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 D3.3

Hybrid Analysis for Evolvability

Due date of deliverable: (T0+45)

Actual submission date: 1st January 2013

Start date of the project: 1st March 2009

Duration: 48 months

Organisation name of lead contractor for this deliverable: KUL

Final version

<table>
<thead>
<tr>
<th>Dissemination level</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PU Public</td>
<td>✓</td>
</tr>
<tr>
<td>PP Restricted to other programme participants (including Commission Services)</td>
<td></td>
</tr>
<tr>
<td>RE Restricted to a group specified by the consortium (including Commission Services)</td>
<td></td>
</tr>
<tr>
<td>CO Confidential, only for members of the consortium (including Commission Services)</td>
<td></td>
</tr>
</tbody>
</table>
Executive Summary:
Hybrid Analysis for Evolvability

This document summarises Deliverable D3.3 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).

This deliverable reports on mechanisms for expressing the evolution of systems in the language ABS and for the analysis of such systems. As the full details of evolving systems are not known at compile time or because the configuration is based on runtime decisions, the analysis consists of a combination of static and dynamic techniques.

The approaches considered in this deliverable consist of several extensions to the ABS language. One integrates component updates with the group mechanism to ensure that updates occur safely. Another uses groups to structure objects into something akin to services and uses service discovery to bind to objects. A combination of static and dynamic checking ensures that method calls always dispatch correctly. A final approach uses a dynamic version of delta modelling that allows the selected features to be modified at runtime, resulting in a change of the deltas applied. To complement this dynamic delta modelling approach, a conflict detection type system for delta modelling is also presented.

This report also describes MetaABS, a meta-programming facility for ABS, and the new dynamic back end upon which it is based. MetaABS provides a framework for adapting the ABS runtime from within the language itself, allowing many ABS extensions and checking mechanisms to be implemented without having to change the compiler or the backend. MetaABS thus provides a strong unifying framework for implementing many different extensions to ABS. Some of these extensions have been developed in the context of this task, namely, the dynamic application of delta modules, which lies at the core of dynamic delta modelling, and user-defined COG-specific schedulers.

List of Authors

Dave Clarke (KUL)
Einar Broch Johnsen (UIO)
Radu Muschevici (KUL)
José Proença (KUL)
Michiel Helvensteijn (CWI)
Michael Lienhardt (BOL)
Contents

1 Introduction 5
   1.1 Runtime Evolution Of Object Groups ............................................. 5
      1.1.1 Component Model ................................................................. 5
      1.1.2 Adaptive Object Groups ...................................................... 6
   1.2 Hybrid Analysis of Delta Modules .................................................. 6
      1.2.1 Conflict-Free Delta Modules ............................................... 6
      1.2.2 Dynamic Modelling of Product Lines ...................................... 7
   1.3 The Dynamic ABS Back End and MetaABS ........................................ 7
   1.4 Deviations from the DoW .............................................................. 8
   1.5 List of Papers Comprising Deliverable D3.3 .................................... 8

2 The ABS Component Model 10
   2.1 Related Work .............................................................................. 10
   2.2 The ABS Approach ...................................................................... 11
   2.3 Example ...................................................................................... 12
   2.4 Conclusion .................................................................................. 13

3 A Type-Safe Model of Adaptive Object Groups .................................. 14
   3.1 Introduction .................................................................................. 14
   3.2 Related Work .............................................................................. 15
   3.3 A Kernel Language for Adaptive Object Groups ............................... 16
      3.3.1 The Syntax ........................................................................... 16
      3.3.2 Example ............................................................................... 17
   3.4 A Type and Effects System ............................................................. 17
   3.5 Operational Semantics ................................................................... 18
   3.6 Conclusion ................................................................................... 19

4 Conflict Detection in Delta-Oriented Programming ....................... 21
   4.1 Introduction .................................................................................. 21
   4.2 Related Work .............................................................................. 22
   4.3 Delta-Oriented Programming .......................................................... 24
   4.4 Conflicts ...................................................................................... 24
   4.5 Type System ................................................................................ 25
   4.6 Conclusion ................................................................................... 28

5 Dynamic Modelling of Product Lines ............................................. 29
   5.1 Introduction .................................................................................. 29
   5.2 Related Work .............................................................................. 29
   5.3 Example ...................................................................................... 30
   5.4 Dynamic Product Lines ................................................................. 30
   5.5 Cost and Optimisation ................................................................. 31
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.6 Conclusion</td>
<td>33</td>
</tr>
<tr>
<td>6.1 Introduction</td>
<td>34</td>
</tr>
<tr>
<td>6.2 MetaABS Interface</td>
<td>35</td>
</tr>
<tr>
<td>6.2.1 MetaABS Types</td>
<td>35</td>
</tr>
<tr>
<td>6.2.2 Usage Example</td>
<td>35</td>
</tr>
<tr>
<td>6.3 A Dynamic Back End for ABS</td>
<td>36</td>
</tr>
<tr>
<td>6.3.1 Design</td>
<td>36</td>
</tr>
<tr>
<td>6.3.2 Usage</td>
<td>37</td>
</tr>
<tr>
<td>6.3.3 Code Generation</td>
<td>37</td>
</tr>
<tr>
<td>6.4 Applications</td>
<td>38</td>
</tr>
<tr>
<td>6.4.1 User-Defined Process Schedulers</td>
<td>38</td>
</tr>
<tr>
<td>6.4.2 Dynamic Product Reconfiguration</td>
<td>39</td>
</tr>
<tr>
<td>6.5 Conclusion</td>
<td>39</td>
</tr>
<tr>
<td>7 Conclusion</td>
<td>40</td>
</tr>
<tr>
<td>Bibliography</td>
<td>40</td>
</tr>
<tr>
<td>Glossary</td>
<td>47</td>
</tr>
<tr>
<td>A An Object Group-Based Component Model</td>
<td>48</td>
</tr>
<tr>
<td>B A Type-Safe Model of Adaptive Object Groups</td>
<td>64</td>
</tr>
<tr>
<td>C Conflict Detection in Delta-Oriented Programming</td>
<td>80</td>
</tr>
<tr>
<td>D Dynamic Delta Modeling</td>
<td>96</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

As long-living systems evolve during their life time, it is essential to maintain consistency between component and interface specifications, even when the code base changes in unforeseen directions: errors are discovered, new scenarios appear, new requirements are added, etc. The Abstract Behavioural Specification Language (ABS) \cite{ABS} is extended to support these unforeseen changes in a safe way. This deliverable reports on mechanisms designed to implement and analyse software evolution in ABS. We approach the problem from three main perspectives.

- **Runtime evolution of object groups**: we introduce two approaches for evolving groups of objects. One introduces components as special objects that can be reconfigured at runtime. Unlike normal objects, components have ports to capture variability points, critical sections that must not be interrupted for reconfiguration, and have associated locations that can dynamically change. The other approach allows objects to move between groups in a type safe manner and uses service discovery to bind via an interface to an appropriate object in a group.

- **Hybrid analysis of dynamic delta modules**: we propose a technique to analyse and apply delta modules to systems already running. Firstly, we describe a type-system for delta modules that guarantees the composition of well-typed deltas without conflicts. Then we describe a model of dynamic product lines which evolve based on changes in the feature selection using delta modules.

- **Implementation of dynamic ABS**: to cope with the need to modify ABS code at runtime, we introduce a new reflective layer that allows introspection and manipulation of running code. This layer is exposed in a language extension called MetaABS. The new dynamic back end and the MetaABS language provide a framework for tightly integrating activities that rely on dynamic aspects of ABS.

1.1 Runtime Evolution Of Object Groups

Objects in ABS are grouped in so-called concurrent object groups (COG), which are structures that restrict how different concurrent tasks can be scheduled and how objects can communicate with each other. In this task we take a different perspective of groups of objects, seen initially as units of deployment or later as sets of common interfaces. We start by grouping objects following the component model introduced in Deliverable D2.1 \cite{D2.1} with a new focus on the safe reconfiguration of components. We then explore object groups extended with a service discovery mechanism, associating each group to a given (extendable) interface, and describe a type-safe mechanism to manage groups and support mobility between groups.

1.1.1 Component Model

Components are units in a software architecture, used to achieve unplanned dynamic reconfiguration. In many implementations of components \cite{14,22,69,70,84} the models that describe the component structure and the program execution are decoupled. Using independent models for these two concepts makes it difficult
to dynamically modify the structure of components by instructions in objects. For example, Click \[69\] does not allow runtime modifications, and OSGi \[70\] supports addition of new elements but neither modification nor removal. Fractal \[14\] supports a coherent reconfiguration of the component structure, but it is a complex model that lacks a formal presentation.

In our approach we extend the ABS language with the notion of components, which differs from the COMP component model introduced in Deliverable D2.1 \[32\]. With respect to the COMP model the new component model focuses on safe reconfiguration and on the integration with ABS, and it is described in detail in Chapter 2.

In the component model a concurrent group of objects can be associated to a location to describe, for example, that it is executed in some computer or network. A component is an object extended with special constructors to reconfigure the structure of components and to manage locations. More specifically, a component can have special fields, called ports, that can be modified only via reconfiguration. Modification of ports is known as rebinding. Methods in components can be marked as critical, and special guards can check that no critical method is being executed. Finally, objects can create new locations and move groups of objects between locations.

### 1.1.2 Adaptive Object Groups

Decoupling classes and objects is often considered a good software design practice, typically achieved by programming to interfaces, to object groups, and to service-oriented abstractions. Interfaces describe how to interact with an object providing some set of services, object groups organise a collection of objects with a dynamic notion of membership \[53\], and service abstractions, such as service discovery, allow objects to be found dynamically. These approaches to decoupling classes and objects provide a good support for adaptability. At runtime it is possible to discover members of these groups, to change membership of objects, and to create new groups. This work provides a kernel language to reason about these dynamic groups, equipped with static and runtime type systems, which guarantee that calls to members of a group always obey their interface. This is aligned closely with the description of work of this task.

### 1.2 Hybrid Analysis of Delta Modules

In software product line (SPL) engineering, a feature model describes valid combinations of features, each describing a desired software product that can be automatically built from feature selections. The choice for the ABS language was to build software products from feature selections following the delta-oriented programming approach \[77\]. This approach improves modularity, reuse, and flexibility when building SPLs with respect to other approaches.

The same delta-based mechanism used to generate new software products can be reused to transform products at runtime. In this section we focus on the hybrid analysis of delta-oriented programs. Hybrid in the sense that we present a new static approach to validate delta modules and techniques for exploring and optimising the runtime evolution of software products.

#### 1.2.1 Conflict-Free Delta Modules

Software product lines (SPLs) attempt to provide optimal code reuse when building families of software products that share several features. Using the delta-oriented programming approach for developing SPLs \[17, 18\], a set of delta modules are selected and combined for a given choice of valid features \[77\]. A delta module describes a set of transformations to a core software product to produce a new product. The final software product is built by applying the combined delta module to the core code base.

However, the selected deltas must not be in conflict: two delta modules are in conflict if they modify a shared code space in incompatible ways and there is no explicit order imposed between them. Schaefer et al. \[79\] addressed the validation of delta-oriented programs by using a type-based approach where delta-oriented product lines are encoded as constraints that must hold. The approach described here overcomes
some of their limitations to detect conflicts, such as the computational complexity and the need for a total order over delta modules, and cover a larger set of conflicts. The approach that we propose uses a type system based on row-polymorphism \cite{75} that can guarantee freedom of conflicts. The validation of delta modules provides trustworthiness guarantees to \textit{ABS} programs that evolve at runtime via delta application, as described in the DoW of this task.

### 1.2.2 Dynamic Modelling of Product Lines

Dynamic product lines \cite{25} explore variability at runtime, by introducing an automata model that connects different products with transitions that specify the transfer of state information, and by extending the language with a new statement declaring when it is safe to reconfigure. In this deliverable, we follow the same dynamic approach to product lines but use a more incremental mechanism to achieve product evolution, by using delta modules based on changes of feature selections. Furthermore, we extend dynamic product lines with a simple cost model added to transitions, which are used to optimise dynamic product lines configuration.

### 1.3 The Dynamic \textit{ABS} Back End and MetaABS

The tool support for the \textit{ABS} language provides several back ends, each providing an executable version of a given \textit{ABS} model. A good overview of this framework is given in Deliverable 1.2 \cite{31}. The most actively used back ends generate code that can be executed in Java and in Maude. The former allows fully interoperability with Java, i.e., Java code can be used by and produced from \textit{ABS} programs. The latter is based on the term-rewriting engine Maude \cite{21} and follows faithfully the operational rules of \textit{ABS}. Using Maude, the execution of an \textit{ABS} program is given by the consecutive application of rewrite rules to a tuple representing the state of the program.

While the Maude back end is more flexible, allowing easy extensions to the runtime semantics, the Java back end is tightly connected to the operational model of Java, making it difficult to extend with dynamic constructs. The ability to modify the runtime execution of an \textit{ABS} program addresses objectives of other tasks of the \textbf{HATS} project as well, including:

- **user defined scheduling** (Task 2.1): changing how concurrent processes are scheduled;
- **dynamic load balancing** (Task 2.1): based on deployment components, dynamically changing the location where concurrent object groups are deployed;
- **modelling and specification of evolvable systems** (Task 3.1): identifying fundamental mechanisms to model systems evolving at runtime; and
- **autonomously evolving systems** (Task 3.5): allowing objects to reconfigure programs where they run during their execution.

To avoid developing a customised implementation for each extension of the \textit{ABS} language requiring runtime capabilities, we present a reflective layer for \textit{ABS}, called MetaABS. MetaABS provides support for meta-programming, exposing to the programmer runtime objects and classes of \textit{ABS} and allowing their modification. Due to the large flexibility given by this meta-programming layer we risk not being able to guarantee trustworthiness of \textit{ABS} programs. Therefore the intention is that it is only made available for developers of tools for \textit{ABS}, and not for the traditional software developer.

The execution of MetaABS in Java is achieved via a new back end, the \textit{dynamic Java back end}. This back end keeps information regarding the state of the running program using data structures, as opposed to convert \textit{ABS} classes directly into Java classes. The result is an integrating layer with runtime support for tools for \textit{ABS}. Safety properties are assured not by MetaABS but MetaABS provides a framework for implementing dynamic checks.
1.4 Deviations from the DoW

When the DoW was written, it was anticipated that adaptability would be phrased in terms of open ABS specifications. In the early phases of the project, the delta-modelling approach and components were adopted to express static and dynamic variability, respectively, and it was therefore natural to recast the work done in Task 3.3 in terms of these technologies instead of open ABS specifications. Another deviation from the DoW is that more focus than originally intended was spent on the meta-programming extension to ABS, MetaABS, and the supporting dynamic Java back end. This is because it became apparent while performing the research that a unifying extension to the runtime of ABS was needed in order to cater for the various mechanisms described in this task (and in others) that needed additional runtime support. This means that new extensions to the ABS language can be implemented in MetaABS, without requiring new compiler and new back end support.

1.5 List of Papers Comprising Deliverable D3.3

This section lists all the papers that comprise this deliverable, indicating where they were published, and explains how each paper is related to the main text of this deliverable. As requested by the reviewers, the papers are not directly attached to Deliverable D3.3. A version of this deliverable with the papers attached to it is available on the HATS web site at the following url: [http://www.hats-project.eu/sites/default/files/Deliverable3.3-with-papers.pdf](http://www.hats-project.eu/sites/default/files/Deliverable3.3-with-papers.pdf).

**Paper 1: An Object Group-Based Component Model**

This paper [56] describes the component model, where components are regarded as special objects that can be connected to other objects, and exploits rebinding and mobility of objects while preserving trustworthiness. This is the main subject of the next chapter.

This paper was written by Michaël Lienhardt, Mario Bravetti, and Davide Sangiorgi, and was published in the proceedings of the International Symposium On Leveraging Applications of Formal Methods, Verification and Validation 2012.

Attached in Appendix A.

**Paper 2: A Type-Safe Model of Adaptive Object Groups**

This paper [11] describes a core language that explores service-oriented abstractions in an object-oriented setting, equipped with a static and runtime type system. Chapter 3 presents an overview of this language.

This paper was written by Joakim Bjerk, Dave Clarke, Einar Broch Johnsen, and Olaf Owe, and it is to appear in the proceedings of the International Workshop on Foundations of Coordination Languages and Self Adaptation 2012.

Attached in Appendix B.

**Paper 3: Conflict Detection in Delta-Oriented Programming**

This paper [57] develops a type system to validate the composition of delta modules, guaranteeing absence of conflicts. Chapter 4 describes this type system.

This paper was written by Michaël Lienhardt and Dave Clarke, and was published in the proceedings of the International Symposium On Leveraging Applications of Formal Methods, Verification and Validation 2012.

Attached in Appendix C.
Paper 4: Dynamic Delta Modeling

This paper explores the runtime evolution of software products by applying delta modules, and optimisation techniques for such systems based on a cost model. This dynamic evolution of delta-oriented programs is investigated in Chapter 5.

This paper was written by Michiel Helvensteijn and was published in the second volume of the proceedings of the International Software Product Line Conference 2012.

Attached in Appendix D.
Chapter 2

The ABS Component Model

Components are an intuitive tool to achieve unplanned dynamic reconfiguration. In a component system, an application is structured into several distinct pieces called components. Each of these components has dependencies on functionality located in other components; such dependencies are collected into a set of output ports. The component itself offers functionality to the other components, which is collected into a set of input ports. Communication from an output port to an input port is possible when a binding between the two ports exists. Dynamic reconfiguration in such a system is then achieved by adding and removing components, or by replacing bindings. Thus updates or modifications of parts of an application are possible while the operation is running.

The work described in this chapter addresses the DoW by providing a mechanism for expressing the evolution of ABS models based on components. Safety is achieved using critical sections and guards that can only succeed when an object is not in a critical section. These integrate well with ABS’s futures and concurrent object groups.

The work was carried out by BOL and it extends the component model described in Deliverables D2.1 [32] and D3.1b [29]. The core difference is that the current work is integrated into ABS and it includes features for synchronising the updates so that they can occur safely.

2.1 Related Work

While the idea of component is simple, bringing it into a concrete programming language is not easy. The informal description of components talks about the structure of a system, and how this structure can change at runtime, but does not mention program execution. Many implementations of components [70, 14, 22, 84, 10, 67, 69] do not integrate the program execution model (generally implemented using a classic object-oriented language like Java or C++) and the component structure (generally described in an Architecture Description Language (ADL)) into one coherent model. Unplanned dynamic reconfigurations become hard, as it is difficult to express modifications of the component structure using objects, since these are just supposed to describe the execution of the programs. For instance, models like Click [69] do not allow runtime modifications, while OSGi [70] only allows the addition of new classes and objects: component deletion or binding modification are not supported. In this respect, a more flexible model is Fractal [14], which reifies components and ports into objects. Using an API, in Fractal it is possible to modify bindings at runtime and to add new components; still, it is difficult for the programmers to ensure that reconfigurations will not create state inconsistencies.

Formal approaches to component models have been studied [16, 54, 83, 63, 62, 61]. These models have the advantage of having a precise semantics, which clearly defines what components, ports and bindings are (when such constructs are included). This helps understanding how dynamic reconfigurations can be implemented and how they interact with the normal execution of the program. In particular, Oz/K [62] and COMP [63] propose a way to integrate in an unified model both components and objects. However, Oz/K has a complex communication pattern, and deals with adaptation via the use of passivation, which is
complicated. In contrast, COMP offers support for dynamic reconfiguration, but integrating it into the semantics of ABS appears to be complex as it requires a total modification of the communication mechanism between objects.

2.2 The ABS Approach

Most component models have a notion of component that is distinct from the objects used to represent the data and the main execution of the software. The resulting language is structured into two layers, one using objects for the main execution of the program, and one using components for the dynamic reconfiguration. Even though this separation seems natural, it makes the integration of the different requests for reconfiguration into the program’s workflow difficult. In contrast, our approach, presented in [56], has a uniform description of objects and components; that is, we enhance objects and object groups—the core ingredients of ABS—with the core elements of components (ports, bindings, consistency and hierarchy) and hereby enable dynamic reconfiguration.

We achieved this by exploiting the similarities between objects (and object groups) and components. Most importantly, the provided methods of an object closely resemble the input ports of a component. In contrast, objects do not have explicit output ports, but the dependencies of an object can be stored in internal fields. Thus, rebinding an output port corresponds to the assignment of a new value to the field. However, classic objects cannot ensure the consistency of the rebinding. Suppose we wished to treat certain object fields as output ports: we could add methods to the object for their rebinding; but it would be difficult in presence of concurrency to ensure that a call to one of these methods does not harm ongoing computations. For instance, if we need to update a field (such as one containing a reference to a printer driver), then we would first want to wait for the termination of all current executions referring to that field (e.g., printing jobs). In ABS, object groups offer a mechanism for consistency by ensuring that there is at most one task running in an object group. This does enable some consistency, but it is insufficient in situations involving several method calls. Another difference between object and component models is that only the latter talks about locations. Locations structure a system, possibly hierarchically, and can be used to express dynamic addition or removal of code, as well as distribution of a program over several computers.

To ensure the consistent modifications of bindings and the possibility to ship new pieces of code at runtime, we add four elements to the ABS core language:

1. A notion of output port distinct from the object’s fields. The former (identified with the keyword \texttt{port}) corresponds to the objects’ dependencies and can be modified only when the object is in a \textit{safe} state, while the latter corresponds to the inner state of the objects and can be modified with ordinary assignments.

2. The possibility to annotate methods with the keyword \texttt{critical}: this specifies that the object, while this method is executing, is not in a safe state.

3. A new primitive to wait for an object to be in a safe state. Thus, it becomes possible to wait for all executions using a given port to finish, before rebinding the port to a new object.

4. Locations. Our semantics structures an ABS model into a tree of locations that can contain object groups. Using locations, it is possible to model the addition of new pieces of code to a program at runtime. Moreover, it is also possible to model distribution (each top-level location is a different computer) and code mobility (by moving a sub-location from a computer to another one).

The resulting language remains close to the underlying ABS language; it is a conservative extension of ABS (i.e., a core ABS model is valid in our language and its semantics is unchanged), and, as shown in our following example, introducing the new primitives into an ABS program is simple.
2.3 Example

We illustrate our approach with an example inspired from the Virtual Office case-study of the HATS project [28]. This case study assumes an open environment with resources like computers, projectors or printers that are used to build different workflows. For the purpose of our example, we suppose that we want to define a workflow that takes a document (a resource modelled with the class Document), modifies it using another resource (modelled with the class Operator) and then sends it to a printer (modelled with the class Printer). We also assume that the protocol used by Operators is complicated, so we isolate it into a dedicated class. Finally, we want to be able to change the protocol at runtime, without disrupting the execution of previous instances of the workflow. Such a workflow is presented in Figure 2.1.

There are two classes: the class OperatorFrontEnd implements the protocol in the method modify; and the class WFController encodes the workflow. The elements _op, _doc and _p are ports, annotated with port, and represent dependencies to external resources. It is only possible to modify their value using the construct rebind, which checks if the object is in a safe state (no critical method in execution) before modifying the port. Moreover, methods modify and newInstanceWF make use of these ports in their code, and are thus annotated as critical as it would be dangerous to rebind ports during their execution.

The key operations of our component model is shown in the two lines of code in the body of the method changeOperator. First is the await statement, which waits for the objects this and _opfe to be in a safe state. By construction, these objects are in a safe state only when there are no running instances of the workflow: it is then safe to modify the ports. Second is the rebind statement; the statement will succeed since the concurrency model of object-groups ensures that no workflow instance can be spawned between the end of the await and the end of the method. Moreover, the second line shows that it is possible to rebind a port of another object, provided that this object is in the same group as the one doing the rebinding.
2.4 Conclusion

The language extension described in this chapter offers facilities for dynamic evolution while remaining close to the underlying ABS language. Indeed, the language is a conservative extension of ABS. In contrast with other component models, our language does not drastically separate objects and components. Three major features of the notion of component — ports, consistency, and location — are present in the language as follows: output ports, taken care of at the level of our enhanced objects; consistency, taken care of at the level of object groups; and information about locations, which is added separately.

This model was successfully implemented in the maude backend of the ABS toolchain.
Chapter 3

A Type-Safe Model of Adaptive Object Groups

This chapter describes an approach to the dynamic evolution of systems based on object groups and service discovery. Rather than having objects communicating directly with other objects, a level of indirection is imposed between objects. This is a group. Simply put, a group is a collection of objects that advertise their interface for other objects to bind to. Objects use service discovery to bind to an appropriately typed object from a group. The type of a group is the collection of interface types of objects publishing in that group. To some degree this can be reasoned about statically (assuming that the set of implemented interfaces increases monotonically). In general, however, programs need to perform a dynamic check to determine whether a given type is available in a group.

This work addresses the DoW in the following way. The separation of otherwise tightly coupled objects into groups with published interfaces accessible via service discovery allows systems to evolve more easily, and the combination of static and dynamic checking ensures that certain vulnerabilities are avoided by trapping them as early as possible.

The work described in this chapter is a result of a collaboration of UIO and KUL in the context of Work Package 3.3. UIO provided expertise on concurrent objects and KUL provided expertise on type systems. This work built upon previous work of these two partners [19].

3.1 Introduction

This chapter explores service-oriented abstractions such as service adaptation, discovery, and querying in an object-oriented setting. We develop a formal model of adaptive object-oriented groups which offer services to their environment. These groups fit directly into the object-oriented paradigm in the sense that they can be dynamically created, they have an identity, and they can receive method calls. In contrast to objects, groups are not used for structuring code. A group exports its services through interfaces and relies on objects to implement these services. Objects may join or leave different groups. Groups may dynamically export new interfaces, they support service discovery, and they can be queried at runtime for the interfaces they support.

This chapter gives an overview of an operational semantics and a static type system for this adaptive group model based on interfaces, interface queries, groups, and service discovery. The type system ensures that well-typed programs do not cause method-not-understood errors at runtime.

The chapter is organised as follows. Section 3.2 discusses related work. Section 3.3 presents the language syntax and a small example. Core fragments of a type and effect system and an operational semantics for the language are presented in Section 3.4 and in Section 3.5. Section 3.6 concludes the chapter.
3.2 Related Work

The most common use of object groups is to provide replicated services in order to offer better fault tolerance. This idea originated in the Amoeba operating system [46], where communication to elements of a group is via multicast. The component model Jgroup/ARM [65] adopts this idea to provide autonomous replication management using distributed object groups. In this setting, members of a group maintain a replicated state for reasons of consistency. The ProActive active object programming model [8] supports abstractions for object groups, which enable group communication—via method call—and various means for synchronising on the results of such method calls, such as wait-for-one and wait-for-all. ProActive is formalised in Caromel and Henrio’s Theory of Distributed Objects [15].

Object groups have been investigated as a modularisation unit for objects which is complementary to components. Groups meet the needs of organising and describing the statics and dynamics of networks of collaborating objects [53]; groups can have many threads of control, they support roles (or interfaces), and objects may dynamically join and leave groups (inspired by CORBA).

Another early work on groups is ActorSpaces [2], which combine Actors with Linda’s pattern matching facility, allowing both one-to-one communication, multicast, and querying. Unlike our approach, groups in ActorSpaces are intensional: all actors with the same interface belong to the same group. Furthermore, ActorSpaces support broadcast communication to a group, which has not been considered in this chapter as it would differentiate communication with an object and with a group.

Object groups have further been used for coordination purposes. For example, CoLaS [23] is a coordination model based on groups in which objects may join and leave groups. CoLaS allows very intrusive coordination of message delivery based on a Coordinator state, and the Coordinator may access the state of participants. The model is implemented in Smalltalk and neither formalisation nor typing is discussed [23]. Concurrent object groups have also been proposed to define collaborating objects with a single thread of control in programming and modelling languages [82, 45]. Concurrent object groups do not have identity and function as runtime restrictions on concurrency rather than as a linguistic concept.

Microsoft’s Component Object Model (COM) supports querying a component to check whether it supports a specific interface, similar to the query-mechanism considered in this chapter. A component in COM may also have several interfaces, which are independent of each other. COM has proven difficult to formalise; Pucella develops $\lambda^{COM}$ [74], a typed $\lambda$-calculus which addresses COM components in terms of their interfaces, and discusses extensions to capture subtyping, querying for interfaces, and aggregation.

A wide range of service discovery mechanisms exist [41]. The programming language AmbientTalk [26] has built-in service discovery mechanisms, integrated in an object-oriented language with asynchronous method calls and futures, but without groups. Various works formalise the notion of service discovery [52], but they often do so in a formalism quite far removed from the object-oriented setting. For example, Fiadeiro et al.’s $\mathcal{M}$ model of service discovery and binding takes an algebraic and graph-theoretic approach, but it lacks the concise operational notion of service discovery formalised in our model. No type system is presented either.

Our earlier work [19] enabled objects to advertise and retract interfaces to which other objects could bind using a primitive service discovery mechanism. A group mechanism was also investigated as a way of providing structure to the services. In that work services were equated with single objects, whereas in the present work a group service is a collection of objects exporting their interfaces, and thus groups can change over time to support more functionality.

The key differences with most of the discussed works is that the model in this chapter remains within the object-oriented approach, multiple groups may implement an advertised service in different ways, and our formalism offers a transparent group-based service discovery mechanism with primitive exclusion policies. Furthermore, our notion of groups has an implicit and dynamically changing interface.
### 3.3 A Kernel Language for Adaptive Object Groups

We study an integration of service-oriented abstractions in an object-oriented setting by defining a kernel of the ABS language. The language has a notion of group which dynamically connects interfaces to implementations. Groups are first-class citizens; they have identities and may be passed around. An object may dynamically join a group and thereby add new services to this group, extending the group’s supported interfaces. Objects may be part of several groups. Both objects and groups may join and leave groups, thereby migrating their services between groups.

#### 3.3.1 The Syntax

The syntax of the kernel language is given in Figure 3.1. As the language is derived from ABS, only novel elements are described. A type \( T \) in the kernel language is either a basic type, an interface describing a service, or a group of interfaces, written \( \text{Group}(T) \), denoting a group that supports the set \( T \) of interfaces.

The expressions \( e \) of the kernel language include two new constructors that are related to service-oriented software. The \( \text{newgroup} \) constructor dynamically creates a new, empty group which does not offer any services to the environment. Service discovery may be localised to a named group \( y \): the expression \( \text{acquire } I \text{ in } y \text{ except } \pi \) finds some group or object \( z \) such that \( z \) offers a service better than \( I \) (in the sense of subtyping) and such that \( z \) is not in the set \( \pi \). If the \( \text{in } y \) clause is omitted, then the service provider \( z \) may be found anywhere in the system.

The statements \( s \) of the kernel language include a facility to dynamically export service interfaces \( T \) through a group \( y \) by the expression \( x \text{ joins } y \text{ as } T \), which states that object or group \( x \) is used to implement the interfaces \( T \) in the group \( y \). Consequently, \( y \) will support the interfaces \( T \) after \( x \) has joined the group. Objects and groups \( x \) may try to withdraw service interfaces \( T \) from a group \( y \) by the expression \( x \text{ leaves } y \text{ as } T \{ s_1 \} \text{ else } \{ s_2 \} \). Withdrawing interfaces from a group can lead to runtime exceptions which need to be handled either by the client or by the service provider. In our approach, the exception is handled on the server side; i.e., withdrawing interfaces \( T \) from \( y \) only succeeds if \( y \) continues to offer all the interfaces of \( T \), exported by other objects or groups. Thus, removals may not affect the type of \( y \). If the removal is successful then branch \( s_1 \) is taken, otherwise \( s_2 \) is taken. In addition, the language includes the statement \( x \text{ subtypeof } I \text{ in } y \{ s_1 \} \text{ else } \{ s_2 \} \) which is used to \text{query} a known group \( x \) about its supported interfaces. The statement works like a conditional and branches the execution depending on whether the query succeeds or not. If \( x \) offers an interface better than \( I \), the expanded knowledge of the group \( x \) becomes available through the variable \( y \) in the scope of the statements \( s_1 \). If \( x \) does not offer an interface as good as \( I \), the branch \( s_2 \) is taken. Remark the introduction of a new name for the group inside the scope, which ensures that the knowledge of the extended type is local.
3.3.2 Example

We illustrate the dynamic organisation of objects in groups using an example of software which provides text editing support (inspired by [74]). This software provides two interfaces: SpellChecker allows the spell-checking of a piece of text and Dictionary provides functionality to update the underlying dictionary with new words, alternate spellings, etc. Apart from an underlying shared catalog of words, these two interfaces need not share state and may be implemented by different classes. Let us assume that the overall system contains several versions of Dictionary, some of which may have an integrated SpellChecker. Consider a class implementing a text editor factory that manages groups implementing these two interfaces. The factory has two methods: makeEditor dynamically assembles such software into a text editor group and replaceDictionary allows the Dictionary to be dynamically replaced in such a group. These methods may be defined as follows:

```java
group<SpellChecker, Dictionary> makeEditor() {
    group<> editor; SpellChecker s; Dictionary d;
    editor = newgroup;
    d = acquire Dictionary except emptyset;
    d subtypeOf SpellChecker ds {
        ds joins editor as Dictionary, SpellChecker;
    } else {
        d joins editor as Dictionary;
        s = new SpellChecker();
        s joins editor as SpellChecker;
    }
    return editor;
}

void replaceDictionary(group<SpellChecker, Dictionary> editor, Dictionary nd) {
    Dictionary od;
    nd joins editor as Dictionary;
    od = acquire Dictionary in editor except nd;
    od leaves editor as Dictionary {skip;} else {skip;};
    return;
}
```

The method makeEditor acquires a top-level service d which exports the interface Dictionary (since there is no in-clause in the acquire-expression). If d also supports the SpellChecker interface, d can join the newly created group editor as both Dictionary and SpellChecker. Otherwise d joins the editor group only as Dictionary. In this case a new SpellChecker object is created and added to the group as SpellChecker. Remark that we assumed the presence of several Dictionary services in the overall system, otherwise the initial acquire-expression may not succeed and execution could be blocked at this point. The kernel language could be extended by a more robust version of acquire which uses branching (similar to subtypeOf); in fact, inside a group g, robustness may be obtained by first checking for the existence of an interface I in g using subtypeOf and then binding to the object or group implementing I in g using acquire.

The method replaceDictionary will replace the Dictionary service in a text editor group. First we add the new Dictionary service nd to the editor group and then we fetch the old service od in the group by means of an acquire, where the except-clause is used to avoid binding to the new service nd. Finally the old service od is removed as Dictionary in the group by a leave statement. The example illustrates group management by joining and leaving mechanisms as well as service discovery.

3.4 A Type and Effects System

The language distinguishes behaviour from implementations by using an interface as a type which describes a service. Classes are not types in source programs. A class can implement a number of service interfaces, so its instances can export these services to clients. A program variable typed by an interface can refer to an instance of any class which implements that interface. A group typed by Group(I) exports the services described by the set I of interfaces to clients, so a program variable of type I may refer to the group if I ∈ I. We denote by Any the “empty” interface, which extends no interface and declares no method signatures. A service described by an interface may consist of only some of the methods defined in a class which implements the interface, so interfaces lead to a natural notion of hiding for classes. In addition to the source program types used by the programmer, class names are used to type the self-reference this; i.e., a class name is used as an interface type which exports all the methods defined in the class.
HATS Deliverable D3.3

Hybrid Analysis for Evolvability

I

is extended with a new variable

effects of the two branches are treated as for the conditional. Rule

the method being analysed. Without this restriction, a field could dynamically extend its type, resulting in

interfaces

if

The runtime syntax is given in Figure 3.3. A runtime configuration \( cn \) is either the empty configuration \( \varepsilon \) or it consists of objects \( obj \) and groups \( grp \). Groups \( grp \) have an identity \( g \) and contain a set \( export \) of interfaces \( I \) associated with the objects \( o \) implementing them. Objects \( obj \) have an identity \( o \), a state \( \sigma \), and a stack \( \rho \) of processes \( proc \).

The operational semantics is given by rules in the style of SOS \([72]\), reflecting small-step semantics, and the most relevant rules are presented in Figure 3.4. Rules involving an object and a group will lock the group in question, thereby disallowing concurrent execution of other objects involving the same group. This

Figure 3.2: A fragment of the type system for term.

Subtyping. The subtype relation \( \prec \) is defined as the transitive closure of the extends-relation on interfaces: if \( I \) extends \( J' \) and \( J' \prec J \) or \( J' = J \), then \( I \prec J \). It is implicitly assumed that all interfaces extends \( \text{Any} \), so we let \( I \prec \text{Any} \) for all \( I \). A group type \( \text{Group}(S) \) is a subtype of \( I \) if there is some \( J \in S \) such that \( J \prec I \), and \( \text{Group}(S) \prec \text{Group}(S') \) if for all \( J \in S' \) there is some \( I \in S \) such that \( I \prec J \). We extend the source language subtype relation by letting a class be a subtype of all its implemented interfaces. The reflexive closure of \( \prec \) is denoted \( \triangleq \).

Typing contexts. Typing contexts and operations on them are defined as usual. In addition, for typing contexts \( \Gamma_1 \) and \( \Gamma_2 \), define the intersection \( \Gamma_1 \cap \Gamma_2 \) by \( \Gamma_1 \cap \Gamma_2(x) = T \) if \( T \) is the best type such that \( \Gamma_1(x) = T_1, \Gamma_2(x) = T_2, \) and \( T_1 \triangleq T \) and \( T_2 \triangleq T \). In particular, we have \((\Gamma_1 \cap \Gamma_2)(x) = \text{Group}(S_1 \cap S_2)\) if \( \Gamma_1(x) = \text{Group}(S_1) \) and \( \Gamma_2(x) = \text{Group}(S_2) \).

The Type and Effect System. Programs in the kernel language are analysed using a type and effect system (e.g., [5, 3, 63]). The inference rules for some novel terms are shown in Figure 3.2. The rules for other features are standard and can be found in the attached paper in Section 1.5. By T-ACQUIRE, service discovery has the desired type, if successful. The premise of the rule is omitted if the statement has no in-clause.

For statements, the typing judgment \( \Gamma \vdash s : \text{ok}(\Delta) \) expresses that the statement \( s \) is well-typed if the variables in \( s \) are typed according to \( \Gamma \) and that the typing context for further analysis should be modified according to the effect \( \Delta \). Empty effects are omitted in the presentation of the rules. By T-JOIN, when an object joins a group \( y \) and contributes interfaces \( T \) to \( y \), the effect is that the type of \( y \) is extended with the interfaces \( T \). Note the requirement local\((y)\), which expresses that \( y \) must be a local variable in the scope of the method being analysed. Without this restriction, a field could dynamically extend its type, resulting in an unsound system. Rule T-LEAVE shows that leaving a group has no effect on the typing context, and the effects of the two branches are treated as for the conditional. Rule T-INSPECT shows how the typing context is extended with a new variable \( y \) which extends the type of the group \( x \) for the scope of the branch \( s_1 \). The overall effect is again the intersection of the effects of the two branches.

3.5 Operational Semantics

The runtime syntax is given in Figure 3.3. A runtime configuration \( cn \) is either the empty configuration \( \varepsilon \) or it consists of objects \( obj \) and groups \( grp \). Groups \( grp \) have an identity \( g \) and contain a set \( export \) of interfaces \( I \) associated with the objects \( o \) implementing them. Objects \( obj \) have an identity \( o \), a state \( \sigma \), and a stack \( \rho \) of processes \( proc \).

The operational semantics is given by rules in the style of SOS [72], reflecting small-step semantics, and the most relevant rules are presented in Figure 3.4. Rules involving an object and a group will lock the group in question, thereby disallowing concurrent execution of other objects involving the same group. This
is crucial in the JOIN and LEAVE1 rules for joins and leaves, which may actually modify the group. In the rule NEW-GROUP, a globally unique group identifier is found by fresh$(g)$. Then an empty group with this identifier is added to the configuration.

The rule JOIN extends the knowledge of a group with the new interfaces from the object’s perspective and correspondingly extends the exports set from the group’s perspective. Service discovery is handled by the ACQUIRE rule. The acquire expression is replaced by a value $v$, which is an object or group identifier satisfying the in and except clauses. If the in clause is omitted from the expression, then the premise $(a \circ l)^V(y) = g$ is omitted from the rule. Note that this rule will block if no matching object or group exists. This could be solved by either returning null (by means of a global check) or by adding an else branch similar to those in QUERY1 and QUERY2. Within the kernel language, the existence of a matching object or group inside a group can be checked using the query mechanisms.

The leaves statement is handled by the rules LEAVE1 for a successful leave and LEAVE2 for an unsuccessful one. A group or object $x$ may leave a group successfully if the group provides the same interface support without $x$. To determine this, we use the function intf$(\text{export})$ which returns a set containing the interfaces of all the pairs in export, removing redundant information. An entry is redundant if a subtype of the entry is present in the set. The type of the group does not change by a leaves statement and hence the object does not need to update information about the group. The branches $s_1$ or $s_2$ are chosen depending on the success. The rules QUERY1 and QUERY2 handle the branching statement that checks if a group exports a given interface. If the test succeeds then a fresh variable $y$ is introduced and is only visible in $s_1$. The type of this variable is the union of what the current object already knew about the group and the new information $I$. If the test fails the $s_2$ branch is chosen by QUERY2.

### 3.6 Conclusion

The chapter has proposed a formal model for adaptive systems, based on a notion of object-oriented groups, where groups are first-class citizens. A main advantage is that one may collect several objects into a group, thereby obtaining a rich interface reflecting a complex service, which can be seen as a single object from the outside. In contrast to objects, groups may dynamically add support for an increasing number of interfaces. The formation of groups is dynamic; join and leave primitives in the kernel language allow the migration of services provided by objects and inner groups as well as software upgrade, provided that interfaces are not removed from a group. An object or group may be part of several groups at the same time. This gives a very flexible notion of group.

Adaptive object groups are combined with service discovery by means of acquire and subtypeOf constructs in the kernel language, which allow a programmer to discover services in an open and unknown environment or in a known group, and to query interface support of a given object or group. These mechanisms are formalised in a general object-oriented setting, based on experiences from a prototype Maude [21] implementation of the group and service discovery primitives. The presented model provides expressive mechanisms for adaptive services in the setting of object-oriented programming with modest conceptual additions, and is one of the candidate targets for the MetaABS model explained in Chapter 6.
Figure 3.4: A fragment of the operational semantics.
Chapter 4

Conflict Detection in Delta-Oriented Programming

A collection of deltas can be in conflict if applying two deltas in different orders results in a different software product. Conflicts need to be detected and resolved statically, generally by imposing an order on the offending deltas, before deploying them in order to avoid runtime errors when applying deltas dynamically (as described in Section 5).

The work described in this chapter addresses the DoW by providing the basis of static techniques for checking the well-formedness of deltas before they are deployed, either statically or dynamically. The work was carried out by KUL, who provided expertise on delta-oriented programming and typing, and BOL, who provided expertise on type-checking using row polymorphism. The results extend work previously reported in Deliverable 2.4 [35].

4.1 Introduction

This chapter discusses the notion of conflict for a variant of Delta-Oriented Programming (DOP) [75, 80] without features, distinguishing hard from soft conflicts. We define a type system based on row-polymorphism that ensures that the computation of a well-typed product will always succeed and has an unambiguous result.

```java
class Settings {
    int coffee;
}

class CMachine {
    Settings conf;
    void make() {...}
    void makeCoffee() {...}
}

delta Choco after Sugar {
    modifies class Settings {
        adds int chocolate;
    }
    modifies class CMachine {
        adds makeChoco() {...}
        modifies make() {...}
    }
}

delta Sugar {
    modifies class Settings {
        adds int sugar;
    }
    modifies class CMachine {
        modifies make {...}
    }
}

delta ColourPrint {
    modifies class CMachine {
        modifies make() {...}
        modifies makeCoffee() {...}
        modifies makeChoco() {...}
    }
}

product p_s {Choco Sugar}
product p_h {Choco ColourPrint}
```

Figure 4.1: Soft and Hard Conflicts
In DOP, if delta modules are applied to a core product in a different order, it is not necessarily the case that all computations give the same result. This is illustrated by the code in Figure 4.1. This example models a coffee machine with a core comprised of a class `Setting` storing the type of coffee to brew, and a class `CMachine` with a generic `make` method and a method `makeCoffee`, called by `make` to prepare coffee. In addition to this core, there are three deltas: `Choco` adds the capability of brewing hot chocolate; `Sugar` adds the possibility of setting the quantity of sugar; and `ColourPrint` changes the `make*` methods so that messages are printed in colour. Finally, there are two different products: \( p_s \) applies deltas `Choco` and `Sugar` to the core program, and product \( p_h \) is constructed by applying deltas `Choco` and `ColourPrint` to the core. The order in which the deltas are applied is free in this example, and thus \( p_s \) and \( p_h \) can either be computed by applying either delta `Choco` or the other one first.

Applying first `Choco` and then `Sugar` in \( p_s \) results in a product with the method `make` defined by the delta `Sugar`, whereas if `Sugar` is applied first, the method `make` is defined by `Choco`: the computation of \( p_s \) is *ambiguous* (i.e., it can have different results), caused by a *soft conflict* between the deltas `Choco` and `Sugar`. Such soft conflicts can be dealt with in two ways, exemplified in Figure 4.2: (1) by imposing a partial order between deltas or (2) by introducing a delta resolving the conflict. In the first solution (on the left) `Choco` is redefined to be always applied after `Sugar`, whenever both deltas are selected, by using the keyword `after`. In the second solution (on the right) a new delta `SweetChoco` replaces a previous implementation of `make` by `Choco` and `Sugar`.

```plaintext
delta Choco after Sugar {
  ...
  modifies class CMachine {
    adds makeChoco() {...}
    modifies make() {...}
  }
}
```

```plaintext
delta SweetChoco after Choco Sugar {
  modifies class CMachine {
    modifies make() {...}
  }
}
```

Figure 4.2: Solutions for soft constraints: imposing an order (left) and introducing a resolving delta (right)

The product \( p_h \) presents another kind of conflict, called *hard conflicts*. While first applying the delta `Choco` and then `ColourPrint`, the computation succeeds without any error, first applying `ColourPrint` results in an error because `ColourPrint` tries to modify `makeChoco` before it exists. Such hard conflicts can only be resolved by imposing an ordering on the deltas specifying that `ColourPrint` must be applied after `Choco`.

**Roadmap.** The chapter is structured as follows. Section 4.2 presents related work. Section 4.3 describes a DOP language focusing on deltas and conflicts. Section 4.4 presents a formal definition of soft and hard conflicts. Section 4.5 introduces our type system to capture runtime errors and conflicts. Section 4.6 concludes the chapter.

## 4.2 Related Work

The goal of type checking the code base of a software product line is to ensure that the generated products are type safe, up to the degree of type safety provided by the base language, *without* having to actually generate the products. Other static analysis techniques can instead be employed to check for other potential deficiencies, without aiming to ensure complete type safety.

In particular, the issue of validating delta-oriented programs has not fully been addressed. Schaefer et al. [79] propose to generate a collection of constraints for delta-oriented product lines, ensuring that the manipulations done on the core product are sound and the resulting products are type safe. However, this work has several limitations: i) as it is based on constraints, the types do not reflect the structure of the deltas; ii) it presupposes that the order in which the deltas are applied on a core is totally specified; and iii) it generates a set of constraints per product, which means that the complexity is exponential in the number
of deltas. More recently, Lienhardt and Clarke proposed an approach \cite{59} that addresses the first of these limitations using row polymorphism \cite{75} to capture the structure of products and the semantics of deltas in the types. The underlying computational model takes a collection of classes as its basis and applies deltas in some order to update them. This chapter presents an extension of this approach to deal with the second limitation.

Thaker et al. \cite{86} describe an informally specified approach to the safe composition of software product lines that guarantees that no reference to an undefined class, method or variable will occur in the resulting products. The approach is presented modulo variability given in the feature model and deals especially with the resulting combinatorics. The lack of a comprehensive formal model of the underlying language and type system was rectified with Lightweight Feature Java (LFJ) \cite{27}. Underlying LFJ is a constraint-based type system whose constraints describe composition order, the uniqueness of fields and methods, the presence of fields and methods along with their types, and feature model dependencies.

A formal model of a feature-oriented Java-like language called Featherweight Feature Java (FFJ) \cite{5} presents a similar base language that also formalises Thaker et al.’s \cite{86} approach to safe composition, although for this system type checking occurs only on the generated product. Coloured Featherweight Java \cite{47}, which employs a notion of colouring of code analogous to, but more advanced than \#ifdefs, lifts type checking from individual products to the level of the product line and guarantees that all generated products are type safe. More recent work \cite{4} refines the work on FFJ, expressing code refinements as modules rather than low-level annotations. The resulting type system again works at the level of the product line and enjoys soundness and completeness results, namely, that a product line is well-typed if and only if all of its derived products are well-typed.

In the above mentioned work the refinement mechanisms are monotonic, so no method/class removal or renaming is possible. Kuhlemann et al. \cite{51} addresses the problem of non-monotonic refinements, though their approach does not consider type safety. They consider the presence of desired attributes depending upon which features are selected. Checking is implemented as an encoding into propositional formulas, which are fed into a SAT solver. Recent work addresses non-monotonic refinement mechanisms that can remove or rename classes and methods. An alternative approach due to Schaefer et al. \cite{79} generates detailed dependency constraints for checking delta-oriented software product lines. The checking of the constraint is performed per product, rather than at the level of product lines. This approach to typing delta-oriented programs is complementary to our work, providing part of the checking we have omitted.

A number of static analysis techniques have been developed for design models or code of software product lines. Heidenreich \cite{42} describes techniques for connecting feature models, solution-space models, and problem-space models, which is realised in the FeatureMapper tool. In this tool, models are checked for well-formedness against their meta-model. Similarly, Czarnecki and Pietroszek \cite{24} provide techniques for ensuring that no ill-structured instance of a feature-based model template will be generated from a correct configuration. Apel et al. \cite{6} present a general, language independent, static analysis framework for reference checking—checking which dependencies are present and satisfied. This is one of the key tasks of type checking a software product line. Similar ideas are applied in a language-independent framework for ensuring the syntactic correctness of all product line variants by checking only the product line itself, again without having to generate all the variants \cite{49}. Clarke et al. \cite{17} present an abstract framework for describing conflicts between code refinements and conflict resolution in the setting of delta-oriented programming. Their framework, Abstract Delta Modelling, considers only soft conflicts (otherwise hard conflicts did not cause an error), though the theory could easily encompass both. Padmanabhan and Lutz \cite{71} describe the DECIMAL tool, which performs a large variety of consistency checks on software product line requirements specifications, in particular, when a new feature is added to an existing system. Techniques developed for the analysis and resolution of interference of aspects in AOP \cite{19,36} address similar problems to analyses of software product line conflicts, but they do not consider variability.
4.3 Delta-Oriented Programming

In the rest of the chapter, we will use the term member for either a method or a field of a class. The syntax of our delta-oriented programming language is presented in Figure 4.3. The language represents a variation of the ABS language [1], abstracting away the actual definition of methods and omitting the feature model. Contrarily to ABS, in our language products are explicit selections of deltas instead of being selections of features that trigger associated deltas.

A product line \( PL \) is a classic object-oriented core program (i.e. a list of classes \( CL \)) extended by a sequence of element declarations \( PLE \). An element can either be a delta \( \text{delta } \delta \text{ after } DL \{\text{COL}\} \), where \( DL \) is used to construct the partial order between deltas and \( COL \) is the body of \( \delta \)\(^1\) or a product \( \text{product } p \{ DL \} \), where \( DL \) are the deltas to be applied to the core to produce \( p \).

\( CO \) and \( MO \) are operations on classes and members, respectively. It is possible to add, remove and modify both classes and members. The modification of a class is done with a sequence of operations on members \( MOL \), while the modification of members is not specified in our language as it only focuses on the manipulation on the structure of the cores, not their behaviour.

**Semantics.** The full semantics of the language is presented elsewhere [20, 58]. The computation of a specific product in our language, that is, the process where deltas given by a selected product are applied, is straightforward. The delta names in \( DL \) are sorted to match the order given by the keyword \( \text{after} \). When the order is not total, several sequences of deltas are possible, creating the possibility of conflicts. Then, the code of the deltas are applied in order to the core, thus computing the product.

4.4 Conflicts

Clarke et al. [17] define the notion of conflict for an abstract notion of delta, but they do not capture hard conflicts. This section proposes a more precise definition based on the notion of action.

Deltas and operations on deltas are encoded as elements and actions which perform the operations on elements.

\[
\begin{align*}
E & ::= c \mid c.m \\
A & ::= \bot \mid \text{add} \mid \text{rem} \mid \text{mod}
\end{align*}
\]

Given a class \( c \) or a member \( c.m \) (qualified with its class name \( c \)), a delta can either do nothing with it (\( \bot \)); add it (\( \text{add} \)); remove it (\( \text{rem} \)); or modify it (\( \text{mod} \)). A function \( \text{act} \), defined in Figure 4.4, encodes each operation described by \( CO \) and \( MO \) to a mapping from classes and members of classes \( E \) to actions \( A \), composed sequentially with the operator \( \triangleright \). The notation \( S \rightarrow A \) denotes a function that, when applied to a class or member \( s \in S \), returns \( A \) if \( s \in S \) and \( \bot \) otherwise. The composition operator \( \triangleright \) is defined for actions, and it is lifted to functions from elements to actions in a natural way: \( (f \triangleright g)(e) = f(e) \triangleright g(e) \).

The action of a product line \( PL \), written as \( \text{act}(PL) \), maps delta names from \( PL \) to their corresponding elements and actions, and is defined by combining the actions of all deltas in \( PL \).

Definition 4.4.1 below captures the two kinds of conflict, where \( A \) denote the set of all actions \( A \):

\(^1\)In our examples we write \( \text{delta } \delta \{\ldots\} \) as a shorthand for \( \text{delta } \delta \text{ after } e \{\ldots\} \).
that are analogous to the types of deltas but over types of members

\( \rho \)

variables do not have an inner structure. Deltas are typed with functional types

for members of classes, with the difference that it only records if a member is present or not, as members are not ordered.

the type of the delta \( \text{TML} \) is present and contains the field \( \text{coffee} \), the class \( \text{CMachine} \) is present as well, etc. The type of the delta is a little more subtle, as it is structured in two parts: what kind of program it expects as input (on the left hand side of \( \rightarrow \)), and what it will return after the modification (on the right hand side of \( \rightarrow \)).
Thus stops, pointing out the error it found. If the conflict is soft however, it may be solved by future deltas. If the conflict is hard, it corresponds to a possible error during the generation of the product, and the analysis thus stops, pointing out the error it found. If the conflict is soft however, it may be solved by future deltas.

The second step of the conflict detection works as follows: for each product of the product line, we take an element (resp. product) of the form $\gamma; (d, A)$, where $d$ is the name of the delta performing the operation, $A$ is the performed action, and $\gamma$ is a variable representing past actions done on $E$. Using type unification, sequential composition of operators on the same element $E$ result in annotations of the form $\gamma; (d, A_1); \ldots; (d, A_n)$, which are transformed using a rewriting relation into $\gamma; (d, A_1 \triangleright \ldots \triangleright A_n)$, corresponding to the action of $d$ on the element $E$.

The second step of the conflict detection works as follow: for each product of the product line, we take an element of the core program’s type, and apply to it the deltas for that product, in an order that validates the partial order defined by the programmer. During this application, the annotations on elements are compared using a function detect presented in Figure 4.7, together with the partial order, to look for soft and hard conflicts. We illustrate in Figure 4.7 how this second step detects the soft conflict (resp. hard conflict) in the product $p_a$ (resp. $p_b$) of Figure 4.1. For the product $p_a$, we focus our presentation on the method make of the class CMachine. Initially, this method is present and has not been manipulated by any delta: its annotation is thus $\bot$. When we apply Choco, which modifies that method, we look for possible conflicts: as not previous manipulation was performed, everything is fine and we just add the (Choco, mod) to the annotation. When we apply Sugar, which modifies that method, we look for possible conflicts: there was a previous modification on that method, done by Choco on which the user didn’t specify any order. We thus have a soft conflict. The principle of the analysis for $p_a$ is the same, except that when we detect the conflict, we see that Choco performed an addition on makeChoco, not a simple modification, and thus the conflict is hard, not soft.

After detecting a conflict, our analysis has a different behavior depending on the nature of the conflict. If the conflict is hard, it corresponds to a possible error during the generation of the product, and the analysis thus stops, pointing out the error it found. If the conflict is soft however, it may be solved by future deltas.
1. Core Program

\[ \text{make : Pre}_1; \]

2. Application of Choco

\[ \text{make : Pre}_{1,(\text{choco,mod})}; \]

3. Application of Sugar

\[ \text{make : Pre}_{1,(\text{choco,mod}),(\text{sugar,mod})}; \]

We have a soft conflict, because Choco and Sugar can commute and both modify make.

4. Core Program

\[ \text{makeChoco : Abs}_1; \]

5. Application of Choco

\[ \text{makeChoco : Pre}_{1,(\text{choco,add})}; \]

6. Application of ColourPrint

\[ \text{makeChoco : Pre}_{1,(\text{choco,add}),(\text{ColourPrint,mod})}; \]

We have a hard conflict, because Choco and ColourPrint can commute and one (Choco) does an action different from mod.

Conflict Detection in \( p_s \)

\[ \begin{align*}
\Phi & = \bot \Rightarrow (J = J' = \bot) \\
\Phi & = d, \gamma \Rightarrow (J = \gamma; (d, \text{add}) \land J' = \gamma)
\end{align*} \]

\[ \begin{align*}
\Phi & = T_1 \ f_1 \ \text{def}_1; \ldots; T_n \ f_n \ \text{def}_n; (f_1 : \text{Pre}_{f}; \ldots; f_n : \text{Pre}_{f}; \text{Abs}_f)
\end{align*} \]

Conflict Detection in \( p_h \)

\[ \begin{align*}
\Phi & = \bot \Rightarrow (J = \bot) \\
\Phi & = d, \gamma \Rightarrow (J = \gamma; (d, \text{add}) \land \gamma' = \gamma)
\end{align*} \]

\[ \begin{align*}
\Phi & = ML_i : \text{TML}_i; i \in 1..n \\
\Phi & = \bot \Rightarrow (J = \bot) \\
\Phi & = d, \gamma \Rightarrow (J = \gamma; (d, \text{add}) \land \gamma' = \gamma)
\end{align*} \]

\[ \begin{align*}
\Phi & = \text{class} \ c_1 \ \{ML_1\} \ldots \text{class} \ c_n \ \{ML_n\}; (c_1 : \text{Pre}_f(\text{TML}_1); \ldots; c_n : \text{Pre}_f(\text{TML}_n); \text{Abs})
\end{align*} \]

Figure 4.7: Conflict Detection of the two product from Figure 4.1

Figure 4.8: Typing Core Products

The annotation to capture this conflict is then transformed (in our example “\( \bot; (\text{Choco, mod}); (\text{Sugar, mod}) \)” becomes “\( \bot; (\text{Choco, Sugar}) \)” to show that we have a conflict between Choco and Sugar) before continuing the analysis.

Finally, to ensure the correctness of the conflict detection algorithm, past actions done on deleted members need to be remembered, even when the class itself has been deleted. This means that the type of the deletion of the class \( c \) should be able to identify each member \( m_i \) in \( c \), take their annotation \( J_i \), and specify that the output product is typed with \( c : \text{Abs}_J(\text{Abs}_{J_1}(\text{Abs}_{J_2}(\text{abs}_{J_3}(\ldots)))) \). The unification of the annotations \( J_i \) is achieved using local substitutions [61, 62], which allow conflict variables \( \gamma \) to be substituted locally into an element \( E \). For instance, it is possible to type the removal of class \( c \) with

\[ (c : \text{Pre}_{\gamma}(\rho_{\gamma}')) \rightarrow (c : \text{Abs}_{\gamma}(\text{abs}_{\gamma'}(\text{abs}_{\gamma'}))). \]

It is possible to type the application of this operator to a product with class \( c \) by first unifying \( \rho \) with the structure of \( c \), producing an input type of the form \( (c : \text{Pre}_{\gamma}(\text{abs}_{\gamma'}(\text{abs}_{\gamma'}))); \ldots; m_n : \text{Pre}_{\gamma}(\text{abs}_{\gamma'})). \) and then unifying each instance of variable \( \gamma' \) with the annotation local to each member.

In the rest of this chapter, we present the main rules of our type system, structured in three parts: classes and core products; operators on classes and products; and product lines, i.e., deltas, core definitions and products. A full presentation can be found in the original publication by Lienhardt and Clarke [67].

Core Products. The typing rules for core products are presented in Figure 4.8. The rule T:ML types the body of a class with a mapping stating that all the members of the class are present. The rule T:CL types a core product with a mapping stating that all the classes of the core are present, with their bodies typed with the previous rule.

Operators. A selection of the typing rules for operators are presented in Figure 4.9. For instance, the addition of a member \( m \) by a delta \( d \), typed with the rule T:ADDMEM, states that the operator expects: as
input, a class with the member \( m \) absent and annotated with a variable \( \gamma \) to capture manipulation done by previous deltas; as output, the same class with member \( m \) added (i.e., present) and with \( \langle d, add \rangle \) added to \( \gamma \), thus capturing the addition action performed by \( d \) after the previous manipulations stored in \( \gamma \).

**Product Lines.** The type rules for product lines are presented in Figure 4.10. The rule T:D types delta declarations by: i) computing the type \( TD \) of the delta’s body \( COL \); ii) computing the action of the delta: the statement \( Norm \vdash TD \) means that the annotations in \( TD \) have been completely normalised; and iii) continuing the typing of the product line with the environment \( \Gamma \) extended with a mapping between the delta and its type. The rule T:K types core declarations by typing the declared classes and continuing the typing of \( PL \) with the extended typing environment. The rule T:P types a product by typing the list of deltas \( DL \), ensuring that the core \( k \) is a valid input for \( DL \), and adding the type of the product to the rest of the mapping \( \Pi \). The rule T:NAME is used to type names, where \( n \) is either a core or a delta name. The type of a list of delta names \( DL \) is constructed using the last three typing rules. The list is typed inductively using T:DL-E for the empty list and T:DL-D to add new deltas to the list. The rule T:DL-D types the deltas in sequence, with the additional application of the function \( \text{detect}(d, TP_3) \) to detect conflicts added or resolved by \( d \) using the annotations in \( TP_3 \). The details of the \( \text{detect} \) function can be found in the paper by Lienhardt and Clarke [57].

The type system is sound, which means that a well-typed product line is conflict free.

### 4.6 Conclusion

This chapter presented a simple language for delta-oriented programming and defines notions soft and hard conflicts, a type system based on row polymorphism to capture errors and on a new concept of annotations to capture conflicts. This chapter also shows that, in contrast to what is suggested by Clarke et al. [17], the notion of conflict is not simple and accurately detecting them is not easy.
Chapter 5

Dynamic Modelling of Product Lines

The ABS programming language includes support for delta modelling as the core technology for implementing software product lines. This approach is static in that deltas are applied at compile time and therefore have no runtime representation. The work presented in this chapter addresses the DoW by employing deltas dynamically to represent a software product line whose behaviour changes at runtime. The work also introduces a cost model to guide the change of deltas at runtime.

The work described in this chapter was carried out by CWI in consultation with KUL, building on work of KUL and CTH on delta-oriented programming and abstract delta modelling. This prior work was reported in Deliverable D2.2b [30].

5.1 Introduction

This chapter gives an overview of a formalisation of dynamic product lines in the context of abstract delta modelling (ADM). We show how to transform a static product line, as described in [17, 18], into a dynamic product line. The dynamic product line takes the form of a Mealy Machine, a finite automaton with an input symbol and an output symbol on every transition. In our case, the input symbol corresponds to a feature that has been turned on or off and the output symbol corresponds to the delta that has to be applied to the current product to bring it up-to-date.

Based on this representation of dynamic product lines, we introduce a cost model. We assume that monitoring a specific feature for change has a certain cost, and that some features are more costly than others. We then describe how to optimise dynamic product lines by selectively removing transitions from them, effectively disregarding costly features until they become relevant.

We also introduce a novel case study to demonstrate this approach for a concrete domain. To illustrate both dynamic product lines and the versatility of ADM, we do not use a traditional software product line. Instead, we use a profile manager for modern mobile devices, such as smartphones; a common use case these days. By monitoring personal data such as time, location and schedule, a smartphone can automatically adjust its internal settings based on user-defined rules. We argue that delta modelling and dynamic delta modelling are a natural fit for this scenario.

5.2 Related Work

Delta Modelling [81, 76, 77] is designed as a technique for implementing software product lines [73]: a way to optimally reuse code between software products that differ only in the features they support. The code is divided into units called deltas. Incremental application of a specific set of deltas can mechanically transform a core product into a specific product from the product line. Each delta has an application condition that indicates for which combinations of features the delta should be applied. The legal combinations of features are expressed through a feature model [9]. Such legal combinations of features are commonly referred to as feature configurations. Clarke et al. [17, 18] described delta modelling in an abstract algebraic setting called
the Abstract Delta Modelling (ADM) approach. It gives a formal description of deltas and how they can be combined and linked to the feature model.

Traditionally, a feature configuration is chosen once at build-time. Its corresponding product is then generated and does not change at runtime. Dynamic (software) product lines (DPL) are product lines for which the feature configuration is not fixed at runtime. It can change dynamically, in order to meet changing requirements for continuously running systems. Hallsteinsen et al. introduce several properties they believe to constitute a dynamic software product line. The work described in this chapter allows several of these, such as ‘dynamic variability’, ‘changes binding several times over lifetime’ and ‘context awareness’, but does not yet model others, such as ‘variation point change during runtime’ and ‘deals with unexpected changes during runtime’. In our approach, even though the current feature configuration can change during runtime, the set of available feature configurations is still fixed at build time.

Dynamic product lines have already been discussed by Damiani and Schaefer in the context of delta modelling. Their work is developed in a concrete object-oriented setting, and complements our work. The transition systems described in this chapter can be enriched with reconfigurations, as described by Damiani and Schaefer, to obtain a variation of their reconfiguration automata. It is also possible to include their proposed reconfigure statement to ensure consistency during reconfiguration.

5.3 Example

The example used in this chapter comes from the realm of profile management on modern smartphones and other mobile devices. Modern smartphones and tablets, such as those based on Android, iOS or Windows Phone, have access to a great variety of data with regard to the current circumstances of their user: They know the current time and their current physical location. They know which application the user is currently running, what their scheduled appointments are, and much more. This sort of information can be used to automatically adjust the device settings based on user-defined rules, such as: “when my headphones are plugged in, play music” or “when my battery is running low, turn down screen brightness”. This is known as automated profile management. We show that delta modelling is a natural way to model such rules. A profile management application for Android which uses dynamic delta modelling is currently in the final stages of development (called Delta Profiles).

The idea behind the profile manager application is that the user manually creates a set of rules using the app’s graphical interface, after which it is put into effect as a dynamic product line, regulating the device’s profiles. We give an example set of rules:

- If I am within 1 km of \([-52° 21’ 23”, +4° 57’ 8”]\) and the current time is between 9:00 and 17:00 then
  - set ‘volume’ to 5.

- If I currently have a meeting scheduled then
  - set ‘volume’ to 0 and
  - set ‘foreground app’ to ‘meeting minutes’

and this rule has priority over the previous rule.

Note that we need to establish a priority between the two rules, as they might otherwise conflict with each other on the volume setting.

5.4 Dynamic Product Lines

A naive way to turn a static product line into a dynamic product line (DPL), i.e. to dynamically switch from one feature configuration to another and keep the current product consistent with that feature configuration is to generate the product in the traditional ‘static’ way each time the feature configuration changes. However,
this can be rather costly, and hurt performance. The other extreme is to pre-generate all products, and to continually switch between them. However, the number of possible products can be exponential in the number of features, so this is infeasible for non-trivial product lines.

Instead, we represent a DPL as a Mealy Machine. A Mealy Machine is a finite automaton with an input symbol and an output symbol on each transition [64]. We assume that in a DPL the feature configuration changes dynamically when individual features are sequentially turned on and off with respect to the current feature configuration. These features are used as input symbols for the Mealy Machine. The output symbols are deltas, which, when applied to the current product, yield a new product consistent with the new feature configuration.

A dynamic product line has to be generated only once, and can then be run indefinitely. Figure 5.1 shows a graphical representation of the dynamic product line generated from the example in Section 5.3. In this diagram, $l$, $t$, and $m$ represent the conditions on location, time, and scheduled meeting respectively. The various $d$ are deltas that can transform one profile into another.

We first describe the most straightforward strategy for ‘running’ a DPL. We assume that there is some target state, indicating the feature configuration we ‘want’ to be in. Before the Mealy Machine starts, we set the current state equal to the target feature configuration. The current product is then generated in the traditional static way.

When the machine is running we assume that, when in the current state, every feature that is accepted as input from there is monitored by the system. When that feature is turned on or off, an input symbol representing that feature is generated to make the current state equal to the target state again. And in that way a stream of input symbols is generated for the Mealy Machine, which returns the appropriate deltas to keep the product up to date.

## 5.5 Cost and Optimisation

We assume that occupying a state in a DPL has a cost: the cost of monitoring the features of the outgoing transitions for change. We posit that monitoring some features will be more expensive than monitoring others. For example, in Section 5.3 it will cost more power to continually monitor the current ‘GPS location’ ($l$) than it will to monitor the current ‘time’ ($t$).

Basically, we minimise the cost of a DPL by removing costly transitions, but only where this would not ‘break’ the machine. This means we only need to monitor features when they become relevant. For example,
in the profile manager from Section 5.3, we want to modify the profile when we are in a certain ‘GPS location’ (l) at a certain ‘time’ (t). Either condition satisfied on its own does not modify the profile. So it makes sense to only start monitoring ‘GPS location’ (the more costly factor), when it is already the right ‘time’.

We need to find conditions under which transitions may be removed, as well as an accompanying strategy for walking through a reduced DPL.

Two states are equivalent if they represent the same product (such as ∅, {t} and {l} from Figure 5.1). In theory it might be enough if a state from every equivalence class remains reachable from every other equivalence class. The accompanying strategy for walking through it would then have to be some systematic search algorithm in order to locate the new target state every time. However, we want configuration switches to be swift and a search would probably be too expensive. So we need some middle ground. We are going to reduce dynamic product lines only so far that we can still reach a target equivalent state by nondeterministically firing from a set of available transitions which is easily calculable. This implies at least that we may only remove transitions with ϵ output.

Figure 5.2 shows a reduced version of the dynamic product line of Figure 5.1 with equivalence classes marked (basically, wherever an ϵ transition was/is).

We would like to remove l/ϵ transitions first, then m/ϵ transitions, then t/ϵ transitions, as long as we can still reach every equivalence class by nondeterministic choice out of the available transitions. As can be seen in Figure 5.2 we were able to remove 10 transitions, significantly reducing the overall cost of the DPL. Intuitively, the transitions between ∅ and l could be removed, because the ‘GPS location’ does not become relevant until it is the right ‘time’. The other 8 transitions could be removed because d2 completely overwrites the effect of d1, so it does not matter what happens with t and l while we are in a scheduled meeting. Profile manager rules often look like this in practice, so the DPL can often be significantly optimised.

With an example walk in Figure 5.2 we show how we can always reach a state equivalent to the one targeted. We start in state $cfc = tfc = ∅$. Say we reach specified GPS coordinates before 9:00. Transition l would be fired, if it were available. As it is, $cfc = ∅$ and $tfc = \{l\}$. If we then start a scheduled meeting, m is fired, and $cfc = \{m\}$ and $tfc = \{l, m\}$. If the meeting ends (bringing us back to the previous situation), and it becomes 9:00, $cfc = tfc = \{t, l\}$, because t and l both fire. In all cases, we have $cfc \equiv tfc$, so we always reach the correct product.
5.6 Conclusion

Dynamic Delta Modelling is an extension of Abstract Delta Modelling [17, 18] which includes a formal framework for modelling dynamic product lines. Mealy Machines describe the behaviour of product lines with dynamic feature configurations, while remaining on an abstract level. We have defined a cost-model, and shown an optimisation opportunity for certain kinds of dynamic product lines. We have described the practical case-study of profile management on modern mobile devices directly in our formal framework, illustrating the versatility of ADM and the applicability of its dynamic extension.
Chapter 6

MetaABS

This chapter presents MetaABS, a meta-programming facility for the ABS language, and a dynamic Java back end that supports it. The purpose of MetaABS is to provide a unified interface for various runtime model analysis tasks that are being developed as part of Task 3.3. Adding meta-programming capabilities to ABS means that certain model analysis tasks can be encoded in ABS and carried out automatically while the model is executing.

We designed MetaABS based on requirements provided by several runtime analysis use cases. An example is supporting application-level scheduling in ABS. UIO reported (cf. Deliverable 2.1 [32]) on user-defined schedulers for Real-Time ABS [12], in which the developer specifies class and active object-specific scheduling functions using annotations. Previously, such models could execute only within the abstract ABS interpreter implemented in the Maude [21] rewrite engine. We implemented application-level scheduling at the level of COGs; schedulers are configured using MetaABS. Now, such models are executable on the standard Java VM.

MetaABS is also designed to enable models of dynamic software product lines in ABS. Work done in this package was mostly carried out by KUL (with the help of UKL on details of the back end implementation), which involved extensive work on implementing a new back end for ABS. In collaboration with FRG, KUL added support for runtime product reconfiguration, which is based on dynamic delta application. We further explored dynamic evolution of SPLs, where the feature model changes and code is removed or added to a model at runtime. These results will be reported in Deliverable 3.5. Other uses for MetaABS include the implementation of the ABS component model developed by BOL (cf. Chapter 2), and resource analysis using deployment components by UIO [32]. In the future we also plan to explore user-configurable method dispatch in ABS. An advantage of using MetaABS for these tasks, beyond having a “standard” interface for accessing model internals, is that it does not require extending the ABS language, by means of annotations, or otherwise.

6.1 Introduction

Meta-programming is generally understood as the ability to observe and modify the structure and behaviour of a program from within a program, either statically or at runtime. A meta-programming interface exposes basic elements of the programming language and the runtime environment to the programmer, enabling their inspection and modification. While it exposes these elements, it also abstracts away from their implementation.

Languages that support meta-programming commonly achieve this by providing reflection, that is, the ability of a program to inspect and modify itself at runtime. Thus the meta-program (the program transforming program) and the program that is transformed are the same. Reflection is decomposed into introspection, meaning the ability of a program to examine itself, and intercession, which enables a program to modify its state and behaviour. In other words, introspection and intercession provide, respectively, read and write access to elements of the language. For example, the Java Reflection API is a meta-programming interface
that provides methods to examine, and, to a very limited extent, modify the runtime properties of objects including their class, interfaces, fields and methods.

**MetaABS** comprises a set of operations (a *meta-object protocol*) that expose internals of ABS models, such as classes, methods, object state, concurrent object groups (COGs), task schedulers and message queues, making it possible to observe and modify a model while it is executed. Analysis tasks of particular interest within HATS include the scheduling of tasks inside concurrent object groups; the dynamic reconfiguration of software products; deployment component configuration; and runtime method dispatch.

Metaprogramming has been explored in ABS previously as part of the “Analysis” Task 1.3 (cf. Chapter 4 of Deliverable 1.3 [34]). The main difference with that work is that here the meta-language is ABS itself (using reflection), whereas in D1.3 it is Rascal. This provides a justification for this work: since reflection allows manipulating ABS programs at run-time, it is especially useful for evolvability.

The following Section 6.2 details the MetaABS API. In order to allow the modification of model elements, the support of the ABS back end is required. Therefore, in addition to adding introspection capabilities to the standard ABS Java back end, we design a so-called *dynamic Java back end*, which readily enables the modification of a model’s structural and behavioural elements. The ABS dynamic Java back end is presented in Section 6.3. Finally, Section 6.4 presents some applications of MetaABS for runtime model analysis.

### 6.2 MetaABS Interface

MetaABS is a largely object-oriented reflective interface to the ABS language. It provides an abstraction of the underlying ABS runtime. MetaABS is implemented as a library alongside the ABS standard library. It is easily extensible should new requirements arise. Extending MetaABS does not require changing the ABS language itself. This section lists the main types that MetaABS introduces and shows the provided operations in Figure 6.1.

#### 6.2.1 MetaABS Types

**Object Mirrors** An *ObjectMirror* reflects on an existing ABS object. One obtains an object mirror by invoking the built-in function `reflect(object)` on any given ABS object. The object mirror provides a set of reflective operations such as for getting or setting the object’s class and its concurrent group affiliation (cf. Figure 6.1). We opted for a mirror based design [13] in order to achieve a separation between an object’s regular interface, determined by its type, and its reflective interface.

**Object** *Object* is the type of ABS objects. One can use reflective operations on objects by first using the function `reflect(object)` and then calling a reflective operation on the returned *ObjectMirror*. *ObjectMirror* provides a `getObject()` method that returns the *Object* it reflects upon.

**Classes** A *Class* type represents an ABS class. Its interface includes operations to add and remove methods.

**Cog** A *Cog* represents a concurrent object group (COG), which in ABS is the unit of concurrency and distribution. A COG has a processor which runs at most one process at any given time, a queue of processes waiting to execute, and a process scheduler, which determines which process from the queue will run at each scheduling point. The process scheduler determines the scheduling policy and is configurable.

**Scheduler** Schedulers provide a `schedule()` method which returns a *Process* from the COG’s queue. It is possible to define a custom scheduler and attach it to a COG using `setScheduler()`.

#### 6.2.2 Usage Example

The following example illustrates how reflective operations are accessed from an object mirror.
6.3 A Dynamic Back End for ABS

A back end that fully implements the MetaABS API has been contributed to the ABS compiler tool chain. The key idea behind its design is to use dynamic structures in the target language (Java) to represent ABS language elements. The main difference from the standard ABS Java back end is how language elements are represented when translated to Java. Whereas the standard Java back end (cf. Chapter 2 of Deliverable 1.4 [33]) represents ABS classes, functions and data types as Java classes and ABS interfaces as Java interfaces, the dynamic Java back end uses Java (singleton) objects to represent interfaces, classes, methods, objects, object fields, COGs, data types, functions, etc. Such a representation trades execution performance for fully malleable ABS models. This section details the design of the dynamic ABS back end and illustrates the code generation process by examples.

6.3.1 Design

Figure 6.2 shows the main types used to represent ABS model elements in Java. When compiling a model using the dynamic ABS Java backend, MetaABS operations (Figure 6.1) are mapped to this interface.

ABS classes are represented as objects of type ABSDynamicClass, which provide operations to set or modify the class name, initialisation block (constructor), methods, fields and class parameters. Methods and constructors of classes are represented as objects of type ABSClosure. ABSClosure is an abstract class whose exec
method serves as a placeholder for each method’s specific behaviour. To create a method, a concrete subclass of **ABSClosure** overriding **exec** needs to be provided. Fields are represented as objects of type **ABSField**. Concrete fields inherit from **ABSField** and provide a specific initialisation expression by overriding the **init** method. **ABS** objects are instances of **ABSDynamicObject**, which offers an interface through which it is possible to modify their class and COG associations, update their fields and call methods. **ABSCog** objects are associated to objects and mainly control the scheduling of tasks. By modifying a COG’s **TaskScheduler** one can configure its scheduling policy, implemented as a **TaskSchedulingStrategy**.

### 6.3.2 Usage

To generate Java bytecode using the dynamic **ABS** Java back end, the **ABS** compiler is invoked using the `-dynamic` switch. For example, the following command generates Java code for the **ABS** program **PeerToPeer.abs** (provided as an example in Deliverable 1.2 [31]) into the `javagen` directory:

```bash
class PeerToPeer.abs
```

To execute the code generated from the PeerToPeer.abs example, one can use the following command line:

```bash
class PeerToPeer.Main
```

These tasks can be performed equally using the Eclipse IDE with the **ABS** plugin installed.

### 6.3.3 Code Generation

To illustrate code generation for the dynamic **ABS** Java back end, we show how a few simple **ABS** code examples (blue boxes) compile to Java using the dynamic Java back end (green boxes), and compare it to the code generated for the regular, static Java back end (grey boxes).

The example in Figure 6.3 shows the code generation process for class declarations. The generated static Java code is very similar to the original **ABS** code: the generated Java class **C_c** corresponds to **ABS** class **C**, and the generated method **ABSInteger getX()** corresponds to **Int getX()**. In the dynamic setting, **ABS** classes are represented as singleton instances of class **ABSDynamicClass**. The static method **C_c.singleton** (line 3)

Figure 6.2: Dynamic **ABS** Java back end interface (partial view)
creates an \texttt{ABSDynamicClass} object (line 5) and adds class C's fields and methods. These are represented as subclasses of \texttt{ABSField} and \texttt{ABSClosure}. For each method, a new class inheriting from \texttt{ABSClosure} is created, which, by overriding the \texttt{exec} method, encodes the method's specific behaviour. Similarly, \texttt{ABSField} is subclassed for each field, with the overriding \texttt{init} method defining the field's specific initialisation expression. Instances of these classes are passed as arguments to \texttt{addField} (line 6) and \texttt{addMethod} (line 8).

The example in Figure 6.4 shows object creation and method calling. In the static setting code generation is straightforward. In the dynamic setting, an object of the predefined class \texttt{ABSDynamicObject} is created with a reference to our \texttt{ABSDynamicClass} object representing class C (line 1). Calling C's method then amounts to calling the \texttt{dispatch} method on the \texttt{ABSDynamicObject} with the name of the method as an argument (line 2). The \texttt{dispatch} method returns a generic \texttt{ABSValue} that needs to be cast to the method's specific return type.

6.4 Applications

6.4.1 User-Defined Process Schedulers

Task scheduling in ABS is by default non-deterministic. The ABS Java compiler back end provides a flexible configuration mechanism to define the scheduling strategies that are used during the execution of an ABS system (cf. Chapter 7 of Deliverable 1.2 [31]). We make this configuration mechanism accessible at runtime for ABS models that are compiled to Java.

Real Time ABS introduced user defined schedulers as a means to control the scheduling policy at the level of active objects [12]: the user can define scheduling policies as functions that operate on a \texttt{queue} data type.
representing the available processes, and return a single process from that queue. Objects can be annotated with a dedicated scheduling function, which can also access the state of the object in question.

**MetaABS** provides operations to access the COG of a particular object, which can be assigned a user-defined scheduler object. User defined schedulers need to implement the `Scheduler` interface, which provides a `schedule` method that returns a `Process` from the (builtin) `queue` data type. Schedulers can be given access to program state through class parameters.

### 6.4.2 Dynamic Product Reconfiguration

**MetaABS** and a runtime model that supports runtime adaptation are useful for various scenarios in which the ABS model needs to change dynamically. One such scenario is modelling dynamic SPLs, which introduce the ability of reconfiguring products at runtime. Runtime reconfiguration is understood as the transformation of a product into another valid product defined by the SPL, all without the need to re-compile and deploy the system. Adding this facility to ABS complements the static product configuration that ABS has provided since Deliverable 1.2 [31]. Static product generation introduced support for configuring a particular SPL product at compile time by taking an ABS core model and a set of `delta modules` and “flattening” them to obtain an executable core ABS model of that single product. We add support for runtime product reconfiguration to ABS by adding a dynamic representation of delta modules and deferring the flattening process to the runtime. Product reconfiguration takes the runtime representation of a product and applies a set of dynamic deltas to obtain a different product of the SPL.

The design and implementation of Automatic Dynamic Product Reconfiguration into ABS is currently in progress and a more detailed report will be given in Deliverable 3.5.

### 6.5 Conclusion

This chapter presented **MetaABS**, a reflective interface for the ABS language, and a dynamic back end, both designed to facilitate tasks concerned with runtime model analysis and adaptation. Notable examples of such tasks are user-controlled scheduling policies and dynamic software product lines that enable dynamic product reconfiguration.
Chapter 7

Conclusion

This report describes the work done in Task 3.3 towards the hybrid analysis of evolving systems in the ABS language. Two different core approaches were developed to express evolution, one exploiting components and another exploiting a dynamic version of deltas. These were supported by combined static/dynamic analysis techniques and by a meta-programming language extension to ABS, called MetaABS.

The main contributions of this deliverable are:

- the ABS component model (Chapter 2) and adaptive group model (Chapter 3) that express the dynamic evolution of systems at two different levels of abstraction;
- a dynamic product lines formalism (Chapter 5) and a complimentary conflict detection type system (Chapter 4), both based on delta-modelling, which express and check the evolution of software product lines; and
- the MetaABS meta-language (Chapter 6) which provides a framework for adapting the runtime of the ABS language from within the language.

MetaABS provides a strong unifying framework for implementing many different extensions to ABS. Some have been done in the context of this task, namely, the dynamic application of delta modules and user-defined COG-specific schedulers, but others remain for future work (component configuration). In particular, dynamic reconfiguration of products with support for state transfer and synchronous object updates is being explored at the moment in Task 3.5 and will be reported in Deliverable 3.5.
Bibliography


[18] Dave Clarke, Michiel Helvensteijn, and Ina Schaefer. Abstract delta modeling. Accepted to Special Issue of MSCS, To appear.


[38] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. Design Patterns. Addison-Wesley, November 1994.


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.

COG Concurrent Object Group, the unit of parallelism in ABS.

Core ABS The behavioural functional and object-oriented core of the ABS modeling language

Compiler back end The functional entity of a compiler that is mainly concerned with generating code for a specific machine.

Delta Synonymous with delta module

Delta module A specification of modifications to core ABS language elements (classes, methods, interfaces, etc.)

Dynamic software product line (DSPL) A set of software products that can be adapted dynamically by adding and removing features.

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

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.

IDE Integrated Development Environment

Scheduling The act of choosing one of a set of processes for execution.

Software component A modelling abstraction reflecting the logical units of composition, which provides isolation, mobility, and data-flow reconfiguration capacities.

Software product A software systems with a well-defined set of features.

Software product reconfiguration The process of adding and removing features from a software system at runtime.

Software product line (SPL) A set of software products that share a number of core properties, and differ on other aspects.

Software product line engineering A development methodology for software product lines.
Appendix A

An Object Group-Based Component Model

The paper “An Object Group-Based Component Model” follows.
An Object Group-Based Component Model

Michaël Lienhardt, Mario Bravetti, and Davide Sangiorgi

Focus Team, University of Bologna, Italy
{lienhard,bravetti,davide.sangiorgi}@cs.unibo.it

Abstract. Dynamic reconfiguration, i.e. changing at runtime the communication pattern of a program is challenging for most programs as it is generally impossible to ensure that such modifications won’t disrupt current computations. In this paper, we propose a new approach for the integration of components in an object-oriented language that allows safe dynamic reconfiguration. Our approach is built upon futures and object-groups to which we add: i) output ports to represent variability points, ii) critical sections to control when updates of the software can be made and iii) hierarchy to model locations and distribution. These different notions work together to allow dynamic and safe update of a system. We illustrate our approach with a few examples.

1 Introduction

Components are an intuitive tool to achieve unplanned dynamic reconfigurations. In a component system, an application is structured into several distinct pieces called components. Each of these components has dependencies towards functionalities located in other components; such dependencies are collected into output ports. The component itself, however, offers functionalities to the other components, and these are collected into input ports. Communication from an output port to an input port is possible when a binding between the two ports exists. Dynamic reconfiguration in such a system is then achieved by adding and removing components, and by replacing bindings. Thus updates or modifications of parts of an application are possible without stopping it.

Related Work. While the idea of components is simple, bringing it into a concrete programming language is not easy. The informal description of components talks about the structure of a system, and how this structure can change at runtime, but does not mention program execution. As a matter of fact, many implementations of components [1, 3, 5, 15, 2, 11, 13] do not merge into one coherent model i) the execution of the program, generally implemented using a classic object-oriented language like Java or C++, and ii) the component structure, generally described in an annex Architecture Description Language (ADL). This approach makes it simple to add components to an existing standard program. However, unplanned dynamic reconfigurations become hard, as

* Partly funded by the EU project FP7-231620 HATS.
it is difficult to express modifications of the component structure using objects (since these are rather supposed to describe the execution of the programs). For instance, models like Click [13] do not allow runtime modifications while OSGi [1] allows addition of new classes and objects, but no component deletions or binding modifications. In this respect, a more flexible model is Fractal [3], which reifies components and ports into objects. Using an API, in Fractal it is possible to modify bindings at runtime and to add new components; Fractal is however rather complex, and it is informally presented, without a well-defined model.

Formal approaches to component models have been studied e.g., [4, 8, 14, 12, 10, 9]. These models have the advantage of having a precise semantics, which clearly defines what is a component, a port and a binding (when such a construct is included). This helps understanding how dynamic reconfigurations can be implemented and how they interact with the normal execution of the program. In particular, Oz/K [10] and COMP [9] propose a way to integrate in a unified model both components and objects. However, Oz/K has a complex communication pattern, and deals with adaptation via the use of passivation, which, as commented in [7], is a tricky operator — in the current state of the art it breaks most techniques for behavioral analysis. In contrast, COMP offers support for dynamic reconfiguration, but its integration into objects appears complex.

Our Approach. Most component models have a notion of component that is distinct from the objects used to represent the data and the main execution of the software. The resulting language is thus structured in two different layers, one using objects for the main execution of the program, one using components for the dynamic reconfiguration. Even though such separation seems natural, it makes difficult the integration of the different requests for reconfiguration into the program’s workflow. In contrast, in our approach we tried to have a uniform description of objects and components. In particular, we aim at adding components on top of the Abstract Behavioral Specification (ABS) language [6], developed within the EU project HATS. Core ingredients of ABS are objects, futures and object groups to control concurrency. Our goal is to enhance objects and object groups with the basic elements of components (ports, bindings, consistency and hierarchy) and hence enable dynamic reconfigurations.

We try to achieve this by exploiting the similarities between objects and object groups with components. Most importantly, the methods of an object closely resemble the input ports of a component. In contrast, objects do not have explicit output ports. The dependencies of an object can be stored in internal fields, thus rebinding an output port corresponds to the assignment of a new value to the field. Objects, however, lack mechanisms for ensuring the consistency of the rebinding. Indeed, suppose we wished to treat certain object fields as output ports: we could add methods to the object for their rebinding; but it would be difficult, in presence of concurrency, to ensure that a call to one of these methods does not harm ongoing computations. For instance, if we need to update a field (like the driver of a printer), then we would want to wait first that all current execution using that field (like some printing jobs) to finish first. This way we ensure that the update will not break those computations.
In Java, such consistency can be achieved using the *synchronized* keyword, but this solution is very costly as it forbids the interleaving of parallel executions, thus impairing the efficiency of the program. In ABS, object groups offer a mechanism for consistency, by ensuring that there is at most one task running in an object group. This does ensure some consistency, but is insufficient in situations involving several method calls. A further difference between objects and components is that only the latter talks about *locations*. Locations structure a system, possibly hierarchically, and can be used to express dynamic addition or removal of code, as well as distribution of a program over several computers.

To ensure the consistent modifications of bindings and the possibility to ship new pieces of code at runtime, we add four elements to the ABS core language:

1. A notion of output port distinct from the object’s fields. The former (identified with the keyword *port*) corresponds to the objects’ dependencies and can be modified only when the object is in a *safe* state, while the latter corresponds to the inner state of the objects and can be modified with the ordinary assignments.
2. The possibility of annotating methods with the keyword *critical*: this specifies that the object, while an instance of the method is executing, is not in a safe state.
3. A new primitive to wait for an object to be in a safe state. Thus, it becomes possible to wait for all executions using a given port to finish, before rebinding the port to a new object.
4. A hierarchy of locations. Thus an ABS program is structured into a tree of locations that can contain object groups, and that can move within the hierarchy. Using locations, it is possible to model the addition of new pieces of code to a program at runtime. Moreover, it is also possible to model distribution (each top-level location being a different computer) and code mobility (by moving a sub-location from a computer to another one).

The resulting language remains close to the underlying ABS language. Indeed, the language is a conservative extension of ABS (i.e., an ABS program is a valid program in our language and its semantics is unchanged), and, as shown in our following example, introducing the new primitives into an ABS program is simple. In contrast with previous component models, our language does not drastically separate objects and components. Three major features of the informal notion of component — ports, consistency, and location — are incorporated into the language as follows: (i) output ports are taken care of at the level of our enhanced objects; (ii) consistency is taken care of at the level of object groups; (iii) the information about locations is added separately.

We believe that the separation between output ports and fields is meaningful for various reasons:

- Output ports represent dependencies of an object towards its environment (functionalities needed by the object and implemented outside it, and that moreover might change during the object life time). As such they are logically different from the internal state of the object (values that the object may have to consult to perform its expected computation).
• The separation of output ports allows us to have special constructs for them. Examples are the constructs for consistency mentioned above. Moreover, different policies may be used for updating fields and output ports. For instance, in our model while a field of an object \( o \) may be updated only by \( o \), an output port of \( o \) may be modified by objects in the same group as \( o \). This difference of policy is motivated in Section 3.1.

• The separation of output ports could be profitable in reasoning, in particular in techniques of static analysis.

• The presence of output ports may be useful in the deployment phase of a system facilitating, for instance, the connection to local communication resources.

Roadmap. §2 describes the core ABS language. §3 presents our extension to the ABS language. §4 presents the semantics of the language. The main features of core ABS and our extensions are illustrated along the document with several examples.

2 Core ABS

We present in Figure 1 the object core of the ABS language. For the full description of the language, including its functional aspect, see [6]. We assume an overlined element to be any finite sequence of such element. A program \( P \) is defined as a set of interface and class declarations \( I \) and \( C \), with a main function \( \{ T x; s \} \). The production \( T \) types objects with interface names \( I \) and futures with future types \( \text{Fut}(T) \), where \( T \) is the type of the value returned by an asynchronous method call of the kind \( e!m(e) \) (versus \( e.m(e) \) representing synchronous calls): the actual value of a future variable can be read with a \( \text{get} \). An interface \( I \) has a name \( I \) and a body declaring a set of method headers \( S \). A class \( C \) has a name \( C \), may implement several interfaces, and declares in its body its fields with \( F \) and its methods with \( M \). In the following examples: for simplicity we will omit “?” in await guards (in ABS “e?” guards are used for expressions “e” returning a future, instead simple “e” guards are used for boolean expressions) and we will follow the ABS practice to declare the class constructor like a method, named \( \text{init} \).

\[
P ::= I P \mid C P \mid \{ T x; s \} \quad F ::= T x
\]
\[
T ::= I \mid \text{Fut}(T) \quad S ::= T m(T x)
\]
\[
I ::= \text{interface } I \{ S \} \quad M ::= S\{ T x; s \}
\]
\[
C ::= \text{class } C(Tx) [\text{implements } I] \{ F M \}
\]
\[
s ::= \text{skip} \mid s; s \mid e \mid x = e \mid \text{await}(g) \mid \text{if } e \{ s \} \text{ else } \{ s \}
\]
\[
e ::= v \mid x \mid \text{this} \mid \text{new } [\text{cog}] C(\overline{e}) \mid e.m(\overline{e}) \mid e!m(\overline{e}) \mid \text{get}(e)
\]
\[
v ::= \text{null} \mid \text{true} \mid \text{false} \mid 1 \mid \ldots
\]
\[
g ::= e \mid e? \mid g \land g
\]

Fig. 1. Core ABS Language
Object Groups and Futures. One of the main features of ABS is its concurrency model which aims to solve data races. Objects in ABS are structured into different groups called cogs which are created with the new cog command. These cogs define the concurrency of a program in two ways: i) inside one cog, at most one object can be active (i.e. execute a method); ii) all cogs are concurrent (i.e. all method calls between objects of different cogs must be asynchronous). Concurrency inside a cog is achieved with cooperative multitasking using the await statement, and synchronization between concurrent executions is achieved with the await and get statements, based on futures.

We illustrate this concurrency model with a simple class Printer in Figure 2, modeling a printer driver with a job queue stored in a Status s. The principle of the print method of Printer is as follow: i) the printing request is added to the queue of jobs, which returns the identifier for that new job; ii) the method waits until all previous jobs have been processed; iii) the method does the actual printing (using the method printPhy) and waits for its completion, which returns a code describing if the printing was successful or not; and iv) the job is removed from the queue and the code is returned to the user.

3 Component Model

3.1 Ports and Bindings

The ABS concurrency model as it is cannot properly deal with runtime modifications of a system, in particular with unplanned modifications. Let us consider the client presented in Figure 3. This class offers a little abstraction over the Printer class with three extra features: i) the possibility to change printer; ii) some notification messages giving the current status of the printing job (count being the identifier of the job); and iii) the possibility to get the number of jobs handled by this object.
This class is actually erroneous: let us consider the scenario where a printing job is requested, followed by the modification of the printer. The `print` method sends the job to the first printer $p_1$, then waits for the notification from $p_1$’s status. While waiting, the printer gets modified into $p_2$: the following requests will fail as they will be directed to $p_2$ and not $p_1$. A possible solution would be to forbid the interleaving of different methods execution by replacing the `await`s by `get`s, which corresponds to the `synchronized` in Java.

We overcome this inconsistency problem by forbidding the modification of the field $p$ while it is in use. For this, we combine the notions of output port (from components) and of critical section. Basically the field $p$, which references an external service that can change at runtime, is an output port; the `print` method that needs stability over this port, creates a critical section to avoid the modification of $p$ while it is executing; the `count` field and the `GetNumberOfJobs` method, that have no link to an external service, remain unchanged.

The syntax for our manipulation of output port and critical section is as follows.

$$
F ::= \ldots \mid \text{port } T f
$$

$$
S ::= \ldots \mid \text{critical } T m(T_x)
$$

$$
s ::= \ldots \mid \text{rebind } e.x = e
$$

$$
g ::= \ldots \mid \|e\|
$$

Here, a field can be annotated with the keyword `port`, which makes it an output port, supposedly connected to an external service that can be modified at
class PrintClient {
    port Printer p;
    int count;

    void setPrinter(Printer pr) {
        await (∥this∥);
        rebind p = pr
    }

    critical void print(File f) { ... }

    int GetNumberOfJobs() { return count; }

    void init() { count = 0 }
}

Fig. 4. An improved Printing Client

runtime. Moreover, methods can be annotated with the keyword critical, which ensures that, during the execution of that method, the output ports of the object will not be modified.

Output ports differ from ordinary fields in two aspects:

1. output ports cannot be freely modified. Instead one has to use the rebind statement that checks if the object has an open critical section before changing the value stored in the port. If there are no open critical sections, the modification is applied; otherwise an error in a form of a dead-lock is raised;
2. output ports of an object o can be modified (using the rebind statement) by any object in the same object-group of o. This capacity is not in opposition to the classic object-oriented design of not showing the inner implementation of an object: indeed, a port does not correspond to an inner implementation but exposes the relationship the object has with independent services. Moreover, this capacity helps achieving consistency as shown in the next examples.

Finally, to avoid errors while modifying an output port, one should first ensure that the object has no open critical sections. This is done using the new guard ∥e∥ that waits for the object e not to be in critical section. Basically, if an object o wants to modify output ports stored in different objects o_i, it first waits for them to close all their critical section, and then can apply the modifications using rebind.

3.1.1 Examples
Printing Client. In Figure 4 we show how to solve our previous example (from Figure 3). The changes are simple: i) we specify that the field p is a port; ii) we annotate the method print with critical (to protect its usage of the port p); and iii) we change the method setPrinter that now waits for the object to be in a consistent state before rebinding its output port p.
class OperatorFrontEnd {
    port Operator _op;

critical Document modify(Document doc) { ... }

void init(Operator op) { rebind _op = op; }
}

class WFController {
    port Document _doc;
    port Printer _p;
    OperatorFrontEnd _opfe;

critical void newInstanceWF() { ... }

void changeOperator(Operator op) {
    await (∥this∥ ∧ ∥_opfe∥);
    rebind _opfe._op = op;
}

void init(Document doc, Operator op, Printer p) {
    rebind _doc = doc;
    rebind _p = p;
    _opfe = new OperatorFrontEnd(op);
}
}

Fig. 5. Dynamic Reconfiguration Example

Workflow Controller. For the purpose of this example, we suppose we want to define a workflow that takes a document (modeled by an instance of the class Document), modifies it using an Operator and then sends it to a Printer. We suppose that the protocol used by Operator objects is complex, so we isolate it into a dedicated class. Finally, we want to be able to change protocol at runtime, without disrupting the execution of previous instances of the workflow. Such a workflow is presented in Figure 5.

We thus have two classes: the class OperatorFrontEnd implements the protocol in the method modify; the class WFController encodes the workflow. The elements _op, _doc and _p are ports, and correspond to dependencies to external resources. In consequence they are annotated as port. It is only possible to modify their value using the construct rebind, which checks if the object is in a safe state (no critical method in execution) before modifying the port. Moreover, methods modify and newInstanceWF make use of these ports in their code, and are thus annotated as critical as it would be dangerous to rebind ports during their execution.

The key operations of our component model are shown in the two lines of code describing the method changeOperator. First is the await statement, which waits for the objects this and _opfe to be in a safe state. By construction, these objects
are in a safe state only when there are no running instances of the workflow: it is then safe to modify the ports. Second is the \texttt{rebind} statement; the statement will succeed since the concurrency model of object-groups ensures that no workflow instance can be spawned between the end of the \texttt{await} and the end of the method. Moreover, the second line shows that it is possible to rebind a port of another object, provided that this object is in the same group as the one doing the rebinding.

3.2 Locations

The final layer of our language introduces \textit{locations} that are used to model the different elements of our virtual office, like printers, computers, rooms and buildings. The idea is that components stand at a certain location. Thus every location, e.g. a room, is endowed with its own resources/services, e.g. printers, scanners, etc..., and a worker computer that stands at a certain location may exploit the location information to use resources at the same location.

Locations themselves are structured into trees according to a sublocation relation, such that we can have several locations at the top level (roots of trees) and object groups can only occur as leaves of such trees (and not as intermediate nodes).

We modify slightly the syntax of our previous calculus to introduce locations in it. We use \( l \) to represent location names. We represent with \((l, g)\) and \((l, l')\) the father-to-son sublocation relation where object groups can only appear as leaves of such trees (and not as intermediate nodes).

The additions are presented as follows.

\[
\begin{align*}
  s & ::= \ldots | \text{move } e \text{ in } e \\
  e & ::= \ldots | \text{new loc}
\end{align*}
\]

First, we add the possibility to create a new location (with a fresh name \( l \)) with a command \texttt{new loc}, then we add the possibility of modifying the father of a location/group \( n \) returned by an expression (or to establish a father in the case \( n \) does not possess one, or to remove the father of \( n \)) with the command \texttt{move } \texttt{n in } \texttt{l}_\perp: \texttt{the new father becomes the location } \texttt{l}_\perp \texttt{(returned by an expression). Technically, we also introduce a new type for location values, called location, which is added to the syntax of types } T.

3.2.1 Examples

In the Virtual Office case study we use locations to express the movement of a worker from a location to another one. The worker moves with his laptop, in which we suppose a workflow document has been previously downloaded. The worker component has a set of output ports for connection to the services at the current worker location, which are needed to execute the downloaded workflow. Therefore the worker movement from a location to another one requires rebinding all such output ports, which can only be done if the workflow (a critical method) is not executing. Therefore, compared to previous examples, we need to model simultaneous rebinding of multiple output ports.
**Example 1.** We represent the movement of a worker to a different environment as the movement of the worker to a new location, which includes:

- a set of object groups representing the devices that the worker needs to perform the workflow (here represented by services "ServiceA" and "ServiceB")
- possibly, a local registry component, providing to the worker laptop component the links to the devices above; this will be modeled in Example 2.

More precisely, whenever the worker moves to a location $l$, first we wait for possible current workflow executions to be terminated, then we rebind to the (possibly discovered, see Example 2) new devices in the new location.

We represent the worker component as an object group composed by two objects:

- a “ServiceFrontEnd” object endowed by all the required output ports (here ports “a” and “b” for services “ServiceA” and “ServiceB”, respectively),
- a “manager” object, called “WorkerFrontEnd” which: changes the ports in the “ServiceFrontEnd” object (possibly performing the service discovery enquiring the local service registry, see Example 2).

Finally, in the example code below, we make use of a primitive function “group” which is supposed to yield the group of a given object.

```java
class ServiceA { ... }
class ServiceB { ... }

class ServiceFrontEnd {
    port ServiceA a;
    port ServiceB b;
    critical void workflow() { ... }
}
class WorkerFrontEnd {
    ServiceFrontEnd s;

    void changeLocation(location l2, ServiceA a2, ServiceB b2) {
        await ||s||;
        move group(this) in l2;
        rebind s.a = a2;
        rebind s.b = b2;
    }

    void init(location l, ServiceA a, ServiceB b) {
        move group(this) in l;
        s = new ServiceFrontEnd();
        rebind s.a = a;
        rebind s.b = b;
    }
}
```
Example 2. In this example we also model the local registry component for each location, providing links to the local devices for the worker component, and the global root registry (which has a known address) which, given a location, provides the link to the local register at that location.

More precisely, whenever the worker moves to a location \( l \), first we have a discovery phase via a global root register so to obtain the local registry at location \( l \), then we wait for possible current workflow executions to be terminated, then a discovery phase via the registry component of the new location, and finally a rebinding to the discovered devices in the new location.

```java
class ServiceA { ... }
class ServiceB { ... }

class Register {
    ServiceA discoverA() { ... }
    ServiceB discoverB() { ... }
}

class RootRegister {
    Register discoverR(location \( l \)) { ... }
}

class ServiceFrontEnd {
    port ServiceA a;
    port ServiceB b;
    critical void workflow() { ... }
}

class WorkerFrontEnd {
    RootRegister rr;
    ServiceFrontEnd s;

    void changeLocation(location \( l_2 \)) {
        Fut<Register> fr=rr!discoverR(l2); await(fr); Register r=get(fr);
        await ||s||;
        move group(this) in \( l_2 \);
        rebind s.a = get(r!discoverA());
        rebind s.b = get(r!discoverB());
    }

    void init(location \( l \), RootRegister rr2) {
        rr = rr2;
        Fut<Register> fr=rr!discoverR(l); await(fr); Register r=get(fr);
        move group(this) in \( l \);
        s = new ServiceFrontEnd();
        rebind s.a = get(r!discoverA());
        rebind s.b = get(r!discoverB());
    }
}
```
4 Semantics

We present in this section the semantics of our language. Our semantics is described as a virtual machine based on i) a runtime syntax that extends the basic language; ii) some functions and relations to manipulate that syntax; and iii) a set of reduction rules describing the evolution of a term.

4.1 Runtime Syntax

The runtime syntax consists of the language extended with constructs needed for the computations, like the runtime representation of objects, groups, and tasks. Figure 6 presents the global runtime syntax. Configurations \( N \) are sets of classes, interfaces, objects, concurrent object groups (cogs), futures, invocation messages and hierarchy statements between components. The associative and commutative union operator on configurations is denoted by a whitespace and the empty configuration by \( \epsilon \). An object is a term of the form \( \text{ob}(o, \sigma, K_{idle}, Q) \) where \( o \) is the object’s identifier, \( \sigma \) is a substitution representing the object’s fields, \( K_{idle} \) is the active task of the object (or \( K_{idle} = \text{idle} \), when the object is idle and it is not executing anything), and \( Q \) is the queue of waiting tasks (the union of such queue, denoted by the whitespace, is associative with \( \epsilon \) as the neutral element). A cog is a term of the form \( \text{cog}(c, o_\epsilon) \) where \( c \) is the cog’s identifier, \( o_\epsilon \) is either \( \epsilon \), which means that there is nothing currently executing in the cog, or an object identifier, in which case there is one task of the object \( o \) executing in \( c \). A future is a pair of the name of the future \( f \) and a place \( v_\perp \) where to store the value computed for this future. An invocation message \( \text{invoc}(o, f, m, v) \) specifies that some task called the method \( m \) on the object \( o \) with the parameters \( \overline{v} \), this call corresponding to the future \( f \). An hierarchy statement \( (\gamma_\perp, \gamma) \) states that the component \( \gamma \) is a child of the component \( \gamma_\perp \) (\( \perp \) being the name of the top level component). A task \( K \) consists of a pair with a substitution \( \sigma \) of local variable bindings, and a statement \( s \) to execute. A substitution \( \sigma \) is a mapping from variable names to values. For convenience, we associate the declared type of the variable with the binding, and, in case of substitutions directly included in objects, we also use substitutions to store, the “this” reference, the class, the cog of an object and an integer denoted by \( \text{nb}_{cr} \) which, as we will see, will be used for critical section management. Finally, we extend the values \( v \) with object and future identifiers.

\[
\begin{align*}
N &::= \epsilon \mid I \mid C \mid NN \mid \text{ob}(o, \sigma, K_{idle}, Q) \mid \text{cog}(c, o_\epsilon) \mid \text{fut}(f, v_\perp) \mid \text{invoc}(o, f, m, v) \mid (\gamma_\perp, \gamma) \\
Q &::= \epsilon \mid K \mid QQ \\
K &::= \{ \sigma, s \} \\
v &::= \text{null} \mid o \mid f \mid 1 \mid \ldots \\
\sigma &::= \epsilon \mid \sigma; T \ x \ v \\
\gamma_\perp &::= \gamma \mid \perp
\end{align*}
\]

Fig. 6. Runtime Syntax; here \( o, f \) and \( c \) are object, future, and cog names
4.2 Reduction Relation

The semantics of the component model is an extension of the semantics of core ABS in [6]. It uses a reduction relation \( \rightarrow \) over configurations, \( N \rightarrow N' \) meaning that, in one execution step, the configuration \( N \) can evolve into \( N' \). We extend that relation in four different aspects. First, we extend the reduction definition with three reduction rules that define the semantics of the \texttt{Rebind} and \texttt{subloc} operator.

\begin{align*}
\text{Rebind-Local} & \quad \sigma(\text{nb}_{cr}) = 0 \\
& \quad \text{ob}(o, \sigma, \{ \sigma', \text{rebind} \ o.f = v; \ s \ }, Q) \rightarrow \text{ob}(o, \sigma[f \mapsto v], \{ \sigma', \ s \ }, Q)
\end{align*}

\begin{align*}
\text{Rebind-Global} & \quad \sigma(o(\text{nb}_{cr}) = 0) \quad \sigma(o(\text{cog}) = \sigma(o'(\text{cog})) \\
& \quad \text{ob}(o, \sigma, K_{\text{idle}}, Q) \rightarrow \text{ob}(o, \sigma[f \mapsto v], K'_{\text{idle}}, Q)
\end{align*}

\begin{align*}
\text{Loc-Move} & \quad (\gamma_{\perp}, \gamma) \text{ob}(o, \sigma, \{ \sigma', \text{move} \ \gamma \ \text{in} \ \gamma_{\perp}; \ s \ }, Q) \rightarrow (\gamma'_{\perp}, \gamma) \text{ob}(o, \sigma, \{ \sigma', \ s \ }, Q)
\end{align*}

The rule \texttt{Rebind-Local} is applied when an object rebinds one of its own ports. The rule first checks that the object is not in a critical section by testing the special field \texttt{nb}_cr for zero and then updates the value of the field. The rule \texttt{Rebind-Global} is applied when an object rebinds a port of another object and is similar to the previous one. The rule \texttt{Loc-Move} moves a location \( \gamma \) (initially put inside the location \( \gamma_{\perp} \)) inside another location \( \gamma'_{\perp} \).

The second aspect of our extension defines the semantics of our new expression, the creation of location \texttt{new loc}. In [6], the reduction rules defining the semantics of expressions are written using statements of the form \( \sigma \vdash e \rightarrow \sigma \vdash e' \) to say that in the context \( \sigma \) mapping some variables to their values, \( e \) reduces to \( e' \). Because expression \texttt{new loc} has a side effect (adding the new location to the configuration), we extend this statement to include the configuration: \( N, \sigma \vdash e \rightarrow N', \sigma \vdash e' \).

\begin{align*}
\text{New-Location} & \quad \gamma \text{fresh} \\
& \quad N, \sigma \vdash \text{new loc} \rightarrow N' (\perp, \gamma), \sigma \vdash \gamma
\end{align*}

That rule simply states that the \texttt{new loc} commands creates a new location and returns it.

The third aspect of our extension concerns method call. In our system, we indeed have two kinds of methods: normal ones and critical ones, the second ones creating a critical section on the callee. We model opened critical sections with the special hidden field \texttt{nb}_{cr}, that is initialized to zero, incremented each time a critical section is opened, and decremented each time a critical section is closed. Then, when an object calls a method, it creates an \texttt{invoc} message describing who is the callee, the method to execute, the parameters and the
return future. This message is then reduced into a task in the queue of the callee using the function bind that basically replaces the method by its code. To give the semantics of our critical methods, we extend this bind function to add, to the code of a critical method, some statements that manipulate the \( \text{nb}_{\text{cr}} \) field.

The rule NM-BIND corresponds to the normal semantics of the bind function, while the rule CM-BIND is the one used to bind a critical function. Basically, the first thing a critical method does is to increment the field \( \text{nb}_{\text{cr}} \), opening the critical section, and the last thing it does is to decrement the field, thus closing it.

Finally, the last aspect of our extension concerns our guard extension \( \| e \| \).

These two rules simply state that, when the object \( o \) has its field \( \text{nb}_{\text{cr}} \) different from zero, it has a critical section opened.

4.3 Properties

Important properties that show the adequateness of our machinery for port rebinding are: (i) we never modify a port while being in a critical section (this property is a consequence of the reduction rule Rebind: the execution of the rebind expression can only occur when the object’s lock is 0) and (ii) when \( \text{wait} \) statements are not used in between, modification of several ports is atomic (due to cooperative concurrency in the object group model): this can be used, like in the second example of the location extension, to ensure consistency.

References


Appendix B

A Type-Safe Model of Adaptive Object Groups

A Type-Safe Model of Adaptive Object Groups

Joakim Bjørk  
University of Oslo, Norway  
joakimbj@ifi.uio.no

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

Einar Broch Johnsen  
University of Oslo, Norway  
einarj@ifi.uio.no

Olaf Owe  
University of Oslo, Norway  
olaf@ifi.uio.no

Services are autonomous, self-describing, technology-neutral software units that can be described, published, discovered, and composed into software applications at runtime. Designing software services and composing services in order to form applications or composite services requires abstractions beyond those found in typical object-oriented programming languages. This paper explores service-oriented abstractions such as service adaptation, discovery, and querying in an object-oriented setting. We develop a formal model of adaptive object-oriented groups which offer services to their environment. These groups fit directly into the object-oriented paradigm in the sense that they can be dynamically created, they have an identity, and they can receive method calls. In contrast to objects, groups are not used for structuring code. A group exports its services through interfaces and relies on objects to implement these services. Objects may join or leave different groups. Groups may dynamically export new interfaces, they support service discovery, and they can be queried at runtime for the interfaces they support. We define an operational semantics and a static type system for this model of adaptive object groups, and show that well-typed programs do not cause method-not-understood errors at runtime.

1 Introduction

Good software design often advocates a loose coupling between the classes and objects making up a system. Various mechanisms have been proposed to achieve this, including programming to interfaces, object groups, and service-oriented abstractions such as service discovery. By programming to interfaces, client code can be written independently of the specific classes that implement a service, using interfaces describing the services as types in the program. Object groups loosely organize a collection of objects that are capable of addressing a range of requests, reflecting the structure of real-world groups and social organizations in which membership is dynamic [18]; e.g., subscription groups, work groups, service groups, access groups, location groups, etc. Service discovery allows suitable entities (such as objects) that provide a desired service to be found dynamically, generally based on a query on some kind of interface. An advantage of designing software using these mechanisms is that the software is more readily adaptable. In particular, the structure of the groups can change and new services can be provided to replace old ones. The queries to discover objects are based on interface rather than class, so the software implementing the interface can be dynamically replaced by newer, better versions, offering improved services.

This paper explores service-oriented abstractions such as service adaptation, discovery, and querying in an object-oriented setting. Designing software services and composing services in order to form...
applications or composite services require abstractions beyond those found in typical object-oriented programming languages. To this end, we develop a formal model of adaptive object-oriented groups that also play the role of service providers for their environment. These groups can be dynamically created, they have identity, and they can respond to methods calls, analogously with objects in the object-oriented paradigm. In contrast to objects, groups are not used for executing code. A group exports its services through interfaces and relies on objects to implement these services. From the perspective of client code, groups may be used as if they were objects by programming to interfaces. However, groups support service-oriented abstractions not supported by objects. In particular, groups may dynamically export new interfaces, they support service discovery, and they can be queried at runtime for the interfaces they support. Groups are loosely assembled from objects: objects may dynamically join or leave different groups. In this paper we develop an operational semantics and a static type system for this adaptive group model based on interfaces, interface queries, groups, and service discovery. The type system ensures that well-typed programs do not cause method-not-understood errors at runtime.

The paper is organized as follows. Section 2 presents the language syntax and a small example. A type and effect system for the language is proposed in Section 3 and an operational semantics in Section 4. Section 5 defines a runtime type system and shows that the execution of well-typed programs is type-safe. Section 6 discusses related work and Section 7 concludes the paper.

2 A Kernel Language for Adaptive Object Groups

We study an integration of service-oriented abstractions in an object-oriented setting by defining a kernel object-oriented language with a Java-like syntax, in the style of Featherweight Java [14]. In contrast to Featherweight Java, types are different from classes in this language: interfaces describe services as sets of method signatures and classes generate objects which implement interfaces. By programming to interfaces, the client need not know how a service is implemented. For this reason, the language has a notion of group which dynamically connects interfaces to implementations. Groups are first-class citizens; they have identities and may be passed around. An object may dynamically join a group and thereby add new services to this group, extending the group’s supported interfaces. Objects may be part of several groups. Both objects and groups may join and leave groups, thereby migrating their services between groups. The kernel language considers concurrent objects which interact by synchronous method calls. Concurrent activities are triggered by instantiating classes with \texttt{run} methods (similar to overriding the \texttt{run} method of Java’s Thread class). This simple concurrency model is relevant for service-oriented systems.

2.1 The Syntax

The syntax of the kernel language is given in Figure 1. A type \( T \) in the kernel language is either a basic type, an interface describing a service, or a group of interfaces. A program \( P \) consists of a list \( \mathcal{T} \) of interface declarations, a list \( \mathcal{C} \) of class declarations, and a main block \( \{ T \ \pi.s \} \). The main block introduces a scope with local variables \( \pi \) typed by the types \( T \), and a sequence \( s \) of program statements. We conventionally denote by \( \pi \) a list or set of the syntactic construct \( x \) (in this case, a program variable), and furthermore we write \( T_1 \pi \) for the list of typed variable declarations \( T_1 \ x_1; \ldots; T_n \ x_n \) where we assume that the length of the two lists \( T \) and \( \pi \) is the same. The types \( T \) are the basic type \( \text{Bool} \) of Boolean expressions, the empty interface \( \text{Any} \), the names \( I \) of the declared interfaces, and group types \( \text{Group}(\mathcal{T}) \) which state that a group supports the set \( \mathcal{T} \) of interfaces. The use of types is further detailed in Section 3, including the subtyping relation and the type system.
Interface declarations IF associate a name $I$ with a set of method signatures. These method signatures may be inherited from other interfaces $T$ or they may be declared directly as $Sg$. A method signature $Sg$ associates a return type $T$ with a name $m$ and method parameters $\pi$ with declared types $T$.

Class declarations CL have the form $\text{class } C(T \pi) \text{ implements } T \{ T_1 \pi_1; \{ T_2 \pi_2; s \}; M \}$ and associates a class name $C$ to the services declared in the interfaces $T$. In $C$, these services are realized using methods to manipulate the fields $\pi_1$ of types $T_1$. The constructor block $\{ T_2 \pi_2; s \}$ initializes the fields, based on the actual values of the formal class parameters $\pi$ of types $T$. Remark that the constructor block is executed asynchronously. Consequently, it can be used to trigger concurrent activities starting in a new instance of a class. The methods $M$ have a signature $Sg$ and a method body $\{ T \pi; s; \text{return } x; \}$ which introduces a scope with local variables $\pi$ of types $T$ where the sequence of statements $s$ is executed, after which the expression $e$ is returned to the client.

The expressions $e$ of the kernel language consist of Java-like expressions for reading program variables $x$, method calls $x.m(\pi)$ where the actual method parameters are given by $\pi$, and object creation $\text{new } C(\pi)$ where the actual constructor parameters are given by $\pi$. Method calls are synchronous and in contrast to Java all method calls are synchronized; i.e., a caller blocks until a method returns and a callee will only accept a remote call when it is idle. For simplicity, the kernel language supports self-calls but not re-entrance (which could be addressed using thread identities as in Featherweight Java [14]). In addition, we consider two expressions which are related to service-oriented software: newgroup dynamically creates a new, empty group which does not offer any services to the environment. Service discovery may be localized to a named group $y$: the expression $\text{acquire } I \text{ in } y \text{ except } \pi$ finds some group $g$ or object $o$ such that $g$ or $o$ offers a service better than $I$ (in the sense of subtyping) and such that $g$ or $o$ is not in the set $\pi$. If the in $y$ clause is omitted, then the service provider $g$ or $o$ may be found anywhere in the system.

The statements $s$ of the kernel language include standard statements such as skip, assignments $x = e$, sequential composition $s_1 ; s_2$, conditionals, and while-loops. To simplify the kernel language, we keep a flat representation of expressions; i.e., expressions must be assigned to program variables before they can be used in other statements. Service interfaces $T$ are dynamically exported through a group $y$ by the expression $x \text{ joins } y \text{ as } T$, which states that object or group $x$ is used to implement the interfaces $T$ in the group $y$. Consequently, $y$ will support the interfaces $T$ after $x$ has joined the group. Objects and groups $x$ may try to withdraw service interfaces $T$ from a group $y$ by the expression $x \text{ leaves } y \text{ as } T \{ s_1 \} \text{ else } \{ s_2 \}$. Withdrawing interfaces from a group can lead to runtime exceptions which need to be handled either by the client or by the service provider. In our approach, the exception is handled on the server side; i.e., withdrawing interfaces $T$ from $y$ only succeeds if $y$ continues to offer all the interfaces of $T$, exported by other objects or groups. Thus, removals may not affect the type of $y$. If the removal is successful then branch $s_1$ is taken, otherwise $s_2$ is taken. In addition, the language includes the statement $x \text{ subtypeOf } I \{ s_1 \} \text{ else } \{ s_2 \}$ which is used to query a known group $x$ about its supported interfaces. The statement works like a conditional and branches the execution depending on whether the query succeeds or not. If $x$ offers an interface better than $I$, the expanded knowledge of the group $x$ becomes available through the variable $y$ in the scope of the statements $s_1$. If $x$ does not offer an interface as good as $I$, the branch $s_2$ is taken. Remark the introduction of a new name for the group inside the scope, which ensures that the knowledge of the extended type is local. (By syntactic sugar, the variable $y$ need not appear in the surface syntax).

2.2 Example

We illustrate the dynamic organization of objects in groups by an example of software which provides text editing support (inspired by [22]). This software provides two interfaces: SpellChecker allows
A Type-Safe Model of Adaptive Object Groups

Figure 1: Syntax of the kernel language. The type names $T$ include interfaces names $I$ and $\text{Bool}$. Square brackets $[]$ denotes optional elements.

the spell-checking of a piece of text and $\text{Dictionary}$ provides functionality to update the underlying dictionary with new words, alternate spellings, etc. Apart from an underlying shared catalog of words, these two interfaces need not share state and may be implemented by different classes. Let us assume that the overall system contains several versions of $\text{Dictionary}$, some of which may have an integrated $\text{SpellChecker}$. Consider a class implementing a text editor factory, which manages groups implementing these two interfaces. The factory has two methods: $\text{makeEditor}$ dynamically assembles such software into a text editor group and $\text{replaceDictionary}$ allows the $\text{Dictionary}$ to be dynamically replaced in such a group. These methods may be defined as follows:

```java
Group⟨SpellChecker,Dictionary⟩ makeEditor() {
    Group⟨⟩ editor; SpellChecker s; Dictionary d;
    editor = newgroup;
    d = acquire Dictionary except emptyset;
    d subtypeOf SpellChecker ds {
        ds joins editor as Dictionary, SpellChecker;
    } else {
        d joins editor as Dictionary;
        s = new SpellChecker();
        s joins editor as SpellChecker;
    }
    return editor;
}

void replaceDictionary(Group⟨SpellChecker,Dictionary⟩ editor, Dictionary nd) {
    Dictionary od;
    nd joins editor as Dictionary;
    od = acquire Dictionary in editor except nd;
    od leaves editor as Dictionary {skip;} else {skip;};
    return;
}
```

The method $\text{makeEditor}$ acquires a top-level service $d$ which exports the interface $\text{Dictionary}$ (since there is no $\text{in}$-clause in the $\text{acquire}$-expression). If $d$ also supports the $\text{SpellChecker}$ interface, we let $d$ join the newly created group $\text{editor}$ as both $\text{Dictionary}$ and $\text{SpellChecker}$. Otherwise $d$ joins the $\text{editor}$ group only as $\text{Dictionary}$. In this case a new $\text{SpellChecker}$ object is created and added to the group as $\text{SpellChecker}$. Remark that we assumed the presence of several $\text{Dictionary}$ services in the overall system, otherwise the initial $\text{acquire}$-expression may not succeed and execution could be
Figure 2: The type system for expressions.

blocked at this point. The kernel language could be extended by a more robust version of \texttt{acquire} which uses \texttt{subtypeof}; in fact, inside a group \( g \), robustness may be obtained by first checking for the existence of an interface \( I \) in \( g \) using \texttt{subtypeof} and then binding to the object or group implementing \( I \) in \( g \) using \texttt{acquire}.

The method \texttt{replaceDictionary} will replace the \texttt{Dictionary} service in a text editor group. First we add the new \texttt{Dictionary} service \texttt{nd} to the \texttt{editor} group and then we fetch the old service \texttt{od} in the group by means of an \texttt{acquire}, where the \texttt{except}-clause is used to avoid binding to the new service \texttt{nd}. Finally the old service \texttt{od} is removed as \texttt{Dictionary} in the group by a \texttt{leave} statement. The example illustrates group management by joining and leaving mechanisms as well as service discovery.

3 A Type and Effects System

The language distinguishes behavior from implementations by using an interface as a type which describes a service. Classes are not types in source programs. A class can implement a number of service interfaces, so its instances can export these services to clients. A program variable typed by an interface can refer to an instance of any class which implements that interface. A group typed by \texttt{Group}(\( S \)) exports the services described by the set \( S \) of interfaces to clients, so a program variable of type \( I \) may refer to the group if \( I \in S \). We denote by \texttt{Any} the “empty” interface, which extends no interface and declares no method signatures. A service described by an interface may consist of only some of the methods defined in a class which implements the interface, so interfaces lead to a natural notion of hiding for classes. In addition to the source program types used by the programmer, class names are used to type the self-reference \texttt{this}; i.e., a class name is used as an interface type which exports all the methods defined in the class.

Subtyping. The subtype relation \( \prec \) is defined as the transitive closure of the extends-relation on interfaces: if \( I \) extends \( J' \) and \( J' \prec J \) or \( J' = J \), then \( I \prec J \). It is implicitly assumed that all interfaces extends \texttt{Any}, so we let \( I \prec \texttt{Any} \) for all \( I \). A group type \texttt{Group}(\( S \)) is a subtype of \( I \) if there is some \( J \in S \) such that \( J \prec I \), and \texttt{Group}(\( S \)) \( \prec \) \texttt{Group}(\( S' \)) if for all \( J \in S \) there is some \( I \in S' \) such that \( I \prec J \). We extend the source language subtype relation by letting a class be a subtype of all its implemented interfaces. The reflexive closure of \( \prec \) is denoted \( \preceq \).

Typing contexts. A typing context \( \Gamma \) binds variable names to types. If \( \Gamma \) is a typing context, \( x \) a variable, and \( T \) a type, we denote by \texttt{dom}(\( \Gamma \)) the set of names which are bound to types in \( \Gamma \) (the domain of \( \Gamma \)) and by \( \Gamma(x) \) the type bound to \( x \) in \( \Gamma \). Define the update \( \Gamma[x \mapsto T] \) of a typing context \( \Gamma \) by \( \Gamma[x \mapsto T](x) = T \) and \( \Gamma[x \mapsto T](y) = \Gamma(y) \) if \( y \neq x \). By extension, if \( \overline{x} \) and \( \overline{T} \) denote lists \( x_1, \ldots, x_n \) and
A Type-Safe Model of Adaptive Object Groups

\[ \Gamma \vdash \text{skip} : \text{ok} \]

\[ \Gamma \vdash e : \Gamma(x) \]

\[ \Gamma \vdash x = e : \ldots \text{premise of the rule is omitted if the statement has no in-clause.} \]

Rule T-SUB captures subtyping in the type system.

\[ \begin{align*}
\text{match} & \quad \text{local}(x) \\
\Gamma & \vdash \text{if } x \{x_1\} \text{ else } \{x_2\} : \Gamma(\Delta_{\cap} \Delta_{\cap})
\end{align*} \]

\[ \Gamma \vdash x \text{ leaves } y \text{ as } \{x\} \text{ else } \{x_2\} : \Gamma(\Delta_{\cap} \Delta_{\cap}) \]

\[ \Gamma \vdash \text{leaves } y \text{ as } \{x\} \text{ else } \{x_2\} : \Gamma(\Delta_{\cap} \Delta_{\cap}) \]

\[ \begin{align*}
\text{match} & \quad \text{local}(x) \\
\Gamma & \vdash \text{if } x \{x_1\} \text{ else } \{x_2\} : \Gamma(\Delta_{\cap} \Delta_{\cap})
\end{align*} \]

\[ \Gamma \vdash x \text{ leaves } y \text{ as } \{x\} \text{ else } \{x_2\} : \Gamma(\Delta_{\cap} \Delta_{\cap}) \]

\[ \Gamma \vdash \text{leaves } y \text{ as } \{x\} \text{ else } \{x_2\} : \Gamma(\Delta_{\cap} \Delta_{\cap}) \]

Figure 3: The type and effect system for statements, methods, classes, and programs.

\[ T_1, \ldots, T_n \text{, we may write } \Gamma[\varphi \mapsto \vartheta] \text{ for the typing context } \Gamma[\varphi \mapsto \vartheta] \text{ and } \Gamma[\varphi \mapsto \vartheta] \text{ for } \Gamma[\varphi \mapsto \vartheta]. \]

For typing contexts \( \Gamma_1 \) and \( \Gamma_2 \), we define \( \Gamma_1 \circ \Gamma_2 \) such that \( \Gamma_1 \circ \Gamma_2(x) = \Gamma_2(x) \) if \( x \in \text{dom}(\Gamma_2) \) and \( \Gamma_1 \circ \Gamma_2(x) = \Gamma_1(x) \) if \( x \notin \text{dom}(\Gamma_2) \).

For typing contexts \( \Gamma_1 \) and \( \Gamma_2 \), we define the intersection \( \Gamma_1 \cap \Gamma_2 \) by \( \Gamma_1 \cap \Gamma_2(x) = T \) if \( T \) is the best type such that \( \Gamma_1(x) = T_1 \), \( \Gamma_2(x) = T_2 \), and \( T_1 \leq T \) and \( T_2 \leq T \). In particular, we have \( \Gamma_1 \cap \Gamma_2(x) = \text{Group}(S_1 \cap S_2) \) if \( \Gamma_1(x) = \text{Group}(S_1) \) and \( \Gamma_2(x) = \text{Group}(S_2) \).

The Type and Effect System. Programs in the kernel language are analyzed using a type and effect system (e.g., [2, 19, 24]). The inference rules for expressions are given in Figure 2 and for statements, methods, classes, and programs in Figure 3.

Expressions are typed by the rules in Figure 2. Let \( \Gamma \) be a typing context. A typing judgment \( \Gamma \vdash e : T \) states that the expression \( e \) has the type \( T \) if the variables in \( e \) are typed according to \( \Gamma \). By T-VAR, variables must be typed in \( \Gamma \). Method calls to a method \( m \) on a variable \( x \) are typed to \( T \) if \( x \) has the (interface) type \( T' \) such that the types \( \vartheta \) of the actual parameters \( \varphi \) give a match for \( m \) in \( T' \) with parameter types \( \vartheta \) and the declared return type of \( m \) in \( T' \) is \( T \). In T-NEW, \( \text{new } C \) has type \( I \) if the types of the actual parameters to the class constructor can be typed to the declared types of the formal parameters of the class, by means of the auxiliary function \( \text{ptypes} \), and the class implements \( I \), expressed by \( C \prec I \). We omit the definitions of the auxiliary functions \( \text{match} \) and \( \text{retType} \) here, these are straightforward lookup functions on the program’s interface table which perform the matching and retrieve the return type of a method in a class, respectively. Similarly, \( \text{ptypes} \) retrieves the types of the formal parameters to a class in the program’s class table. By T-GROUP, a new group has the empty group type (with no exported interfaces). By T-ACQUIRE, service discovery has the obvious type, if successful. The premise of the rule is omitted if the statement has no in-clause. Rule T-SUB captures subtyping in the type system.
Statements are typed by the rules in Figure 3. Let \( \Gamma \) and \( \Delta \) be typing contexts. A typing judgment 
\[
\Gamma \vdash s : \text{ok}(\Delta)
\]
expresses that the statement \( s \) is well-typed if the variables in \( s \) are typed according to \( \Gamma \) and that the typing context for further analysis should be modified according to the effect \( \Delta \). Empty effects are omitted in the presentation of the rules. The typing of statements \( \text{skip} \) and \( x = e \) are standard. These judgments have no effects. The statement \( \text{return} \ x \) has a return type and is typed in the effect of typing the statements of the method body. The use of effects can be seen in rule T-COMPOSITION, where the second statement is type checked in the typing context modified by the effect of analyzing the first statement, and the effects are accumulated in the conclusion of the rule. Rules T-CONDITIONAL and T-WHILE propagate effects from the subexpressions; in the case of T-CONDITIONAL the resulting effect is approximated by taking the intersection of the effects of the branches. By T-JOIN, when an object joins a group \( y \) and contributes interfaces \( T \) to \( y \), the effect is that the type of \( y \) is extended with the interfaces \( T \). Note the requirement \( \text{local}(y) \), which expresses that \( y \) must be a local variable in the scope of the method being analyzed. (We omit the definition, which is again a lookup in the class table of the program). Without this restriction, a field could dynamically extend its type, resulting in an unsound system; e.g., an assignment \( \varepsilon = e \) in a statically well-typed method could become unsound if the type of \( \varepsilon \) were extended. However extending the type \( \tau \) of a local variable which copies the value of \( \varepsilon \) to a type \( \tau' \) and assigning the result back to a field \( \varepsilon' \) is allowed, as \( \varepsilon' \) would need to be of the extended type \( \tau' \) and \( \varepsilon \) would remain of type \( \tau \) as required by the other method. (For comparison, the needed restriction to local variables is handled differently in the query statement \( \text{subtypeof} \), which introduces a fresh local variable.) Rule T-LEAVE shows that leaving a group has no effect on the typing context, and the effects of the two branches are treated as for the conditional. Rule T-INSPECT shows how the typing context is extended with a new variable \( y \) which extends the type of the group \( x \) for the scope of the branch \( s_1 \). The overall effect is again the intersection of the effects of the two branches.

Programs, classes, and methods are typed in the standard way. Methods do not have effects, which reflects that effects are constrained to local variables inside methods. Likewise, classes and programs do not have effects. (For simplicity, the standard type checking of interface declarations is omitted in the presentation.) The body of a class constructor and the main method of a program may have the same effects as the body of a method.

4 Operational Semantics

The runtime syntax is given in Figure 4. A runtime configuration \( cn \) is either the empty configuration \( \varepsilon \) or it consists of objects \( \text{obj} \) and groups \( \text{grp} \). Groups \( \text{grp} \) have an identity \( g \) and contain a set \( \text{export} \) of interfaces \( I \) associated with the objects \( o \) implementing them. Objects \( \text{obj} \) have an identity \( o \), a state \( \sigma \), and a stack \( \rho \) of processes \( \text{proc} \). When an object has processes to execute, it executes the process at the top of its stack. The stack grows with self-calls and shrinks at method returns. The empty stack is denoted \( \text{idle} \). A state \( \sigma \) maps program variables \( x \) to their types \( T \) and values \( v \). A process \( \text{proc} \) can be \( \text{error} \) or it has a local state \( \sigma \) and a sequence \( s \); \( \text{return} \ x \); of statements to be executed. The expression \( \text{wait}(o, m) \) encodes a \( \text{lock} \), expressing that the object is waiting for the return value of method \( m \) in another object \( o \) (or on an auxiliary self-call). Values \( v \) include object and group names, and Booleans.

The operational semantics is given by rules in the style of SOS [21], reflecting small-step semantics. Each rule describes one step in the execution of an object. Concurrent execution is given by standard SOS context and concurrency rules (not shown here), and we assume associative and commutative matching over configurations (as in rewriting logic [7]). Thus objects execute concurrently, with the following exceptions: The rule for synchronous remote call \( \text{CALL1} \) refers to both the caller and callee objects.
and therefore the two objects must synchronize and the caller will be blocked by the wait statement. Furthermore rules involving an object and a group will lock the group in question, thereby disallowing concurrent execution of other objects involving the same group. This is crucial in the JOIN and LEAVE1 rules for joins and leaves, which may actually modify the group.

We define the lookup of a program variable \( x \) in a state \( \sigma \) by \( \sigma(x) = (T, v) \), with the projections \( T(x) = T \) and \( V(x) = v \). Thus, for a state \( \sigma \), \( T \) gives the associated mapping of program variables to their types and \( V \) the mapping of program variables to their values. The SKIP rule is standard and states that a skip has no effect. The effect of assignment is divided into two rules, ASSIGN1 for local variables, updating \( l \), and ASSIGN2 for fields, updating \( a \). In the rule NEW-GROUP, a globally unique group identifier is found by fresh(\( g \)). Then an empty group with this identifier is added to the configuration. The two rules COND1 and COND2 handle the two cases of the conditional statement.

Method calls are handled by CALL1 for calls to other objects, CALL2 for self calls, and CALL3 for calls to groups. When a call is made to another object in CALL1, the called object must be in an idle state. The caller blocks until the generated wait statement can be executed. In the wait statement, the callee and method name are recorded, which allows the runtime system to infer the proper type of the return value from method \( m \) in the proper class. Let \( \text{bind}(m,C,V) \) denote the process resulting from the activation of method \( m \) in \( C \), in which \( l \) maps the parameters of \( m \) to their declared types and values \( V \), and the local variables to their declared types and default values. The callee gets the process \( \text{bind}(m,C,(a \circ l)^V(V)) \), where \( C \) is the class of the callee, pushed onto its process stack \( \rho \). With self calls in CALL2, the process stack cannot be idle, but a wait statement replaces the call statement and an instance of the called method is pushed to the stack. In CALL3, a call to a group is reduced to a call to a group or an object inside the callee which exports an appropriate interface to the group. By appropriate we mean that the called method is supported by the interface (formally, \( m \in mtd(I) \)). RETURN1 handles returns from remote calls. Here the blocking wait statement is replaced by the returned value. Returns from self calls are handled in a similar way by the RETURN2 rule. (Remark that the generalization to concurrent objects with asynchronous calls and futures is straightforward as in [6, 15] whereas the extension to multi-threaded programs would require re-entrant lock as in [14]).

The new statement is handled by the NEW-OBJECT rule, where fresh(\( o',C \)) asserts that \( o' \) is a new name in the global configuration such that classOf(\( o' \)) = \( C \). An object with this name is created. The mapping attrs(\( C,V \)) maps the declared fields of class \( C \) to their declared types and default values, this to \( C \), and the class parameters to declared types and actual values. The process init(\( C \)) corresponds to the
init-block of C, which instantiates local variables to their declared types and default values. The process of the new object is the initial process of its class. Note that an init-block is executed independently from the creator, so it may trigger active behavior; for instance, the init-block can call a run method.

The rule JOIN extends the knowledge of a group with the new interfaces from the object’s perspective and correspondingly extends the exports set from the group’s perspective. Service discovery is handled by the ACQUIRE rule. The acquire expression is replaced by a value v, which is an object or group identifier satisfying the in and except clauses. If the in clause is omitted from the expression, then the premise \((a \circ l)^\Gamma(y) = g\) is omitted from the rule. Note that this rule will block if no matching object or group exists. This could be solved by either returning null (by means of a global check) or by adding an else branch similar to those in QUERY1 and QUERY2. Within the kernel language, the existence of a matching object or group inside a group can be checked using the query mechanisms.

The leaves statement is handled by the rules LEAVE1 for a successful leave and LEAVE2 for an unsuccessful one. A group or object x may leave a group successfully if the group provides the same interface support without x. To determine this, we use the function \(\text{inf}(\text{export})\) which returns a set containing the interfaces of all the pairs in export, removing redundant information. An entry is redundant if a subtype of the entry is present in the set. The type of the group does not change by a leaves statement and hence the object does not need to update information about the group. The branches \(s_1\) or \(s_2\) are chosen depending on the success. The rules QUERY1 and QUERY2 handle the branching statement that checks if a group exports a given interface. If the test succeeds then a fresh variable y is introduced and is only visible in \(s_1\). The type of this variable is the union of what the current object already knew about the group and the new information \(l\). If the test fails the \(s_2\) branch is chosen by QUERY2.

The initial state. For a program \(P = \overline{TFC}CL\{T \alpha, s\}\), we define the initial state to be \(o(\epsilon, \text{main}\{\alpha \mapsto (T, \text{default}(T))\}; s)\) where \(o\) is such that fresh\((o, \text{Main})\).

5 Type Safety

This section extends the type system of Section 3 to runtime configurations and shows that the execution of well-typed programs remains well-typed.

5.1 Well-Typed Configurations

The extension of the type system to runtime configurations is given in Figure 6. The typing context \(\Gamma\) stores the types of all constant values (object and group identities) at runtime. By RTT-CONFIG, a configuration is well-typed if all objects and groups are well-typed. By RTT-GROUP, a group is well-typed if all the objects which export interfaces through the group implement these interfaces (checked by RTT-EXPS and RTT-EXP). By RTT-OBJECT, an object is well-typed if its class is its type in \(\Gamma\) and its state and stack are well-typed in the context of the types of the fields. Substitutions (the state of fields and local variables) are checked by RTT-SUBS and RTT-SUB. The stack is well-typed by RTT-STACK if all its processes are well-typed by RTT-PROC; i.e., the state of local variables and the method body sr are well-typed. Observe that due to the query-mechanism of the language, the types of program variables in two processes which stem from activations of the same method, may differ at runtime. For this reason, the typing context used for typing runtime configurations cannot rely on the statically declared types of program variables. This explains why RTT-PROC extends \(\Gamma\) with the locally stored typing information \(l^T\) to type check \(l^T\) and sr. The effects of the static type system are not needed here, as they are reflected by how the operational semantics updates this local type information. For consistency in the presentation,
A Type-Safe Model of Adaptive Object Groups

Figure 5: The operational semantics.
the typing of fields is represented in the same way, although these types are not altered by the execution. The rules from the static type checking are reused as appropriate.

5.2 Subject Reduction

The type system guarantees that the type of \textit{fields} in an object never changes at runtime (in particular, recall the restriction \textit{local}(y) in rule T-JOIN). This allows us to establish in Lemma 1 from the static typing of methods in well-typed programs that method binding, if successful, results in a well-typed process at runtime. To show that the \textit{error} process cannot occur in the execution of well-typed programs, it suffices to show that substitutions are always well-typed. Lemma 2 shows that this is the case for the initial configuration and Lemma 3 shows that one execution step preserves runtime well-typedness. Together, these lemmas establish a subject reduction theorem for the language, expressing that well-typedness is preserved during the execution of well-typed programs and in particular that method binding always succeeds. Here, $\rightarrow^*$ denotes the reflexive and transitive closure of the reduction relation $\rightarrow$.

Lemma 1 Assume that a well-typed program has a class \textit{C} which defines a method \textit{m} with formal parameters \textit{T} of type \textit{T} and return type \textit{T}. Let \textit{o} be an object such that \textit{classOf}(\textit{o}) = \textit{C} and $\Gamma \vdash o(a,\rho) : \textit{ok}$. If $\Gamma \vdash T : T$, then $\Gamma \vdash o(a) + \textit{bind}(m,C,T) : T$.

Lemma 2 Let \textit{P} be a program such that $\Gamma \vdash P : \textit{ok}$ and let \textit{cn} be the initial state of \textit{P}. Then $\Gamma \vdash \textit{cn} : \textit{ok}$.

Lemma 3 If $\Gamma \vdash \textit{cn} : \textit{ok}$ and \textit{cn} $\rightarrow \textit{cn}'$ then there is a $\Gamma'$ such that $\Gamma' \vdash \textit{cn}' : \textit{ok}$ and $\Gamma \subseteq \Gamma'$.

Theorem 1 (Subject reduction) Let $\Gamma \vdash P$ and let \textit{cn} be the initial runtime state of \textit{P}. If \textit{cn} $\rightarrow^*$ \textit{cn}' then there is a $\Gamma'$ such that $\Gamma' \vdash \textit{cn}' : \textit{ok}$ and $\Gamma \subseteq \Gamma'$.

6 Related Work

Object orientation is well-suited for designing small units which encapsulate state with behavior, but does not directly address the organization of more complex software units with rich interfaces. Two approaches to building flexible and adaptive complex software systems involve, independently, object groups and service discovery. Our work unifies these two approaches in a formal, type-safe setting.

The most common use of object groups is to provide replicated services in order to offer better fault tolerance. Communication to elements of a group is via multicast. This idea originated in the Amoeba

![Figure 6: The runtime type system.](image-url)
operating system [16]. The component model Jgroup/ARM [20] adopts this idea to provide autonomous replication management using distributed object groups. In this setting, members of a group maintain a replicated state for reasons of consistency. The ProActive active object programming model [3] supports abstractions for object groups, which enable group communication—via method call—and various means for synchronizing on the results of such method calls, such as wait-for-one and wait-for-all. ProActive is formalized in Carmel and Henrio’s Theory of Distributed Objects [4]. These notions of group differ from ours in two respects. Firstly, in these approaches communication with groups is via multicast, whereas in our approach each message will be delivered to exactly one object, and secondly, in the formal theory, groups are fixed upon creation. Furthermore, there is no notion of service discovery associated with groups.

Object groups have been investigated as a modularization unit for objects which is complementary to components. Groups meet the needs of organizing and describing the statics and dynamics of networks of collaborating objects [18]; groups can have many threads of control, they support roles (or interfaces), and objects may dynamically join and leave groups. Lea [18] presents a number of common usages for groups and discusses their design possibilities, inspired from CORBA. Groups have been used to provide an abstraction akin to a notion of component. For example, in Oracle Siebel 8.2 [8], groups are used as units of deployment, units of monitoring, and units of control when deploying and operating components on Siebel servers. Our approach abstracts from most of these details, though groups are treated as first class entities in our calculus.

Another early work on groups is ActorSpaces [1], which combine Actors with Linda’s pattern matching facility, allowing both one-to-one communication, multicast, and querying. Unlike our approach, groups in ActorSpaces are intensional: all actors with the same interface belong to the same group. Furthermore ActorSpaces support broadcast communication to a group, which has not been considered in this paper as it would differentiate communication with an object and with a group. Compared to our paper, these works do not give a formalization of group behavior or discuss typing.

Object groups have further been used for coordination purposes. For example, CoLaS [9] is a coordination model based on groups in which objects may join and leave groups. CoLaS goes beyond the model in our paper by allowing very intrusive coordination of message delivery based on a coordinator state. In our model, the groups don’t have any state beyond the state of their objects. Similar to our model, objects enroll to group roles (similar to interfaces). However, unlike our model objects may leave a group at any time, and the coordinator may access the state of participants. The model is implemented in Smalltalk and neither formalization nor typing is discussed [9]. Concurrent object groups have also been proposed to define collaborating objects with a single thread of control in programming and modeling languages [15,23]. Concurrent object groups do not have identity and function as runtime restrictions on concurrency rather than as a linguistic concept.

Microsoft’s Component Object Model (COM) supports querying a component to check whether it supports a specific interface, similar to the query-mechanism considered in this paper. A component in COM may also have several interfaces, which are independent of each other. In contrast to the model presented in our paper, COM is not object-oriented and the interfaces of a component are stable (i.e., they do not change). COM has proven difficult to formalize; Pucella develops $\lambda^{COM}$ [22], a typed $\lambda$-calculus which addresses COM components in terms of their interfaces, and discusses extensions to the calculus to capture subtyping, querying for interfaces, and aggregation.

A wide range of service discovery mechanisms exist [13]. The programming language AmbientTalk [10] has built-in service discovery mechanisms, integrated in an object-oriented language with asynchronous method calls and futures. In contrast to our work, AmbientTalk is an untyped language, and lacks any compile time guarantees. Various works formalise the notion of service discovery [17],
but they often do so in a formalism quite far removed from the standard setting in which a program using service discovery would be written, namely, an object-oriented setting. For example, Fiadeiro et al.’s [11] model of service discovery and binding takes an algebraic and graph-theoretic approach, but it lacks the concise operational notion of service discovery formalized in our model. No type system is presented either.

Some systems work has been done that combines groups and service discovery mechanisms, such as group-based service discovery mechanisms in mobile ad-hoc networks [5, 12]. In a sense our approach provides language-based abstractions for a mechanism like this, except that ours also is tied to interface types to ensure type soundness and includes a notion of exclusion to filter matched services.

Our earlier work [6] enabled objects to advertise and retract interfaces to which other objects could bind, using a primitive service discovery mechanism. A group mechanism was also investigated as a way of providing structure to the services. In that work services were equated with single objects, whereas in the present work a group service is a collection of objects exporting their interfaces. In particular, this means that the type of a group can change over time as it comes to support more functionality.

The key differences with most of the discussed works is that the model in this paper remains within the object-oriented approach, multiple groups may implement an advertised service in different ways, and our formalism offers a transparent group-based service discovery mechanism with primitive exclusion policies. Furthermore, our notion of groups has an implicit and dynamically changing interface.

7 Conclusion

The paper has proposed a formal model for adaptive service-oriented systems, based on a notion of object-oriented groups. We develop a kernel object-oriented language in which groups are first-class citizens in the sense that they may play the role of objects; i.e., a reference typed by an interface may refer to an object or to a group. A main advantage is that one may collect several objects into a group, thereby obtaining a rich interface reflecting a complex service, which can be seen as a single object from the outside. Although objects in our language are restricted to executing one method activation at the time, a group may serve many clients at the same time due to inner concurrency.

In contrast to objects, groups may dynamically add support for an increasing number of interfaces. The formation of groups is dynamic; join and leave primitives in the kernel language allow the migration of services provided by objects and inner groups as well as software upgrade, provided that interfaces are not removed from a group. An object or group may be part of several groups at the same time. This gives a very flexible notion of group.

Adaptive object groups are combined with service discovery by means of acquire and subtypeOf constructs in the kernel language, which allow a programmer to discover services in an open and unknown environment or in a known group, and to query interface support of a given object or group. These mechanisms are formalized in a general object-oriented setting, based on experiences from a prototype Maude [7] implementation of the group and service discovery primitives. The presented model provides expressive mechanisms for adaptive services in the setting of object-oriented programming with modest conceptual additions. We have developed an operational semantics and type and effects system for the kernel language, and show the soundness of the approach by a proof of type-safety.

The combination of features proposed in this paper suggests that our notion of a group can be made into a powerful programming concept. The work presented in this paper may be further extended in a number of directions. The overall goal of our work is to study an integration of service-oriented and object-oriented paradigms based on a formal foundation. In future work, we plan to extend the
proposed kernel language to multi-thread concurrency and study in more detail how different usages of object groups such as replication, resource, and access groups (see, e.g., [18]) may be captured using the proposed primitives. It is also interesting to study the integration into the kernel language of more service-oriented concepts such as for example error propagation and handling, as well as high-level group management operations such as group aggregation.

References


Appendix C

Conflict Detection in Delta-Oriented Programming

The paper “Conflict Detection in Delta-Oriented Programming” [57] follows.
Conflict Detection in Delta-Oriented Programming

Michäel Lienhardt\textsuperscript{1} and Dave Clarke\textsuperscript{2}

\textsuperscript{1} University of Bologna, Italy
lienhard@cs.unibo.it
\textsuperscript{2} IBBT-DistriNet Katholieke Universiteit Leuven, Belgium
Dave.Clarke@cs.kuleuven.be

Abstract. This paper studies the notion of conflict for a variant of Delta-Oriented Programming (DOP) without features, separating out the notions of hard and soft conflict. Specifically, we define a language for this subset of DOP and give a precise, formal definitions of these notions. We then define a type system based on row-polymorphism that ensures that the computation of a well-typed product will always succeed and has an unambiguous result.

1 Introduction

Delta-oriented programming (DOP) \cite{21,22} is a recent approach to developing Software Product Lines (SPLs) \cite{6} that addresses several limitations of previous approaches: it completely dissociates feature models from feature modules (now called deltas), which allows features to be implemented using more than one delta and deltas to be used by several features, thus improving modularity, reuse and flexibility; moreover, DOP enables non-monotonic modifications of the core architecture, including the removal of fields, methods and even classes. DOP is flexible and enables the modular construction of SPLs. However, tool support for DOP is not as mature as for other SPL approaches. In particular, the issue of validating delta-oriented programs has not fully been addressed. Schaefer et al. \cite{20} propose to generate a collection of constraints for delta-oriented product lines, ensuring that the manipulations done on the core product are sound and the resulting products are type safe. However, this work has several limitations: i) as it is based on constraints, the types do not reflect the structure of the deltas; ii) it presupposes that the order in which the deltas are applied on a core is totally specified; and iii) it generates a set of constraints per product, which means that the complexity is exponential in the number of deltas. More recently, the present authors proposed an approach \cite{16} that addresses the first of these limitations using row polymorphism \cite{19} to capture the structure of products and the semantics of deltas in the types. The underlying computational model takes

\textsuperscript{*} This research is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (http://www.hats-project.eu).
Conflict Detection in Delta-Oriented Programming 179

```plaintext
core k {
class Settings {
    int coffee;
}
class CMachine {
    Settings conf;
    void make() {...}
    void makeCoffee() {...}
}
delta Choco {
    modifies class Settings {
        adds int chocolate;
    }
    modifies class CMachine {
        adds makeChoco() {...}
        modifies make() {...}
    }
}
delta Sugar {
    modifies class Settings {
        adds int sugar;
    }
    modifies class CMachine {
        modifies make {...}
    }
}
delta ColorPrint {
    modifies class CMachine {
        modifies make() {...}
        modifies makeCoffee() {...}
        modifies makeChoco() {...}
    }
}
product ps {Choco Sugar} k
product ph {Choco ColorPrint} k
```

Fig. 1. Soft and Hard Conflicts

a collection of classes as its basis and applies deltas in some order to update
them. This paper presents an extension of this approach to deal with the second
limitation.

In DOP, if feature modules are applied to a core product in a different order
it is not necessarily the case that all computations give the same result. This is
illustrated by the code in Figure 1.

This example models a coffee machine with core k comprised of a class Setting
storing the type of coffee to brew, and a class CMachine with a generic make
method and a method makeCoffee, called by make to prepare coffee. In addi-
tion to this core, there are three deltas: Choco adds the capability of brewing
hot chocolate; Sugar adds the possibility of setting the quantity of sugar; and
ColorPrint changes the make* methods so that messages are printed in color. Fi-
nally, there are two different products: ps applies deltas Choco and Sugar on the
core k, and product ph is constructed by applying deltas Choco and ColorPrint
to the core. The order in which the deltas are applied is free in this example,
and thus ps and ph can either be computed by applying either delta Choco or
the other one first.
Applying first Choco and then Sugar in \( p_s \) results in a product with the method \( \text{make} \) defined by the delta Sugar, whereas if Sugar is applied first, the method \( \text{make} \) is defined by Choco: the computation of \( p_s \) is ambiguous (i.e. it can have different results), caused by a soft conflict between the deltas Choco and Sugar. Such soft conflicts can be dealt with in two ways: by defining a partial order between delta: for instance, by stating that Choco is always applied after Sugar, the computation of \( p_s \) is unambiguous; and by defining another delta to resolve the conflict: adding to \( p_s \) delta SweetChoco that replaces the method \( \text{make} \) after applying Choco and Sugar will also make the computation unambiguous.

The product \( p_h \) presents another kind of conflict, called hard conflicts. While first applying the delta Choco and then ColorPrint, the computation succeeds without any error, first applying ColorPrint results in an error because ColorPrint tries to modify method \( \text{makeChoco} \) before it exists. Such hard conflicts can only be resolved by imposing an ordering on the deltas specifying that ColorPrint must be applied after Choco. The work on Abstract Delta Modelling discussed only soft conflicts—otherwise hard conflicts did not cause an error—though the theory could easily encompass both [4].

Roadmap. The paper is structured as follows. Section 2 describes a DOP language focusing on deltas and conflicts. Section 3 presents a formal definition of soft and hard conflicts. Section 4 introduces our type system to capture runtime errors and conflicts. Section 5 compares our approach with related work and Section 6 concludes the paper.

2 Delta-Oriented Programming

In the rest of the paper, we will use the term member for either a method or a field of a class. The syntax of our delta-oriented programming language is presented in Figure 2. A product line \( PL \) is a sequence of element declaration \( PLE \). An element can either be a delta \( \text{delta } d \text{ after } DL \{ \text{COL} \} \), where \( DL \) is used to construct the partial order between deltas (see Definition 2) and \( \text{COL} \) is the body of \( d \); a core product \( \text{core } k \{ \text{CL} \} \), where \( \text{CL} \) is the set of classes defining the core \( k \); or a product \( \text{product } p = \{ \text{DL} \} k \), where \( \text{DL} \) are the deltas to be applied to the core \( k \) to produce \( p \).
CO and FO are the operations on classes and members, respectively. It is possible to add, remove and modify both classes and members. The modification of a class is done with a sequence of operations on members FOL, while the modification of members is not specified in our language that only focuses on the manipulation on the structure of the cores, not their behavior.

Free names. The declaration of a new delta d, or a new core k or of a new product p, respectively binds d, k or p in the rest of the program. This notion of binders and free names implicitly creates a form of α-conversion on our PL terms. We note PL =α PL’ when PL is α-equivalent to PL’. Using this notion of α-conversion, we can assume that all declared deltas, cores and products of a product line PL have different names.

Definition 1. A product line PL with no free names is said to be closed. We denote P\text{cl} the set of all closed product lines.

Semantics. The full semantics of the language is presented elsewhere [5,15]. The principle of the computation of a product in our language is quite simple. The delta names in DL are sorted to match the order given by the keyword after. When the order is not total, several sequences of deltas are possible, creating the possibility of conflicts. Then, the code of the deltas are applied in order to the core, thus computing the product. The following definition presents the formal construction of the order between deltas, which is necessary to define the notion of conflicts (hard and soft).

Definition 2. A general context G is a product line with a hole •. Given a product line PL and a term t given by one of the productions in Figure 2. Say that t is in PL (denoted t \in PL) if there exists a general context G such that PL = G[t]. The relation <\text{PL} between delta names is defined as the smallest transitive relation satisfying the following property

\[
\text{delta } d \text{ after } d_1 \ldots d_n \{\text{COL}\} \in PL \Rightarrow d_i <_{PL} d.
\]

Finally, the next definition presents the equivalence relation used to sort delta names, which is used in our type system.

Definition 3. A relation R is closed under general context iff

\[
\forall G, x, y, \quad x \; R \; y \quad \Rightarrow \quad G[x] \; R \; G[y]
\]

Define the relation ≡\text{dl} as the smallest equivalence relation closed under general context validating the following rule (where d₁ and d₂ are delta names):

\[
DL \equiv \text{dl} \; DL' \quad \frac{d_1 \; d_2 \; DL \equiv \text{dl} \; d_2 \; d_1 \; DL'.}{d_1 \; d_2}
\]

3 Conflicts

Clarke et al. [4] define the notion of conflict for an abstract notion of delta, but they do not capture hard conflicts. This section proposes a more precise definition based on the notion of action.
**Actions.** The following syntax gives the elements $E$ manipulated by a delta and how they can be manipulated:

\[
E ::= c \mid c.f
\]

\[
A ::= \bot \mid \text{add} \mid \text{rem} \mid \text{mod}
\]

Given a class $c$ or an member $c.f$ (annotated with its class name $c$), a delta can either do nothing with it ($\bot$); add it (\text{add}); remove it (\text{rem}); or modify it (\text{mod}). Denoting the set of all elements $E$ (resp. all actions $A$) by $\mathcal{E}$ (resp. $\mathcal{A}$), we define the \textit{effect} (or \textit{action} by language abuse) of a delta $d$ as the function $f : \mathcal{E} \to \mathcal{A}$ which maps every element $E$ to the action performed by $d$ on $E$. The action of the code of a delta is defined inductively in Fig. 3, based on the following notions.

**Definition 4.** Given a set of elements $S \subseteq \mathcal{E}$ and an action $A$. Use $S \to A$ to denote the function $f : \mathcal{E} \to \mathcal{A}$ such that $f(E) = A$ when $E \in S$ and $f(E) = \bot$ for all $E \notin S$. Given two functions $f, g : \mathcal{E} \to \mathcal{A}$. Use $f \triangleright g$ to denote the function $h : \mathcal{E} \to \mathcal{A}$ such that $h(E) = f(E) \triangleright g(E)$, where $\triangleright$ is defined in Fig. 3.

Adding a class corresponds to the action $\text{add}$ on the class and on all of its members (denoted by the set $\text{members}(c, FL)$). Removal corresponds to the action $\text{rem}$ on the class and on all of its possible members (noted $\mathcal{F}$). Modification corresponds to $\text{mod}$, plus all the actions done on the member level. Sequential composition of operators is handled by the operator $\triangleright$.

Finally, the action of a product line that maps all delta names to their actions for a closed product line is defined as follows.
Fig. 4. Type Syntax

Definition 5. Given a closed product line PL. The action of PL, denoted act(PL), is a function that maps all the deltas declared in PL to their code’s action:

\[
act(PL)(d) \begin{cases} 
\text{act}_c(COL) \text{ if delta } d \text{ after } DL \{COL\} \in PL \\
\emptyset \to \bot \text{ otherwise} 
\end{cases}
\]

Conflicts. The following definition captures the two kinds of conflict:

Definition 6. Given a closed product line PL, an element E, a product product p = {DL} k ∈ PL, and two delta names d₁, d₂ ∈ DL such that d₁ \not<_{PL} d₂ ∧ d₂ \not<_{PL} d₁. Product line PL has a soft conflict, denoted PL ⊨^E\ E\ d₁ \not\sim d₂, if E is a member c.f and act(PL)(d₁)(E) = act(PL)(d₂)(E) = \text{mod}. There is a hard conflict, denoted PL ⊨^E\ E\ d₁ \not\sim \sim d₂, iff both deltas are acting on E and one of them is not doing a simple modification. That is,

\[
(\text{act}(PL)(d₁)(E), \text{act}(PL)(d₂)(E)) \notin \{(\bot) \times A\} \cup (A \times \{\bot\}) \cup \{(\text{mod},\text{mod})\}
\]

A conflict occurs when two operations on the same element may not produce the same result. An example soft conflict results from two modifications of an member: the two possible sequences can produce a different result, thus causing ambiguity. Hard conflicts produce an error during the computation of a product. For instance, first modifying an element and then removing it is correct, whereas trying to modify an element that was removed is erroneous. Finally, it is possible to resolve soft conflicts with another delta that acts on the element after the conflict:

Definition 7. Given a soft conflict PL ⊨^E\ E\ d₁ \not\sim d₂. This conflict is resolved iff there exists d ∈ dep(p) such that d₁ <_{PL} d, d₂ <_{PL} d and act(PL)(d)(E) \neq \bot, with

\[
\text{dep}(p) \triangleq \{d₁ \ldots dₙ \mid \text{product } p = \{d₁ \ldots dₙ\} k \in PL\}
\]

Theorem 1. A closed product line PL with no unresolved conflicts is unambiguous.
4 Type System

The type system extends our previous work [16] to capture conflicts. Its syntax is presented in Figure 4.

Row types [19] capture the structure of products and classes, row polymorphism is used to type deltas and annotations, \( J \), capture the action of deltas and conflicts. The type of a product \( TP \) consists of a mapping between class names \( c \) and presence information that can either be \( \text{Pre}_J(\text{TC}) \), meaning that the class is present, where \( \text{TC} \) specifies which members are present and which are absent from the class, and \( J \) specifies the previous actions done on the class; or \( \text{Abs}_J(\text{TC}) \), meaning that the class is not part of the product, where \( \text{TC} \) specifies all members absent and stores the past actions done to them—normally this component would not be present, but the type system needs to track the previously applied actions, even when a member has been removed. Row polymorphism is enabled with variables \( \rho \) which stands for an unknown mapping.

The structure of the type \( \text{TC} \) is similar to \( TP \) with two differences: as members do not have an inner structure, presence information for them do not contain an inner type; and empty rows \( \text{Abs}_f^J \) (resp. row variables \( \rho_f^\gamma \)) are annotated with a general annotation \( J \) (resp. a conflict variable \( \gamma \)) to solve the technical difficulty of storing the past actions done on the members of a deleted class. More details can be found in the companion report [15]. Mappings \( \text{TCL}^c \) (resp. \( \text{TFL}^f \)) are annotated with sets of class names \( c \) (resp. member names \( f \)) to ensure that they are just defined once in a type. Deltas are typed with functional types \( \text{TD} \) where \( \alpha \) can either be a row variable \( \rho \) or a conflict variable \( \gamma \).

Annotations, \( J \), are used for conflict detection, which is structured into two steps. We first define the action of deltas inductively on their structure: each simple operator acting on an element \( E \) is typed with an annotation on \( E \) of the form \( \gamma; (d, A) \), where \( d \) is the name of the delta performing the operation, \( A \) is the performed action, and \( \gamma \) represents past actions done on \( E \). Using type unification, sequential composition of operators on the same element \( E \) result in annotations of the form \( \gamma; (d, A_1); \ldots; (d, A_n) \), which are transformed using a rewriting relation into \( \gamma; (d, A_1 \Delta \ldots \Delta A_n) \), corresponding to the action of \( d \) on the element \( E \). These annotations are used later, during the typing of products, to detect if they contain conflicts. This detection is done inductively on the structure of the list of delta \( DL \) declared in each product: given a list \( d_1 \ldots d_n \) that result in annotation \( J \) on an element \( E \) and a delta \( d \) that performs the action \( A \neq \perp \) on \( E \), using type unification, the list \( d_1 \ldots d_n \) \( d \) results in annotation \( J; (d, A) \) on \( E \), which is then checked and possibly transformed to record soft conflicts using helper function called \text{detect}. Terms of the form \( J; (d_1 \ldots d_n) \) generally represent soft conflicts involving the \( n \) deltas \( d_i \) (but can also be used to store delta names in some specific cases). Hard conflicts, corresponding to possible errors, are ill-typed and thus have no syntactic representation in \( J \).

Finally, to ensure the correctness of the conflict detection algorithm, past actions done on deleted members need to be remembered, even when the class itself has been deleted. This means that the type of the deletion of the class \( c \) should be able to identify each member \( f_i \) in \( c \), take their annotation \( J_i \), and specify that
the output product is typed with \( c : \text{Abs}\{\{f_i : \text{Abs}J_i; (d, \text{rem})\}\} \). As discussed in the companion report [15], it is not possible using standard unification to compute such a type. We solve this problem by using local substitutions [17,15], which allow conflict variables \( \gamma \) to be substituted locally into an element \( E \). For instance, it is possible to type the removal of class \( c \) with

\[
\{c : \text{Pre}_\gamma(\{\rho_\gamma'\})\} \rightarrow \{c : \text{Abs}_\gamma; (d, \text{rem})(\{\text{Abs}_{\gamma'}; (d, \text{rem})\})\}.
\]

It is possible to type the application of this operator to a product with class \( c \) by first unifying \( \rho \) with the structure of \( c \), producing an input type of the form \( \{c : \text{Pre}_J(\{f_1 : \text{Pre}_{\gamma'}; \ldots; f_n : \text{Pre}_{\gamma'}; \text{Abs}_{\gamma'}\})\} \), and then unifying each instance of variable \( \gamma' \) with the annotation local to each member.

**Relations.** We define two relations on our type syntax: a structural equivalence that identifies types with the same semantics, and the rewriting relation used for the computation the action of a delta.

**Definition 8.** A type context \( T \) is any type term with a hole \( \bullet \). Moreover, we say that a relation \( R \) is closed under type context iff

\[
\forall T, x, y, \quad x R y \Rightarrow T[x] R T[y]
\]

The structural equivalence \( \equiv \) between types is the smallest equivalence closed under type context satisfying the following rules, where \( a \) denotes either a class name or a member name, \( K \) denotes either \( \text{Abs} \), \( \text{Pre} \) or \( \text{Pre}(TC) \) and \( W \) denotes either a class list \( \text{TCL}^c \) or a member list \( \text{TFL}^f \):

\[
\begin{align*}
\text{Abs}^c \cup \{c\} & \equiv c : \text{Abs}_\perp; \text{Abs}^c \\
\text{Abs}_f^l \cup \{f\} & \equiv f : \text{Abs}_f; \text{Abs}_f^l.
\end{align*}
\]

The rewriting relation \( \triangleright \) is the smallest reflexive and transitive relation that is closed under type context satisfying the following rules:

\[
(d, A); (d, A') \triangleright (d, A \triangleright A')
\]

This equivalence relation states that the order in which the classes and members are typed is not important, and that the empty row \( \text{Abs}^f \) corresponds to classes and members being absent. Moreover, the rewriting relation states that when we have two consecutive actions for the same delta \( d \) (typically coming from two operators used in the definition of \( d \)), we can combine them using the operator \( \triangleright \), thus computing the action of \( d \). Finally, let \( \text{Norm} \vdash TD \) denote when the annotations \( J \) in \( TD \) have been fully reduced with the rewriting relation, i.e. when the actions of the deltas have been computed.

**Typing Rules.** The rules defining our type system are structured in three parts: classes and core products; operators on classes and products; and product lines, i.e. deltas, core definitions and products.
Core Products. The typing rules for core products are presented in Figure 5. The rules for cores and classes have the form $\Phi \vdash E : T$, where $\Phi$ is a delta context, either a pair $(d, \gamma)$ or $\perp$, $E$ is the typed term and $T$ its type. When $\Phi$ is a pair $(d, \gamma)$, we are currently typing the addition of some classes performed by delta $d$, where $\gamma$ is the previous actions done on the elements. The annotation $J$ is $\gamma; (d, \text{add})$. Otherwise, when $\Phi$ is $\perp$, we are typing a core and the annotation is $\perp$. The rule T:FL types the body of a class with a mapping stating that all the members of the class are present. The rule T:CL types a core product with a mapping stating that all the classes of the core are present, with their bodies typed with the previous rule.

Operators. The typing rules for operators are presented in Figure 6. Our typing statements for operators have the form $d \vdash E : T$ where $d$ is the delta in which the operators are declared, $E$ is the typed operator and $T$ is its type. The typing rules for operators on members and core products are almost identical to the ones presented in our previous work [16], with the addition of the annotations $J$ describing the actions performed by the operator on the typed element. For instance, the addition of a member $f$ by a delta $d$, typed with the rule T:AddMem, states that the operator expects: as input, a class with the member $f$ absent and annotated with a variable $\gamma$ to capture manipulation done by previous deltas; as output, the same class with member $f$ added (i.e. present) and with $(d, \text{add})$ added to $\gamma$, thus capturing the addition action performed by $d$ after the previous manipulations stored in $\gamma$.

To avoid duplication of typing rules, in the six last typing rules, we use $E$ for any language term, $T$ for any type and $\Lambda$ for either a delta name $d$ or a typing environments $\Gamma$ that map delta names and core names to their types (these environments are used to type product lines). The rule T:Seq types the sequential composition of operators. The rules T:Equiv and T:Rew introduce the usage of the structural equivalence $\equiv$ and rewriting relation $\prec$ in our type system. Finally, the rules T:Inst, T:Gen and T:Subst deal with type generalization and substitution, which can be freely applied.

Product Lines. The type rules for product lines are presented in Figure 7. The typing judgements have the form $\Gamma \vdash E : T$, where $\Gamma$ is the typing environment storing the type of deltas and cores, $E$ is the typed term and $T$ is its type in the
context $\Gamma$. The type $T$ can either be the type of a delta $TD$ or a mapping $II$ between product names and their type. The rule $T:D$ types delta declarations by: i) computing the type $TD$ of the delta’s body $COL$; ii) computing the action of the delta; the statement $Norm \vdash TD$ means that the annotations in $TD$ have been completely rewritten by $\triangleright$ (by construction of $\triangleright$, every annotation thus corresponds to the action performed by the delta on the annotated element); and iii) continuing the typing of the product line with the environment $\Gamma$ extended with a mapping between the delta and its type. The rule $T:K$ types core declarations by typing the declared classes and continuing the typing of $PL$ with the extended typing environment. The rule $T:P$ types a product by typing the list of deltas $DL$, ensuring that the core $k$ is a valid input for $DL$, and adding the type of the product to the rest of the mapping $II$. The rule $T:NAME$ is used to type names, where $n$ is either a core or a delta name.

The type of a list of delta names $DL$ is constructed using the four last typing rules; it works as follows. Using the typing rule $T:Eq$ based on the relation $\equiv_{dl}$,
\[
\begin{array}{l}
\text{T:D} \quad \frac{d \vdash COL : TD \quad \text{Norm} \vdash TD \quad \Gamma ; d:TD \vdash PL : \Pi}{\Gamma \vdash \text{delta } d \text{ after } DL \{COL\} PL : \Pi} \\
\text{T:K} \quad \frac{\bot \vdash CL : TP \quad \Gamma ; k : TP \vdash PL : \Pi}{\Gamma \vdash \text{core } k \{CL\} PL : \Pi} \\
\text{T:P} \quad \frac{\Gamma \vdash DL : TP \rightarrow TP' \quad \Gamma \vdash k : TP \quad \Gamma \vdash PL : \Pi}{\Gamma \vdash \text{product } p = \{DL\} k PL : \Pi ; p : TP'} \\
\text{T:DL-D} \quad \frac{\Gamma \vdash d : TP \rightarrow TP_3 \quad \Gamma \vdash d_1 \ldots d_n : TP_1 \rightarrow TP_2}{\forall 1 \leq i \leq n, d \not\subseteq_{PL} d_i \quad \text{detect}(d, TP_3) = TP_4} \\
\text{T:NAME} \quad \frac{\Gamma \vdash n : \Gamma(n)}{\Gamma \vdash \epsilon : \forall \rho . \{\rho\}} \\
\text{T:DL-E} \quad \frac{\Gamma \vdash \text{core } k \{CL\} PL : \Pi}{\Gamma \vdash \text{delta } d \text{ after } DL \{COL\} PL : \Pi} \\
\text{T:Eq} \quad \frac{\Gamma \vdash d_1 \ldots d_n : TP_1 \rightarrow TP_4}{\Gamma \vdash \text{product } p = \{DL\} k PL : \Pi ; p : TP'} \\
\end{array}
\]

**Fig. 7. Typing Product Lines**

we sort the list of delta names to match \(<_{PL} : DL\) is thus replaced by \(DL'\), corresponding to a valid computation of the product. This sorting is enforced by the statement \(\forall d' \in D_1 \ldots D_n, d \not\subseteq_{PL} d'\) in rule T:DL-D. Then, the list is typed inductively using T:DL-E for the empty list and T:DL-D to add new deltas to the list. Finally, the rule T:DL-D types the deltas in sequence, with the additional application of the function detect\((d, TP_3)\) to detect conflicts added or resolved by \(d\) using the annotations in \(TP_3\). This function traverses type \(TP_3\), applying detectClass on all annotations at the class level, and detectMem on all annotations at the member level. We present in Figure 8 these two functions detectClass and detectMem (we omit the straightforward presentation of detect) together with an helper function check which is used to check for hard conflicts between the delta in input and all the previous actions performed on the element. Note that because there are soft conflicts only at the member level, the function detectMem returns a modified annotation or an error ERR when an hard conflict is detected; detectClass returns only either OK when there are no conflicts, or ERR otherwise.

**Properties.** The type system combines the properties of the classic row types system and the conflict detection performed using annotations.

**Definition 9.** A type \(\Pi\) is said to be conflict free if there are no product name \(p\), type context \(T\) and annotation \(J\) of the form \(J' ; (d_1 \ldots d_n)\) such that \(\Pi(p) = T[J]\).

**Definition 10.** A product line \(PL\) has a delta application error iff during its reduction, either: i) a delta \(d\) that adds an element \(E\) is applied on a core that already contains \(E\); ii) a delta \(d\) that removes or modifies an element \(E\) is applied on a core that does not contain \(E\).
detectClass(d, J) {
  if J = J′;(d, A) then
    if J′ = J″;(d′, A′) then
      if d’ <PL d then
        OK
      else if A = A′ = mod then
        check(d, J″)
      else
        ERR
    else OK
  else OK
}
detectMem(d, J) {
  if J = J′;(d, A) then
    if J′ = J″;(d′, A′) then
      if d’ <PL d then J
    else if A = A′ = mod then
      if check(d, J″) = OK then
        J′′;(d1, d2)
      else
        ERR
    else
      ERR
  else if J′ = J″;(d1 ... dn) then
    S ← {dj | 1 ≤ j ≤ n ∧ dj ∉ PL d}
    if S = ∅ then
      J
    else if A = mod then
      if check(d, J″) = OK then
        S′ ← {dj | 1 ≤ j ≤ n} \ S
        J″;(S′); (d, S)
      else
        ERR
    else ERR
  else J
}

check(d, J) {
  if J = J′;(d′, A) then
    if d’ <PL d then OK
  else if A = mod then
    check(d, J′)
  else ERR
  else if J = J′;(d1 ... dn) then
    if ∃i, di < d then OK
  else check(d, J′)
  else OK
}

Theorem 2 (Soundness). Given a closed product line PL and type Π such that \( \emptyset \vdash PL : \Pi \) holds, there exists PL' with \( PL' \equiv_{d1} PL \) such that PL' does not have any delta application error.

Proof. Consider the derivation K of \( \emptyset \vdash PL : \Pi \). K can be transformed it into a derivation K' where we apply the rule T:Eq only once, at the end. Consider the term PL' typed by K' just before the application of T:Eq: this term will not have an error as our type system without T:Eq is more restrictive than [19]. □

Theorem 3 (Freedom). Given a closed product line PL and conflict free type Π such that \( \emptyset \vdash PL : \Pi \) holds. Then PL is conflict free.

5 Related Work

The goal of type checking the code base of a software product line is to ensure that the generated products are type safe, up to the degree of type safety provided by the base language, without having to actually generate the products. Other static analysis techniques can instead be employed to check for other potential deficiencies, without aiming to be ensure complete type safety.

Thaker et al. [23] describe an informally specified approach to the safe composition of software product lines that guarantees that no reference to an undefined
class, method or variable will occur in the resulting products. The approach is presented modulo variability given in the feature model and deals especially with the resulting combinatorics. The lack of a comprehensive formal model of the underlying language and type system was rectified with Lightweight Feature Java (LFJ) [8]. Underlying LFJ is a constraint-based type system whose constraints describe composition order, the uniqueness of fields and methods, the presence of field and methods along with their types, and feature model dependencies. The soundness of LFJ’s type system was validated using theorem prover Coq.

A formal model of a feature-oriented Java-like language called Featherweight Feature Java (FFJ) [2] presents a similar base language that also formalizes Thaker et al. [23]’s approach to safe composition, although for this system type checking occurs only on the generated product. Coloured Featherweight Java [11], which employs a notion of colouring of code analogous to but more advanced than #ifdefs, lifts type checking from individual products to the level of the product line and guarantees that all generated products are type safe. More recent work [1] refines the work on FFJ, expressing code refinements as modules rather than low-level annotations. The resulting type system again works at the level of the product line and enjoys soundness and completeness results, namely, that a product line is well-typed if and only if all of its derived products are well-typed.

In the above mentioned work the refinement mechanisms are monotonic, so no method/class removal or renaming is possible. Kuhlemann et al. [14] addresses the problem of non-monotonic refinements, though their approach does not consider type safety. They consider the presence of desired attributes depending upon which features are selected. Checking is implemented as an encoding into propositional formulas, which are fed into a SAT solver. Recent work addresses non-monotonic refinement mechanisms that can remove or rename classes and methods. An alternative approach due to Schaefer et al. [20] generate detailed dependency constraints for checking delta-oriented software product lines. The checking of the constraint is performed per product, rather than at the level of product lines. This approach to typing delta-oriented programs is complementary to our work, providing part of the checking we have omitted.

A number of static analysis techniques have been developed for the design models or code of software product lines. Heidenreich [10] describes techniques for ensuring that the correspondence between feature models, solution-space models, and problem-space models, which is realised in the FeatureMapper tool. In this tool, models are checked for well-formedness against their meta-model. Similarly, Czarnecki and Pietroszek [7] provide techniques for ensuring that no ill-structured instance of a feature-based model template will be generated from a correct configuration. Apel et al. [3] present a general, language independent, static analysis framework for reference checking—checking which dependencies are present and satisfied. This is one of the key tasks of type checking a software product line. Similar ideas are applied in a language-independent framework for ensuring the syntactic correctness of all product line variants by checking only the product line itself, again without having to generate all the variants [12].
Clarke et al. [4] present an abstract framework for describing about conflicts between code refinements and conflict resolution in the setting of delta-oriented programming. Padmanabhan and Lutz [18] describe the DECIMAL tool, which performs a large variety of consistency checks on software product line requirements specifications, in particular, when a new feature is added to an existing system. Techniques developed for the analysis and resolution of interference of aspects in AOP [13,9] address similar problems to analyses of software product line conflicts, but they do not consider variability.

6 Conclusion

This paper presented a simple language for delta-oriented programming and defines notions soft and hard conflicts, a type system based on row polymorphism to capture errors and on a new concept of annotations to capture conflicts. This paper also shows that, in contrast to Clarke et al. [4], the notion of conflict is not simple and accurately detecting them is not easy. Much work remains to be done. First, we need to extend our type system to ensure type safety not only of delta application, but also of the generated products. Then, we need to extend our calculus and type system to include features models. Whether this can be done while keeping the time complexity of our type system polynomial in the size of the product line remains to be seen.

References


Appendix D

Dynamic Delta Modeling

Abstract. Abstract Delta Modeling offers an algebraic description of how a (software) product line may be built so that every product can be automatically derived by structured reuse of code. In traditional application engineering a single valid feature configuration is chosen, which doesn’t change during the lifetime of the product. However, there are many useful applications for product lines that change their configuration at run time. We present a new technique for generating efficient dynamic product lines from their static counterparts. We use Mealy machines for their dynamic reconfiguration. Furthermore, we posit that monitoring some features will be more expensive than monitoring others, and present techniques for optimizing a Mealy machine to minimize cost. We illustrate these techniques through a mobile application for Android, which dynamically changes a devices profile based on environmental factors.

1 Introduction

Delta Modeling [13,11,12] is designed as a technique for implementing software product lines [10]: a way to optimally reuse code between software products which differ only by which features they support. The code is divided into units called deltas, which can incrementally transform a core product in order to mechanically generate a product in the product line. Each delta has an application condition, which indicates for which combinations of features the delta should be applied. The legal combinations of features are expressed through a feature model [2]. Such legal combinations of features are commonly referred to as feature configurations.

Clarke et al. [3, 4] described delta modeling in an abstract algebraic manner called the Abstract Delta Modeling (ADM) approach. In that work, delta modeling is not restricted to software product lines, but rather product lines of any domain. It gives a formal description of deltas, how they can be combined, how they can be linked to features from the feature model, as well as how to avoid and resolve implementation conflicts.

Traditionally, a feature configuration is chosen once at build-time, its corresponding product is generated, and it does not change at runtime. Dynamic (software) product lines [7] are product lines for which the feature configuration is not fixed at runtime. It can change dynamically, in order to meet changing requirements for continuously running systems. Dynamic product lines have already been discussed in the context of delta modeling [5], but only in a concrete object oriented setting (called delta oriented programming).
We now formalize dynamic product lines in the context of ADM. We show how to transform a static product line, as described in [3, 4], into a dynamic product line. It takes the form of a Mealy Machine, a finite automaton with an input symbol and an output symbol on every transition. In our case, the input symbol corresponds to a feature that has been turned on or off and the output symbol corresponds to the delta that has to be applied to the current product in order to bring it up to date.

Based on this representation of dynamic product lines, we introduce a cost model. We assume that monitoring a specific feature for change has a certain cost, and that some features are more costly than others. We then describe how to optimize dynamic product lines by selectively removing transitions from them, effectively disregarding costly features until they become relevant.

We also introduce a novel case study to demonstrate this approach for a concrete domain. To illustrate both dynamic product lines and the versatility of ADM, we do not use a traditional software product line. Instead, we use a profile manager for modern mobile devices, such as smartphones; a much more common use case these days. By monitoring personal data such as time, location and schedule, a smartphone can automatically adjust its internal settings based on user defined rules. We show that delta modeling and, by extension, dynamic delta modeling, are a natural fit for this use-case.

The rest of the paper is structured as follows. Section 2 summarizes the relevant theory of ADM. Section 3 introduces our case study, both as motivation and illustration. Dynamic product lines are formally described in Section 4 as an extension to ADM. Section 5 describes the cost model as well as our optimization strategy. Finally, Section 6 concludes the paper and discusses related and future work.

2 Preliminaries

To make this paper self-contained, we now repeat the relevant theory from ADM. For more detailed information, we refer the reader to [3, 4]. Readers familiar with the theory can skip this section.

2.1 Products and Deltas

Firstly, we assume a set of products, \( \mathcal{P} \). The set of possible modifications to products forms a delta monoid, as follows.

**Definition 1 (Delta Monoid).** A delta monoid is a monoid \((\mathcal{D}, \cdot, \epsilon)\), where \( \mathcal{D} \) is a set of product modifications (referred to as deltas), and the operation \( \cdot : \mathcal{D} \times \mathcal{D} \rightarrow \mathcal{D} \) 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.

Applying a delta to a product results in another product. This is captured by the notion of delta application.
Definition 2 (Delta Action). Delta action is an operation $-(-) : D \times P \to P$. If $d \in D$ and $p \in P$, then $d(p) \in P$ is the product resulting from applying delta $d$ to product $p$. It satisfies the conditions $(y \cdot x)(p) = y(x(p))$ and $\epsilon(p) = p$.

This all leads to the notion of a deltoid, which describes all building blocks necessary to create a product line in a concrete domain.

Definition 3 (Deltoid). A deltoid is a 5-tuple $(P, D, \cdot, \epsilon, -(-))$, where $P$ is a product set, $(D, \cdot, \epsilon)$ is a delta monoid and $-(-)$ is a delta application operator.

Finally, we describe a useful notion of expressiveness.

Definition 4 (Maximal Expressiveness). A deltoid $(P, D, \cdot, \epsilon, -(-))$ is said to be maximally expressive iff $\forall p, p' \in P : \exists x \in D : x(p) = p'$.

Having this property means that any product can be transformed into any other product by applying the right delta. From this point on, we assume that every deltoid is maximally expressive. This is often true in practice.

2.2 Product Lines

We will now formally define the representation of a product line. The following concepts are built upon some deltoid $(P, D, \cdot, \epsilon, -(-))$, which we assume as given in the definitions below.

A product line consists of several ingredients. There is a finite set of relevant features $F$. From a delta modeling perspective, these features have no inherent semantic meaning and are treated as labels.

Definition 5 (Feature Model). A feature model $\Phi \subseteq P(F)$ is a set of sets of features from $F$. Each $F \in \Phi$ is a set of features corresponding to a valid feature configuration, i.e. a set of features that may be active together.

A product line contains a core product $c \in P$ and an underlying delta model, the selective application of which to $c$ should be able to generate any product in the product line. A delta model describes the set of deltas required to build a specific product, along with a strict partial order on those deltas, restricting the order in which they may be applied.

Definition 6 (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 $x$ must be applied before $y$, though not necessarily directly before.

The partial order represents the intuition that a delta applied later has full access to earlier deltas and more authority over modifications to the product.

To connect features and deltas, we use application conditions.

Definition 7 (Application Function and Condition). Given a set of deltas $D \subseteq D$ and feature model $\Phi$, an application function $\gamma : D \to \mathcal{P}(\Phi)$ is used to map each delta $x \in D$ to its application condition. An application condition $\gamma(x) \subseteq \Phi$ determines for which feature configurations delta $x$ needs to be applied.
So a product line describes all possible products, and how to generate them.

**Definition 8 (Product Line).** A product line is a tuple $(F, \Phi, c, D, \prec, \gamma)$ where $F$ is a feature set, $\Phi \subseteq \mathcal{P}(F)$ is a feature model, $c \in \mathcal{P}$ is a core product, $(D, \prec)$ is an underlying delta model and $\gamma : D \rightarrow \mathcal{P}(\Phi)$ is an application function.

If some feature configuration $F$ is valid according to $\Phi$, its corresponding product can be generated by $\text{prod}(PL, F)$, which uses all elements of the product line. For a detailed description of this process, we refer the reader to [3, 4].

### 3 Case Study

Traditionally a product line only exists in the development stage [10]. It is a way to efficiently reuse code between otherwise separate products. However, the delta modeling formalism described in Section 2 is a surprisingly good fit for an entirely different kind of use-case. That of a dynamic product line, which reconfigures itself at runtime.

#### 3.1 Profile Management

The first thing that may come to mind when thinking about dynamic delta modeling is a software product line with features that can be dynamically turned on and off, based, for example, on the personal preferences of who is using the software at the time, or because of changing requirements. [7, 5]

While that would be a valid use-case, we choose to go a different direction in motivating and explaining the research in this paper: profile management on modern smartphones and other mobile devices. This may prove more interesting, as it departs from what a product line is traditionally supposed to be.

Modern smartphones and tablets, such as those based on Android [6], iOS [1] or Windows Phone [9], have access to a great variety of data with regard to the current circumstances of their user: They know the current time and their current physical location. They know which application the user is currently running, what their scheduled appointments are, and much more. This sort of information can be used to automatically adjust the devices settings based on user defined rules, such as: “when my headphones are plugged in, play music” or “when my battery is running low, turn down screen brightness”. This is known as automated profile management. We show that delta modeling is a natural way to model such rules.

The basic idea is this: a profile, consisting of a specific configuration of the devices settings, is a product in the product line. Deltas can be applied to modify those settings. Features are now specific conditions on (environmental) factors monitored by the device. Application conditions are sets of feature configurations, as before, and now represent the exact conditions under which certain settings should be modified.

A profile management (Delta Profiles [8]) application for Android [6] is currently in the final stages of development. It uses the theory in this section.
3.2 Profile Deltoid

We now formalize these notions by defining a concrete deltoid (Definition 3). Some formerly abstract constructs are refined in this section, such as what a product, a feature and a delta is. Note, however, that the remainder of this paper is still in an abstract setting, and these refinements are only used when referring to Section 3 for illustration.

We start by narrowing down, for a specific device, what it is capable of monitoring and modifying for us:

**Definition 9 (Device).** A device is a 4-tuple \((FC, S, V, vl)\) where \(FC\) contains the names of the factors that the device is capable of monitoring (e.g. ‘time’, ‘gps location’, ‘battery status’), \(S\) contains the names of the settings the device controls (e.g. ‘volume’, ‘brightness’, ‘im status’), \(V\) is the encompassing set of possible values for these factors and settings and \(vl : (FC \cup S) \to \mathcal{P}(V)\) maps each factor and setting to the set of values it is allowed to take on.

To simplify matters, we assume that \(FC\) and \(S\) are disjoint, although in practice this will not be the case. i.e., we might want to modify the devices settings based on monitoring other settings. This may introduce a host of new challenges that we plan to address in future work.

Each possible device defines a concrete deltoid upon which (dynamic) product lines can be created. From this point on, we assume that some specific device \((FC, S, V, vl)\) is given, and define a deltoid based in that device.

**Definition 10 (Profiles).** We refine the set of possible products to the set of possible profiles, mapping settings to values:

\[
P \equiv S \to V
\]

such that for each \(s \in S\) we have \(P(s) \in vl(s)\). (see Definition 9)

**Example 1 (Profile)** As an example, consider the following profile:

\[
P_{e} = \begin{cases} 
'volume' \mapsto 10, \\
'bluetooth' \mapsto \text{on}, \\
'brightness' \mapsto 3, \\
'foreground app' \mapsto 'clock', \\
\vdots 
\end{cases}
\]

Since \(S\) is usually quite large, we show only the relevant parts of profiles.

**Definition 11 (Profile Deltas).** We refine the set of deltas to the set of profile deltas, which can modify profiles:

\[
\mathcal{D} \equiv S \to V.
\]

such that for each \(s \in \text{dom} \mathcal{D}\) we have \(\mathcal{D}(s) \in vl(s)\) (see Definition 9). \(\mathcal{D}\) is similar to \(\mathcal{P}\), as both map settings to values, but \(\mathcal{D}\) is a partial function. Settings that are not mapped are not modified by the delta. Neutral delta \(\epsilon\) doesn’t map any settings, so it modifies nothing.
Example 2 (Profile Delta) As an example, take the following profile delta:
\[
d_e = \{ 'volume' \mapsto 5, 'foreground app' \mapsto 'calendar' \}.
\]
We assume that settings that are not mentioned, are not mapped.

When a delta is applied to a profile, it overwrites that profile’s values:

**Definition 12 (Profile Delta Action).** Profile delta action \( -(-) : \mathcal{D} \times \mathcal{P} \to \mathcal{P} \) is defined as follows, for all \( d \in \mathcal{D} \), \( p \in \mathcal{P} \) and \( s \in \mathcal{S} \):

\[
(d(p))(s) \overset{\text{def}}{=} \begin{cases} 
  d(s) & \text{if } s \in \text{dom}(d) \\
  p(s) & \text{otherwise}
\end{cases}
\]

Example 3 (Profile Delta Action) For example, delta \( d_e \) (from Example 2) applied to product \( p_e \) (from Example 1) results in the following product:

\[
d_e(p_e) = \begin{cases} 
  'volume' \mapsto 5, 'bluetooth' \mapsto \text{on}, 'brightness' \mapsto 3, 'foreground app' \mapsto 'calendar'. \\
  \vdots
\end{cases}
\]

Profile delta composition \( \cdot : \mathcal{D} \times \mathcal{D} \to \mathcal{D} \) is implicitly defined by Definition 12, and can be seen as function composition. Deltas that are applied later can overwrite settings from deltas that are applied earlier.

This defines a concrete deltoid \((\mathcal{P}, \mathcal{D}, \cdot, \epsilon, -(\cdot))\) for each device \((FC, S, V, vl)\).

### 3.3 Rule Sets as Product Lines

The idea behind the profile manager application is that the user manually creates a set of rules using the app’s graphical interface, after which it is put into effect as a dynamic product line, regulating the devices’ profiles.

Let us examine the parts of a profile manager product line more closely. Recall that an abstract product line consists of \((\mathcal{F}, \Phi, c, \prec, \gamma)\). From these, \(\Phi\), \(\prec\) and \(\gamma\) are based directly on user-defined rules. The rest is implicitly derived.

Before we describe the ingredients of such a product line, we refine the notion of ‘feature’ that has to be used:

**Definition 13 (Conditions).** We restrict the set of features for any product line for a device \((FC, S, V, vl)\) to the set of possible conditions over factors:

\[
\mathcal{F} \subseteq FC \times \mathcal{P}(V)
\]

such that for each \((fc, V') \in \mathcal{F}\) we have \(V' \subseteq vl(fc)\) (see Definition 9).
Example 4 (Condition) As an example, consider the following conditions:

\[ f_{e1} = ('time', \text{ between 9:00 and 17:00}) \]
\[ f_{e2} = ('gps\ location', \text{ within 1 km of } [+52° 21' 23", +4° 57' 8"]). \]

Note that these names are mapped to a set of values. A condition is satisfied (a feature is ‘on’) if the actual value is contained within that set.

The user can input rules which include conditions and effect. These form the application function \( \gamma \) (Definition 7) and the delta set \( D \) respectively.

Example 5 (Rule) For example, the rule: “if I am within 1 km of [+52° 21' 23", +4° 57' 8"] and the time is between 9:00 and 17:00, set the volume to 5 and set the calendar application to the foreground” is encoded as follows (using Examples 2 and 4):

\[ D \ni d_e \]
\[ \gamma(d_e) = \{ F \in \Phi \mid \{ f_{e1}, f_{e2} \} \subseteq F \} \]

A partial order \( \prec \) can be defined between the deltas to encode rule priority, in order to avoid and resolve conflicts. Basically, a delta can overwrite values of other deltas if it is greater in the partial order.

The other elements of the product line are implicitly derived, as follows. \( F \) contains all conditions that appear in application function \( \gamma \):

\[ F = \bigcup_{d \in D} \gamma(d) \]

The feature model \( \Phi \) (Definition 5) is implicitly defined by the conditions in \( F \). Basically, \( \Phi \) consists of all feature configurations that do not contain any contradictory conditions but do contain all implied conditions. Finally, for our profile manager, we assume the core product \( c \) to be the devices profile as the user has manually set it before any deltas are applied. In effect, in \( c \), every value is set to ‘manual’ \( \in V \).

3.4 Example Rule Set

We now give an example set of rules and the corresponding product line:

- If I am within 1 km of [+52° 21' 23", +4° 57' 8"] and the current time is between 9:00 and 17:00 then
  - set ‘volume’ to 5.
- If I currently have a meeting scheduled then
  - set ‘volume’ to 0 and
  - set ‘foreground app’ to ‘meeting minutes’
and this rule has priority over the previous rule.
Note that we need to establish a priority between the two rules, as they might otherwise conflict with each other on the volume setting. These are the deltas:

\[ D \ni d_1 = \{ \text{‘volume’} \mapsto 5 \}, \]
\[ D \ni d_2 = \{ \begin{align*}
\text{‘volume’} & \mapsto 0, \\
\text{‘foreground app’} & \mapsto \text{‘meeting minutes’}
\end{align*} \}. \]

with \( d_1 \prec d_2 \). And these are the application conditions:

\[ \gamma(d_1) = \{ F \in \Phi \mid \{ t, l \} \subseteq F \}, \]
\[ \gamma(d_2) = \{ F \in \Phi \mid \{ m \} \subseteq F \} \]

where

\[ t = \{ \text{‘time’, between 9:00 and 17:00} \}, \]
\[ l = \{ \text{‘gps location’, within 1 km of [+52° 21’ 23”, +4° 57’ 8”]} \}, \]
\[ m = \{ \text{‘ongoing meeting’, yes} \}. \]

Based on this, \( F = \{ t, l, m \} \) and the feature model \( \Phi = \mathcal{P}(F) \) contains all combinations of conditions, as none of them exclude or imply each other. So there are 8 possible profiles. We will use this product line for illustration later.

## 4 Dynamic Product Lines

A naive way to turn a static product line \( PL \) into a dynamic product line (DPL), i.e. to dynamically switch from one feature configuration \( (F) \) to another \( (F') \) and keep the current product consistent with that feature configuration, is to generate the product in the traditional way, namely \( \text{prod}(PL, F') \), each time the feature configuration changes. However, this can be rather costly, and hurt performance. The other extreme is to pre-generate all products, and to continually switch between them. However, the number of possible products can be exponential in the number of features, so this is infeasible for non-trivial product lines.

### 4.1 Dynamic Product Lines as Mealy Machines

Instead, we represent a DPL as a Mealy Machine. A Mealy Machine is a finite automaton with an input symbol and an output symbol on each transition. We assume that in a DPL the feature configuration changes dynamically by individual features being sequentially turned on and off with regard to the current feature configuration. These features are used as input symbols for our Mealy Machine. The output symbols are deltas, which, when applied to the current product, yield a new product consistent with the new feature configuration.

**Definition 14 (Mealy Machine).** A Mealy Machine is represented as a 5-tuple \((S, \Sigma, \Lambda, \mathcal{T}, \mathcal{G})\) where \( S \) is a set of states, \( \Sigma \) is an input alphabet, \( \Lambda \) is an output alphabet, \( \mathcal{T} : S \times \Sigma \rightarrow S \) maps a state and input symbol onto a next state
and \( G : S \times \Sigma \rightarrow \Lambda \) maps a state and input symbol onto an output symbol. \( T \) and \( G \) are partial functions but are defined for the same inputs, i.e. \( \text{dom}(T) = \text{dom}(G) \). Sometimes the definition of Mealy Machine includes an initial state, but in our case the initial state is not fixed, so we do not include one.

**Notation 1 (Symmetric Difference)** Because input symbols in our Mealy Machines will turn a feature from a feature configuration either on or off, we often use symmetric difference between sets, which is denoted:

\[
F_1 \ominus F_2 \overset{\text{def}}{=} (F_1 \cup F_2) \backslash (F_1 \cap F_2).
\]

**Notation 2 (Product Difference)** Note that since we assumed our deltoid to be maximally expressive (Definition 4), any product can be transformed into any other by application of the right delta. The delta that transforms \( p \) into \( p' \) is denoted \( p \mapsto p' \):

\[
(p \mapsto p')(p) = p'
\]

**Definition 15 (Dynamic Product Line).** Given a product line \( PL = (F, \Phi, c, D, <, \gamma) \), we define its dynamic product line as a Mealy Machine \((\Phi, F, D, T, G)\) where:

- Every state is a valid feature configuration in \( \Phi \).
- Every input symbol is a feature in \( F \).
- Every output symbol is a delta in \( D \).
- \( T(F, f) = F \ominus \{f\} \). A feature input is an event that triggers that feature to be added to or removed from the originating state, resulting in a target state.
- \( G(F, f) = \text{prod}(PL, F) \mapsto \text{prod}(PL, T(F, f)) \). The output generated by a transition is the delta required to transform the product corresponding to the originating state to the product corresponding to the target state.

\( T(F, f) \) and \( G(F, f) \) are only defined if \( F \ominus \{f\} \in \Phi \).

A dynamic product line has to be generated only once, and can then be run indefinitely. Figure 1 shows a graphical representation of the dynamic product line generated from our Section 3.4 example. Besides the conditions \( t, l, m \) and deltas \( d_1, d_2 \) given there, the three extra deltas that appear in the diagram are:

\[
\begin{align*}
d_1' &= \{ \text{‘volume’} \mapsto \text{‘manual’} \}, \\
d_2' &= \{ \text{‘volume’} \mapsto \text{‘manual’}, \text{‘foreground app’} \mapsto \text{‘manual’} \}, \\
d_2'' &= \{ \text{‘volume’} \mapsto 5, \text{‘foreground app’} \mapsto \text{‘manual’} \}.
\end{align*}
\]

Note from Figure 1 that we have entered a very different view of our product line than the one formed in Section 2.2. Whereas there deltas and their partial order were the main focus, we assume now that those have been processed, and we are looking at the different products of the product line, and how they relate to each other.
Fig. 1. The dynamic product line of the Section 3.4 example. Note that when looking at it as a cube, each dimension represents one feature.

**Notation 3 (Outgoing Transitions)** We write the set of outgoing transitions from state $F \in \Phi$ as:

$$\text{out}(F) \overset{\text{def}}{=} \{ f \mid (F, f) \in \text{dom}(T) \}$$

We now prove that if you start with a product corresponding to some state, walking any path from that state in the DPL and applying the output deltas to that product results in a product consistent with the new state.

**Theorem 1 (Product Consistency).** Given a dynamic product line $DPL = (\Phi, F, D, T, G)$ generated from $PL$. For any two states $F, G \in \Phi$ and any path $f_1, \ldots, f_n \in F^*$ such that $T(\cdots T(F, f_1) \cdots , f_n) = G$, deltas $d_1, \ldots, d_n \in D^*$ are encountered as output such that $d_n(\cdots d_2(d_1(\prod(PL, F)))) \cdots ) = \prod(PL, G)$.

**Proof.** Induction on $n$:

- $n = 0$. Immediate.
- $n > 0$. Take $F' = T(F, f_1)$:

$$\begin{align*}
& d_n(\cdots d_2( d_1(\prod(PL, F))) \cdots ) = (\text{Definition 15}) \\
& d_n(\cdots d_2( G(F, f_1)(\prod(PL, F))) \cdots ) = (\text{Definition 15}) \\
& d_n(\cdots d_2( (\prod(PL, F) \mapsto \prod(PL, F'))(\prod(PL, F))) \cdots ) = (\text{Notation 2}) \\
& d_n(\cdots d_2( \prod(PL, F')) \cdots ) = (\text{induction}) \\
& \prod(PL, G) \square
\end{align*}$$
4.2 Event Based Strategy

We first describe the most straightforward strategy for ‘running’ a DPL.

We assume that there is some target state $tfc \in \Phi$, indicating the feature configuration we ‘want’ to be in. Before the Mealy Machine starts, we set the current state $cfc \in \Phi$ equal to the target feature configuration. The current product $cp$ is then generated in the traditional static way, consistent with $tfc$:

$$
cfc \leftarrow tfc
$$
$$
.cp \leftarrow \text{prod}(PL, cfc).
$$

When the machine is running we assume that, when in the current state $cfc$, every feature $f$ that is accepted as input from there is monitored by the system somehow. When that feature is turned on or off, giving $tfc = cfc \oplus \{f\}$, an input symbol $f$ is generated to make the current state equal to the target state again. And in that way a stream of input symbols is generated for our Mealy Machine, which returns the appropriate deltas to keep our product up to date. So, for every input symbol $f$, do the following:

1. update the product $cp \leftarrow G(cfc, f)(cp)$
2. set the next state $cfc \leftarrow T(cfc, f) = tfc$

After that, the DPL has stabilized, until $tfc$ changes again.

We need to make sure that any strategy we use results in a new current product consistent with the target state after the Mealy Machine has stabilized.

**Definition 16 (Correct Target Product).** We say that a certain strategy for running a DPL reaches a correct target product iff whenever $cfc \neq tfc$, the Mealy Machine will stabilize after which $cp = \text{prod}(PL, tfc)$.

**Theorem 2.** The event based strategy always reaches a correct target product.

**Proof.** By initialization, $cp$ is consistent with $tfc$ in the beginning. Whenever $tfc$ changes, there is an $f \in \text{out}(cfc)$ (Notation 3) such that $tfc = cfc \oplus \{f\}$ (Definition 15). Feature event $f$ is fired, giving $cp \leftarrow G(cfc, f)(cp)$ and $cfc \leftarrow T(cfc, f) = tfc$. So by Theorem 1, $cp = \text{prod}(PL, cfc) = \text{prod}(PL, tfc)$. $\square$

5 Optimization

In this section, we discuss the cost of dynamic product lines, and the minimization of that cost.

5.1 Cost Model

We assume that occupying a state in a DPL has a cost: the cost of monitoring the features of the outgoing transitions for change. We posit that monitoring some features will be more expensive than monitoring others. For example, in Section 3.4 it will cost more power to continually monitor the current ‘gps location’ $(f_l)$ than it will to monitor the current ‘time’ $(f_t)$.

First we introduce some cost domain $C$ over time (such as power in watt).
Definition 17 (Cost). Given dynamic product line \((\Phi, F, D, T, G)\) we introduce a function \(\text{cost} : \Phi \times F \rightarrow C\). The function \(\text{cost}(F, f)\) expresses the cost of monitoring feature \(f\) from state \(F\). A feature is only monitored from a state if it has an outgoing transition there, so if \((F, f) \notin \text{dom}(T)\) then \(\text{cost}(F, f) = 0\).

In the Section 3 profile manager, we assume that the cost of monitoring a condition depends only on the factor being checked, so we can use a short notation for that: \(\text{cost}(f) = \text{cost}(F; (f, V'))\) for all \(F \in \Phi, f \in FC\) and \(V' \subseteq V\).

Basically, we minimize the cost of a DPL by removing costly transitions, but only where this would not ‘break’ the machine. This means we only need to monitor features when they become relevant. For example, in the Section 3 profile manager, we want to modify the profile when we are in a certain ‘gps location’ \((f_l)\) at a certain ‘time’ \((f_t)\). Either condition satisfied on its own does not modify the profile. So it makes sense to only start monitoring ‘gps location’ (the more costly factor), when it is already the right ‘time’.

We need to find conditions under which transitions may be removed, as well as an accompanying strategy for walking through a reduced DPL. We give possible solutions in Sections 5.2 and 5.3.

5.2 Reduced Dynamic Product Lines

We first establish some useful notions. We set up an equivalence relation between different feature configurations that correspond to the same generated product.

Definition 18 (Feature Configuration Equivalence). Given a product line \(PL = (F, \Phi, c, D, \prec, \gamma)\), two feature configurations \(F, G \in \Phi\) are equivalent, denoted \(F \equiv G\), iff \(\text{prod}(PL, F) = \text{prod}(PL, G)\).

Realize that while running a DPL, by Definition 16, we don’t care whether we end up in the exact target state. It is sufficient that we always reach an equivalent state. That is why this will be a useful notion once we start removing transitions.

When walking a DPL from which transitions are removed, it is possible that our path will be blocked, preventing current state \(cfc\) to become equal to target state \(tfc\). This means that even though \(tfc\) will only change by one feature at a time, a gap greater than one feature may appear between the two. In Section 5.3, we will explain how to walk through such a machine by nondeterministically firing a transition whenever some are available, to bring \(cfc\) closer to \(tfc\).

Definition 19 (Available Transitions). For current state \(cfc \in \Phi\) and target state \(tfc \in \Phi\), the set of available transitions is defined as:

\[
\text{at}(cfc, tfc) \overset{\text{def}}{=} (cfc \odot tfc) \cap \text{out}(cfc)
\]

We give a new transition function which takes a target state, rather than a single feature, and returns the set of possible resulting states. It is defined recursively.
**Definition 20 (Target Based Transition Function).** Given dynamic product line \((\Phi, F, D, T, G)\), the target based transition function \(\overline{T} : \Phi \times \Phi \rightarrow \mathcal{P}(\Phi)\) is defined as:

\[
\overline{T}(cfc, tfc) \overset{\text{def}}{=} \bigcup_{f \in \text{at}(cfc, tfc)} \overline{T}(T(cfc, f), tfc)
\]

It always terminates, as the set \(\text{at}(cfc, tfc)\) becomes smaller when a step is taken.

We now define what a reduced dynamic product line is:

**Definition 21 (Reduced Dynamic Product Line).** The Mealy Machine \(DPL' = (\Phi, F, D, T', G')\) is a reduced dynamic product line if there is a dynamic product line \(DPL = (\Phi, F, D, T, G)\) such that:

- \(\forall (F, f) \in \text{dom}(T') : T'(F, f) = T(F, f)\)
- \(\forall (F, f) \in \text{dom}(G') : G'(F, f) = G(F, f)\)
- \(\forall cfc, tfc \in \Phi : \forall F \in \overline{T}(cfc, tfc) : F \equiv tfc\)

The first two conditions indicate that the set of transitions in \(DPL'\) should be a subset of the transitions in \(DPL\). The third condition makes sure that despite the ‘missing’ transitions, we still always reach a correct target product. Note that by that third condition, we may only remove transitions with \(\epsilon\) output.

Figure 2 shows a reduced version of the dynamic product line of Figure 1 with equivalence classes marked (basically, wherever an \(\epsilon\) transition was).
Let us assume that for the device we are considering:

\[ \text{cost('time') < cost('ongoing meeting') < cost('gps location')} \]

So we would like to get rid of \( \ell/\epsilon \) transitions first, then \( m/\epsilon \) transitions, then \( t/\epsilon \) transitions, so long as the third condition from Definition 21 continues to hold.

As you can see in Figure 2, we were able to remove 10 transitions, significantly reducing the overall cost of the DPL. Intuitively, the transitions between \( \emptyset \) and \( l \) could be removed, because the ‘gps location’ does not become relevant until it is the right ‘time’. The other 8 transitions could be removed because \( d_2 \) completely overwrites the effect of \( d_1 \), so it does not matter what happens with \( t \) and \( l \) while we are in a scheduled meeting. Profile manager rules often look like this in practice, so the DPL can often be significantly optimized by this process.

5.3 Target Based Strategy

The strategy of Section 4.2 is most straightforward, but it will no longer reach a correct target product (Definition 16) after we have removed transitions, because we might disregard a feature event when we are not monitoring it, and then neglect to apply it when it does becomes relevant. The solution is to modify our strategy to take this into account, to safely walk a reduced DPL (Definition 21).

The target state \( tfc \) will gradually change by single features being turned on and off, as before. But now the current state \( cfc \) will always try to ‘mimic’ the target feature configuration by generating feature events based on the current difference between the two (Definition 19) and walking the Mealy Machine accordingly. For example, if the current state is \( \{f, g\} \) and the target state is \( \{g, h\} \), the \( f \) and \( h \) events may be triggered in a nondeterministic order until \( cfc = tfc \) or until we are blocked from going any further by missing transitions.

The target based strategy works as follows. Given the reduced dynamic product line \( DPL = (\Phi, F, D, T, G) \) created originally from product line \( PL \), we first set the initial state \( cfc \) to equal \( tfc \), and its corresponding product \( cp \) is generated in the familiar static way as before:

\[
\begin{align*}
    cfc & \leftarrow tfc \\
    cp & \leftarrow \text{prod}(PL, cfc)
\end{align*}
\]

When we start running the DPL, \( tfc \) may change and the set of available transitions \( at(cfc, tfc) \) will become non-empty. We then initialize an empty delta \( d \):

\[
d \leftarrow \epsilon
\]

and repeat the following until \( at(cfc, tfc) \) is empty again:

1. nondeterministically choose a feature \( f \in at(cfc, tfc) \),
2. compose the resulting delta \( d \leftarrow G(F, f) \cdot d \) and
3. set the next state \( cfc \leftarrow T(cfc, f) \).
This is consistent with the target based transition function from Definition 20.

When \( at(cfc, tfc) \) is once again empty, and the dynamic delta model has stabilized, we apply the accumulated deltas to the current product:

\[
cp \leftarrow d(cp)
\]

We now prove that the target based strategy always results in a correct target product (Definition 16) for reduced dynamic product lines.

**Theorem 3.** The target based strategy applied on a reduced dynamic product line always reaches a correct target product.

The proof of this theorem relies heavily on the third condition of Definition 21, which is admittedly a strong condition. Finding an efficient algorithm for deciding whether or not a transition may be removed is future work.

**Proof.** By initialization, \( cp \) is consistent with \( tfc \) in the beginning. Then, whenever \( tfc \) changes, \( cfc \) will become one of the states in \( T(cfc, tfc) \) after stabilization. By Theorem 1, we have \( cp = prod(PL, cfc) \). By Definition 21, we will have \( cfc \equiv tfc \) and thus \( cp = prod(PL, tfc) \).

With an example walk in Figure 2, we show how we can always reach a state equivalent to the one targeted. We start in state \( cfc = tfc = \emptyset \). Say we reach specified gps coordinates before 9:00. Transition \( l \) would be fired, if it were available. As it is, \( cfc = \emptyset \) and \( tfc = \{l\} \). If we then start a scheduled meeting, \( m \) is fired, and \( cfc = \{m\} \) and \( tfc = \{l, m\} \). If the meeting ends (bringing us back to the previous situation), and it becomes 9:00, \( cfc = tfc = \{t, l\} \), because \( t \) and \( l \) both fire. In all cases, we have \( cfc \equiv tfc \), so we always reach the correct product.

### 6 Conclusion

Dynamic Delta Modeling is an extension of Abstract Delta Modeling [3, 4] which includes a formal framework for modeling dynamic product lines. Using Mealy Machines, we accurately describe the behavior of product lines with dynamic feature configurations, while remaining on an abstract level. We have defined a cost-model, and shown an optimization opportunity for certain kinds of dynamic product lines. We have described the practical case-study of profile management on modern mobile devices directly in our formal framework, illustrating the versatility of ADM and the applicability of its dynamic extension.

### 6.1 Related Work

Hallstein et al. [7] introduce several properties they believe constitute a dynamic software product line. Dynamic Delta Modeling allows several of these, such as ‘dynamic variability’, ‘changes binding several times over lifetime’ and ‘context awareness’, but does not yet model others, such as ‘variation point change during runtime’ and ‘deals with unexpected changes during runtime’. In
our approach, even though the current feature configuration can change during runtime, the set of available feature configurations is still fixed at build time.

The work of Damiani and Schaefer [5] complements our work. A Mealy Machine generated by the technique in this paper, in a concrete object oriented setting, may be enriched by their reconfiguration translations, basically embedding a reconfiguration automaton into our DPL, and in that way reconfiguring existing objects in the heap to be consistent with the change in code. To be fully compatible with their technique, stabilization of the DPL should wait until the reconfigure statement is encountered by the program.

6.2 Future Work

We have made a beginning, but there is plenty of opportunity for future work. Optimization of the dynamic product line is still a complex endeavor, so there is need for an efficient algorithm. Also, we now assume that the target feature configuration can only change by single features being turned on and off and we would like to loosen that restriction, a goal that introduces several complications. On a more concrete level, it would be interesting to enrich the profile manager to allow monitoring – as well as modifying – settings. As delta application would be able to trigger a new feature configuration change, some sort of termination analysis would be required.

References