, 19]), that is, the semantics of a method/field introduced in a class by a trait/record is identical to the semantics of the same method/field defined directly within the class. For instance, the semantics of the class SyncAccount in Listing 2 is identical to the semantics of the JAVA class:

class SyncAccount implements IAccount { int balance; Lock lock; void unsyncUpdate(int x) { balance = balance + x; } void update(int x) { lock.lock(); unsyncUpdate(x); lock.unlock(); } }

Listing 2: Artifacts for the SYNC_ACCOUNT product

The mandatory Base feature represents the basic functionality of any bank account allowing to store the current balance and to update it. This functionality can be extended by the optional Sync(hronized) feature guaranteeing synchronized access to the account. The features Retirement and Investment that provide the possibility to store an additional bonus for the account are optional and mutually exclusive. The optional feature With Holder adds a reference to the holder of the account and requires the presence of either the Retirement or the Investment feature.

Products in a SPL are constructed from a common artifact base. In our approach, the artifact base for a SPL consists of records, traits, interfaces and the classes assembled thereof. Products use different classes depending on the features they provide. The record, trait, interface and class that capture the functionality of the account providing the Base feature are given in Listing 1. The product ACCOUNT (providing the mandatory feature Base) is specified by the declaration

Listing 3: Artifacts for the RET_ACCOUNT product

class RetAccount implements IBonusAccount by RRetAccount and TRetAccount

class RetAccount implements IBonusAccount { }

class RetAccount implements IBonusAccount { } // 1st product

A product providing several features can be realized by composing and/or modifying records, traits and interfaces contained in the artifact base. Listing 2 contains the records, traits and class required to implement the Sync feature. The product SYNC_ACCOUNT (providing the features Base and Sync) is specified by the declaration

product SYNC_ACCOUNT uses SyncAccount // 2nd product

3. IMPLEMENTING SPL

As a running example to demonstrate how product line variability is implemented in our trait-based approach, we use the SPL of bank accounts considered in [13]. The products of a SPL are defined by their features. A feature is a designated characteristic of a product and represents a unit of product functionality. Figure 1 shows the feature model of the bank account SPL determining the different products by possible combinations of features. The mandatory Base feature represents the basic functionality of any bank account allowing to store the current balance and to update it. This functionality can be extended by the optional Sync(hronized) feature guaranteeing synchronized access to the account. The features Retirement and Investment that provide the possibility to store an additional bonus for the account are optional and mutually exclusive. The optional feature With Holder adds a reference to...

```java
interface IBonusAccount extends IAccount { void addBonus(int b); }
record RBonus is RAccount[balance renameTo 401kBalance] TBonus is TAccount[balance renameTo 401kBalance, update renameTo addBonus]

interface IInvestAccount extends IAccount { }
record RInvestAccount is RAccount[balance renameTo Retirement] TInvestAccount is TAccount[balance renameTo Retirement]

class RetAccount implements IBonusAccount by RRetAccount and TRetAccount

class RetAccount implements IBonusAccount { } // 1st product

A product providing several features can be realized by composing and/or modifying records, traits and interfaces contained in the artifact base. Listing 2 contains the records, traits and class required to implement the Sync feature. The product SYNC_ACCOUNT (providing the features Base and Sync) is specified by the declaration

product SYNC_ACCOUNT uses SyncAccount // 2nd product
```

3. IMPLEMENTING SPL

As a running example to demonstrate how product line variability is implemented in our trait-based approach, we use the SPL of bank accounts considered in [13]. The products of a SPL are defined by their features. A feature is a designated characteristic of a product and represents a unit of product functionality. Figure 1 shows the feature model of the bank account SPL determining the different products by possible combinations of features. The mandatory Base feature represents the basic functionality of any bank account allowing to store the current balance and to update it. This functionality can be extended by the optional Sync(hronized) feature guaranteeing synchronized access to the account. The features Retirement and Investment that provide the possibility to store an additional bonus for the account are optional and mutually exclusive. The optional feature With Holder adds a reference to the holder of the account and requires the presence of either the Retirement or the Investment feature.

Products in a SPL are constructed from a common artifact base. In our approach, the artifact base for a SPL consists of records, traits, interfaces and the classes assembled thereof. Products use different classes depending on the features they provide. The record, trait, interface and class that capture the functionality of the account providing the Base feature are given in Listing 1. The product ACCOUNT (providing the mandatory feature Base) is specified by the declaration

```java
interface IAccount { void update(int x); }
record RAccount is { int balance; /* provided field */ }
trait TAccount is ...
```
trait TInv is TBonus + {
int 401balance; /* required field */
void originalUpdate(int x); /* required method */
void update(int x) { x = x/2; originalUpdate(x); 401balance += x; }
/* provided methods */
}

interface IInvAccount extends IBonusAccount {
record RInvAccount is RBonus + RAccount
trait TInvAccount is TInv + TAccount[update renameTo originalUpdate]
}

class InvAccount implements IInvAccount by RInvAccount and TInvAccount

Listing 4: Artifacts for the INV_ACCOUNT product

interface IClient { void payday(int x, int bonus); }
record RClient is { IBonusAccount a; /*provided fields*/
trait TCClient is {
IBonusAccount a; /* required field */
void payday(int x, int bonus) { a.addBonus(bonus); a.update(x); }
/* provided methods */
}

class Client implements IClient by RClient and TCClient

Listing 5: Artifacts for the _ACCOUNT_WH products

product RET_ACCOUNT uses RetAccount // 3rd product

Listing 4 contains the code artifacts to implement the Investment feature. The InvAccount class implements a variant of the basic bank account which has a 401balance field in addition to the usual balance of the account. When the balance is updated by the update method, the input is split between the basic balance field and the 401balance field. This is realized in the trait TInv. The addBonus method increments the 401balance field directly. The interface IInvAccount, the record RInvAccount and the trait TInvAccount provide the public methods, the fields and the methods of the class InvAccount. The record RInvAccount is composed from the records RBonus and RAccount. The trait TInvAccount is built by composing the trait TInv and the trait TAccount where the update method is renamed to originalUpdate to work with the TInv trait. The product INV_ACCOUNT (providing the features Base and Investment) is specified by the declaration

product INV_ACCOUNT uses InvAccount // 4th product

The With Holder feature is implemented by adding a class Client, representing the owner of an account in a field a of type IBonusAccount. The owner can access his account via the methods update and addBonus of the IBonusAccount interface. The payday method in the TCClient trait increments both the balance and 401balance fields by the input amount. The corresponding artifacts are given in Listing 5. This feature requires the presence of a feature that implements the IBonusAccount interface, i.e., either Retirement or Investment. The corresponding products INV_ACCOUNT_WH and RET_ACCOUNT_WH are specified by the declarations

product INV_ACCOUNT_WH uses InvAccount, Client // 5th product
product RET_ACCOUNT_WH uses RetAccount, Client // 6th product

The product SYNC_ACCOUNT (providing the features Base, Sync and Retirement) implements an account where all public methods are synchronized (cf. Listing 6). First, we introduce the trait TSync2 that synchronizes two methods on the same lock. In

trait TSync2 is TSync + TSync[m renameTo m1, sync_m renameTo synch_m1]

Listing 6: Artifacts of the SYNCRETACCOUNT product

product SYNCRETACCOUNT uses SyncRetAccount // 7th product

The product SYNCRETACCOUNT (providing the features Base, Sync and Investment) is implemented in a similar way by the code artifacts in Listing 7. The product is specified by the declaration

product SYNCRETACCOUNT uses SyncRetAccount // 8th product

The last two products of the SPL, obtained by adding the Sync feature to the 5th and 6th product, respectively, are specified by the declarations

// 9th and 10th products
product SYNCRETACCOUNT_WH uses SyncRetAccount, Client
product SYNCRETACCOUNT_WH uses SyncRetAccount, Client

This example shows that the proposed approach can be used to flexibly model product line variability without limitations by a class hierarchy. The composition operators on records and traits support the fine-grained reuse of artifacts, e.g., to express different features accessing the same fields, features removing fields that are no longer required, or different features redefining the same methods.
4. THE FRTJ CALCULUS

In this section, we outline the FRTJ calculus, a minimal core calculus (in the spirit of FJ [15]) for interfaces, records, traits and classes.

As pointed out in [10], using trait names as types limits the reuse potential of traits, because method exclusion and renaming operations would break the type system. Moreover, if class names are not used as types, interface and record declarations are independent from classes, and the dependencies of trait declarations on classes are restricted to object creation. Thus, in the FRTJ calculus, trait, record and class names are not types. The only user-defined types are interface names. In this way, the reuse potential of traits and records is increased to appropriately capture product line variability.

Syntax. The syntax of FRTJ is given in Figure 2. We also consider a calculus, FFRTJ (FLAT FRTJ), obtained by removing the portions of the syntax highlighted in gray. We use the overbar notation according to [15]. For instance, the pair “I\overbar{x}” stands for “I_1 x_1, ..., I_n x_n”, and “I\overbar{f}” stands for “I_1 f_1; ...; I_n f_n”. The empty sequence is denoted by “∅”.

In FRTJ, there are no constructor declarations. Like in FJ, the syntax of the constructor of a class is fixed with respect the field order, types and names: in every class C, we assume the implicit constructor C[|{this.f = z}|] where I\overbar{f} are the fields of C. Note that FFRTJ is indeed a subset of JAVA. The FFRTJ class implements I\overbar{f} by |{I.f | (I.g, x, y)}| (where the fields I\overbar{g} are a subset of the fields I\overbar{f}) can be understood as the JAVA class class implements I\overbar{f} |{I.f | (I.g, x, y)}|.

A class table CT is a map from class names to class declarations. Similarly, an interface table IT, a record table RT and a trait table TT map interface, record and trait declarations, respectively. A FRTJ program is a 5-tuple |{IT, RT, TT, CT, e}|, where e is the expression to be executed. For the type system and the operational semantics, we assume fixed, global tables IT, RT, TT, and CT. We also assume that these tables are well-formed, i.e., they contain an entry for each interface/record/trait/class mentioned in the program, and that the interface subtyping and record/trait reuse graphs are acyclic.

Types, Subinterfacing and Subtyping. Nominal types, ranged over by \eta, are either class names or interface names. The subinterfacing relation is the reflexive and transitive closure of the intermediate subinterfacing relation declared by the extends clauses in the interface table IT. It is formalized by the judgment |{I_1 \sqsubseteq I_2}| to be read: “I_1 is a subinterface of I_2”. The subtyping relation for nominal types is the reflexive and transitive closure of the relation obtained by extending subinterfacing with the interface implementation relation declared by the implements clauses in the class table CT. It is formalized by the judgment |{\eta_1 \sqsubseteq \eta_2}| to be read: “\eta_1 is a subtype of \eta_2”.

Well-Typed FRTJ programs. We write |{IT, RT, TT, CT, e}| : \eta to be read: “the program |{IT, RT, TT, CT, e}| is well-typed with type \eta”. To mean that the interfaces in IT, the records in RT, the traits in TT and the classes in CT are well-typed, and the expression e is well-typed with type \eta.

Reduction. Following FJ [15], the semantics of FRTJ is given by means of a reduction relation of the form e → e’, to be read “expression e reduces to expression e’ in one step”. We write e →* to denote the reflexive and transitive of →. Values are defined by the following syntax: |{v := newC(V)}|.

Properties. Type soundness can be proved by using the standard technique of subject reduction and progress theorems.

Theorem 4.1 (FRTJ Type Soundness).

\hl{If e : \eta and e \rightarrow* e’ with e’ a normal form, then e’ is: either a value v of type C and C <: \eta; or an expression containing (I) newC(\overbar{e}) |\{I.e\}| where C <: I.}

A formulation of traits in a JAVA-like setting has support to type-checking of traits in isolation from the classes or traits that use them, so that it is possible to type-check a method defined in a trait only once (instead of having to type-check it in every class or trait using that trait). The FRTJ type system supports the above property through a suitable combination of nominal and structural typing. Within a basic trait expression, the use of method parameters are type-checked according to the nominal notion of typing defined by the interface hierarchy, while the uses of the this pseudo-variable are type-checked according to a structural notion of typing that takes into account the fields and methods required by the trait and the methods provided by the trait. The following theorem can be established by inspecting the FRTJ typing rules.

Theorem 4.2 (FRTJ Type-Checking).

A program can be type-checked by type-checking only once its interfaces, record, traits, and classes.

5. IMPLEMENTING SPL IN FRTJ

In this section, the methodology to implement SPL in FRTJ is presented. A SPL consists of a set of products that are constructed from common artifacts in the SPL artifact base. A product is specified by the set of classes it uses. A product specification PS is a declaration

\hl{product P uses C}

where P is the name of the product and C is a sequence of class names. The set of products contained in the SPL is captured in its product table. A product table PT is a map from product names P to product specifications PS. A SPL L is a 5-tuple |{IT, RT, TT, CT, PT}| where PT is the product table of the SPL and |{IT, RT, TT, CT}| represents the artifact base. The artifact base contains the interfaces, records, traits and classes used to specify products. We assume that the tables IT, RT, TT and CT are well-formed, i.e., the tables IT/RT/TT/CT contain an entry for each interface/record/trait/class used in the SPL and the interface subtyping and record/trait reuse graphs are acyclic.

The code of the product P is the 4-tuple |{LT_P, RT_P, TT_P, CT_P}|, where LT_P, RT_P, TT_P and CT_P are the subtables of IT, RT, TT and CT containing exactly the entries for the interfaces, records, traits
and classes reachable from the classes contained the specification of P. We assume that each product specification product P uses $\mathcal{C}$ of PT is well-formed, that is: $\mathcal{C}$ contains exactly classes in $\mathcal{C}_P$.

Figure 3 depicts the relations between a SPL artifact base, the products and the SPL. On the lowest level, the traits $T_1$, $T_2$, $T_3$, the records $R_1$, $R_2$, $R_3$ and interfaces $I_1$, $I_2$, $I_3$ are used to build classes $C_1$, $C_2$, $C_3$ constituting the SPL artifact base. The classes are then used to build the products $P_1$, $P_2$ and $P_3$. The products are contained in two different SPL $L_1$ and $L_2$.

**Example 5.1.** Consider the bank account SPL introduced in Section 3. The SPL $\text{BankLine}$, described by the feature model in Figure 1 where the feature Sync is removed, is formalized as a 5-tuple $(\mathcal{T}, \mathcal{R}, \mathcal{T}^t, \mathcal{C}, \mathcal{P})$ containing the entries for all interfaces/records/trait/classes given in Listings 1, 3, 4 and 5 and the five product specifications

$\text{product ACCOUNT uses Account}$

$\text{product INV_ACCOUNT uses InvAccount}$

$\text{product RET_ACCOUNT uses RetAccount}$

$\text{product INV_ACCOUNT_WH uses InvAccount, Client}$

$\text{product RET_ACCOUNT_WH uses RetAccount, Client}$

Using the type system of FRTJ introduced in Section 4, we define type safety of a product line. We write $\vdash L$ OK, to be read: “the SPL L is well-typed”, i.e., the code of every product P in L is well-typed. In most approaches (with the exception of e.g., [30, 131]), the only way to verify that all the products of a SPL are type-safe is to generate and type-check all products individually. As a consequence of Theorem 4.2, in FRTJ, the type safety of a SPL can be verified without type-checking all its products individually, since it is enough to type-check only once each artifact in the artifact base.

**Theorem 5.2.** (FRTJ SPL TYPE-CHECKING). A SPL L can be type-checked by type-checking only once its interfaces, records, traits and classes.

The formalization of SPL and the FRTJ calculus easily allow extending SPL with further products. These products can also use traits, records, interfaces and classes that are not contained in the original artifact base. The SPL $L' = (\mathcal{T}', \mathcal{R}', \mathcal{T}^t', \mathcal{C}', \mathcal{P}')$ is an extension of the SPL $L = (\mathcal{T}, \mathcal{R}, \mathcal{T}^t, \mathcal{C}, \mathcal{P})$ if L has been obtained from L by adding interfaces, records, traits and classes and products (that is if $\mathcal{I} \subseteq \mathcal{I}'$, $\mathcal{R} \subseteq \mathcal{R}'$, $\mathcal{T}^t \subseteq \mathcal{T}^t'$, $\mathcal{C} \subseteq \mathcal{C}'$ and $\mathcal{P} \subseteq \mathcal{P}'$ hold). In Figure 3, the SPL $L_2$ is an extension of SPL $L_1$.

**Example 5.3.** The SPL $\text{BankLine}'$, described by the feature model in Figure 1 with the Sync feature, extends the SPL $\text{BankLine}$ of Example 5.1. It can be formalized by adding to the SPL BankLine the code in Listings 2, 6 and 7 and the five product specifications

$\text{product SYNC_ACCOUNT uses SyncAccount}$

$\text{product SYNC_INV_ACCOUNT uses SyncInvAccount}$

$\text{product SYNC_RET_ACCOUNT uses SyncRetAccount}$

$\text{product SYNC_INV_ACCOUNT_WH uses SyncInvAccount, Client}$

$\text{product SYNC_RET_ACCOUNT_WH uses SyncRetAccount, Client}$

A further consequence of Theorem 4.2 is that for ensuring the type safety of the extended SPL $\text{BankLine}'$ only the newly added records, traits, interfaces, classes, and products must be type-checked.

**Theorem 5.4.** (FRTJ SPL EXTENSION TYPE-CHECKING). Let the SPL $L' = (\mathcal{T}', \mathcal{R}', \mathcal{T}^t', \mathcal{C}', \mathcal{P}')$ be an extension of the SPL $L = (\mathcal{T}, \mathcal{R}, \mathcal{T}^t, \mathcal{C}, \mathcal{P})$. If $L'$ has been already type-checked (so that the typings of all its artifacts are available), then the products in $\mathcal{T}^t' - \mathcal{P}$ can be type-checked without type-checking the artifacts of $L'$ and by type-checking only once the interfaces, records, traits, classes in $\mathcal{T}^t' - \mathcal{T}$, $\mathcal{R}' - \mathcal{R}$, $\mathcal{T}^t' - \mathcal{T}$, $\mathcal{C}' - \mathcal{C}$, respectively.

6. RELATED WORK

Trails are well suited for designing libraries and enable clean design and reuse which has been shown using SMALLTALK/SQUEAK (see, e.g., [8, 11]). Recently, [6] pointed out limitations of the trait model caused by the fact that methods provided by a trait can only access state by accessor methods (which become required methods of the trait). To avoid this, traits are made stateful (in a SMALLTALK/SQUEAK-like setting) by adding private fields that can be accessed from the clients possibly under a new name or merged with other variables. In FRTJ traits are stateless. By their required fields, however, it is possible to directly access state within the methods provided by a trait. Moreover, the names of required fields (in traits) and provided fields (in records) are unimportant because of the field rename operation. Since field renaming works synergically with method renaming, exclusion and aliasing, FRTJ has more reuse potential.

The approaches to implementing the variability of SPL in the object-oriented paradigm can be classified into two main directions [18]. First, *annotative approaches*, such as conditional compilation, frames [4] and COLORED FEATHERWEIGHT JAVA (CFJ) [16], mark the source code of the whole SPL with respect to product features and remove marked code depending on the feature configuration. Second, *compositional approaches* (like the calculus FRTJ presented in this paper) assemble products from artifacts in a common artifact base. Compositional implementations of SPL in the object-oriented paradigm use a variety of mechanisms, such as aspects [17], mixins [28], or features modules in the AHEAD framework [5]. In [22], product line variability is implemented in SCALC [25] using traits that are realized by mixin-based inheritance. The compositional approaches closest to FRTJ are FEATHERWEIGHT FEATURE JAVA (FFJ) [3] and LIGHTWEIGHT FEATURE JAVA (LFJ) [13]. Both calculi aim at a formalization of feature-based product composition with static guarantees.

FFJ and LFJ use specific linguistic constructs to implement features according to the feature-oriented paradigm [5]. A feature can introduce new classes and refine existing ones. The ordering in
which features are composed is restricted. A feature that refines a class can be added only after the class to be refined has been introduced. Refinement of classes is unavoidable, since the class hierarchy may have to be changed radically for implementing product variability. While refining a class does not change its name, its definition may change completely (even its superclass can be altered). Therefore, class refinement can break code in client classes built before the refinement. The case of a class that is refined by adding new fields is particularly interesting. Both FFJ and LFJ propose a way to initialize fields that are added to a class by refinement: (1) FFJ requires that all fields are initialized by a single constructor call with the values to be assigned to the fields as arguments (as in FJ). Newly added fields are initialized by ensuring that, whenever a superset constructor in a client class built before the refinement is invoked, the additional fields are initialized to default values. (2) LFJ initializes newly added fields by a default constructor without arguments associated to each class that assigns default values (as in JAVA), and relies on assignment operations to set the fields properly. Both ways have the subtle drawback that, when a class refinement adds new fields, the code in client classes built before the refinement still type-checks, even if (due to a faulty SPL implementation) no code for the proper initialization of the new fields is inserted. In the presented approach, a class name refers to the same definition entity in all the products. If (due to faults in the SPL implementation) the code of a product invokes the constructor of a class not listed in the product specification, the error is automatically detected during type-checking assuming the well-formedness of the product table.

FFJ has a type system to check single product specifications, while LFJ supports the type-checking of a complete SPL. LFJ introduces a constraint-based type system (similar to the one in [2]) that supports the type-checking of feature modules in isolation. The type safety of a SPL can be verified by checking the validity of a generated propositional formula expressing its type safety. The FRTJ type system ensures that a SPL is type-safe by type-checking the artifacts in the artifact base only once. Furthermore, it allows type-checking of an extension of a (type-safe) SPL just by considering only the newly added parts.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a novel approach to implement product line variability by trait and record composition. FRTJ programs may look more verbose than standard class-based programs; however, the degree of re-use provided by records and traits is higher than the re-use potential of standard static class-based hierarchies. The FRTJ type system is able to ensure type-safety of a SPL by type-checking its artifacts only once and to ensure type-safety of an extension of a (type-safe) SPL by checking only the newly added parts. An extended version of this paper is available as [7]. A prototypical implementation of a language based on the FRTJ calculus is currently under development.

Our linguistic constructs which are lower-level than standard OO mechanisms can be used to introduce derived linguistic concepts in order to reduce the amount of code to write. For future work, we plan to investigate the possibility of adding a feature module construct (like the one of LFJ [13]) to FRTJ in order to lift the reuse potential beyond class level. Additionally, we aim at developing a process for building up an artifact base supporting as much code reuse as possible for implementing a particular SPL and evaluate this at larger case examples. The process will include guidelines on how features in a feature model can be represented best by traits, records and interfaces and how the resulting classes should be assembled. An IDE that allows viewing the different code artifacts from the perspective of the feature model of the SPL has to be developed assisting the programmer in managing the created artifacts.

Acknowledgments. We are grateful to Viviana Bono for initial collaboration on the subject of this paper and thank the OOPS 2010 referees and the referees of an earlier version of this paper for insightful comments and pointers to related work.

8. REFERENCES

Appendix L

A Programming Language with Records and Traits

A Programming Language with Records and Traits

Lorenzo Bettini1, Ferruccio Damiani1, Ina Schaefer2⋆⋆, and Fabio Strocco1

1 Dipartimento di Informatica, Università di Torino, C.so Svizzera, 185 - 10149 Torino, Italy
2 Chalmers University of Technology, 412 96 Gothenburg, Sweden

Abstract. Traits are units of fine-grained behavior reuse in the object-oriented paradigm. In this paper, we present the prototypical implementation of SUGARED WELTERWEIGHT RECORD-TRAIT JAVA (SWRTJ), a dialect of JAVA, where object functionality is implemented by traits and by records (a construct that complements traits to model the state part of objects). Records and traits are assembled in classes that are instantiated to build objects. Records and traits can be composed by explicit linguistic operations, allowing code manipulations to achieve fine-grained code reuse. SWRTJ is implemented using XTEXT, an Eclipse framework for the development of programming languages as well as other domain-specific languages. Our implementation comprises an Eclipse-based editor for SWRTJ with typical IDE functionalities, and a stand-alone compiler, which translates SWRTJ programs into standard JAVA programs.

1 Introduction

Traits have been designed to play the role of units for behavior fine-grained reuse in order to counter the problems of class-based inheritance with respect to code reuse. A trait is a set of methods, completely independent from any class hierarchy. The common behavior (i.e., the common methods) of a set of classes can be factored into a trait. Traits were introduced and implemented in the dynamically-typed class-based language SQUEAK/SMALLTALK [37, 19]. Various formulations of traits in a JAVA-like setting can be found in the literature (see, e.g., [38, 30, 12, 36, 13, 28]). Also two recent programming languages, SCALA [31] and FORTRESS [5], incorporate forms of the trait construct.

Records have been proposed in [12] as the counterpart of traits with respect to state to play the role of units for state fine-grained reuse. A record is a set of fields, completely independent from any class hierarchy. The common state (i.e., the common fields) of a set of classes can be factored into a record.

In this paper, we present the prototypical implementation of SUGARED WELTERWEIGHT RECORD-TRAIT JAVA (SWRTJ), a JAVA-like programming language aiming
at interface-based polymorphism and using traits and records as composable units of behavior and state reuse, respectively. SWRTJ is based on the calculus presented in [10] whose complete formalization and proof of type safety is available as a technical report in [9]. In SWRTJ, the declarations of object type, state, behavior and generator are completely separated. Namely, SWRTJ considers:

- **Interfaces**, as pure types, defining only method signatures.
- **Records**, as pure units of state reuse, defining only fields.
- **Traits**, as pure units of behavior reuse, defining only methods.
- **Classes**, as pure generators of instances, implementing interfaces by using traits and records, and defining constructors.

Classes cannot be reused themselves. In particular, there are no class hierarchies. Therefore, as first outcome of the complete separation of the roles played by interfaces/records/traits/classes, problems of fragility in a class hierarchy (that arise with class-based and mixin-based inheritance, see [19] for a detailed discussion) are avoided a priori. Multiple inheritance with respect to methods is obtained via the trait construct, and multiple inheritance with respect to fields is obtained via the record construct. Thus, another outcome of the complete role separation is that multiple inheritance is subsumed by ensuring that, in the spirit of the original trait proposal in SQUEAK/SMALLTALK, the composite unit has complete control over the composition and must resolve conflicts explicitly.

SWRTJ is equipped with a JAVA-like nominal type system (where the only types are interface names) supporting the typechecking of traits in isolation from the traits and classes that use them. In SCALA [31] and FORTRESS [5] each trait, like each class, also defines a type. However, as a matter of fact, the role of unit of reuse and the role of type are competing.\(^5\) For instance, in order to define the subtyping relation on traits in such a way that a trait (or a class) is always a subtype of the component traits, both SCALA and FORTRESS rule out method exclusion (the operation that forms a new trait by removing a method from an existing trait), limiting the reuse potential of traits. In SWRTJ, traits (as well as records and classes) do not define types, therefore method exclusion can be supported. This will be illustrated in Examples 2 and 3 of Section 2.

SWRTJ is implemented using XTEXT [4], a framework for development of programming languages as well as other domain-specific languages (DSLs). In XTEXT, the syntax of SWRTJ is defined using an EBNF grammar. The XTEXT generator creates a parser, an AST-meta model (implemented in EMF [40]) as well as a full-featured Eclipse-based editor for SWRTJ. Further, a SWRTJ program will be translated into a standard JAVA program. Although, the syntax of SWRTJ is JAVA-like, the record and trait concepts themselves are not necessarily bound to JAVA, and we could translate a SWRTJ program also into other object-oriented languages, such as C++. The compiler can also be used as a command-line program, outside of the Eclipse IDE. The implementation described in the paper is based on [42]. The implementation of SWRTJ is available as an open source project at \(\text{http://swrtj.sourceforge.net}\).

\(^5\) The distinction between the role of type and the role of unit of reuse, described in terms of type and class, dates back at least to Snyder [39] (see also Cook et al. [18]).
Organization of the Paper. In Section 2 we illustrate record and traits as supported in SWRTJ. In Section 3 and Section 4 we describe SWRTJ and its implementation. In Section 5 we discuss the features of our language and its impact on programming, and report on our experience with XTEXT. Related work is discussed in Section 6. We conclude by outlining some future work.

2 Traits and Records in SWRTJ

In SWRTJ, a trait consists of methods, of required methods which parametrize the behavior, and of required fields that can be directly accessed in the body of the methods, along the lines of \[10, 13\].\(^6\) Traits are building blocks to compose classes and other, more complex, traits. A suite of trait composition operations allows the programmer to build classes and composite traits. A distinguished characteristic of traits is that the composite unit (class or trait) has complete control over conflicts that may arise during composition and must solve them explicitly. Traits do not specify any state. Therefore a class composed by using traits has to provide the required fields. The trait composition operations considered in SWRTJ are as follows:

- A basic trait defines a set of methods and declares the required fields and the required methods.
- The symmetric sum operation, \(+\), merges two traits to form a new trait. It requires that the summed traits must be disjoint (that is, they must not provide identically named methods)\(^7\) and have compatible requirements (two method/field requirements are compatible if they are identical).
- The operation exclude forms a new trait by removing a method from an existing trait.
- The operation aliased forms a new trait by giving a new name to an existing method. When a recursive method is aliased, its recursive invocation refers to the original method (as in \[19\]).\(^8\)
- The operation renameTo creates a new trait by renaming all the occurrences of a required field name or of a required/provided method name from an existing trait.\(^9\)

\(^6\) Field requirements are not present in most formulation of traits in the SMALLTALK/SQUEAK-like and JAVA-like settings. They were introduced in the formulation of traits in a structurally typed setting by Fisher and Reppy [20].

\(^7\) The disjoint requirement for composed traits was proposed by Snyder [39] for multiple class-based inheritance (see also the JIGSAW framework [14]). According to other proposals, two methods with the same name do not conflict if they are syntactically equal [19, 30] or if they originate from the same subtrait [28].

\(^8\) The variant of aliasing proposed in [28] (where, when a recursive method is aliased, its recursive invocation refers to the new method) can be straightforwardly encoded by exclusion, renaming and symmetric sum.

\(^9\) Method renaming and required field renaming are not present in most formulation of traits in the SMALLTALK/SQUEAK-like and JAVA-like settings. Method renaming has been introduced in the formulation of traits in a structurally typed setting by Reppy and Turon [35]. Renaming operations were already present in the JIGSAW framework [14] in connection with module composition and in the EIFFEL language [29] in connection with multiple class-based inheritance.
Therefore, in SWRTJ, the actual names of the methods defined in a trait (and also the names of the required methods and fields) are irrelevant, since they can be changed by the `renameTo` operation.

Records are building blocks to compose classes and other, more complex, records by means of operations analogous to the ones described above for traits. The record composition operations considered in SWRTJ are as follows:

- A basic record defines a set of fields.
- The `symmetric sum` operation, `+`, merges two disjoint records to form a new record.
- The operation `exclude` forms a new record by removing a field from a record.
- The operation `renameTo` creates a new record by renaming a field in a record.

In the following, we illustrate the record and traits constructs of SWRTJ through some examples. The first example considers the implementation of a bank account which may have synchronized access.

**Example 1.** Consider the implementation of a class `Account` providing the basic functionality to update the balance of an account with the interface:

```java
interface IAccount { void update(int x); }
record RAccount is { int balance; /∗ provided field ∗/ }
trait TAccount is {
   int balance; /∗ required field ∗/
   void update(int x) { balance = balance + x; } /∗ provided method ∗/
}
class Account implements IAccount by RAccount and TAccount {
   Account(int b) { this.balance = b; }
}
```

Listing 1: A standard `Account` class (without synchronization).

A class `SyncAccount` implementing a variant of the bank account that guarantees synchronized access can be developed by introducing a record `RSync` that provides a field for a lock and a trait `TSync` that provides a method that wraps the code for synchronization around a non-synchronized method. Based on these and the record `RAccount` and trait `TAccount` for the basic account, a record `RSyncAccount` and a trait `TSyncAccount` can be defined providing the fields and methods of the class `SyncAccount`. The interface `ILock` and the corresponding class `CLock` are part of the library of the language, see Section 3. The corresponding code is shown in Listing 2.

The record `RSync` and the trait `TSync` are completely independent from the code of the basic account. Because of the trait and record operations to rename methods and fields, they can be reused for synchronizing any method or several methods on the same
record RSync is { ILock lock; }
trait TSync is {
   ILock lock; /* required field */
   void m(int x); /* required method */
   void sync_m(int x) { lock.lock(); m(x); lock.unlock(); }
}
record RSyncAccount is RSync + RAccount
trait TSyncAccount is TAccount[update renameTo unsyncUpdate] + TSync[m renameTo unsyncUpdate, sync m renameTo update]

class SyncAccount implements IAccount by RSyncAccount and TSyncAccount {
   SyncAccount(int b) { this.balance = b; lock = new CLock(); }
}

Listing 2: An class with synchronization.

lock (provided the signature of each method is the same as in TSync). SWRTJ extends method reusability of traits to state reusability of records and fosters a programming style relying on small components that are easy to reuse.

For instance, if the class Account is also supposed to have other methods such as, e.g., `increase(int i)` and `decrease(int i)` that we want to synchronize, using the same lock for `update`, we could write the SyncAccount class as follows:

class SyncAccount implements IAccount by RSyncAccount and TAccount[update renameTo unsyncUpdate, increase renameTo unsyncIncrease, decrease renameTo unsyncDecrease] + TSync[m renameTo unsyncUpdate, sync m renameTo increase] + TSync[m renameTo unsyncDecrease, sync m renameTo decrease]

Traits/records satisfy the so called flattening principle [30] (see also [26, 25]), that is, the semantics of a method/field introduced in a class by a trait/record is identical to the semantics of the same method/field defined directly within the class. For instance, the semantics of the class SyncAccount in Listing 2 is identical to the semantics of the JAVA class:

class SyncAccount implements IAccount {
   { int balance;
      ILock lock;
      void unsyncUpdate(int x) { balance = balance + x; }
      void update(int x) { lock.lock(); unsyncUpdate(x); lock.unlock(); }
}

Example 2. This example illustrates the use of the method exclusion operation for traits. Consider the task of developing a class Stack that implements the interface:

interface IStack {
   boolean isEmpty();
   void push(Object o); Object pop();
}

In JAVA, the corresponding implementation would be a class as follows:

\footnote{Note that our locking implementation guarantees that if the same thread calls lock() on the same lock instance twice it will not deadlock.}
class Stack implements IStack {
    List l;
    Stack() { l = new LinkedList(); }
    boolean isEmpty() { return (l.size() == 0); }
    void push(Object o) { l.addFirst(); }
    Object pop() { Object o = l.getFirst(); l.removeFirst(); return o; }
}

Suppose that afterwards a class Lifo implementing the following interface should be developed:

interface ILifo {
    boolean isNotEmpty();
    void push(Object o);
    void pop();
    Object top();
}

In JAVAv there is no straightforward way to reuse the code in class Stack, as it would not be possible to override the pop method changing the return type from Object to void. If the class Stack was originally developed in a language with records and traits, it would have been written as follows (the interface IList, not shown here, has all the standard methods of lists):

record RElements is { IList l; }

trait TStack is {
    IList l; /* required field */
    boolean isEmpty() { return (l.size() == 0); }
    void push(Object o) { l.addFirst(o); }
    Object pop() { Object o = l.getFirst(); l.removeFirst(); return o; }
}

class Stack implements IStack by RElements and TStack {
    Stack() { l = new LinkedList(); }
}

Based on this, a programmer would be able to write the class Lifo by defining a trait TLifo that reuses the trait TStack by exploiting the method exclusion operation:

trait TLifo is (TStack exclude pop) + {
    IList l; /* required field */
    boolean isNotEmpty() { return l.isEmpty(); }
    void pop() { l.removeFirst(); }
    Object top() { return l.getFirst(); }
}

class Lifo implements ILifo by RElements and TLifo {
    Lifo() { l = new LinkedList(); }
}

The body of trait TLifo satisfies the requirements of trait sum operation described at the beginning of the section: the method pop is excluded thus it does not generate a conflict and the field requirement is identical to the one of TStack.

This is a paradigmatic example of trait composition that does not preserve structural subtyping. If traits were types and composed traits were subtypes of the component traits (as in SCALA and FORTRESS), then the declaration of the trait TLifo would not typecheck, since:

- the trait TLifo should provide all the methods provided by TStack, and
- a method with signature Object pop() and a method with signature void pop() could not belong to the same trait/class.
Example 3. This example illustrates the use of the method renaming operation. A class Queue that implements the interface

```java
interface IQueue { boolean isNotEmpty(); void enqueue(Object o); Object dequeue(); }
```

can be written by defining a trait TQueue that reuses the record RElements and the trait TStack (introduced in Example 2) by renaming the method `pop`, excluding the method `push` and providing the methods `enqueue` and `isNotEmpty`:

```java
trait TQueue is (TStack rename pop to dequeue) exclude push
  + { IList l; /* required field */
    boolean isEmpty(); /* required method */
    boolean isNotEmpty() { return !(isEmpty()); }
    void enqueue(Object o) { l.addLast(o); } }
```

class Queue implements IQueue by RElements and TQueue

```java
{ Queue() { l = new CLinkedList(); } }
```

Note that, if traits were types and composed traits were subtypes of the component traits, then the declaration of the trait TQueue would not typecheck.

### 3 The SWRTJ Programming Language

In this section we describe the SWRTJ programming language, and we sketch the main features of its type system.
Syntax. The syntax of SWRTJ is illustrated in Table 1 (without primitive types and file imports that are not relevant for this description). An interface can extend one or more interfaces. A class must implement one or more interfaces. In the syntax, the overline bar indicates a (possibly empty) list, as in Featherweight Java [24]. For instance, $I = I_1, \ldots, I_n$, $n \geq 0$. The parameter declarations are denoted by $I \times x$ to indicate $I_1 x_1, \ldots, I_n x_n$. The same notation can be applied to the other lists. In the syntax, $m$ denotes a method name, $f$ a field name and $x$ a local variable or parameter. Note that the grammar contains only the field access $\text{this}.f$ (even for field assignment) because a field can be used only within a method (trait method or constructor) due to its private visibility. Variables and methods can only have an interface type. Trait expressions in trait composition operations do not have to be names of already defined traits: they can also be “anonymous” traits \{ F; S; M \}. The same holds for record expressions. This is in particular useful for traits, when a trait expression has to define some “glue” code for other traits used in a trait composition operation. In this way, there is no need to define a trait with a name only for that. An example of such usage is in the stream scenario, shown later, and in the examples of Section 2.

The entry point of an SWRTJ program is specified by

\[
\text{program } <\text{name}> \{ \<\text{expression}> \}
\]

In the scope of the program, the implicit object $\text{args}$ is the list of the program’s command line arguments.

In SWRTJ traits, records and classes are not types in order to subdivide the roles of the different constructs. Moreover, as illustrated in Example 2, traits and records support exclusion operations that violate subtyping. Therefore, a type can be only an interface or a primitive type (e.g., int, boolean etc.). For simplicity, method overloading is not supported. Constructor overloading allows only the definition of constructors with different numbers of parameters within a class.

Visibility modifiers are ruled out since they are not necessary. The traditional visibility modifiers are implied by the constructs used in SWRTJ as follows:

- **private**: every instance variable is private. Since class is not a type, fields can be accessed only from the \text{this} parameter. Every provided method is private if it does not appear in the interfaces implemented by a class. Interfaces are the only way to make a method accessible from the outside.
- **protected**: is not necessary since inheritance is ruled out.
- **public**: every method declared in the interface implemented by a class is public. Fields cannot be public in order to support information hiding.

The system library of SWRTJ provides interfaces such as $\text{IObject}$ (implicitly extended by every interface), $\text{IInteger}$, $\text{IString}$ etc. Every interface has a corresponding class implementing it, e.g., $\text{CObject}$, $\text{CInteger}$, etc. For synchronization mechanisms, we provide the interface $\text{ILock}$ (and the corresponding class $\text{CLock}$) with methods $\text{lock}$ and $\text{unlock}$ avoiding to introduce specific concurrency features in the language. In order to deal with collections, $\text{IList}$ is an interface with typical list operations. $\text{CArrayList}$ is the corresponding implementation class. The interfaces
IPrintStream and IScanner, and their implicit “global” instances out and in can be used to perform basic operations such as writing to standard output and reading from standard input, respectively. Standard basic types, such as int, boolean, etc., are also provided. Methods can be declared as void with the usual meaning.

In Listing 3, we show the interfaces, records and traits for implementing a stream library (along the lines of the examples of [19]), together with the corresponding classes and the main program. Note that the TReadWriteStream reuses the previously defined traits and resolves conflicts by method renaming. Furthermore, it contains “glue” code (in an anonymous trait) for the initialization of the fields of the composed traits.

Type System. In this paper, we do not present the meta-theory of SWRTJ. We refer to [9] for the formalization of the FRTJ calculus on which SWRTJ is based. In this section, we provide an introduction to the typing of SWRTJ intended for the programmer. For simplicity, we will not consider void.

The SWRTJ type system supports type-checking records and traits in isolation from classes that use them. Therefore, each trait and record definition has to be typechecked only once, i.e., every class can use a trait or a record without type-checking it again. This is more efficient and convenient in practice, e.g., if the trait/record source is not available. The basic idea of the type system is to collect constraints when checking traits and records, and to establish that these constraints hold when a class is declared, in order to ensure that the pseudo-variable this in all the methods of used traits can be used safely.

In the SWRTJ type system, a nominal type is either a class name or an interface name. Note that in SWRTJ classes are not source types. Class names cannot be used as types in the code written by the programmer, but are used only internally by the compiler. Indeed, the type of this is an inferred structural type, useful to check constructor call compatibility with respect to interfaces, for instance, in a cast on an expression such as (I)new C(). The syntax for expression types is as follows θ ::= I | C | ⟨F, σ⟩ where I is an interface name, C is a class name and the ⟨F, σ⟩ is the structural type for this, which contains all the fields (F) and the signatures (σ) of the methods that can be selected on this in the context where the expression occurs. If the expression is a constructor call, such as new C(), its type is the class C; if the expression is this, its type is ⟨F, σ⟩; otherwise, the type of the method is an interface, for instance, if the expression is a method call, such as x.m(), the type is the interface that I declares as return type of the method m.

SWRTJ type system checks all requirements by inferring constraints. A constraint is a triple ⟨F, S, I⟩ consisting of required fields, method signatures and interfaces collected while analyzing an expression. The constraints contain the types of every field and method selected on this and the name of every interface used, either as type in the methods parameters to which this is passed as argument or as return type in the methods in which this is returned.

The subinterfacing relation is the reflexive and transitive closure of the immediate subinterfacing relation declared by the extends clauses in the interface definitions. The subtyping relation for nominal types is the reflexive and transitive closure of the relation obtained by extending subinterfacing with the interface implementation relation declared by the implements clauses in the class definitions.
interface IStream { void close(); }
interface IWriteStream extends IStream { void write(IString data); }
interface IReadStream extends IStream { IString read(); }
interface IReadWriteStream extends IWriteStream, IReadStream {}

record ReadResource is {IScanner resource;}
record WriteResource is {IPrintStream resource;}
record ReadWriteResource is
    ReadResource[resource renameTo readResource] +
    WriteResource[resource renameTo writeResource]

trait TReadStream is {
    IScanner resource;
    void init() { this.resource = in; }
    IString read() { return this.resource.nextLine(); }
    void close() {} }

trait TWriteStream is {
    IPrintStream resource;
    void init() { this.resource = out; }
    void write(IString data) { this.resource.println(data); }
    void close() {} }

trait TReadWriteStream is
    TReadStream[init renameTo readInit, resource renameFieldTo readResource, close renameTo readClose] +
    TWriteStream[init renameTo writeInit, resource renameFieldTo writeResource, close renameTo writeClose] +
    {  // required methods
        void readInit(); void writeInit(); void readClose(); void writeClose();
    // provided methods
        void init() { this.readInit(); this.writeInit(); }
        void close() { this.readClose(); this.writeClose(); }
    }

class CReadStream implements IReadStream by ReadResource and TReadStream
    { CReadStream() { this.init(); } }

class CWriteStream implements IWriteStream by WriteResource and TWriteStream
    { CWriteStream() { this.init(); } }

class CReadWriteStream implements IReadWriteStream by ReadWriteResource
    and TReadWriteStream
    { CReadWriteStream() { this.init(); } }

program StreamExample
    { IReadWriteStream stream = new CReadWriteStream();
        stream.write("Please insert a string");
        stream.write("You wrote: ".concat(stream.read()));
        stream.close();
    }

Listing 3: Stream implementation.
4 Implementing SWRTJ

In this section, we describe the implementation of SWRTJ using the XTEXT [4] framework for Eclipse. Although Eclipse itself provides a framework for implementing an IDE for programming languages, this procedure is still quite laborious and requires a lot of manual programming. XTEXT eases this task by providing a high-level framework that generates most of the typical and recurrent artifacts necessary for a fully-fledged IDE on top of Eclipse.

The first task in XTEXT is to write the grammar of the language using an EBNF-like syntax. Starting from this grammar, XTEXT generates an ANTLR parser [32]. The generation of the abstract syntax tree is handled by XTEXT as well. In particular, during parsing, the AST is generated in the shape of an EMF model (Eclipse Modeling Framework [40]). Thus, the manipulation of the AST can use all mechanisms provided by EMF itself. There is a direct correspondence between the names used in the rules of the grammar and the generated EMF model JAVA classes. For instance, if we have the following grammar snippet

```
Message : MethodInvocation | FieldAccess;
MethodInvocation : method=ID
                  '(': (argumentList+=Expression (',' argumentList+=Expression)*)? ')'?
```

the XTEXT framework generates the following EMF model JAVA interface (and the corresponding implementation class):

```
public interface MethodInvocation extends Message
{
    MethodName getMethod();
    EList<Expression> getArgumentList();
}
```

Besides, XTEXT generates many other classes for an editor for the language to be defined. The editor contains syntax highlighting, background parsing with error markers, outline view, code completion. Further, XTEXT provides the infrastructure for code generation. Most of the code generated by XTEXT can already be used off the shelf, but other parts can or have to be adapted by customizing some classes used in the framework. The usage of the customized classes is dealt with by relying on Google-Guice [1], so that the programmer does not have to maintain customized abstract factories [21]. In this way it is very easy to insert custom implementations into the framework ("injected" in Google-Guice terminology), with the guarantee that the custom classes will be used consistently throughout the code of the framework.

The validation mechanisms for the language must be be provided by the language developer. In our case, this is the SWRTJ type system. Implementing the validation mechanism in a compiler usually requires to write specific visitors for the abstract syntax tree. EMF already simplifies this task by providing a switch-like functionality to efficiently execute methods with dynamic dispatch according to the actual type of an

---

11 The reader who is familiar with ANTLR will note that the syntax of XTEXT grammars is very similar to ANTLR’s syntax.
AST node. Thus, there is no need to add code to implement a visitor structure [21].

XTEXT leverages this mechanism by only requiring methods with a `@Check` annotation, that will be called automatically for validating the model according to the type of the AST node being checked. The validation takes place in the background, together with parsing, while the user is writing a SWRTJ program, so that an immediate feedback is available, as usually in IDEs.

Binding the symbols (e.g., the binding of a field reference to its declaration) is important in compiler development. EMF uses “proxies” to represent references. It can delay the resolution (binding) of references when they are accessed. XTEXT already provides an implementation for binding references, which basically binds a reference to $n$ to the first element definition with name $n$ occurring in the model. This usually has to be adapted in order to take the visibility of names in a program into account. For instance, a field is visible only in the methods of a class, such that different hierarchies can safely have fields with the same name. XTEXT supports the customization of binding in an elegant way with the abstract concept of “scope”. The actual binding is still performed by XTEXT, but it can be driven by providing the scope of a reference, i.e., all declarations that are available in the current context of a reference. Note that this also permits to filter out elements according to their kind, e.g., in order not to mix field names with method names if we need to resolve a reference to a field.

The programmer can provide a customized `AbstractDeclarativeScopeProvider`. XTEXT will search for methods to invoke, using reflection, according to a convention on method name signatures. Suppose, we have a rule `ContextRuleName` with an attribute `ReferenceAttributeName` assigned to a cross reference with `TypeToReturn` type, that is used by the rule `ContextType`. You can create one or both of the following two methods:

```java
public IScope scope_<ContextRuleName>_<ReferenceAttributeName>(<ContextType> ctx, EReference ref)
public IScope scope_<TypeToReturn>(<ContextType> ctx, EReference ref)
```

The XTEXT binding mechanism looks for the first method (by reflection), if this does not exist, then it looks for the second. If no such method exists, the default linking semantics (see above) is used.

For instance, if we consider the grammar rule for method invocation illustrated at the beginning of this section, we can drive the resolution of the method name in a method invocation statement in any expression where such statement can occur by defining the following method (The code should be understandable without the knowledge of XTEXT):

```java
public IScope scope_MethodInvocation_method(Expression context, EReference ref) {
  ExpressionType expressionType = ExpressionType.createInstance(context.getReceiver());
  Collection<MethodName> methodList = null;
  if (expressionType != null)
    methodList = expressionType.getInvokableMethods();
  else
    methodList = new LinkedList<MethodName>();
  return Scopes.scopeFor(methodList);
}
```
The scope provider will be used by XTEXT not only to solve references, but also to implement code completion. Thus, a programmer achieves two goals by implementing the abstract concept of scope. Note that the code above can also return an empty scope, e.g., if the receiver expression in a method call cannot be typed. In that case, the XTEXT framework generates an error due to an unresolvable method name during validation, and an empty code completion list in case the programmer requests content assistance when writing the method name of a method invocation expression. This mechanism is handled by the framework itself, so that the programmer is completely relieved from these issues, once the correct scope provider is implemented.

XTEXT provides a (mostly) automatic support for file import/inclusion in the developed language by using grammar rules like the following:

```
Import : 'import' importURI=STRING;
```

SWRTJ programs can be split into separate files, and include other SWRTJ files using the `import` keyword. The corresponding dependencies among source files are handled by XTEXT itself. Thus, the EMF model for the AST corresponding to an included file is available automatically in the current edited source. Moreover, the modification of an included file $f$ automatically triggers the re-validation of all the files including $f$.

Finally, the code generation phase is dealt with in XTEXT by relying on XPAND [3], a code generation framework based on “templates”, specialized for code generation based on EMF models. This generation phase reuses the lookup functions and the type system functions used during validation. In our implementation of SWRTJ, code generation produces standard JAVA programs, which do not need any additional libraries to be compiled and executed. Our code generation phase implements the flattening procedure sketched in Section 2. However, by providing different templates, we could also generate C++ code (this is subject of future work), or code in another (possibly class-based) existing language.

XTEXT generates three plugin projects: one for the language parser and corresponding validators, one for the code generator, and one for the user interface IDE parts. The first two plugins do not depend on the third. Thus, it is straightforward to build a standalone compiler for SWRTJ to be executed outside Eclipse on the command line, which we also provide. Figure 1 shows a screenshot of the SWRTJ editor. Note the code completion functionalities, the outline, and the error markers. The project view also shows the generated JAVA files.

5 Evaluation

SWRTJ programs may look more verbose than standard class-based programs. However, the degree of reuse provided by records and traits is higher than the reuse potential of standard static class-based hierarchies. The distinction of each programming concept in a separate entity pays off in the long run, since each component is reusable in different contexts, in an unanticipated way. This does not happen so easily with standard class-based OO linguistic constructs. Class hierarchies need to be designed from the start with a specific reuse scenario in mind. In particular, for single inheritance,
design decisions should be made from the very beginning. This design might be hard to change, if not impossible, forcing either to refactoring or to code duplication.

Our linguistic constructs are lower-level than standard OO mechanisms. However, some syntactic sugar can be added to reduce the amount of code to write in some situations. For instance: class names might be used as types, fields might also be declared directly in classes, etc. Along the same lines, we can simulate class inheritance with our linguistic constructs. This can be easily achieved by inverting the “flattening” concept. For instance, if we start from the flattened \texttt{SyncAccount} class in Section 2, we can easily separate fields and methods into automatically generated records and traits, and generate the corresponding SWRTJ class declaration. Similarly, inheritance and method overriding can be simulated with inherited interfaces and trait method renaming, respectively. The \texttt{super} call can be simulated with a call to the renamed version of the method. However, this would decrease the level of reuse for such components.

This issue is related to the debate of whether it is better to have a pure or hybrid programming language. In this respect, the purity of a language usually imposes more verbose solutions than a hybrid language. For instance, consider the amount of code for writing the \texttt{main} static public method of a public class in \texttt{JAVA}, with simple form of the corresponding function in C++ which also provides functions besides classes and methods. The goal of reducing verbosity often led to additional language constructs which may break the pure linguistic features, e.g., the addition of imperative features to a purely functional language (see, e.g., Objective Caml, [34]). The general debate
between pure and hybrid languages is out of the scope of the present paper. Nonetheless, we argue that having linguistic constructs for reusable code development eases adding other high-level linguistic constructs which otherwise may often only be possible at the cost of code duplication and of the resulting complexity of code maintenance.

Our experience with XTEXT was in general quite positive. As usual, some time is required to get acquainted with the concepts of the framework. In particular, XTEXT relies on EMF. Thus, one should be familiar with EMF concepts as well, especially, when it comes to analyse the model for validation and code generation. However, after this knowledge is achieved, developing a language compiler and an IDE using XTEXT is extremely fast. XTEXT seems to be the right tool to experiment with language design and to develop implementations of languages. Furthermore, experimenting with new constructs in the language being developed can be handled straightforwardly. It requires to modify the grammar, regenerate XTEXT artifacts and to deal with the cases for the new constructs. Finally, XTEXT leaves the programmer with the possibility of customizing every aspect of the developed language implementation by specialized code (which is flexibly “injected” in XTEXT using Google Guice), even though XTEXT hides many internal details of IDE development with Eclipse. Even, EMF mechanisms are still open to adaptation. For instance, we developed a customized EMF resource factory for synthesizing the interfaces and classed of the internal library described in Section 3. This facilitates making interfaces and classes such as IList and CArrayList transparently available in every program (represented as an EMF model), without having to treat them differently in program validation.

6 Related Work

Traits are well suited for designing libraries and enable clean design and reuse which has been shown using SMALLTALK/SQUEAK (see, e.g., [11, 15]). Recently, [7] pointed out limitations of the trait model caused by the fact that methods provided by a trait can only access state by accessor methods (which become required methods of the trait). To avoid this, traits are made stateful (in a SMALLTALK/SQUEAK-like setting) by adding private fields that can be accessed from the clients possibly under a new name or merged with other variables. In SWRTJ traits are stateless. By their required fields, however, it is possible to directly access state within the methods provided by a trait. Moreover, the names of required fields (in traits) and provided fields (in records) are unimportant because of the field rename operation. Since field renaming works synergically with method renaming, exclusion and aliasing, SWRTJ has more reuse potential.

In [36], a variant of traits that can be parameterized by member names (field and methods), types and values is proposes. Thus, the programmer can write trait functions that can be seen as code templates to be instantiated with different parameters. This enhances the code reuse provided by traits already. It could be interesting to adapt this approach to our context and to extend the parameterization functionalities also to interfaces, records and classes. This will be the subject of future work. However, an important difference between our proposal and the one in [36] is that, in the latter, traits play also the competing role of type, which is avoided in SWRTJ. Another feature of SWRTJ is that structural types are used only “internally” on this, i.e., the program-
mer works with nominal types (interfaces) alone. We believe this is an important feature from a practical point of view, as it reduces the distance between the classical JAVA-like languages and our linguistic constructs, from the perspective of the programmer.

With respect to XTEXT, there are other tools for implementing text editors (and IDE functionalities) in Eclipse. Tools like IMP (The IDE Meta-Tooling Platform) [2] and DLTK (Dynamic Languages Toolkit) [16] only deal with IDE functionalities and leave the parsing mechanism completely to the programmer, while XTEXT starts the development cycle right from the grammar itself. Another framework, closer to XTEXT is EMFText [23]. EMFText basically provides the same functionalities. But, instead of deriving a meta-model from the grammar, it does the opposite, i.e., the language to be implemented must be defined in an abstract way using an EMF meta model. (A meta model is a model describing a model, e.g., an UML class diagram describing the classes of a model). Note that XTEXT can also connect the grammar rules to an existing EMF meta model, instead of generating an EMF meta model starting from the grammar. XTEXT seems to be better documented than EMFText (indeed, both projects are still young and always under intense development), and more flexible, especially since it relies on Google Guice. On the other hand, EMFText offers a “language zoo” with many examples that can be used to start the development of another language. In this respect, the examples of languages implemented using XTEXT, we found on the web are simpler DSLs, and not programming languages like SWRTJ. Thus, this paper can also be seen as a report of effective usage of XTEXT for implementing more complex programming languages.

7 Conclusions and Future Work

In this paper, we presented the programming language SWRTJ and its implementation. SWRTJ is based on the calculus presented in [10]. In that paper, we considered mechanisms for code reuse for implementing Software Product Lines (a set of software systems with well-defined commonalities and variabilities [17]). We explored a novel approach to the development of SPL, which provides flexible code reuse with static guarantees. In order to be of effective use for SPL, the type-checking has to facilitate the analysis of newly added parts without re-checking already existing products. The SWRTJ type system (Section 3) satisfies this requirement since it supports type-checking records and traits in isolation from classes that use them.

A special form of reuse is at the base of the contemporary agile software development methodologies, which are based on an iterative approach, where each iteration may include all of the phases necessary to release a small increment of a new functionality: planning, requirements analysis, design, coding, testing, and documentation. Another example is the use of Extreme Programming [6], where team members work on activities simultaneously. While an iteration may not add enough functionality to guarantee the release of a final product, an agile software project intends to be capable of releasing new software at the end of every iteration. However, this means that the next iteration will reuse the software produced in the previous ones. We believe that an interesting future research direction is to investigate whether the programming lan-
guage features proposed in this paper may help in writing software following an agile methodology.

In [8], we presented a tool for identifying the methods in a JAVA class hierarchy that could be good candidates to be refactored in traits (by adapting the Smalltalk analysis tool of [27] to a JAVA setting). It will be interesting to investigate the application of this approach also for detecting possible candidates for records and traits in the context of porting existing JAVA code to SWRTJ code.

Traits have been proposed as a lightweight mechanism for fine-grained reuse, with a level of granularity finer than classes. They support reuse at the granularity of methods. In future work, we are aiming to investigate language features for coarse-grained reuse, on level of granularity coarser than classes that smoothly integrate with traits. A good starting point for this line of work is the notion of components described in [33].

Acknowledgements. We are grateful to the developers of XTEXT, in particular, Sven Efftinge and Sebastian Zarnekow, for their prompt help and support during the development of SWRTJ. We are grateful to Viviana Bono for initial collaboration on the subject of this paper and thank Stéphane Ducasse for many insightful discussions about traits, and Alexandre Bergel for many interesting comments on the technical report [9].

References

19. S. Ducasse, O. Nierstrasz, N. Schärdi, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *ACM TOPLAS*, 28(2):331–388, 2006.
21. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
Appendix M

A Model-Based Framework for Automated Product Derivation

The paper “A Model-Based Framework for Automated Product Derivation” [105] follows.
A Model-Based Framework for Automated Product Derivation∗

Ina Schaefer, Alexander Worret, Arnd Poetzsch-Heffter

Software Technology Group
TU Kaiserslautern
Kaiserslautern, Germany
{inschaef | worret | poetzsch}@cs.uni-kl.de

Abstract—Software product line engineering aims at developing a set of systems with well-defined commonalities and variabilities by managed reuse. This requires high up-front investment for creating reusable artifacts, which should be balanced by cost reductions for building individual products. We present a model-based framework for automated product derivation to facilitate the automatic generation of products. In this framework, a model-based design layer bridges the gap between feature models and implementation artifacts. The design layer captures product line variability by a core design and ∆-designs specifying modifications to the core for representing product features. This structure is mapped to the implementation layer guiding the development of code artifacts capable of automatic product derivation. We evaluate the framework for a CoBox-based product line implementation using extended UML class diagrams for the design and frame technology for the implementation layer.

Keywords-Software Product Lines; Automated Product Derivation; Model-based Development; Frame Technology

I. INTRODUCTION

A software product line is a set of software systems with well-defined commonalities and variabilities [1]. Software product line engineering aims at developing these systems by managed reuse in order to reduce time to market and to increase product quality. The creation of reusable artifacts requires a high up-front investment which should be balanced by cost reductions for building individual products. Currently, derivation of single products requires manual intervention during application engineering, especially for product implementation, which can be tedious and error-prone [2]. Hence, it cannot be guaranteed that the overall development costs are reduced by product line engineering when compared to other reuse approaches.

Automated product derivation (or software mass customization [3]) is an approach to create single products by removing the need for manual intervention during application engineering. Besides, automated product derivation allows centralized product line maintenance and product line evolution, because modifications of the artifacts can automatically be propagated to the products. In order to be able to create products automatically, product line variability is restricted to configurative variability [4]. The different product configurations are captured in a feature model where features are designated product characteristics. Automated product derivation means that a product implementation for a particular feature configuration is automatically generated from the reusable product line artifacts. Software product line engineering processes, such as PuLSE [5] or KobrA [6], focus on managing product line variability in all software development phases, but leave product derivation as a manual activity. In [7], only organizational and technical requirements for automated product derivation are considered. Some approaches [8], [9] aim at automatically deriving design documents. However, no approach provides guidance for the design and implementation of product line artifacts capable of automated product derivation.

To overcome this problem, we propose a model-based framework for automated product derivation. A design layer bridges the gap between feature models and product implementations. During domain engineering, it guides the development of implementation artifacts capable of automated product derivation. On the design layer, a product line is described by a core design and a set of ∆-designs. The core design represents a product with a basic set of features. The ∆-designs define modifications to the core design that are necessary to incorporate specific product characteristics. ∆-designs can cover combinations of features. This makes the presented approach very flexible because modifications caused by several features can be designed differently from modifications caused by one of these features. In order to obtain a design for a product with a particular feature configuration during application engineering, the modifications specified by the respective ∆-designs are applied to the core. A design can be validated and verified before code artifacts are developed. Furthermore, designs can be refined based on the principles of model-driven development [10]. Refinements are orthogonal to product line variability because they can be performed in both core and ∆-designs equally. Therefore, the proposed approach serves as a basis for model-driven development of software product lines with automated product derivation.

In order to develop reusable code artifacts capable of automated product derivation, the structure of the design layer is mapped to the implementation layer. A product

∗This work has been partially supported by the Rheinland-Pfalz Research Center for Mathematical and Computational Modelling (CNaM)² and by the European project HATS, funded in the Seventh Framework Program.
line implementation consists of a core implementation of the product described by the core design and a set of $\Delta$-implementations corresponding to the $\Delta$-designs which specify the modifications to the core implementation to realize the designated product characteristics. The core design and core implementation refer to a complete product and can be developed by single application engineering techniques. The implementation of a product for a particular feature configuration is obtained automatically during application engineering by applying the modifications of the respective $\Delta$-implementations to the core implementation. The design layer is independent of a specific implementation technique. The only requirement for a concrete implementation technique is that the modifications of the core can be represented appropriately and applied automatically. The separation of design and implementation artifacts into core and $\Delta$-designs/implementations allows a stepwise development of the software product line. The approach can easily deal with evolving software product lines by capturing new features in additional $\Delta$-designs/implementations.

We have evaluated the proposed model-based framework at the development of a shopping system product line. In order to consider variable deployments, the implementation layer is based on the CoBox component model [11]. We developed a notation for CoBox-based core and $\Delta$-designs. For implementing the product line variability, we applied frame technology [12]. The core implementation is captured by core frames and the $\Delta$-implementations by sets of $\Delta$-frames specifying the modifications to the core implementation.

The main advantages of the model-based framework for automated product derivation are:

- The separation of core and $\Delta$-designs/implementations allows an evolutionary development of product lines.
- Product variability can be handled very flexibly because $\Delta$-designs/implementations allow representing modifications caused by combinations of features.
- The design layer facilitates model-based validation and verification before implementation.
- The framework can be used with different implementation techniques to exploit their strengths in particular application domains.
- The framework serves a basis for model-driven development of software product lines with automated product derivation because refinements are orthogonal to product line variability.

This paper is organized as follows: In Section II, we review related work. In Section III, we present our model-based framework for automated product derivation that is realized in Section IV and evaluated in Section V. Section VI concludes the paper with an outlook to future work.

II. RELATED WORK

Model-driven development [10] is increasingly used in software product line engineering. Many approaches focus on modeling product line variability. In KobrA [6], UML diagrams are annotated with variant stereotypes to describe variation points in models. In [13], a UML profile for representing product line variability is introduced. However, resolving the modeled variabilities requires additional documents and manual intervention.

In [14], [15], the general idea to use model-driven development for product derivation is advocated. Models in the problem domain, which correspond to feature models of product lines, are stepwise transformed to models in a solution domain, i.e. models of products or product implementations. However, these approaches rely on manual intervention for configuring and performing model transformations. [4] proposes the integration of model-driven development and aspect-oriented concepts. The introduced notion of positive variability refers to a core model to which selectively certain parts are added. The difference of this notion to $\Delta$-designs/implementations is that the latter can also contain modifications and removals of design and implementation artifacts. Model transformations in [4] are realized by aspect-oriented composition of artifacts which also extends to the implementation by means of aspect-oriented programming concepts. However, the manual implementation of certain product parts is explicitly included in the approach which is not considered in our framework for automated product derivation.

Most approaches for automated product derivation consider only the design layer or the implementation layer. For automated derivation of product designs, [8] proposes an approach to automatically generate UML class and activity diagrams via annotations from variability models of the complete product line. In [9], product architectures are automatically derived from a common domain architecture model by means of model transformations. [16] considers an automated derivation of UML class diagrams by resolving explicitly specified feature-class dependencies.

There are different technologies for automated code generation applied in the context of software product lines, such as conditional compilation, frame technology [12], [17], generative programming [18] or code annotations [19]. Also, compositional approaches, such as aspect-oriented programming [20], feature-oriented programming [21] or mixins [22], are used to automatically generate product implementations from reusable artifacts. However, in order to generate products, it is assumed that the necessary code artifacts already exist. A systematic process how to design these artifacts is not provided.

The model-based framework for automated product derivation presented in [23] is structurally similar to the framework proposed in this paper. It contains a modeling layer describing the relation between product features and implementation artifacts. Because the implementation is based on aspect-oriented programming, the models define how classes and aspects are composed for feature con-
figurations. Product derivation is fully automated, but in contrast to the work presented in this paper, the approach is conceptually restricted to aspect-oriented techniques. This limits the means for dealing with product variability to the expressiveness of aspect-oriented concepts that can, for instance, not deal appropriately with features removing code.

III. MODEL-BASED AUTOMATED PRODUCT DERIVATION

In order to provide a standardized technique how to design and implement product line artifacts suitable for automated product derivation, we propose a model-based framework. This approach is based on a model-based design layer that links product line variability declared in a feature model with the underlying implementation layer.

Overview. The proposed approach is structured into three layers (see Figure 1). During domain engineering, the variability of the software product line is captured by a feature model on the feature layer. Based on the feature model, reusable design and code artifacts are developed representing the product line variability on the underlying design and implementation layers. The design and the implementation artifacts are separated into a core design/implementation and \( \Delta \)-designs/implementations, respectively, that can be configured automatically for a specific feature configuration during application engineering. The design concepts can be chosen such that relevant system aspects in each design stage can be adequately expressed. The design layer is independent of a concrete implementation technique, but provides the structure of the implementation artifacts. Designs can be refined based on the principles of model-driven development [10], until they are detailed enough for implementation. A product line design can be validated and verified before the development of code artifacts such that errors can be corrected less costly.

Feature Layer. The products of a software product line are described by a feature model. Features can represent functional behavior of products, but can also refer to non-functional aspects, such as deployment issues. A feature model declares the configurative variability of the product line, i.e., the commonalities of all products are captured by mandatory features, possible variabilities are modeled by optional features, and constraints between features are defined. The set of possible products of a product line is described by the set of valid feature configurations.

Design Layer. The design of a product line is split into a core design and a set of \( \Delta \)-designs that are developed during domain engineering. The core design corresponds to a product of the product line with a basic set of features. This core can be developed according to well-established single application design principles. The variability of the product line is handled by \( \Delta \)-designs. The \( \Delta \)-designs declare modifications to the core design in order to represent specific product characteristics. The step from the feature model to the design artifacts is a creative process because product line variability can be represented in different ways in a design.

In order to find a core design for a product line, a suitable basic feature configuration has to be identified. Mandatory features are always contained in the basic configuration, as they have to be present in all valid configurations. For optional features, the guideline adopted is that \( \Delta \)-designs should add rather than remove functionality. If an optional feature only adds entities to the design, the feature should not be a part of the basic configuration. However, if an optional feature is included in many products, adding it to the core configuration can be beneficial because it can be tested thoroughly without considering product line variability. If selecting an optional feature causes that functionality is excluded from products, this feature should be contained in the core configuration to keep the core as small as possible. Alternative features represent options where at least one or exactly one feature has to be included in a valid configuration. Since the core configuration has to be valid, a choice between these options is necessary. If a feature selection requires to pick at least one feature, for the core exactly one feature should be chosen. The decision which option to include in the core can be based on an estimation which feature is most likely contained in many configurations.

\( \Delta \)-designs define modifications of the core design to incorporate specific product characteristics. The modifications caused by \( \Delta \)-designs comprise additions of design entities, removals of design entities and modifications of the existing design entities. The \( \Delta \)-designs contain application conditions determining under which feature configurations the specified modifications have to be carried out. These application conditions are Boolean constraints over the features contained in the feature model and build the connection
between features in the feature model and the design level. A $\Delta$-design does not necessarily refer to exactly one feature, but potentially to a combination of features. For example, if the feature model contains two features $A$ and $B$, the constraint $(A \land \neg B)$ attached to a $\Delta$-design denotes that the modifications are only carried out for a feature configuration if feature $A$ is selected and feature $B$ is not selected.

The general application constraints allow very flexible $\Delta$-designs as combinations of features can be handled individually. The number of $\Delta$-designs that are created for a feature model depends on the desired granularity of the application conditions. The application conditions of all $\Delta$-designs can be checked if all features are addressed in at least one design. In order to obtain a design for a particular product during application engineering, all $\Delta$-designs whose constraints are valid under the respective feature configuration are applied to the core. This can involve different $\Delta$-designs that are applicable for the same feature in isolation as well as in combinations with other features. To avoid conflicts between modifications targeting the same design entities, first all additions, then all modifications and finally all removals are performed.

**Implementation Layer.** In order facilitate automated product derivation, the structure of the design is mapped to the structure of the implementation artifacts that are developed during domain engineering. The implementation artifacts are separated into a core implementation and $\Delta$-implementations. The core design is implemented by the core implementation. As the core design is a complete product, single application engineering methods can be applied for implementing the core. This implementation can also be validated and verified thoroughly by well-established principles. $\Delta$-designs are implemented by $\Delta$-implementations which have the same structure as the $\Delta$-designs. The additions, modifications and removals of code specified in $\Delta$-implementations capture the corresponding additions, modifications, and removals declared in the $\Delta$-designs. The application condition attached to a $\Delta$-implementation determines under which feature configurations the code modifications are to be carried out. The conditions directly refer to the application condition of the implemented $\Delta$-designs. The process to obtain a product implementation for a specific feature configuration during application engineering is the same as for the design. The modifications specified by all $\Delta$-implementations with a valid application conditions under a specific feature configuration are applied to the core. Again, first all additions, then all modifications and finally all removals of code are carried out. This analogous priority rule ensures that a product implementation generated for a specific feature configuration is an implementation of the corresponding product design.

The close correspondence between design layer and implementation layer provides a general approach to create reusable artifacts during domain engineering that suitable for automated product derivation during application engineering. The design layer provides the structure for the corresponding code artifacts. Because core design and core implementation are complete products, they can be developed by well-established principles from single application engineering. The independence of $\Delta$-designs and $\Delta$-implementations from core designs and core implementations, respectively, yields the potential of incremental, evolutionary product line development. Refinement of designs along the lines of model-driven development can easily be incorporated into the proposed framework, because refinement is orthogonal to the concepts for capturing product line variability. Since the design layer is independent of the implementation layer, the proposed model-based framework can be used with different concrete implementation techniques, as long as the concrete implementation technique allows expressing the desired modifications and supports automatic code generation.

**IV. A Framework for Model-based Automated Product Derivation**

In order to evaluate the proposed approach, we realized the model-based framework for automated product derivation for developing an information system product line. As application domain for the product line, we use the Common Component Modeling Example (CoCoME) [24] that describes a software system for cash desks dealing with payment transactions in supermarkets. Information systems involving clients-server communications are generally distributed and highly concurrent. To deal with this inherent complexity, we implement our system in the object-oriented, data-centric CoBox component and concurrency model [11]. A CoBox is a runtime component consisting of a (non-empty) set of runtime objects, i.e., other CoBoxes or instances of ordinary classes. Each CoBox at runtime executes a set of tasks, of which at most one can be active at any time. A task is active as long as it has not finished or willingly suspends its execution. Thus, inside a CoBox all code is executed sequentially. A CoBox communicates with other CoBoxes outside of its own CoBox via asynchronous messages. CoBoxes allow flexible deployment because the location where a CoBox is instantiated does not influence its functional behavior. This allows considering also variability of deployment besides variability of functionality in the product line to be developed.

**A. Feature Layer**

For representing product line variability on the feature layer, we use feature diagrams [25]. In a feature diagram, the set of possible product configurations is determined by a hierarchical feature structure. A feature can either be mandatory, if it is connected to its parent feature with a filled circle, or optional, if it is connected with an empty circle. Additionally, a set of features can form an alternative...
selection in which at least one (filled triangle) or exactly one (empty triangle) feature has to be included in a valid configuration. Furthermore, constraints between features can be represented by explicit links.

To use CoCoME [24] as an example for a product line, we extended the application scenario with functional and deployment variabilities keeping the original system as one possible configuration. The feature model for the CoCoME software product line is shown in Figure 2. A CoCoME system has different payment options. First, it is possible to pay by cash or by one of the non-cash payment options, i.e., credit card, prepaid card or electronic cash. At least one payment option has to be chosen for a valid configuration. Product information can be input using a keyboard or a scanner where at least one option has to be selected. Furthermore, the system has optional support to weigh goods, either at the cash desks themselves or at separate facilities. With respect to deployment, there is the alternative option to have a single-desk system with only one cashier or a multi-desk system with a set of cashiers. The multi-desk system can optionally comprise an express mode which requires cash payment or a self-service mode requiring non-cash payment.

B. Design Layer

Since we aim at a CoBox-based design and implementation of the product line, the design layer has to capture all relevant aspects for specifying CoBoxes. This includes the CoBoxes that classes belong to as well as deployment information for the CoBoxes. We introduce an extension to UML class diagrams [26] to express the additional information. Usually, UML diagrams are extended by stereotype annotations. This, however, would drastically impair the readability of the diagrams. With the extended notation, a CoBox design consists of a set of CoBoxes and ordinary classes. Graphically, CoBoxes are represented by a rounded box named the same as the owning CoBox class. Ordinary classes are denoted as usual UML classes. Both, CoBox classes and ordinary classes have member variables and methods. CoBoxes can contain other CoBoxes and other ordinary classes. UML relations describe relations between CoBoxes and classes. In addition, deployment information is provided by determining on which deployment targets CoBoxes should be instantiated. This is expressed by a doubled-headed arrow from a deployment target to a CoBox.

The design layer handles the variability of the feature model by a core design and a set of \( \Delta \)-designs. The core design of a CoBox-based product line is denoted by a CoBox design. \( \Delta \)-designs require additional notation to specify the modifications to the core design. In a \( \Delta \)-design, it is defined which CoBoxes or classes are added or removed and which member variables or methods in existing CoBoxes or classes are added, removed or modified. The + symbol marks additions, − marks removals and * denotes modifications.

As UML class diagrams already use the + and − symbols for public and private members, we attach the alteration symbols to the right top corner of an altered CoBox, an altered class or of a rectangle surrounding the altered class members. Additionally, each \( \Delta \)-design contains its application condition, a Boolean constraint over the features in the feature model, to determine for which configurations the \( \Delta \)-design is applied to the core. The application condition is displayed in an angular box at the top of the design.

The core configuration of the CoCoME software product line includes cash payment, keyboard input, and is a multi-desk system because cash payment and keyboard input are features of almost any cash desk system and most shops comprise more than one cashier. Other optional features are not incorporated into the core in order to keep it as...
small as possible. The resulting CoBox core design is shown in Figure 3. The design specifies the CashDesk and StoreServer CoBoxes for realizing the core functionality. Instances of the CashDesk and StoreServer CoBoxes are created on the deployment targets CashDesk Client and Store Server, respectively, that have to be physically connected. The logical connection is established via an additional ConnectionAgent CoBox created on each deployment target.

Figure 4 depicts the ∆-design containing the modifications for credit card payment, that is not included in the core configuration. To provide credit card functionality, a CoBox Bank has to be added to the system. Further, the CoBox CashDesk has to be extended by a CoBox CardReader and further class members to take care of the credit card payment. Also, the ConnectionAgent gets further member variables and methods to handle the communication with the Bank. This is denoted by the + symbol attached to the respective classes and members. Additionally, two methods of the CashDesk CoBox are modified, which is shown by the * symbol. Deployment information for the Bank CoBox is provided relative to the overall system. The deployment target Bank Server on which the Bank CoBox is to be instantiates has to establish a physical connection to the deployment target on which CashDesk CoBox is executed. This allows dealing with deployment modifications caused by other ∆-designs. The angular box in the top right corner of the ∆-design shows the application condition. This condition determines that the ∆-design is applied in all feature configurations in with the CreditCard feature is included. During application engineering, we obtain a design for a multi-desk system containing cash payment, credit card payment and keyboard input by applying the modifications specified in the ∆-design to the core design. This allows performing model-based validation and verification of this product already on the design level before the implementation is derived.

C. Implementation Layer

The implementation layer for the CoCoME software product line is realized by frame technology [12]. Frames structure source code into parts with pre-defined break points. The break points can be adapted by inserting code from other frames or by removing code from break points. In our model-based framework, the structure of the CoBox-based design is directly mapped to the frame structure on the implementation layer. The core design of a product line is realized by a set of core frames. Each CoBox in the core design is implemented by a core frame. The code in this core frame also contains break points that are necessary for modifications caused by ∆-frames. For each ∆-design in which the CoBox is altered, a ∆-frame is constructed that contains the respective modifications to this CoBox. Additionally, for each CoBox newly created by a ∆-design, a ∆-frame is generated that contains its implementation. The application conditions of the ∆-frames are the same as the ones of the implemented ∆-designs. Special build frames capture in which feature configurations the modifications of a ∆-frame are applied to the core frames.

XVCL [17] is a programming language-independent implementation of frame technology using an XML-dialect for defining frames, break points and break point adaptations. We use XVCL to realize the implementation layer of the CoCoME product line. The XVCL core frame for the CashDesk CoBox is depicted in Listing 5. This frame implements the design specified in Figure 3. The frame contains XVCL break tags for including additional attributes and methods that are specified by feature frames targeting this core frame. The ∆-frame in List-

Listing 5. Core Frame for the CashDesk CoBox

```xml
<cf-frame name="CashDesk_Core">
  public cobox class CashDesk {
    ...
    private Keyboard _keyboard;
    private Order _currentOrder;
    <break name="CashDesk_AdditionalAttributes"/>
    ...
    public void selectCashPayment() {
      _keyboard!setStateCashPayment();
    }
    ...
    <break name="CashDesk_AdditionalMethods"/>
  </cf-frame>
```

Figure 4. ∆-Design for Credit Card Payment
We realized the CoCoME product line with the proposed model-based framework for automated product derivation [27]. The CoBox-based implementation of the product line is carried out in JCoBox\(^1\) that is compiled to standard Java. As XVCL is programming language-independent, it is straightforward to use it for the JCoBox implementation of the CoCoME product line. In the current implementation, 168 different products of the CoCoME product line can be derived automatically by the XVCL-based two-step derivation process. The implementation of the CoCoME software product line consists of 12 core frames, 21 \(\Delta\)-frames and 12 build frames. The derivation process requires 6 additional meta frames not containing any source code to guide the derivation. Non-variable system parts are implemented in 5 regular source code files.

The advantage of the presented approach is that it is not limited to a particular implementation language or technique and applicable in a variety of scenarios. The development of the product line core allows using established single application engineering principles. Manual product-specific intervention is explicitly avoided such that modifications in any of the product line artifacts can be fully automatically propagated to existing products. There is no need for additional customization of products after product derivation. Modifications of the product line artifacts, however, affect all three layers. This introduces the need for additional synchronization mechanisms in case of parallel modifications.

\(^1\)http://softech.informatik.uni-kl.de/Homepage/JCoBox
feature models and product implementations. Its structure guides the development of implementation artifacts capable of automated product derivation. We realized and evaluated the proposed framework with an extended version of UML for the design and frame technology for the implementation.

For future work, we will realize the introduced framework with different implementation techniques to evaluate its general applicability. A first candidate is the trait-based language presented in [28]. Additionally, we will improve the tool support following our prototypical implementation. In order to analyze the effects of product line evolution for automated product derivation, we will formalize our approach to give a formal account how the design and implementation layers are affected by newly added features.

REFERENCES


Appendix N

Variability Modelling for Model-Driven Development of Software Product Lines

The paper “Variability Modelling for Model-Driven Development of Software Product Lines” follows.
Variability Modelling for Model-Driven Development of Software Product Lines

Ina Schaefer
Dept. of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden
schaefer@chalmers.se

Abstract—Model-driven development of software-intensive systems aims at reducing design complexity by shifting the focus during system development from implementation to modelling. A model is an abstraction of a system with respect to certain system aspects. In model-driven development, an initial system model is successively refined by adding details relevant in particular design phases. A software product line [2], [3] is a set of systems with well-defined commonalities and variabilities. In order to use model-driven development in software product line engineering, the variability of the different products has to be represented within the used modelling concepts and preserved under model refinement.

The variability of products in software product lines is currently predominantly captured by feature models [4]. Features represent important product characteristics. A feature model determines a set of products by the set of valid feature configurations. However, features at the level of a feature model are merely labels [5]. Hence, feature-based variability has to be mapped to the modelling concepts used on each modelling level [6] in order to design a product line by a model-driven development process.

Existing approaches to integrate feature-based variability into modelling languages can be classified in two main directions [7]. First, negative variability-based approaches consider one model for all products of a product line that is augmented with variant annotations determining which model elements are present in which products [8], [9], [10], [11]. Second, positive variability-based approaches [6], [7], [12], [13], [14] associate model fragments to features and compose them for a given feature configuration. However, most approaches only focus on modelling concepts used on one modelling level and do not consider how the variability representation can be preserved under model refinement.

In order to define a seamless model-driven development process for software product lines, we propose $\Delta$-modelling, a general concept integrating variability modelling with model refinement. On each modelling level, product line variability is represented by a core model and a set of $\Delta$-models. The core model represents a valid product of the product line. $\Delta$-models specify changes of the core model, i.e., additions, modifications and removals of model fragments, in order to capture further products. An application condition attached to a $\Delta$-model determines for which feature configuration a $\Delta$-model is applicable. A product model for a feature configuration is obtained by applying the modifications specified by the $\Delta$-models with valid application conditions.

For refinement, the core model and every $\Delta$-model are transformed independently into a more detailed core model or $\Delta$-model, respectively. The internal structure of the core and $\Delta$-models, as well as the application conditions of the $\Delta$-models are preserved. If the specified modifications in the refined $\Delta$-models satisfy local refinement compatibility conditions, a refined product model for a feature configuration can be obtained in two ways: first, the product model is configured on the higher level of abstraction and afterwards transformed to a refined model; or second, the core and $\Delta$-models are refined and afterwards configured by applying the modifications of the refined $\Delta$-models to the refined core model. The commutativity of model refinement and model configuration builds the basis for incremental model-driven development of software product lines.

The $\Delta$-modelling concept provides an integrated variability modelling approach for model-driven development of software product lines. Its main characteristics are:

- $\Delta$-modelling is independent of a concrete modelling or implementation language. It can be instantiated to concrete modelling or implementation languages by

---

*This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) and by the European project HATS, funded in the Seventh Framework Program.

© SSE, University Duisburg-Essen, Germany
defining the semantics of \( \Delta \)-application.

- Combinations of features can be explicitly captured by flexible application conditions attached to \( \Delta \)-models.
- Modular and evolutionary system development is facilitated by adding \( \Delta \)-models to an existing model.
- Model refinement is orthogonal to variability modelling. Core and \( \Delta \)-models are refined independently such that variability is expressed by the same structural concepts on all modelling levels.
- The commutativity of model configuration by \( \Delta \)-application and model refinement provides the basis for an incremental development process by stepwise refinement of core and \( \Delta \)-models.

The outline of this paper is as follows: In Section II, we review related work. In Section III, we explain the \( \Delta \)-modelling approach at an example of a trading system product line. In Section IV, we formalize \( \Delta \)-modelling and extend this formalization to model refinement in Section V. Section VI concludes with an outlook to future work.

II. RELATED WORK

Model-driven engineering for software product line development is proposed in [10], [15] in order to resolve product variability by model transformations. A model in the problem domain, usually a feature model, is transformed into a model in the solution domain, e.g., a product model [7], product architecture [16] or product implementation [17].

The existing approaches to represent feature-based variability can be classified into two main directions [7]. Annotative approaches specify negative variability. They consider one model representing all products of a product line. Variant annotations, e.g., using UML stereotypes [8], [9], [10], [11], define which parts of the model have to be removed to derive the model of a concrete product. [5] associates presence conditions to modelling elements to be removed in certain feature configurations.

Compositional approaches capture positive variability. Model fragments are associated with features and composed for a particular feature configuration. A prominent example is the AHEAD [12] approach. A product is built by stepwise refinement of a base module with a sequence of feature modules. In [6], [7], [13], models are constructed by aspect-oriented modelling techniques. [14] applies model superposition to compose model fragments. In [18], a product model is obtained by composition and refined by model transformation. [19] propose to represent model variability by a base model and associated variability and resolution models determining how modelling elements of the base model have to be replaced for a particular product model. The base model is similar to the core model in the \( \Delta \)-modelling approach while variability and resolution models correspond to \( \Delta \)-models, but are not directly connected product features.

Most of the above approaches only focus on the representation of variability on a single modelling layer. In [9], different modelling levels during system development are considered, but variability resolution is based on textual decision models that are separated from the system models. In contrast, \( \Delta \)-modelling facilitates a seamless representation of variability inbetween different modelling layers.

The notion of program deltas is introduced in [20] to describe the modification of an object-oriented program, e.g., by introduction of new fields or extension of methods. The mapping of collaborative features to models in [6] is similar to \( \Delta \)-models. Collaborative features can modify a core model by additions, removals and modifications, but require a one-to-one relationship to a feature. In [21], \( \Delta \)-modelling is presented as an approach to develop product line artifacts suitable for automated product derivation.

Feature-oriented model-driven development (FOMDD) [22] combines feature-oriented programming (FOP) with model-driven engineering. In FOMDD, a product can, first, be composed from a base module and a sequence of feature modules and afterwards transformed to another product model. Second, the base module and the feature modules can be transformed and then composed to a transformed product model. This is similar to the commutativity between model refinement and model configuration in \( \Delta \)-modelling. FOP can be seen as a special case of \( \Delta \)-modelling. Feature modules are always associated to exactly one feature, whereas \( \Delta \)-models explicitly consider combinations of features. In feature modules, only additions and modifications can be specified. In contrast, \( \Delta \)-models may contain removals of model parts. While the base module in FOP is fixed by the mandatory features, in \( \Delta \)-modelling, any valid product can be chosen as core model enabling a flexible product line design.

III. VARIABILITY MODELLING USING \( \Delta \)-MODELS

Figure 1 shows an overview of the model-based development process for software product lines using the \( \Delta \)-modelling approach. First, an initial model of the product line is created that captures the variability of the feature model. From this initial model on a high level of abstraction, successively refined models are constructed that describe more detailed aspects of the considered products.

On each modelling level, product variability is captured by a core model and a set of \( \Delta \)-models. A core model corresponds to a valid product of the product line. \( \Delta \)-models specify changes to the core model by additions, modifications and removals of model fragments in order to represent further products. An application condition is attached to every \( \Delta \)-model determining for which feature configurations the specified changes are to be carried out. In order to obtain a product model for a feature configuration, the changes specified by \( \Delta \)-models with valid application condition are applied to the core. The concept of \( \Delta \)-modelling to express variability is independent of a concrete modelling language. The modelling constructs used on each modelling level can be chosen to appropriately represent the considered system aspects.
The application conditions attached to the \( \Delta \)-models create the connection between the features in the feature model and product variability on the different modelling levels. If the selection of a feature influences the choice of the modelling language, e.g., if a feature refers to the used implementation framework, core and the \( \Delta \)-models can be seen tuples containing the specifications of the core and \( \Delta \)-models in the respective modelling formalisms, while the general variability structure is preserved.

The step from a feature model of a product line to the initial core model and the set of \( \Delta \)-models is a creative process, since product line variability can in general be represented in different ways. The variability structure provided by the initial modelling level provides the variability structure of the lower, more refined modelling levels (cf. Figure 1). A core model is refined to a more detailed core model. \( \Delta \)-models are refined to more detailed \( \Delta \)-models with the same application condition. An important property of the refinement between two modelling levels is that it commutes with model configuration by \( \Delta \)-application. This means that a refined configured product model can be obtained in the following two ways. First, the product model for a feature configuration is configured from the core model and the applicable \( \Delta \)-models and afterwards refined. Second, the core model and the \( \Delta \)-models are refined, such that afterwards the refined product can be configured. This commutativity property provides the basis for an incremental model-based development process by stepwise model refinement.

**Example** We illustrate variability modelling based on \( \Delta \)-models at the case example of a software product line of trading systems. The Common Component Modeling Example (CoCoME) [23] describes a software system handling payment transactions in supermarkets. It was extended to a software product line in [24]. The variability of the products are expressed in the feature diagram [4] shown in Figure 2. Mandatory features are represented by a filled circle, optional features with an empty circle. Alternative features are specified with a filled triangle if at least one feature has to be selected or by an empty triangle if exactly one features has to be selected. Constraints between features are represented by explicit links. A product in the trading system product line has different payment options, i.e., cash payment or payment by credit card, prepaid card or electronic cash. At least one payment option has to be chosen for a valid configuration. Product information can be entered using a keyboard or a scanner, where at least one option has to be selected. Furthermore, the system has optional support to weigh goods, either at the cash desks or at separate facilities. A trading system can be configured as a single-desk system with only one cashier or as a multi-desk system with several of cashiers. A multi-desk system can optionally comprise an express mode which requires cash payment or a self-service mode requiring non-cash payment.

A core model represents a product for a valid feature configuration. Thus, it can be developed by well-established single application engineering techniques as a standard product model. In the example, the feature configuration containing the keyboard, the cash payment and the single-desk system features is selected as core configuration.

**Component Modelling Level** In our example, we start the model-based development process at the component level by representing the core model by a UML component diagram [25] and its variability by component diagram \( \Delta \)-models that are an extension of UML component diagrams with annotations for the specified changes. In a second step, the component diagrams are refined to UML class diagrams showing in more detail how the components are implemented. Figure 3 depicts the component diagram specifying the core product of the trading system product line with the keyboard, cash payment and single-desk system features. It contains a Cash Desk component dealing with cash payment and an Inventory component keeping the store inventory. Every time a product is entered at the cash desk, the price of the product is requested from the inventory.
Figure 4 depicts the component diagram ∆-model containing the modifications of core component diagram to include credit card functionality. A Bank component has to be added specified by the + annotation at the Bank component. Additionally, the Cash Desk component has to be modified to handle credit card payment which indicated by * annotation at the Cash Desk component. In order to realize the communication with the Bank component, an required interface and a corresponding connection to the bank component have to be added. The application condition of this ∆-model (in the top right hand corner) defines that the modifications are carried out if the credit card feature is selected. In Figure 5, the component diagram for a single-desk system containing keyboard input, cash payment and credit card payment is depicted that results from applying the ∆-model for credit card payment (cf. Figure 4) to the core component diagram (cf. Figure 3).

Class Modelling Level Each component of the trading system product line can be refined to a class diagram. The class diagram represents the internal component structure and can be used as basis for an implementation. The interactions between the components are not considered on the class diagram level because they are already captured on the component modelling level. In the ∆-modelling approach, core and ∆-models are refined independently. A refined product model is obtained by applying the refined ∆-models to the refined core model.

Figure 6 shows the class diagram for the Cash Desk component contained in the core. It comprises a Cash Desk class implementing the main functionality of the cash desk. a Keyboard class handling the input from the keyboard and a Display class providing output to a display. Figure 7 depicts the class diagram ∆-model specifying the modifications of the core class diagram to incorporate the credit card feature. In order to provide credit card payment functionality, a Card Reader class is required. The Cash Desk class is modified by adding a reference to the bank, by adding methods to deal with the credit card payment and by modifying the existing payment methods.

Model Refinement and Configuration The class diagram for the configured Cash Desk component with the basic features and the credit card feature can be obtained in two different ways. First, the core model and the ∆-model on the component modelling level (cf. Figures 3 and 4) can refined to a core and ∆-model on the class diagram level (cf. Figures 6 and 7) and configured to a class diagram by the standard configuration procedure. Second, a component diagram including the Cash Desk component with the basic features and the credit card feature can be configured on the component diagram level (cf. Figure 5). Afterwards, the configured Cash Desk component can be refined to a class diagram specifying the component’s structure in more detail.

Model refinement and model configuration commute in the example because the refinement of the component diagram ∆-models is compatible with model configuration. Compatibility requires that for a component added (or removed) by a component diagram ∆-model, in the class diagram refinement of this ∆-model, all parts of the class diagram are specified as added (or removed) as well. If a component is modified in a component diagram ∆-model, in the refined class diagram, the parts of the class diagram may be specified as added, modified or removed. However,
the change operations resulting from refinement of a modification operation have to satisfy a local refinement compatibility condition. This condition requires that the class diagram obtained by applying the refined component \( \Delta \)-model to the class diagram of a refined core component is the same as the class diagram refinement of the same component configured on the component diagram level.

IV. FORMALIZING \( \Delta \)-MODELLING

Variability modelling using \( \Delta \)-models is a general approach that is not limited to specific modelling concepts, such as component or class diagrams used in Section III. \( \Delta \)-modelling can be applied to any modelling or implementation language by defining the semantics of the change operations specified in the \( \Delta \)-models for the concrete language. The number of modelling layers also depends on the concrete application and is not limited by the \( \Delta \)-modelling approach. For instance, in Section III, use case diagrams separated into core and \( \Delta \)-diagrams could be used to represent the requirements of the set of systems under development and subsequently be refined to component diagrams.

In order to show the general applicability of \( \Delta \)-modelling, we base the following formalization on a general notion of models. A model contains a set of modelling elements \( E \) that can, for instance, represent components or classes (by their names). The set of modelling elements \( E \) describes the (domain-specific) concepts used in the models. Furthermore, a model contains relations between modelling elements representing correspondences, such as connections between required and provided interfaces in component diagrams. For simplicity, we restrict our notion of a model to contain only one binary relation \( R \) over the set of modelling elements \( E \). This allows us to consider relations formally while keeping the model and its formal treatment simple. In a concrete model instantiation, a set of relations can be defined to express different relationships between elements.

Definition 1 (Models): Let \( E \) be a set of modelling elements. A model \( M \) is a tuple \( M = (E, R) \) where \( R \subseteq E \times E \) is a relation over the modelling elements.

A core model represents a product for a valid feature configuration. This allows treating the core model in the same way as any product model. We define the set of valid feature configurations as a subset of the powerset of the set of features.

Definition 2 (Core and Product Models): For a set of features \( F = \{f_1, \ldots, f_n\} \), let \( \mathcal{F} \subseteq 2^F \) denote the set of valid feature configurations. A core model (product model) is a triple \( C = (E, R, f) \) where \( f \in \mathcal{F} \) is a valid feature configuration, and \((E, R)\) is a model representing the feature configuration \( f \).

A \( \Delta \)-model specifies changes to a core model to model other products. For a core model \( C = (E, R, f) \), a \( \Delta \)-model defines additions, modifications and removals of modelling elements \( e \in E \), and additions and removals of tuples in the relation \( R \). A \( \Delta \)-model has an application condition determining under which feature configurations the specified changes have to be carried out. Application conditions are logical (e.g., Boolean) constraints over the features contained in the feature model. A \( \Delta \)-model does not necessarily refer to exactly one feature, but potentially to a combination of features. This allows very flexible \( \Delta \)-models as combinations of features can be handled individually. For example, if a feature model contains two features \( A \) and \( B \), the Boolean constraint \((A \land \neg B)\) denotes that the modifications are only carried out for a feature configuration if feature \( A \) is selected and feature \( B \) is not selected. The granularity of the application conditions determines the number of \( \Delta \)-models that have to be created to ensure that all features present in the feature model are appropriately captured.

Definition 3 (\( \Delta \)-Models): A \( \Delta \)-model over a model \( M = (E, R) \) is a tuple \( \Delta = (\varphi, Op) \) where the application condition \( \varphi \) is a constraint over the set of features \( F = \{f_1, \ldots, f_n\} \) and \( Op = \{op_1, \ldots, op_m\} \) is a set of operation modifications over the model \( M \) with

\[
\text{op}_i ::= \text{add } e \mid \text{mod } e \mid \text{rem } e \mid \text{add } r(e_1, e_2) \mid \text{rem } r(e_1, e_2)
\]

In order to obtain a product model for a feature configuration \( f \in \mathcal{F} \), all \( \Delta \)-models with valid application condition for the feature configuration \( f \) are applied to the core model. This can involve different \( \Delta \)-models that are applicable for the same feature. To limit the occurrence of conflicts between changes targeting the same modelling elements and relations, first all additions, then all modifications and finally all removals are performed. In order to express this ordering formally, we assume that \( \Delta \)-models are normalized, i.e., their change operations contain only additions, only modifications, or only removals. A \( \Delta \)-model \( \Delta = (\varphi, Op) \) can be normalized by splitting it into three disjoint normalized \( \Delta \)-models \( \Delta_a = (\varphi, Op_a), \Delta_m = (\varphi, Op_m) \) and \( \Delta_r = (\varphi, Op_r) \) such that \( Op = Op_a \cup Op_m \cup Op_r \). We call a set of \( \Delta \)-models \( \Delta = \{\Delta_1, \ldots, \Delta_n\} \) sorted if and only if there exist \( i, j \) with \( 1 \leq i \leq j \leq n \), such that \( \Delta_1, \ldots, \Delta_i \) contain only additions, \( \Delta_{i+1}, \ldots, \Delta_j \) contain only modifications, and

![Figure 7. Class Diagram \( \Delta \)-Model](image-url)
\(\Delta_{j+1}, \ldots, \Delta_n\) contain only removals. The operation \(\nu(\Delta)\) transforms a set of \(\Delta\)-models into a set of normalized and sorted \(\Delta\)-models. The application function \(\text{apply}(M, Op)\) modifying a model \(M\) by the change operations \(Op\) is defined in Definition 5.

**Definition 4 (Configuration):** Let \(C = (E, R, f_0)\) be a core model and \(\Delta = \{\Delta_1, \ldots, \Delta_n\}\) be a sequence of normalized and sorted \(\Delta\)-models with \(\Delta_i = (\varphi_i, Op_i)\) for all \(i\). For a feature configuration \(f \in F\), a product model \(P_f = (E_P, R_P, f)\) is configured by \(\text{conf}\{(E, R), \{\Delta_1, \ldots, \Delta_n\}, f\}\) where for a model \(M = (E_M, R_M)\), its configuration is defined by

\[
\text{conf}(M, \{\Delta_1, \ldots, \Delta_n\}, f) =
\begin{cases}
\text{conf}(\text{apply}(M, Op_1), \{\Delta_2, \ldots, \Delta_n\}, f) & \text{if } f \models \varphi_1 \\
\text{conf}(M, \{\Delta_2, \ldots, \Delta_n\}, f) & \text{otherwise}
\end{cases}
\]

and \(\text{conf}(M, \emptyset, f) = (E_M, R_M, f)\).

The ordering in which the change operations specified in a single \(\Delta\)-model are applied to a model is not fixed, which can also be seen as simultaneous application of the specified changes. The same modelling element can be added several times, but only occurs once in the model. Similarly, a tuple in the relation is added only once, if the related modelling elements are contained in the model. The modification of a modelling element also causes that the element is replaced in all relational tuples in which the original modelling element is contained. If an element is removed, all relational tuples containing this element are removed from the relation. The application of a change operation is undefined if a modelling element \(e\) is modified that is not contained in the core or added before by another \(\Delta\)-model, or if a modelling element or a relational tuple is removed, that is not contained in the core or added before by another \(\Delta\)-model.

**Definition 5 (\(\Delta\)-Application):** The application of a set of change operations \(Op = \{op_1, \ldots, op_n\}\) to a model \(M = (E, R)\) is defined by the application function \(\text{apply}\):

- \(\text{apply}(M, \emptyset) = M\)
- \(\text{apply}(M, Op) = \text{apply}(\text{apply}(M, op_1), Op \setminus \{op_1\})\)
- \(\text{apply}(M, \text{add} e) = (E \cup \{e\}, R)\)
- \(\text{apply}(M, \text{mod} e) = (E \setminus \{e\} \cup \{e'\}, R')\), if \(e \in E\) where \(e' \in E\) is the result of the modification of \(e \in E\)
- \(\text{apply}(M, \text{rem} e) = (E \setminus \{e\}, R')\), if \(e \in E\) where \(R' = \{(e_1, e_2) \mid (e_1, e_2) \in R, e_1, e_2 \neq e\} \cup \{(e', e_2) \mid (e_2, e) \in R, e_2 \neq e\} \cup \{(e_2, e') \mid (e_2, e) \in R, e_2 \neq e\} \cup \{(e', e') \mid (e, e) \in R\}\)
- \(\text{apply}(M, \text{add} r(e_1, e_2)) = (E, R \cup \{r(e_1, e_2)\})\), if \(e_1, e_2 \in E\)
- \(\text{apply}(M, \text{rem} r(e_1, e_2)) = (E, R \setminus \{r(e_1, e_2)\})\), if \(e_1, e_2 \in E\) and \(r(e_1, e_2) \in R\).

Despite using normalized and sorted \(\Delta\)-models during configuration, there can still be conflicts between the change operations specified in different \(\Delta\)-models. A conflict occurs if a modelling element or tuple in a relation is added and removed by two different \(\Delta\)-models, if a modelling element is modified and removed by two different \(\Delta\)-models, or if a modelling element is modified by two different \(\Delta\)-models. This indicates that the granularity of the \(\Delta\)-models and their application conditions is too coarse. Conflicts can be removed by splitting \(\Delta\)-models and refining them to explicitly cover the conflicting feature combinations.

**Definition 6 (Conflicts in \(\Delta\)-Models):** A set of \(\Delta\)-models \(\Delta = \{\Delta_1, \ldots, \Delta_n\}\) contains a conflict if for a feature configuration \(f \in F\), there are \(\Delta\)-models \(\Delta_i\) and \(\Delta_j\) with \(i \neq j\), \(f \models \varphi_i\) and \(f \models \varphi_j\), and there exists \(e \in E\), or \(e_1, e_2 \in E\) and \(r(e_1, e_2) \in R\), such that one of the following holds:

- \(\text{add} e \in Op_i\) and \(\text{rem} e \in Op_j\)
- \(\text{add} r(e_1, e_2) \in Op_i\) and \(\text{rem} r(e_1, e_2) \in Op_j\)
- \(\text{mod} e \in Op_i\) and \(\text{rem} e \in Op_j\)
- \(\text{mod} e \in Op_i\) and \(\text{mod} e \in Op_j\)

A core model \(C = (E, R, f_0)\) and a set of \(\Delta\)-models \(\Delta\) are well-defined if for all valid feature configurations \(f \in F\), all applications of \(\Delta\)-operations are defined and there are no conflicts between any two \(\Delta\)-models. Well-definedness is a prerequisite for commutativity of model configuration and model refinement.

V. MODEL REFINEMENT AND CONFIGURATION

Based on the formalization of the \(\Delta\)-modelling approach in Section IV, model refinement of core and \(\Delta\)-models can be defined. A model is transformed to a more detailed model by refining the contained modelling elements to models themselves, as in the example in Section III, components are refined to class diagrams showing their internal structure. Relations between modelling elements are not considered for refinement, such as connections between components are only relevant on the component modelling level.

**Definition 7 (Model Refinement):** The refinement operation \(\text{refine}\) maps every modelling element \(e \in E\) to a model \(M\) such that \(\text{refine}(e) = M_e = (E_e, R_e)\). The refinement \(M'\) of a model \(M = (E, R)\) is defined by \(\text{refine}(M) = \{(M'' \mid M'' = \text{refine}(e), e \in E)\}

\{(r(\text{refine}(e_1), \text{refine}(e_2)) \mid r(e_1, e_2) \in R)\}\)

The core model and all other product models can be refined using the above definition of model refinement. In order to refine \(\Delta\)-models, the specified addition, modification and removal operations on modelling elements and relations have to be refined. The addition of a modelling element is refined to a set of addition operations for the elements and the relational tuples of the model obtained by refining the modelling element. The removal of a modelling element is refined to a set of remove operations for the modelling elements and relational tuples of the model resulting from refining the modelling element. The modification of a modelling element is refined to a set of modification operations for modelling elements and relational tuples obtained by refining the modelling element. Change operations for relations are removed during \(\Delta\)-refinement. The application condition
of a $\Delta$-model remains unchanged such that the variability structure is preserved. In the example in Section III, the component diagram $\Delta$-model (cf. Figure 4) is refined to a class diagram $\Delta$-model (cf. Figure 7) according to the following definition.

Definition 8 ($\Delta$-Refinement): The refinement of a $\Delta$-model $\Delta = (\varphi, Op)$ is defined by $\text{refine}(\Delta) = (\varphi, \text{refine}(Op))$ where $\text{refine}([o_{p_1}, \ldots, o_{p_n}]) = [\text{refine}(o_{p_1}), \ldots, \text{refine}(o_{p_n})]$ and

- $\text{refine}(\text{add } e) = \{\text{add } e’ \mid e’ \in E_c\} \cup \{\text{add } r(e_1’, e_2’) \mid r(e_1’, e_2’) \in R_e, e_1’, e_2’ \in E_c\}$
- $\text{refine}(\text{rem } e) = \{\text{rem } e’ \mid e’ \in E_c\} \cup \{\text{rem } r(e_1’, e_2’) \mid r(e_1’, e_2’) \in R_e, e_1’, e_2’ \in E_c\}$
- $\text{refine}(\text{mod } e) = \{\text{op } e’ \mid e’ \in E_c\} \cup \{\text{op } r(e_1’, e_2’) \mid r(e_1’, e_2’) \in R_e, e_1’, e_2’ \in E_c\}$ where $\text{op} \in \{\text{add}, \text{rem}, \text{mod}\}$
- $\text{refine}(\text{add } r(e_1, e_2)) = \text{refine}(\text{rem } r(e_1, e_2)) = \emptyset$

The configuration of a refined model for a feature configuration $f \in F$ is performed in three steps. First, the original model on the higher abstraction level is configured subject to the feature configuration $f$. Second, for every modelling element $e$ included in the resulting product model, the refined core model restricted to the modelling element $e$ is configured using the refined $\Delta$-models restricted to the modelling element $e$ subject to the feature configuration $f$. If a modelling element $e$ is not contained in the core model, but introduced by a $\Delta$-model, the refined core model restricted to the modelling element $e$ is an empty model. Third, the refined modelling elements replace their non-refined version in the configured original model. The result is a model that contains models as modelling elements and relations between these models. In the example in Section III, a configured, refined model is a component diagram in which the components contain class diagrams showing their detailed internal structure. The restriction of a core model $C = (E, R, f_e)$ to a modelling element $e \in E$ is defined by $C|_e = \{(e, \emptyset) \mid e \in E \text{ and } C|_e = (\emptyset, \emptyset), \text{ otherwise}\}$. Further, we define $\Delta|_e = \{\Delta|_1, \ldots, \Delta|_n\}$ as the set of $\Delta$-models only modifying element $e \in E$ where $\Delta|_e = (\varphi, [\text{op } e \in Op_i])$ for $\text{op} \in \{\text{add}, \text{rem}, \text{mod}\}$.

Definition 9 (Configuration of Refined Models): Let $C = (E, R, f_e)$ be a core model, $\Delta = \{\Delta|_1, \ldots, \Delta|_n\}$ a set of normalized and sorted $\Delta$-models and $f \in F$ a feature configuration. Let $P_f = (E_p, R_p, f) = \text{conf}((E, R), \{\Delta|_1, \ldots, \Delta|_n\}, f)$ be the configured original model for the feature configuration $f$. Further, let $M_e = \text{conf}(|\text{refine}(C|_e), o(\text{refine}(\Delta|_e))|, f)$ be the refined configured modelling element for $e \in E_p$.

The refined configured model is defined by $P_r = \text{conf}|_r(|\text{refine}(E, R), \text{refine}(\Delta), f)$ with $P_r = \{(M_e \mid e \in E_p), \{r(M_{e_1}, M_{e_2}) \mid r(e_1, e_2) \in R_p\}, f\}$

The commutativity of model refinement and model configuration by $\Delta$-application constitutes the basis for the incremental model-based development of software product lines by stepwise refinement of core and $\Delta$-models. The requirement for commutativity is that the refinement of the change operations specified in $\Delta$-models is compatible with model configuration. For addition and removal operations, compatibility is ensured by the definition of $\Delta$-model refinement in Definition 8. For the refinement of the modification operations, a local refinement compatibility condition has to be established. This local refinement compatibility condition requires that the result of applying the refined modification operation to the refined modelling element in core model is the same as the refinement of the modelling element that has been configured on the non-refined modelling level.

Definition 10 (Refinement Compatibility): Let $e’ \in E$ be the result of applying the modification operation $\text{mod } e$ to the modelling element $e \in E$. The local refinement compatibility condition for the operation $\text{mod } e$ holds iff $\text{apply}(\text{refine}(e), \text{refine}(\text{mod } e)) = \text{refine}(e’)$

The following theorem states that model refinement and model configuration commute if all modification operations satisfy refinement compatibility.

Theorem 1 (Commutativity): For a feature configuration $f \in F$, a core model $C = (E, R, f_e)$ and a set of well-defined $\Delta$-models $\Delta = \{\Delta_1, \ldots, \Delta_n\}$ and a refinement $\text{refine}$ on the modelling elements $e \in E$, if all modification operations $\text{mod } e \in Op_i$ satisfy the refinement compatibility condition, then it holds that $\text{refine}(|\text{conf}((E, R), \Delta, f)) = \text{conf}|_r(\text{refine}(E, R), \text{refine}(\Delta), f)$

Proof: By induction on the set of $\Delta$-models and a case distinction on their add, mod, rem operations. For add and rem operations, the definition of $\Delta$-refinement in Definition 8 is used. For mod operations, the refinement compatibility condition from Definition 10 is assumed.

Commutativity of model refinement and model configuration provides that basis for an incremental model-driven development process for software product lines. After the initial core and $\Delta$-models have been created to capture the variability of the feature model, this initial variability structure is preserved on each modelling level by the independent refinement of core and $\Delta$-models.

VI. CONCLUSION

$\Delta$-modelling is an variability modelling approach for model-driven development of software product lines. Product variability is expressed by core and $\Delta$-models on all modelling levels that are preserved under model refinement. For a concrete development process, the semantics of $\Delta$-application has to be defined for the language concepts used on each modelling level. Model configuration by $\Delta$-application can be automated, e.g., by aspect-oriented model weaving techniques [6], [7], [13] or model superimposition [14]. In [21], the $\Delta$-modelling approach has been implemented using frame technology.

For future work, we aim at providing tool support and guidelines how to develop initial core and $\Delta$-models for
a given feature model. This will be complemented by modular analyses establishing conflict-freedom and well-definedness of core and $\Delta$-models using existing variability analysis techniques based on confluence analysis. Besides, we want to extend $\Delta$-modelling with explicit conflict resolution by imposing a partial order between $\Delta$-models in order to avoid the normalization of $\Delta$-models during configuration.

Acknowledgments The author wants to thank Arnd Poetzsch-Heffter and Alexander Worret for initial collaboration on the subject of this work.

REFERENCES


Appendix O

A Programming Language for Software Product Lines

A Programming Language for Software Product Lines *

Lorenzo Bettini¹, Viviana Bono¹, Ferruccio Damiani¹, and Ina Schaefer²**

¹ Dipartimento di Informatica, Università di Torino, C.so Svizzera, 185 - 10149 Torino, Italy
{bettini,bono,damiani}@di.unito.it

² Chalmers University of Technology, 421 96 Gothenburg, Sweden
schaefer@chalmers.se

Abstract. A software product line is a set of software systems with well-defined commonalities and variabilities defined in terms of product features. In order to implement feature-based variability within the object-oriented programming paradigm, we present a novel linguistic approach based on the concept of program deltas. The implementation of a product line is divided into a core module and a set of delta modules. The core module comprises a set of classes that implement a valid product of the product line. Delta modules specify changes to the core module in order to incorporate specific product features. A product implementation is obtained by changing the core module according to the applicable delta modules. We propose a constraint-based type system for core and delta modules that provides static guarantees that the resulting product implementations are safe without having to generate all possible products. In order to show the feasibility of our approach, we use Featherweight Java for the implementation of product classes.

1 Introduction

A software product line (SPL) is a set of software systems with well-defined commonalities and variabilities [7, 20]. The variabilities of the products can be defined in terms of product features [11], which are important product characteristics. A feature model describes the set of possible products of the product line by the set of valid feature configurations, i.e., the set of features a product implements.

In order to represent feature-based variability in the implementation of product lines, two main approaches can be distinguished. First, syntactic approaches, e.g., [3, 12], mark the code of the product line syntactically with product features. A product for a specific feature configuration is obtained by modifying the code on a syntactic level allowing fine-grained modifications of code. Linguistic approaches, most prominently feature-oriented programming [5, 2, 8], associate features with programming language constructs. The code for a product is obtained by linguistic operations. In feature-oriented programming, feature modules are assembled to products by feature composition. While linguistic constructs associate features with more coarse-grained modifications of product code, they can provide static guarantees about the resulting products on the linguistic level, e.g. by type systems [2, 8].

In this paper, we present a novel linguistic approach to implementing feature-based variability of software product lines within the object-oriented programming paradigm.

* The authors of this work have been partially supported by the Ateneo Italo-Tedesco / Deutsch-Italienisches Hochschulzentrum.
** This author has been supported by the Deutsche Forschungsgemeinschaft (DFG).
We transfer the concept of delta modeling [21, 22], a general approach to represent feature-based variability during all phases of product line development, to the programming language level. The implementation of a software product line is divided into a core module and a set of delta modules. The core module comprises a set of classes that implement a valid product of the product line for a specific set of product features. Delta modules specify changes to the core module in order to implement other products. A delta module can add classes to a product implementation and remove classes from a product implementation. Furthermore, existing classes can be modified. Modification of a class comprises the change of the parent class, the change of the constructor, and additions, removals and renamings of fields and methods. A delta module contains an application condition determining for which feature configuration the specified modifications are to be carried out. In order to obtain a product implementation for a particular feature configuration, an incremental composition of the core module and the delta modules is performed by applying the modifications of all delta modules with valid application condition to the core module. In order to ensure that there are no conflicting modifications, the ordering of delta module application can be explicitly defined. Static guarantees that the implementations of the resulting products are safe are provided by a constraint-based type system that supports the analysis of the product line implementation without having to generate all products. The concept of core and delta modules is independent from a concrete programming language. In order to show the feasibility of our approach, we use FJ (FEATHERWEIGHT JAVA) [10] for the implementation of product classes.

The presented programming language combines the possibility to flexibly represent feature-based variability with the ability to effectively provide static guarantees on the resulting products. The core product is a complete product for any valid feature configuration. This allows developing it with well-established techniques from single application engineering. The choice of the core product is not fixed such that a different product line designs are supported within the same approach. The application conditions attached to delta modules do not necessarily refer to a single feature, but can refer to any combination of features. The implementations for combinations of features can be specified separately, thus increasing the expressiveness of code modifications induced by features and avoiding the optional feature problem [16]. The presented programming language supports the modular and evolutionary development of product lines. If additional features should be included in the products of the product line, new delta modules dealing with the effect of these features can be added to the implementation.

**Organization of the Paper.** In Section 2, we present the concepts of the proposed programming language. In Section 3, we demonstrate its flexibility at different implementations of the same product line. Section 4 introduces preliminary concepts for the formal calculus. In Section 5, we present the formal calculus and the semantics for product generation. Section 6 describes the constraint-based type system. In Section 7, we compare with presented language with related approaches. Section 8 summarizes the paper with an outlook to future work.

## 2 Implementing Feature-Based Variability with Delta Modules

As a running example to illustrate our approach, we use a SPL of bank accounts [8]. Figure 1 shows the feature model of the bank account SPL determining the different products by possible combinations of features. The mandatory Base feature represents the
Fig. 1. Feature Model for Bank Account Product Line

basic functionality of any bank account allowing to store the current balance and to update it. This functionality can be extended by the optional Sync(hronized) feature guaranteeing synchronized access to the account. The features Retirement and Investment that provide the possibility to store an additional bonus for the account are optional and mutually exclusive. The optional feature With Holder adds a reference to the holder of the account and requires the presence of either the Retirement or the Investment feature.

Core modules. In our approach, feature-based variability on the implementation level is captured by a core module and a set of delta modules. A core module corresponds to the implementation of a product for a valid feature configuration. It defines the starting point for generating all the other products by delta module application. The core module depends on the underlying programming language used to implement the products. In the context of this work, the core module contains a set of Java classes. To give a structure to the SPL implementation, we enclose this set of classes inside a core block and specify the features of the product implemented by the core module:

```java
core <Feature names> { <Java classes> }
```

As we will show in Section 3, the product represented by the core module can be any valid product. Listing 1 contains a core module for the bank account SPL. It implements the account with only the Base feature by the Account class. In this and the following examples, we will use the full Java syntax, while the calculus presented in Section 5.1 is based on FJ.

Delta modules. Delta modules specify changes to the core module in order to implement other products. The alterations inside a delta module act on a class level (by adding, removing and modifying a class) and on a class structure level (by modifying the internal structure of a class by changing the super class, by changing the constructor and by adding, removing and renaming fields and methods). An application condition is attached to every delta module in its when clause determining for which feature configurations the specified alterations are to be carried out. The application condition creates the link from the features in the feature model to the implementation. Application conditions are Boolean constraints over features. This allows specifying delta modules for combinations

```java
Listing 1: Core module implementing Base feature
```
of features and also to handle explicitly the absence of features which makes the implementation of features very flexible.

Listing 2 shows the delta module defining the changes of the Account in the core module to incorporate the Investment feature. First, the delta module changes the superclass of Account to WaMu (not shown here). It specifies to add a field 401balance. It redefines the method update by renaming the existing version to originalUpdate and by adding a new definition of the method update. Furthermore, a new method addBonus is added. The application condition in the when clause specifies that this \( \Delta \)-module is applied for all feature configurations in which the Investment feature is present.

**Delta application.** In order to obtain a product for a particular feature configuration, the changes specified by delta modules with valid application conditions are applied to the core (delta application). The alternations specified in one delta module are applied simultaneously. In order to ensure, for instance, that a class to be modified exists or that a modification of the same method by different delta modules does not cause a conflict, an ordering on the application of the delta modules can be defined by means of the after clause. This ordering implies that a delta module is only applied to the core module after all delta modules with a valid application condition mentioned in the after clause have been applied. With the after clauses, a partial ordering on the set of delta modules is defined that captures only the necessary dependencies. Note that specifying that a delta module \( \Delta \) has to be applied after the delta module \( \Delta \) does not mean that \( \Delta \) requires \( \Delta \); it only specifies that if a feature configuration satisfies the when clause of both \( \Delta \) and \( \Delta \), then \( \Delta \) must be applied before \( \Delta \) (this is illustrated for the feature configuration with both Base and Sync, later in this section).

The delta module implementing the Sync feature of the Bank Account SPL is presented in Listing 3. It specifies to change the class Account by adding a lock field (whose class Lock is not shown here) and by wrapping the code for synchronization around the method update. The original method update is renamed into unsync_update, and a method update is introduced which calls unsync_update in a synchronized way (locking before the call, and unlocking afterwards). Note that this delta module must be applied after the delta module for the Investment or the Retirement feature (if any of them has a valid application condition), since the latter modify the update method themselves. With the after clause, we ensure that the synchronization takes place on the correct version of the update method.

The generation of a product for a given feature configuration consists of the following steps, performed automatically by the system:

```markdown
delta Dinvestment when Investment {
  modifies class Account extending WaMu {
    adds int 401balance;
    renames update to originalUpdate;
    adds void addBonus (int x) { 401balance += x;}
    adds void update (int x) { x = x/2; originalUpdate(x); addBonus(x);}
  }
}
```

**Listing 2:** Delta module for Investment feature
delta DsyncUpdate after Dretirement, Dinvestment when Sync {
  modifies class Account {
    adds Lock lock;
    renames update to unsync_update;
    adds void update(int x) { lock.lock(); unsync_update(x); lock.unlock(); } 
  }
}

Listing 3: Delta module for Sync feature

class Account extends WaMu {
  int balance;
  int 401balance;
  void original_update(int x) { balance += x; }
  void update(int x) { x = x/2; original_update(x); addBonus(x); }
  void addBonus(int x) { 401balance += x; }
}

Listing 4: Account with Base and Investment features

1. Find all delta modules with a valid application condition according to the feature configuration (specified in the when clause); and
2. Apply the selected delta modules to the core module in any linear ordering respecting the partial order induced by the after clauses.

As an example of a product implementation resulting from delta application, the implementation of an account with the Base and Investment features is shown in Listing 4. It is the result of applying the delta module Dinvestment in Listing 2 to the core module in Listing 1. As stated earlier, the delta module for the Sync feature in Listing 3 must come after the delta module for the Investment feature, in case the feature configuration contains both features, which is not the case here. Indeed in Listing 5, the result of applying the delta module DsyncUpdate to the core module is depicted, representing the product for the feature configuration Base and Sync.

Note that the automatic generation of products by delta application is only performed if the implementation of the SPL is well-formed (a notion which is formally defined in Section 5). Well-formedness requires that all delta modules associated to a valid feature configuration must be applicable to the core module in any order compatible with the partial order provided by the after clauses, which implies, for instance, that all renamed or modified classes, methods and fields exists. Furthermore, the delta modules non comparable with respect to the after partial order must be compatible, i.e., not causing a

class Account extends Object {
  int balance;
  Lock lock;
  void unsync_update(int x) { balance += x; }
  void update(int x) { lock.lock(); unsync_update(x); lock.unlock(); }
}

Listing 5: Account with Base and Sync features
delta Dretirement when Retirement {
    modifies class Account extending Lehman {
        adds int 401balance;
        removes int balance;
        removes void update(int x);
        adds void update (int x) { addBonus(x) ;}
        adds void addBonus (int x) { 401balance += x ;}
    }
}

Listing 6: Delta module for Retirement feature

class conflict. This means that all potential conflicts between modifications targeting the same class, method or field are resolved by the ordering specified with the after clauses.

3 Implementing Software Product Lines

In this section, we show how the programming language constructs presented in the previous section can be used to implement the bank account SPL. We show the flexibility of the introduced concepts to support different product line implementations starting from different core products. Furthermore, we illustrate the design freedom for the creation of delta modules.

Starting from a Simple Core. The core module has to contain an implementation of a valid product of the product line. One possibility is to take only the mandatory features and a minimal number of required alternative features, if applicable. In our example, the Base feature is the only mandatory feature. Listing 1 shows the respective core module if the SPL implementation is started with a product only containing the Base feature.

In order to represent all possible products of the product line, delta modules have to be defined that modify the core product to incorporate further product features. The delta modules for the Investment and the Sync feature are already shown in Listings 2 and 3. In addition to that, we need a delta module for the Retirement feature shown in Listing 6 (the class Lehman serving as new super class is not shown here). The features Investment and Retirement are optional and mutually exclusive (see Figure 1) which is not expressed directly in the when clauses of their delta modules (Listing 2 and 6, respectively). Because only valid feature configurations according to the constraints of the feature model are used for delta application, this will be ensured at delta application level.

Both delta modules act on the method update. By the after clause of the delta module DsyncUpdate in Listing 3, it is ensured that the correct version of the update method is synchronized. Moreover, if we want to have the Sync feature in presence of the Retirement or Investment features, we need to synchronize also the addBonus method. Thus, we have to implement the additional delta module DsyncBonus shown in Listing 7, which must be applied after the delta modules Dretirement and Dinvestment. This example shows that code required to connect the behavior of two optional features can be introduced by an additional delta module solving the optional feature problem [16]. In Listing 8 we show the Account with Base, Sync and Investment. This is the result of applying first delta Dinvestment, then DsyncUpdate and DsyncBonus.\footnote{Note that our example relies on the fact that if the same thread calls lock() on the same Lock instance twice it will not deadlock.}

\footnote{Note that our example relies on the fact that if the same thread calls lock() on the same Lock instance twice it will not deadlock.}
\texttt{delta DsyncBonus after Dretirement, Dinvestment when Sync \&\& (Retirement || Investment) \{ \\
modifies class Account \{ \\
\textbf{renames} addBonus to unsync\textunderscore addBonus; \\
\textbf{adds} void addBonus(int x) \{ lock.lock(); unsync\textunderscore addBonus(x); lock.unlock(); \} \\
\} \\
\}}

Listing 7: Delta module for Sync feature in presence of Retirement or Investment feature

class Account extends WaMu { 
  int balance;
  int 401balance;
  Lock lock;
  \textbf{void} original\textunderscore update(int x) \{ balance += x; \}
  \textbf{void} unsync\textunderscore update(int x) \{ x = x/2; original\textunderscore update(x); addBonus(x); \}
  \textbf{void} unsync\textunderscore addBonus(int x) \{ 401balance += x; \}
  \textbf{void} update(int x) \{ lock.lock(); unsync\textunderscore update(x); lock.unlock(); \}
  \textbf{void} addBonus(int x) \{ lock.lock(); unsync\textunderscore addBonus(x); lock.unlock(); \}
}

Listing 8: Account with Base, Sync and Investment features

Finally, the delta module for the With Holder feature, shown in Listing 9, adds a class Client with an Account field and a method acting on it to a product implementation. Note that the payday method relies on the fact that Account provides the method addBonus. This is ensured by the feature model that specifies that the With Holder feature requires the Retirement or the Investment feature such that the addBonus method is provided by delta application.

The delta modules for the Retirement and Investment features have some similarities: they both add the field 401balance and the addBonus method. Thus, an alternative implementation for these delta modules is to write a delta module DaddBonus comprising the common code, and to rewrite the delta modules for Retirement and Investment as it is shown in Listing 10. It is straightforward to check that the resulting products, e.g., the one depicted in Listing 8, do not change. The additional delta module allows reusing code, but increases the total number of delta modules. Depending on the size of the feature model, this is a tradeoff a programmer can take into consideration, thanks to the flexibility offered for the design of delta modules.

\textbf{Starting from a Complex Core.} The bank account SPL can also be implemented by starting with a core product containing, besides the mandatory Base feature, also an optional feature, like Sync. This is illustrated in Listing 11. Thus, we need the delta module DunSyncUpdate, which removes the synchronization functionalities in order to generate

delta DwithHolder when WithHolder { 
  \textbf{adds} class Client \{ 
    Account a;
    \textbf{void} payday(int x, int bonus) \{ a.addBonus(bonus); a.update(x); \}
  \}
}

Listing 9: Delta module for With Holder feature
delta DaddBonus when (Retirement || Investment) {
  modifies class Account {
    adds int 401balance;
    adds void addBonus(int x) { 401balance += x; }
  }
}

delta Dretirement after DaddBonus when Retirement {
  modifies class Account extending Lehman {
    removes balance;
    removes update;
    adds void update(int x) { addBonus(x); }
  }
}

delta Dinvestment after DaddBonus when Investment {
  modifies class Account extending WaMu {
    renames update to original_update;
    adds void update(int x) { x = x/2; original_update(x); addBonus(x); }
  }
}

Listing 10: Alternative implementation with delta module DaddBonus

a product with only the Base feature. The renaming of method unsync_update to update
and the removal of the update method do not generate problems, since the modification
operations are applied simultaneously.

In order to be able to generate all possible products, the delta modules of the previous
implementation need some modifications. For instance, the delta module for the Invest-
ment feature must be changed to rename the method unsync_update instead of update.
The delta module for the Retirement feature has to be modified similarly. In the delta
module DsyncBonus of Listing 7, the after clause can simply be removed. Again, it is
straightforward to check that the resulting products are the same as the ones which are
generated by the previous core and delta modules.

4 Preliminary Concepts

FJ (F EATHERWEIGHT J AVA) [10] is a minimal calculus for Java, which focuses on a
few basic concepts: mutually recursive class definitions, inheritance, object creation, field
access, method invocation, method recursion through this, subtyping and type casts. The
minimal syntax, typing and semantics make the type safety proof simple and compact, in
such a way that FJ is a handy tool for studying the consequences of extensions with
respect to Java [19].

The abstract syntax of FJ constructs is given in Figure 2. Following [10], we use the
overline notation for possibly empty sequences. We write “e” as short for a possibly empty
sequence of expressions “e₁,...,eₙ” and “MD” as short for a possibly empty sequence of
method definitions “MD₁...MDₙ” (without commas). The empty sequence is denoted by •.
The length of a sequence ¯e is denoted by #(e). We abbreviate operations on sequences
of pairs in similar way, e.g., we write “¯C ¯f” as short for “C₁f₁,...,Cₙfₙ”, “¯C ¯f;” as short for
“C₁f₁,...,Cₙfₙ;” and this. ¯f = ¯f;” as short for “this.f₁ = f₁;...this.fₙ = fₙ;”.

Sequences of named elements (field, method or parameter names, field, method or
class definitions,...) are assumed to contain no duplicate names (that is, the names of the
elements of the sequence must be distinct).
```java
core Base, Sync {
    class Account extends Object {
        int balance;
        Lock lock;
        void unsync_update(int x) { balance += x; }
        void update(int x) { lock.lock(); unsync_update(x); lock.unlock(); }
    }
}
delta DunSyncUpdate after Dretirement, Dinvestment when !Sync {
    modifies class Account {
        removes Lock lock;
        renames unsync_update to update;
        removes update;
    }
}
delta Dinvestment when Investment {
    modifies class Account extending WaMu {
        adds int 401balance;
        renames unsync_update to originalUpdate;
        adds void addBonus (int x) { 401balance += x; }
        adds void unsync_update (int x) { x = x/2; originalUpdate(x); addBonus(x); }
    }
}
```

Listing 11: Alternative implementation starting from a different core.

The set of variables includes the special variable this (implicitly bound in any method declaration), which cannot be used as the name of a method’s formal parameter. This restriction is imposed by the FJ typing rules (see [10]). Note that no special syntax for this is required since, in method bodies, it is treated as an ordinary variable.

A class definition `class C extends D` consists of its name `C`, its superclass `D` (which must always be specified, even if it is `Object`), a list of field names `C f` with their types, a constructor `K` and a list of method definitions `MD`. The instance variables of `C` are added to the ones declared by `D` and its superclasses and are assumed to have distinct names.

The constructor definition `C(a f, b g) { super(f); this.g = g; }` specifies how to initialize the fields of an instance of `C`: there is one parameter for each field (including the inherited ones), with the same name of the field; its body consists of a call to the parent class constructor to initialize the inherited fields `f` with the parameters of the same name, followed by an assignment to the new fields `g` (defined in `C`) of the parameters of the same name. In FJ [10], the order of the constructor parameters `g` is determined by the order in which the corresponding fields are declared in the class `C`. In this paper, we use a slightly more liberal syntax (that will provide more flexibility in programming a SPL for FJ programs). Namely, the order of the constructor parameters `g` may be a permutation of
the order of the corresponding field definitions in C. For instance we admit the following
class definitions that would not be legal in the original formulation of FJ.

```java
class D extends Object {
    A a; B b; C c;
    D(B b, A a, C c) { super(); this.b=b; this.a=a; this.c=c; }
}
```

```java
class E extends D {
    F f; G g;
    E(B b, A a, C c, G g, F f) { super(b,a,c); this.g=g; this.f=f; }
}
```

A method definition MD specifies the name, the signature and the body of a method; a
body is a single return statement since FJ is a functional core of Java.

A class table CT is a mapping from class names to class definitions. The subtyping
relation <: on classes (types) is the reflexive and transitive closure of the extends relation
(the immediate subclass relation, given by the extends clauses in CT). The class Object
has no members and its definition does not appear in CT. We assume that a class table CT
satisfies the following sanity conditions: (i) CT(C) = class C ... for every C ∈ dom(CT)
(ii) for every class name C (except Object) appearing anywhere in CT, we have C ∈
dom(CT); (iii) there are no cycles in the transitive closure of the extends relation.

A FJ program is a class table CT (containing all the class definitions of the program). 4
A class definition CD can be understood as a mapping from the keyword extends to a
class name, from the keyword constructor to class constructor definition and from
field/method names to field/method definitions. We use the metavariable a to range over
field/method names, and the metavariable AD to range over field/method definitions. The lookup of the definition of the field/method a in class C is written aDef(C)(a). Formally,
for every class C in dom(CT), the function aDef(C) is defined as follows:

\[
aDef(C)(a) = \begin{cases}
    CT(C)(a) & \text{if } a ∈ dom(CT(C)) \\
    aDef(D)(a) & \text{if } a \notin dom(CT(C)) \text{ and } CT(C)(extends) = D
\end{cases}
\]

The type of a method is a pair, B → B, of a sequence of argument types B and a return
type B. Given a field definition FD = C f and a method definition MD = C m(C x){ · · · }, we
write signature(FD) to denote the type C of the field f and signature(MD) to denote the type
C → C of the method m. For a constructor definition K = C(λ f, B g){ super(f); this.g =
g; } we write header(K) to denote the constructor header C (C f).

5 The FAJ Calculus

In this section, we present FJA (FEATHERWEIGHT DELTA JAVA), a minimal calculus for
programming a product lines of FJ programs by defining a feature model, a core module
and a set of delta modules.

5.1 Syntax

The abstract syntax of FJA constructs is given in Figure 3. The constructs for class def-

4In [10], a program is a pair (CT, e) of a class table and an expression e (the program’s main
entry point). Such a notion of program can be encoded by adding to CT a distinguished class
class Main { C main() { return e; } }, where C is a suitable type for e.
FMD ::= features \( \varphi \) configurations \( \Pi \) feature model definitions

\( \Pi ::= \varphi \mid II \mid II \&\& II \mid II||II \mid II \rightarrow II \mid II \leftrightarrow II \) configuration propositions

CMD ::= core \( \varphi \{CD\} \) core module definitions

DMD ::= delta \( \delta \) after \( \delta \) when \( \Sigma \{DC\} \) delta module definitions

\( \Sigma ::= \varphi \mid \varphi \mid \Sigma \&\& \Sigma \mid \Sigma \| \Sigma \) application condition

DC ::= adds CD modifies C [extending C] \{ [constructor K] DSC \} delta clauses

DSC ::= adds FD | adds MD removes f | removes m renames f to f | renames m to m delta subclauses

Fig. 3. FΔJ: Syntax of feature model, core-module and delta-module definitions

| features | Base, Sync, Retirement, Investment, With_Holder |
| configurations | Base & & ! (Retirement & & Investment) & & (With_Holder –> (Retirement || Investment)) |

Listing 12: FΔJ code for the feature model in Figure 1

Figure 2. The metavariables \( \varphi, \chi \) and \( \psi \) range over feature names. We write \( \varphi \) as short for the set \( \{ \varphi \} \), i.e., the feature configuration containing the features \( \varphi \). A feature model definition FMD declares the features of a software product line, \( \varphi \), and a propositional formula, \( \Pi \), describing the set of valid feature configurations \( CONFIGURATIONS \subseteq P(\varphi) \). We assume that all feature names occurring in \( \Pi \) occur in \( \varphi \). For example, the FΔJ code for the feature model described by the feature diagram in Figure 1 is given in Listing 12.5

A core module definition CMD consists of the feature configuration \( \varphi \) implemented by the core product and the corresponding classes. A delta module definition DMD can be understood as a mapping from class names to delta-clause definitions, from the keyword after to a possibly empty set of delta module names and from the keyword when to an application condition. A delta clause definition DC can specify the addition, removal or modification of a class. The modification of a class is defined by potentially changing the super class and constructor and a sequence of delta subclauses DSC which can define additions, removals or renamings of fields and methods. The set of delta module names in the after clause are the delta modules that (when applicable) have to be applied before this delta module. The when clause determines for which feature configurations the delta module is applicable by a Boolean constraint over feature names. We write \( \text{DELTAS}(\varphi) \) to denote the set of names of delta modules whose application condition is satisfied by \( \varphi \in CONFIGURATIONS \). A delta module table DMT is a mapping from delta module names to delta module definitions.

A FΔJ SPL is a 4-tuple \( L = (\text{FMD, } \varphi, \text{CT}_\varphi, \text{DMT}) \) consisting of a feature model definition FMD, a feature configuration \( \varphi \) (corresponding to the product implemented by the core-module), a class table \( \text{CT}_\varphi \) (containing the class definitions of the core module) and a delta-module table \( \text{DMT} \) (containing the delta modules of the SPL). To simplify the notation, in the following we always assume a fixed SPL \( L \), that is, fixed feature model definition FMD, core feature configuration \( \varphi \), core class table \( \text{CT}_\varphi \) and delta-module table \( \text{DMT} \). We assume that:

5 The translation of a feature diagram into a propositional formula can be performed in a systematic way (see, e.g., [4]).
The features $\chi$ associated to the core module are defined in $FMD$.

The class table of the core product $CT_\chi$ satisfies the FJ sanity conditions.

The delta-module table $DMT$ satisfies the following sanity conditions: (i) for every $\delta \in \text{dom}(DMT)$, $\text{DMT}(\delta) = \text{delta } \delta \text{ after } \Pi \{\cdots\}$, where $\bar{\delta} \in \text{dom}(DMT)$, all feature names occurring in $\Pi$ are defined in FMD, and for every $C \in \text{dom}(\text{DMT}(\delta))$, $\text{DMT}(\delta)(C) \in \{\text{adds class } C \cdots, \text{modifies } C \cdots, \text{removes } C\}$; (ii) for every class-name $C$ (except $\text{Object}$) appearing in DMT, we have $C \in CT_\chi \cup (\bigcup_{\delta \in \text{dom}(DMT)} \text{dom}(\text{DMT}(\delta)))$; (iii) there are no cycles in the transitive closure of the after relation (given by the after clauses in the delta module table), thus, its reflexive and transitive closure is a partial order.

In the following, instead of $\text{DMT}(\delta) = \text{delta } \delta \cdots$, we write $\text{delta } \delta \cdots$; instead of $\text{dom}(\text{DMT}(\delta))$ we write $\text{dom}(\delta)$; and instead of $\text{DMT}(\delta)(C)$, $\text{DMT}(\delta)(\text{after})$ and $\text{DMT}(\delta)(\text{when})$ we write $\delta(C)$, $\delta(\text{after})$ and $\delta(\text{when})$, respectively.

The adds-domain, the modifies-domain and the remove-domain of a delta module definition $DMD$ are defined as follows:

\[
\begin{align*}
\text{addsDom}(DMD) &= \{C \mid DMD(C) = \text{adds class } C \cdots\} \\
\text{modifiesDom}(DMD) &= \{C \mid DMD(C) = \text{modifies } C \cdots\} \\
\text{removesDom}(DMD) &= \{C \mid DMD(C) = \text{removes } C\}
\end{align*}
\]

A delta modifies-clause $DC$ can be understood as a mapping from the keyword extending to an either empty or singleton set of class names, from the keyword constructor to an either empty or singleton set of class constructers and from field/method names to delta subclauses. The adds-domain, the rename-from-domain, the rename-to-domain and the remove-domain of a delta modifies-clause $DC$ are defined as follows:

\[
\begin{align*}
\text{addsDom}(DC) &= \{a \mid \text{DMD}(a) = \text{adds } a \cdots\} \\
\text{renamesFromDom}(DC) &= \{a \mid \text{DMD}(a) = \text{renames } a \cdots\} \\
\text{renamesToDom}(DC) &= \{a \mid \text{DMD}(a) = \text{renames } \cdots \text{ to } a\} \\
\text{removesDom}(DC) &= \{a \mid \text{DMD}(a) = \text{removes } a\}
\end{align*}
\]

### 5.2 Well-Formed FAJ SPL

In order to define the application of a set of delta modules to a core module for generating product implementations, we define the notion of well-formed FAJ SPL. The first three requirements for well-formedness deal with the structure of the core module and the delta modules. First, the core module has to represent a valid feature configuration. Second: there must not be a delta module that has to be applied for the core feature configuration; each delta module must be applied for at least one configuration; and for all configurations, the sets of delta modules that have to be applied must be distinct and contained in the delta module table. Third, the fields and methods added and renamed-to in every delta module must be disjoint. Thus, given the SPL $L = (FMD, \chi, CT_\chi, DMT)$, these requirements are formalized as follows:

**Requirement 1.** The feature configuration of the core product satisfies the configuration proposition in FMD (i.e., $\chi \in \text{CONFIGURATIONS}$).

**Requirement 2.** The mapping $\text{DELTAS} : \text{CONFIGURATIONS} \to \mathcal{P}(\text{dom}(DMT))$ is injective and such that $\text{DELTAS}(\chi) = \emptyset$ and $\bigcup_{\phi \in \text{CONFIGURATIONS}} \text{DELTAS}(\phi) = \text{dom}(DMT)$. 
Requirement 3. For every $\delta \in \text{dom}(\text{DMT})$ and for every $C \in \text{modifiesDom}(\text{DMT}(\delta))$, we have that $\text{addsDom}(\delta(C)) \cap \text{renamesToDom}(\delta(C)) = \emptyset$.

The fourth well-formedness requirement ensures that there are no conflicting modifications if a set of delta modules is applied to a core module. In order to check conflict-freedom, it suffices to consider only class namespace tables (an abstraction of class tables). A class namespace $\text{CNT}$ is a set of fields/methods names. A class namespace table $\text{CNT}$ is a mapping from class names to class namespaces. We denote the namespace of a class definition $\text{CD}$ by $\text{namespace}(\text{CD})$ and the class namespace table of the classes in the class table $\text{CT}$ by $\text{namespace}(\text{CT})$. We write $\text{namespace}(\cdot)$ as short for $\text{namespace}(\text{CT}(\cdot))$.

A conflict occurs if two (or more) delta modules are not compatible or if a delta module is not applicable to a class table. A set of delta modules is compatible if the sets of added, removed and modified classes are disjoint. A set of (compatible) delta-modules is applicable to a class namespace table $\text{CNT}$ if

- $\text{modifiesDom}(\delta) \cup \text{removesDom}(\delta) \subseteq \text{dom}(\text{CNT})$,
- $\text{addsDom}(\delta) \cap \text{dom}(\text{CNT}) = \emptyset$,
- for every $C \in \text{modifiesDom}(\delta)$ the delta-clause $DC = \delta(C)$ and the class namespace $\text{CNT} = \text{CNT}(C)$ are such that:
  - $(\text{addsDom}(DC) \cup \text{renamesToDom}(DC)) \cap \text{dom}(CS) = \emptyset$, and
  - $\text{removesDom}(DC) \cup \text{renamesFromDom}(DC) \subseteq \text{dom}(CS)$.

The class namespace table $\text{CNT}'$ obtained by applying an applicable delta module $\delta$ to the class namespace table $\text{CNT}$, denoted by $\text{APPLY}(\delta, \text{CNT})$, is defined as follows:

$$
\text{CNT}'(C) = \begin{cases} 
\text{CNT}(C) & \text{if } C \notin \text{dom}(\text{DMT}(\delta)) \\
\text{namespace}(\text{CD}) & \text{if } C \in \text{addsDom}(\delta) \text{ and } \delta(C) = \text{adds CD} \\
\text{APPLY}(\delta(C), \text{CNT}(C)) & \text{if } C \in \text{modifiesDom}(\delta) 
\end{cases}
$$

where, for every $C \in \text{modifiesDom}(\delta)$, the application of the delta-clause $DC = \delta(C) = \text{modifies } C \cdot \{ \cdots \}$ to the class namespace $\text{CN} = \text{CNT}(C)$, denoted by $\text{APPLY}(DC, \text{CN})$, is the class namespace

$$
\text{APPLY}(DC, \text{CN}) = (\text{dom}(\text{CN}) - (\text{removesDom}(DC) \cup \text{renamesFromDom}(DC))) \\
\cup \text{addsDom}(DC) \cup \text{renamesToDom}(DC)
$$

Note that, if the delta-module $\delta$ is applicable to the class namespace table $\text{CNT}$, then

$$
\text{dom}(\text{APPLY}(\delta, \text{CNT})) = (\text{dom}(\text{CNT}) - (\text{removesDom}(\delta) \cup \text{renamesFromDom}(\delta))) \\
\cup \text{addsDom}(\delta) \cup \text{renamesToDom}(\delta)
$$

Given a sequence of delta-modules $\overline{\delta} = \delta_1 \cdots \delta_n$ ($n \geq 0$) and a class namespace table $\text{CNT}$ we will write $\text{APPLY}(\delta, \text{CNT})$ as short for $\text{APPLY}(\delta_1, \text{APPLY}(\cdots, \text{APPLY}(\delta_n, \text{CNT}) \cdots))$. We write $\text{CNT}_\varphi$ to denote the class namespace table obtained by application of $\text{DELTAS}(\varphi)$ to the core module for the feature configuration $\varphi$. 
Proposition 1. For every class table $CT$, if the delta-modules $\delta_1, \cdots, \delta_n$ ($n \geq 0$) are compatible and applicable to the class namespace table $CNT$, then $\text{APPLY}(\delta_1, \cdots, \delta_n, CNT) = \text{APPLY}(\delta_1, \cdots, \delta_n, \text{namespace}(CT)),$ for every permutation $i_1, \ldots, i_n$ of $1, \ldots, n$.

Conflicting modifications in a set of delta modules should not arise if the delta modules are applied in an order respecting the after partial order (that is, the partial order induced by the after clauses). The after partial order can be represented by a direct acyclic graph, that we will call the after-DAG, such that there is an arrow from $\delta_1$ to $\delta_2$ if and only if delta $\delta_1$ after $\delta_2$. The after-level of a delta-module $\delta$, denoted by $\text{afterLevelOf}(\delta)$, is the length of the longest path in the after-DAG ending in $\delta$. A set of delta-modules $\text{DELTA}(\Psi)$ for feature configuration $\Psi$ is conflict-free if the partition of $\text{DELTA}(\Psi)$ into the sets of the delta-modules with the same after-level, $\{\delta^{(1)}, \ldots, \delta^{(k)}\}$ (where $k \geq 0$ and the sets are listed in increasing order with respect to the after-level) is such that, for all $i \in \{1, \ldots, k\}$, the delta-modules in $\delta^{(i)}$ are compatible, and applicable to the class namespace table $\text{APPLY}(\delta^{(i)} \cdots \delta^{(k)}, \text{namespace}(CT))$, where $\text{namespace}(CT) = \text{namespace}(\text{CT}_\Psi)$ is the class namespace table of the core product.

Requirement 4. For every feature configuration $\Psi$ in CONFIGURATIONS, the set of delta modules $\text{DELTA}(\Psi)$ is conflict-free.

Definition 1 (Well-formed SPL). A FJ SPL is well-formed iff it satisfies the Requirements 1, 2, 3 and 4 above.

5.3 Generating FJ Programs from a Well-Formed FJ SPL

Given a delta-module $\delta$ and a class table $CT$ such that $\delta$ is applicable to $\text{namespace}(CT)$, the application of $\delta$ to $CT$, denoted by $\text{APPLY}(\delta, CT)$, is the class table $CT'$ defined as follows:

$$CT'(C) = \begin{cases} 
CT(C) & \text{if } C \notin \text{dom}(\text{DMT}(\delta)) \\
\text{CD} & \text{if } C \in \text{addsDom}(\delta) \text{ and } \delta(C) = \text{adds CD} \\
\text{APPLY}(\delta(C), CT(C)) & \text{if } C \in \text{modifiesDom}(\delta)
\end{cases}$$

where, for every $C \in \text{modifiesDom}(\delta)$, the application of the delta-clause $DC = \delta(C) = \text{modifies } C \cdots \{\cdots\}$ to the class definition $CD = CT(C)$, denoted by $\text{APPLY}(DC, CD)$, is the class definition $CD'$ defined as follows:

$$CD'(\text{extends}) = \begin{cases} 
\text{CD(extends)} & \text{if } DC(\text{extending}) = \emptyset \\
C' & \text{if } DC(\text{extending}) = \{C'\}
\end{cases}$$

$$CD'(\text{constructor}) = \begin{cases} 
\text{CD(constructor)} & \text{if } DC(\text{constructor}) = \emptyset \\
K & \text{if } DC(\text{constructor}) = \{K\}
\end{cases}$$

$$CD'(a) = \begin{cases} 
\text{AD} & \text{if } DC(a) = \text{adds AD} \\
\text{CD(a')} & \text{if } DC(a) = \text{renames a' to a}
\end{cases}$$

Note that, for every class table $CT$, if the delta-module $\delta$ is applicable to $\text{namespace}(CT)$, then it holds that $\text{dom}(\text{APPLY}(\delta, CT)) = \text{dom}(\text{APPLY}(\delta, \text{namespace}(CT)))$, and for every $C \in \text{dom}(\text{APPLY}(\delta, CT))$, $\text{namespace}(\text{APPLY}(\delta, CT)(C)) = \text{APPLY}(\delta, \text{namespace}(CT))(C)$.
Proposition 2. For every class table $CT$, if the delta-modules $\delta_1, \ldots, \delta_n$ ($n \geq 0$) are compatible and applicable to $\text{namespace}(CT)$, then $\text{APPLY}(\delta_1 \cdots \delta_n, CT) = \text{APPLY}(\delta_1 \cdots \delta_n, CT)$, for every permutation $i_1, \ldots, i_n$ of $1, \ldots, n$.

Thus, a well-formed FAJ SPL defines a mapping from each feature configuration $\varphi$ in $\text{CONFIGURATIONS}$ to the class table obtained by applying the delta modules $\text{DELTAS}(\varphi)$ to the class table of the core module according to the after partial order. We write $CT_\varphi$ to denote the class table generated for the feature configuration $\varphi$ and write $<_\varphi$ and $\text{aDef}_\varphi$ to denote the subtype relation and the field/method lookup function associated to the class table $CT_\varphi$, respectively (see Section 4). Note that, for every well-formed FAJ SPL and for every feature configuration $\varphi$ in $\text{CONFIGURATIONS}$, it holds that $\text{namespace}(CT_\varphi) = \text{CNT}_\varphi$.

5.4 Type-Safe, Redundant-Def-Free and Type-Uniform Well-Formed FAJ SPL

A SPL is type-safe if all its products are well-typed programs. Formally, the notion of type-safe FAJ SPL is as follows (we refer to [10] for the notion of well-typed FJ programs).

Definition 2. A well-formed FAJ SPL is type-safe iff for every $\{\varphi\} \in \text{CONFIGURATIONS}$, the class table $CT_\varphi$ represents a well-typed FJ program.

Type-safety of a FAJ SPL can be checked by generating and typechecking each product according the FJ type system. In Section 6, we present a technique for checking whether a FAJ SPL is type-safety without having to generate all products.

A FAJ SPL $L$ is redundant-def-free if it does not contain unused parts. This means that every class definition $C\varphi$, extending clause, constructor definition and field/method definition contained in any delta module of $L$ is used in at least one product. For example, both the variants of the bank account SPL illustrated in Section 3 are redundant-def-free. This property of a FAJ SPL can be checked only using the class namespace tables (by including the extending and constructor keywords in the class namespaces) without generating the products. Formally, the notion of redundant-def-free FAJ SPL is defined as follows.

Definition 3. A well-formed FAJ SPL is redundant-def-free iff for every delta-module name $\delta$ and every class name $C$

- $C \in \text{addsDom}(\delta)$ implies that
  - $CT_\varphi(C)(\text{extends}) = \delta(C)(\text{extends})$ (for some $\varphi \in \text{CONFIGURATIONS}$),
  - $CT_\varphi(C)(\text{constructor}) = \delta(C)(\text{constructor})$ (for some $\varphi \in \text{CONFIGURATIONS}$)
  - for all $a \in \text{dom}(\delta(C))$ there exists $\varphi \in \text{CONFIGURATIONS}$ such that $CT_\varphi(C)(a) = \delta(C)(a)$.

- $C \in \text{modifiesDom}(\delta)$ implies that
  - if $\delta(C)(\text{extending}) \neq \emptyset$, then $\{CT_\varphi(C)(\text{extends})\} = \delta(C)(\text{extends})$ (for some $\varphi \in \text{CONFIGURATIONS}$),
  - if $\delta(C)(\text{constructor}) \neq \emptyset$, then $\{CT_\varphi(C)(\text{constructor})\} = \delta(C)(\text{constructor})$ (for some $\varphi \in \text{CONFIGURATIONS}$)
  - for all $a \in \text{addsDom}(\delta(C))$ there exists $\varphi \in \text{CONFIGURATIONS}$ such that $CT_\varphi(C)(a) = \delta(C)(a)$. 
We say that a SPL is type-uniform to mean that (i) if there are two (or more) products containing a class of name \( C \) with a field/method of name \( a \), then the type of \( a \) must be the same; and (ii) if a class \( C \) is a subtype of a class \( D \) in some product, then there are no products where a class \( D \) is a subtype of a class \( C \). Type-uniformity enforces a communality in the family of the products that helps the design of the SPL. For example, both bank account SPLs shown in Section 3 are type-uniform. However, if a class \( \text{Investor} \) extending the class \( \text{Client} \) was added to the product with the Retirement and With Holder features, and the product with the Investment and With Holder features was modified by making the class \( \text{Client} \) a subclass of the class \( \text{Investor} \), this would break type-uniformity. Formally, the notion of type-uniform \( \text{F} \Delta \text{J} \) SPL is defined as follows.

**Definition 4.** A well-formed \( \text{F} \Delta \text{J} \) SPL is type-uniform iff, for every pair of products \( \text{CT}_{\Psi} \) and \( \text{CT}_{\Psi'} \) it holds that:

- for all \( C \in \text{dom}(\text{CT}_{\Psi}) \cap \text{dom}(\text{CT}_{\Psi'}) \) and for all field/method \( a \), if both \( \text{aDef}_{\Psi}(C, a) \) and \( \text{aDef}_{\Psi'}(C, a) \) are defined, then \( \text{signature}(\text{aDef}_{\Psi}(C, a)) = \text{signature}(\text{aDef}_{\Psi'}(C, a)) \), and
- for all \( C, C' \in \text{dom}(\text{CT}_{\Psi}) \cap \text{dom}(\text{CT}_{\Psi'}) \), if \( C \neq C' \) and \( C <_{\Psi} C' \) then \( C' <_{\Psi'} C \).

The type-uniformity of a well-formed \( \text{F} \Delta \text{J} \) SPL can be checked, without generating the products, by relying on the notions of class signature and class signature table. A class signature \( \text{CS} \) is a mapping from the keyword \( \text{extends} \) to a class name, from the keyword constructor to a class constructor header, from field names to class names and from method names to method types. A class signature table \( \text{CST} \) is a mapping from class names to class signatures. The signature of a class definition \( \text{CD} \) is denoted by \( \text{signature}(\text{CD}) \). The class signature table of the signatures of the classes in the class table \( \text{CT} \) is denoted by \( \text{signature}(\text{CT}) \). We write \( \text{signature}(C) \) as short of \( \text{signature}(\text{CT}(C)) \).

The class signature tables of the products of a well-formed \( \text{F} \Delta \text{J} \) SPL are generated (without generating the products) by applying the transformations described by the delta modules \( \text{DELTA}(\Psi) \) to the class signature table of the core module according to the after partial order, for each feature configuration \( \Psi \) in \( \text{CONFIGURATIONS} \). The formal definition for the application of a delta module to a class signature table can be straightforwardly obtained by mimicking the definition for the application of a delta module to a class table. We write \( \text{CST}_{\Psi} \) to denote the class signature table \( \text{signature}(\text{CT}_{\Psi}) \) of the product for configuration \( \Psi \). The subtyping relation \( <_{\Psi} \) can be read off from the class signature table \( \text{CST}_{\Psi} \) (so that it is possible to check whether there are no cycles in the \( \text{extends} \) relation). The lookup of the type of the field/method \( a \) in class signature in the class \( C \) is denoted by \( a\text{Type}(C)(a) \). Formally, for every class \( C \) in \( \text{dom}(\text{CST}) \), the function \( a\text{Type}(C) \) is defined as follows:

\[
a\text{Type}(C)(a) = \begin{cases} 
\text{CST}(C)(a) & \text{if } a \in \text{dom}(\text{CST}(C)) \\
\text{aType}(D)(a) & \text{if } a \not\in \text{dom}(\text{CST}(C)) \text{ and } \text{CST}(C)(\text{extends}) = D
\end{cases}
\]

### 6 A Type System for \( \text{F} \Delta \text{J} \)

In this section, we present how to check that a type-uniform \( \text{F} \Delta \text{J} \) SPL \( L \) is type safe without generating all products. We present a constraint-based type system for typechecking the core module with respect to the aggregate class signature table \( \text{ACST}_L \) of \( L \) (capturing the signature information for all products of \( L \)) in order to infer a set of class constraints...
and for typechecking every delta module $\delta$ with respect to $\text{ACST}_L$ in order to infer a set of delta clause constraints $\forall_\delta$ that represent a function from a set of class constraints to a set of class constraints. We also present a constraint verification system for checking the set of class constraints $\forall_\phi$ (obtained by applying the sets of delta clause constraints inferred for the delta modules in $\text{DELTAS}(\phi)$ to the set of class constraints $\forall$) against the class signature table $\text{CS}_\phi$.

### 6.1 Aggregate Class Signature Tables

The class signature tables of all the products of a well-formed FJ SPL can be generated without generating the products (see the explanation at the end of Section 5.4). By inspecting the class signature table $\text{CS}_\phi$ it is possible to check, for every class $C$ in $\text{dom}(\text{CS}_\phi)$, whether the names of the fields defined in $C$ are distinct from the names of the fields inherited from its superclasses and whether the formal parameters of the constructor of $C$ correspond to the (defined or inherited) fields of $C$ (as required by FJ, see Section 4). In the following we assume that the class signature tables of all the products satisfy these conditions.

An aggregate class signature $\text{ACS}$ is a mapping from the keyword $\text{extends}$ to a finite non-empty set of class names, from the keyword constructor to a finite non-empty set of constructor headers, from field names to class names and from method names to method types. An aggregate class signature table $\text{ACST}$ is a mapping from class names to aggregate class signatures such that the subtyping relation $\ll_{\text{ACST}}$ induced by the $\text{extends}$ clauses in $\text{ACST}$ is acyclic. The aggregation of the class signature tables of all the products of a type-uniform SPL $L$, denoted by $\text{ACST}_L$, is the aggregate class signature defined as follows.

$$\text{ACST}_L(C) = \text{aggregate}(\{\text{CS}_\phi(C) | \phi \in \text{CONFIGURATIONS} \text{ and } C \in \text{dom}(\text{CS}_\phi)\})$$

where the aggregation $\text{aggregate}(\{\text{CS}_1, \ldots, \text{CS}_k\})$ of a set of class signatures $\{\text{CS}_1, \ldots, \text{CS}_k\}$ ($k \geq 1$), is the aggregate class signature $\text{CS}$ defined as follows:

$$\text{CS}(\text{extends}) = \bigcup_{i \in \{1, \ldots, k\}} \{\text{CS}_i(\text{extends})\}$$

$$\text{CS}(\text{constructor}) = \bigcup_{i \in \{1, \ldots, k\}} \{\text{CS}_i(\text{constructor})\}$$

$$\text{CS}(a) = \text{CS}_b(a) \text{ for any } h \in \{j | j \in \{1, \ldots, k\} \text{ and } a \in \text{dom}(\text{CS}_j)\}$$

The type-uniformity of the SPL $L$ ensures that $\text{ACST}_L$ is an aggregate class signature table. Note that $\text{dom}(\text{ACST}_L) = \bigcup_{\phi \in \text{CONFIGURATIONS}} \text{dom}(\text{CT}_\phi)$ and $\text{dom}(\text{aggregate}(\{\text{CS}_1, \ldots, \text{CS}_k\})) = \bigcup_{i \in \{1, \ldots, k\}} \text{dom}(\text{CS}_i)$.

We write $\text{aType}_L(C)(a)$ to denote the type of the field/method $a$ in class $C$ according to $\text{ACST}_L$. Formally, for every class $C \in \text{dom}(\text{ACST}_L)$, the lookup function $\text{aType}_L(C)$ is defined as follows:

$$\text{aType}_L(C)(a) = \begin{cases} 
\text{ACST}_L(C)(a) & \text{if } a \in \text{dom}(\text{ACST}_L(C)) \\
A & \text{if } a \notin \text{dom}(\text{ACST}_L(C)) \\
\text{aType}_L(D)(a) & \text{for some } D \in \text{ACST}_L(C)(\text{extends}) 
\end{cases}$$

### 6.2 Basic Constraints and Basic Constraint Checking Rules

The syntax of basic constraints, together with an informal explanation of their meaning, is given in Figure 4. A constraint of the form $\text{hasPCP}(C, \bar{f})$ is used to ensure that
Constraint checking:

\[ \begin{align*}
\text{hasField} \left( C, f \right) & \quad \text{class } C \text{ must define or inherit field } f \\
\text{hasMethod} \left( C, m \right) & \quad \text{class } C \text{ must define or inherit method } m \\
\text{subtype} \left( C, D \right) & \quad \text{class } C \text{ must be a subtype of } D \\
\text{hasCAT} \left( C, C_1, \ldots, C_n \right) & \quad \text{class } C \text{ must have a Constructor accepting } n \text{ Arguments of Type } C_1, \ldots, C_n \\
\text{cast} \left( C, D \right) & \quad \text{type } D \text{ must be castable to type } C \\
\text{hasPCP} \left( C, \bar{f} \right) & \quad \text{the Parent class of } C \text{ must have a Constructor with Parameters } \bar{f}
\end{align*} \]

**Fig. 4.** FADJ: Syntax and (informal) meaning of basic constraints

\[
\begin{align*}
\text{hasField} \left( C, f \right) & \quad \text{f} \in \text{dom}(aType_{\varphi}(C)) \\
\text{hasMethod} \left( C, m \right) & \quad m \in \text{dom}(aType_{\varphi}(C)) \\
\text{subtype} \left( C, D \right) & \quad C \prec_{\varphi} D \\
\text{hasCAT} \left( C, C_1, \ldots, C_n \right) & \quad \text{CST}_{\varphi}(\text{constructor}) = C[D; f_1, \ldots, f_n] \\
\text{cast} \left( C, D \right) & \quad \text{CST}_{\varphi}(\text{extends}) = D \\
\text{CST}_{\varphi}(\text{constructor}) & \quad \text{CST}_{\varphi}(\text{extends}) = C[\bar{f}] \quad \text{(for some } \bar{f})
\end{align*}
\]

**Fig. 5.** FADJ: Checking rules for satisfaction of basic constraints

The parameters \( \bar{f} \) that the constructor of the class \( C \) passes to the super class constructor (through the call `super(\( \bar{f} \))`) correspond to the parameters of the constructor of the parent class. According to the explanation at the beginning of Section 6.1, we assume that for every class \( C \) in \( \text{dom}(\text{CST}_{\varphi}) \): (i) the names of the fields defined in each class are distinct from the names of the fields inherited from its superclass, and (ii) the formal parameters of the constructor of \( C \) correspond to the (defined or inherited) fields of \( C \). Therefore, no constraints are needed to ensure conditions (i) and (ii).

The rules for checking the satisfaction of a set of basic constraints with respect to the class signature table of a product \( \text{CST}_{\varphi} \) are given in Figure 5. Most of the rules are self explanatory. Note that, as in the type system for FJ (see [10]), there are three rules for type casts, corresponding to `upcast` (when the subject is a subtype of the target), `downcast` (when the target is a subtype of the subject) and `stupid cast` (when subject and target are unrelated), respectively. We say that a set of basic constraints \( \Phi \) is `cast safe` with respect to a class signature table \( \text{CST} \) to mean that the judgement \( \text{CST}_{\varphi} \models \Phi \) has been established without downcasts or stupid casts (uses of rules with a `downcast` or `stupid cast` premise).

### 6.3 Typing Rules for the Core Module and their Correctness/Completeness

In order to support the application of a set of constraints \( \Phi \) inferred for a delta module to the set of constraints \( \Phi' \) inferred for the core module (thus making it possible to generate the constraints for a product without having to generate the product itself), the typing rules for the core module organize the inferred constraints in a two level hierarchy, corresponding to the structure of the class signature table of the core module. Namely: (i) the typing rules infer a set of `class constraints` (one for each class definition in the core module); (ii) each class constraint consists of the name of the subject class \( C \) and of a set containing a basic constraint of the form \( \text{hasPCP}(C, \ldots) \) (inferred for the constructor of the class) and `method constraints` inferred for the methods defined in the class; and (iii) each method constraint consists of the name of the subject method and of the set of basic constraints.
Method constraints:

\[ m \text{ with } B \text{ method } m \text{ enforces the basic constraints } B \]

(where \( B \) is a set of basic constraints not containing \( \text{hasPCP}(\cdots, \cdots) \))

Class constraints:

\[ C \text{ with } K \text{ class } C \text{ enforces the constraints } K \]

(where \( K \) is a set containing a constraint \( \text{hasPCP}(C, \cdots) \)

and some method constraints)

Fig. 6. FAJ: Syntax of class constraints

inferred for the body of the subject method. Thus, a set of class constraints \( C \) can be
understood as a function from class names to class-constraints, and a class-constraint can
be understood as a function from method names to method-constraints and from the key-
word constructor to a basic constraint of the form \( \text{hasPCP}(C, \cdots) \). The syntax of class
constraints and method constraints is given in Figure 6. The constraint-based typing rules
for FJ expressions, methods and classes and for the core module are given in Figure 7.

The hierarchical organization on a per-class and per-method basis of the constraints
associated to a product is immaterial for the purpose of checking constraint satisfaction.
The flattening function \( \text{FLAT} \), defined below, transforms a set of class-constraints
into a set of basic constraints \( \text{FLAT}(C) \).

\[
\text{FLAT}\left( \bigcup_{i \in \{1, \ldots, n\}} \{ C_i \text{ with } K_i \} \right) = \bigcup_{i \in \{1, \ldots, n\}} \text{FLAT}(K_i)
\]

We could use the FJ typing rules (see [10]) to check whether the core module is a
well-typed FJ program. However (as we will see in Section 6.4), the constraint-based
typing rules infer a set of \( \text{delta removes/adds/modifies-clause constraints} \) (one for each delta clause in the delta module);

Theorem 1 (Correctness and Completeness of the FAJ typing rules for the core
module). Let \( L \) be a type uniform FAJ SPL.

1. If \( \vdash \text{core } \bar{X} \{ \cdots \} : \Gamma \) and \( \text{CST}_\Gamma \models \text{FLAT}(\varphi_\Gamma) \), then the core product \( \text{CT}_\bar{X} \) is
   - a well-typed FJ program, and
   - if \( \text{FLAT}(\varphi_\Gamma) \) is cast safe with respect to \( \text{CST}_\Gamma \), then \( \text{CT}_\bar{X} \) is cast safe.

2. If either the core module core \( \bar{X} \{ \cdots \} \) is not \( \vdash \)-typable or \( \text{CST}_\bar{X} \not\models \text{FLAT}(\varphi_\bar{X}) \), then
   the core product \( \text{CT}_\bar{X} \) is not a well-typed FJ program.

6.4 Typing Rules for Delta Modules and their Correctness/Completeness

Also, the typing rules for delta modules organize the constraints inferred for a delta module
in a two level hierarchy corresponding to the structure of the delta module in order to
support the application of a set of constraints \( D \) inferred for a delta module to the set of
constraints \( C \) inferred for the core module. Namely: (i) the typing rules infer a set of \( \text{delta removes/adds/modifies-clause constraints} \) (one for each delta clause in the delta module);
Expression typing:
\[ \Gamma \vdash x : \Gamma(x) \] 
\[ \Gamma \vdash e : C \mid B \]
\[ aType\Gamma(C,f) = D \]
\[ \Gamma \vdash e.f : D \mid \{ \text{hasField}(C,f) \} \cup B \] 
\[ \text{T-FIELD} \]

\[ \forall i \in 1..n, \Gamma \vdash e_i : C_i \mid B_i \]
\[ aType\Gamma(C)(m) = D_1 \cdots D_n \rightarrow C \] 
\[ \text{T-INV} \]

\[ \forall i \in 1..n, \Gamma \vdash e_i : C_i \mid B_i \]
\[ \Gamma \vdash \text{new} C(e_1,\ldots,e_n) : C \mid \{ \text{hasCAT}(C,C_1,\ldots,C_n) \} \cup (\cup_{i \in 1..n} B_i) \] 
\[ \text{T-NEW} \]

\[ \Gamma \vdash e : D \mid B \]
\[ \Gamma \vdash \text{cast}(C,D) \] 
\[ \text{T-CAST} \]

Method definition typing:
\[ \text{this} : C \times \bar{D} \vdash e : E \mid B \]
\[ \text{ACST}_L(C)(m) = aType\Gamma(C)(m) = \bar{D} \rightarrow D \] 
\[ \text{T-METHOD} \]

Class definition typing:
\[ D \in \text{ACST}_L(C)(\text{extends}) \]
\[ K = C \left( \bar{A} \bar{B} \bar{f} \bar{g} \right) \{ \text{super}(\bar{f}); \text{this}\bar{g} = \bar{g} \} \]
\[ \forall i \in 1..q, \text{this} : C \vdash MD_i : \{ m_i \text{ with } B_i \} \] 
\[ \vdash \text{class } C \text{ extends } D \mid \bar{F} \bar{D} ; K ; MD_1 \cdots MD_q \mid \left\{ C \text{ with } \left( \frac{\{ \text{hasPCP}(C,f) \}}{\cup (\cup_{i \in 1..q} \{ m_i \text{ with } B_i \})} \right) \right\} \] 
\[ \text{T-CLASS} \]

Core module typing:
\[ \forall i \in 1..n, \text{core } \bar{X}_i \mid CD_i : \{ C_i \text{ with } X_i \} \] 
\[ \vdash \text{core } \bar{X} \mid (CD_1 \cdots CD_n) : \cup_{i \in 1..n} \{ C_i \text{ with } X_i \} \] 
\[ \text{T-CORE} \]

Fig. 7. P.A.J: Typing rules for expressions, methods, classes and core modules w.r.t. \text{ACST}_L

(ii) a delta removes-clause constraint is a removes-clause \text{removes} C, a delta adds-clause constraint consists of the keyword \text{adds} followed by a class constraint (defined in Section 6.3), and each delta modifies-clause constraint consists of the name of the subject class C and of a set possibly containing a basic constraint of the form \text{hasPCP}(C,\cdots) (inferred for the constructor of the class) and some delta removes/renames/\text{adds-subclause constraints}; and (iii) a delta remove-subclause constraint is a delta subclause of the shape \text{removes} m, a renames-delta-subclause is a delta subclause of the shape \text{renames} m to m' and a delta adds-subclause constraint consists of the keyword \text{adds} followed by a method constraint (defined in Section 6.3). Thus, a set of delta clause constraints \mathcal{D} can be understood as a function from class names to delta clause constraints, and a delta modifies-clause-constraint can be understood as a function from method names to delta subclause constraints and from the keyword constructor to either a singleton set (containing a basic constraint of the form \text{hasPCP}(C,\cdots)) or the empty set. The syntax of the delta clause constraints is given in Figure 8.

For each feature configuration \mathcal{F} in \text{CONFIGURATIONS}, the class constraints \mathcal{C}_{\mathcal{F}} for the product \text{CT}_{\mathcal{F}} can be generated (without generating the products) by applying the sets of delta clause-constraints inferred for the delta modules \text{DELTA}(\mathcal{F}) to the class constraints of the core module according to the after partial order. The result of the application of a set of delta clause constraints \mathcal{D} to a set of class constraints \mathcal{E}, denoted by \text{APPLY}(\mathcal{D},\mathcal{E}),
Delta subclause-constraints:

- adds m with B: add constraint "method m enforces the basic constraints B"
- removes m: remove constraint "m" with ···
- renames m to m': change constraint "m with B" into "m' with B"

Delta clause-constraints:

- adds C with K: add constraint "class C enforces the constraints K"
- removes C: remove constraint "C" with ···
- modifies C with M: change constraint "C with K" into "APPLY(C with M, C with K)"
  (where M is a set possibly containing hasPCP(C, ···) and some delta subclause constraints)

Theorem 2 (Correctness and Completeness of the $F\Delta J$ type system), Let $L$ be a type-
uniform $F\Delta J$ SPL.

1. If $\vdash$ core $\forall \{\ldots\} : FLAT(\forall)$ and $\vdash$ delta $\delta \ldots : D\delta$ (for all $\delta \in \text{dom}(DMT)$), then
   for every $\forall \in \text{CONFIGURATIONS}$, if $\text{CST}_\forall \models \text{FLAT}(\forall)$ then the $FJ$ program $CT_\forall$
Delta-subclause typing:

\[
\begin{align*}
\text{ACST}_L(f) &= D \\
\text{this : C &\vdash adds D : } &\emptyset &\text{(T-ADD)} \\
\text{this : C &\vdash removes f : } &\emptyset &\text{(T-REMP)} \\
\text{this : C &\vdash renames f to f' : } &\emptyset &\text{(T-RENF)}
\end{align*}
\]

Delta-clause typing:

\[
\vdash CD : \{ C \text{ with } \mathcal{M} \} &\quad (T-ADD) \\
\vdash adds CD : \{ adds C with \mathcal{M} \} &\quad (T-CADD) \\
\vdash removes C : \{ removes C \} &\quad (T-CREM) \\
\vdash modifies C \text{[extending } D \text{]} &\quad (T-MOD)
\]

\[
\begin{align*}
&\{ \text{constructor } K \} \text{ DSC}_1 \ldots \text{DSC}_q : \{ \text{ modifies C with } \}

&\{ \text{ hasPCP( C, f) } \} \cup \{ \text{ super(f) } \} \}

&\{ \cup_{i \in \{1, \ldots, q\}} \{ \text{ dscc}_i \} \}
\end{align*}
\]

Delta-module typing:

\[
\forall i \in 1..n, \vdash \text{DSC}_i : \{ \text{dccc}_i \} &\quad (T-DELTA)
\]

\[
\begin{align*}
&\vdash \delta \text{-delta : } \{ \text{dccc}_1 \ldots \text{dccc}_q \} : \cup_{i \in \{1, \ldots, n\}} \{ \text{dccc}_i \}
\end{align*}
\]

\text{Fig. 9. FJAI: Typing rules for delta subclauses, delta clauses and delta modules w.r.t. ACST}_L

\begin{itemize}
\item is well-typed, and
\item if FLAT(\text{C}) is cast safe with respect to \text{CST}_\text{P}, then \text{CT}_\text{P} is cast safe.
\end{itemize}

2. \text{L redundant-def-free implies that if either the core module or a delta module in}
\text{DELTA}(\text{F}) is not \vdash \text{-typeable or CST}_\text{P} \not\models \text{FLAT(\text{C})}, then the FJ program \text{CT}_\text{P} is not well-typed.

7 Related Work

The approaches to implementing the variability of SPL in the object-oriented paradigm can be classified into two main directions [15]. First, annotative approaches, such as conditional compilation, frames [3] or COLORED FEATHERWEIGHT JAVA (CFJ) [13], mark the source code of the whole SPL with respect to product features on a syntactic level. For a particular feature configuration, marked code is removed. Second, compositional approaches, such as the approach presented in this work, associate code fragments to product features that are assembled to implement a particular feature configuration. In [17], general program modularization techniques, such as aspects [14], mixins [23] or traits [9, 18], are evaluated with respect to their ability to implement features. Although the above approaches are suitable to express feature-based variability, they do not contain designated linguistic concepts for representing features.

The approach which is closest to the presented work is feature-oriented programming (FOP) [5, 25, 8]. In FOP, product features are explicitly represented by a linguistic construct called feature module. Feature modules can introduce new classes or refine existing ones by adding fields and methods or by overriding existing methods. In order to obtain an implementation for a feature configuration, the feature modules are composed incrementally by carrying out the specified class refinements. Because feature modules can only
specify extensions of classes, development always starts from the mandatory features of all products. In contrast, delta modules support additions, modifications and removals of classes which allows to choose any valid product as the core module and facilitates flexible product line design starting from different core products. Feature composition in FOP is carried out in a linear order because a feature that refines a class can be added only after the class to be refined has been introduced. If an already modified method is modified by a later composed feature module, the first modifications are overridden. For delta modules, the partial order only captures essential dependencies to ensure that modified entities exist. Furthermore, it allows explicitly specifying the ordering in which conflicting modifications are carried out. A feature module in FOP is associated to a single product feature. Instead, complex application conditions over the product features can be attached to delta modules such that combinations of features can be handled explicitly. In this way, the optional feature problem [16] is solved by the flexible nature of delta modules (cf. Section 3).

The calculi Featherweight Feature Java (FFJ) [2] and Lightweight Feature Java (LFJ) [8] aim at a formalization of feature-oriented programming [5] with static guarantees by means of a type system. FFJ is based on FJ and extends it with class refinements, but the notion of feature does not appear in the syntax of the language. FFJ has a type system to check product specifications which is an extension of the FJ type system. It requires the generation of all products belonging to the product line in order to ensure their well-typedness.

LFJ extends LJ (Lightweight Java) [24] by a feature module construct representing class refinements of FOP. LFJ introduces a constraint-based type system that supports the type checking of feature modules in isolation and allows for compositional type checking, to guarantee that all products in a SPL are type correct without having to generate them. The type system infers a signature for each feature module which is a set of requirements to be satisfied by a composition including this feature. By the soundness of the LFJ type system, the satisfaction of each feature signature is enough to ensure that a composition of feature modules is a well-formed program with respect to LJ. Each signature can be translated to a propositional formula, whose conjunction $\phi_{safe}$ specifies the requirements for all correct programs. The validity of the formula $FM \land WF_{Spec} \rightarrow \phi_{safe}$ endows that the set of programs satisfying the formula $FM$, a logical representation of the feature model (according to [4]), and the formula $WF_{Spec}$ are type correct. $WF_{Spec}$ encodes the constraints on the precedence and on the subtyping relation which guarantee that the programs can be obtained by feature module composition. With the formula $WF_{Spec}$, the LFJ type system mirrors the requirement of the FOP approach that a suitable linear composition ordering has to be provided. However, a feature model describes the possible products of a product line only by sets of features without any ordering. In this respect, the approach presented in this paper is closer to the variability expressed by feature models, because the constraints inferred from the aggregate signature table $ACST$ (defined in Section 6) do not impose an ordering on delta application. The essential dependencies between delta modules are specified with the after clauses inducing a partial order. Although, this order could also be inferred automatically as in LFJ, the after clauses allow the programmer to explicitly control the order of delta application.
8 Conclusions and Future Work

In this paper, we presented a novel linguistic approach to implementing feature-based variability of software product lines within the object-oriented paradigm. The presented programming language combines the ability to represent flexibly feature-based variability by core and delta modules with means to effectively provide static guarantees on the resulting products by a constraint-based type system. This type system is designed to check type uniform SPLs since type uniformity is a valuable property for SPL implementations. The F∆J constraint-based type system can also be adapted to deal with well-formed non-type uniform SPLs. The idea is to drop the notion of aggregate class signature tables and to enrich the syntax of the constraints to include variables (along the line of the constraint-base system in [1]). These variables are instantiated to class names when checking the constraints inferred for a product with respect to the class signature table of the product.

An implementation of the programming language presented in this paper is currently being developed. In addition, we want to provide an IDE to support the programmer with a global view on the feature model and the relationships between deltas modules for effective software product line development. Furthermore, we aim at investigating the impact of selecting the core product on the resulting product line implementation. We want to provide guidelines for product line development with core and delta modules and integrate it into an model-based development process [22].

The general concept of a delta modules is not bound to a programming language. We have instantiated it on FJ [10] to show the feasibility of our approach. For future work, we are aiming at using other languages for the underlying product implementations. A starting point is the trait-based calculus Featherweight Record-Trait Java (FRTJ) [6]. In FRTJ, classes are assembled from interfaces, records (providing fields) and traits (providing methods) that can be directly manipulated by record or trait composition or using operations such as renaming of fields and methods. The operations on interfaces, records and traits provided by FRTJ make it a good candidate for implementing delta modules in a very expressive way. Moreover, they can help improving code reuse, which is another goal in SPL engineering.

References

9. S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *ACM TOPLAS*, 28(2):331–388, 2006.