Project N°: 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
   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
   2.1 Related Work ............................................................................ 9
   2.2 The ABS Approach ..................................................................... 10
   2.3 Example ..................................................................................... 11
   2.4 Conclusion ................................................................................. 12

3 A Type-Safe Model of Adaptive Object Groups
   3.1 Introduction .............................................................................. 13
   3.2 Related Work ............................................................................ 14
   3.3 A Kernel Language for Adaptive Object Groups ............................ 15
      3.3.1 The Syntax ........................................................................ 15
      3.3.2 Example ............................................................................. 16
   3.4 A Type and Effects System ......................................................... 16
   3.5 Operational Semantics ............................................................... 17
   3.6 Conclusion ............................................................................... 18

4 Conflict Detection in Delta-Oriented Programming
   4.1 Introduction .............................................................................. 20
   4.2 Related Work ............................................................................ 21
   4.3 Delta-Oriented Programming ...................................................... 23
   4.4 Conflicts ................................................................................... 23
   4.5 Type System ............................................................................. 24
   4.6 Conclusion ............................................................................... 27

5 Dynamic Modelling of Product Lines
   5.1 Introduction .............................................................................. 28
   5.2 Related Work ............................................................................ 28
   5.3 Example ..................................................................................... 29
   5.4 Dynamic Product Lines ............................................................. 29
   5.5 Cost and Optimisation ............................................................... 30
5.6 Conclusion

6 MetaABS
6.1 Introduction
6.2 MetaABS Interface
   6.2.1 MetaABS Types
   6.2.2 Usage Example
6.3 A Dynamic Back End for ABS
   6.3.1 Design
   6.3.2 Usage
   6.3.3 Code Generation
6.4 Applications
   6.4.1 User-Defined Process Schedulers
   6.4.2 Dynamic Product Reconfiguration
6.5 Conclusion

7 Conclusion

Bibliography

Glossary
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) [45] 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 [32] 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 [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 that can guarantee freedom of conflicts. The validation of delta modules provides trustworthiness guarantees to 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 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 ABS Back End and MetaABS

The tool support for the ABS language provides several back ends, each providing an executable version of a given ABS model. A good overview of this framework is given in Deliverable 1.2. 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 ABS programs. The latter is based on the term-rewriting engine Maude and follows faithfully the operational rules of ABS. Using Maude, the execution of an 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 ABS program addresses objectives of other tasks of the 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 ABS language requiring runtime capabilities, we present a reflective layer for ABS, called MetaABS. MetaABS provides support for meta-programming, exposing to the programmer runtime objects and classes of 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 ABS programs. Therefore the intention is that it is only made available for developers of tools for ABS, and not for the traditional software developer.

The execution of MetaABS in Java is achieved via a new back end, the dynamic Java back end. This back end keeps information regarding the state of the running program using data structures, as opposed to convert ABS classes directly into Java classes. The result is an integrating layer with runtime support for tools for 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.

**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 Bjørk, 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.

**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.

**Paper 4: Dynamic Delta Modeling**

This paper [44] 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.
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, 31, 16, 57, 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, 64, 88, 68, 62, 60]. 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 [60] 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 \cite{53}. 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 \cite{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 `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 ordinary assignments.

2. The possibility to annotate methods with the keyword `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 \texttt{Document}), modifies it using another resource (modelled with the class \texttt{Operator}) and then sends it to a printer (modelled with the class \texttt{Printer}). We also assume that the protocol used by \texttt{Operator}s 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 \texttt{OperatorFrontEnd} implements the protocol in the method \texttt{modify}; and the class \texttt{WFController} encodes the workflow. The elements \_\texttt{op}, \_\texttt{doc} and \_\texttt{p} are \texttt{ports}, annotated with \texttt{port}, and represent dependencies to external resources. It is only possible to modify their value using the construct \texttt{rebind}, which checks if the object is in a safe state (no critical method in execution) before modifying the port. Moreover, methods \texttt{modify} and \texttt{newInstanceWF} make use of these ports in their code, and are thus annotated as \texttt{critical} as it would be dangerous to rebinding 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 \texttt{changeOperator}. First is the \texttt{await} statement, which waits for the objects \texttt{this} and \_\texttt{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 rebinding 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 \[16\], 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 \[37\] 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 \texttt{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 \texttt{acquire \( I \) in \( y \) except \( T \)} 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 \( T \). If the \texttt{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 \texttt{x joins \( y \) 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 \texttt{x leaves \( y \) as \( T \) \{s\} else \{s\}}. 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 \texttt{x subtypeof \( I \) \{s\} else \{s\}} which is used to \texttt{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:

```
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 \( \text{in-clause} \) in the \( \text{acquire} \)-expression). If \( d \) also supports the SpellChecker interface, \( d \) can join the newly created group \( \text{editor} \) 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 \( \text{acquire} \)-expression may not succeed and execution could be blocked at this point. The kernel language could be extended by a more robust version of \( \text{acquire} \) which uses branching (similar to \( \text{subtypeOf} \)); in fact, inside a group \( g \), robustness may be obtained by first checking for the existence of an interface \( I \) in \( g \) using \( \text{subtypeOf} \) and then binding to the object or group implementing \( I \) in \( g \) using \( \text{acquire} \).

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 \( \text{od} \) in the group by means of an \( \text{acquire} \), where the \( \text{except} \)-clause is used to avoid binding to the new service \( nd \). Finally the old service \( \text{od} \) is removed as Dictionary in the group by a \( \text{leave} \) 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 \( \text{Group}(\mathcal{T}) \) exports the services described by the set \( \mathcal{T} \) of interfaces to clients, so a program variable of type \( I \) may refer to the group if \( I \in \mathcal{T} \). We denote by \( \text{Any} \) 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 \( \text{this} \); 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 Any, so we let $I \prec Any$ 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 of its implemented interfaces. The reflexive closure of $\prec$ is denoted $\preceq$.

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 $T_1 \preceq T$ and $T_2 \preceq T$. 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., [53, 55, 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 17.5. By T-GROUP, a new group has the empty group type (no exported interfaces). 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 $I$ to $y$, the effect is that the type of $y$ is extended with the interfaces $I$. Note the requirement $\text{local}(y)$, 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 $\text{export}$ 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 $\text{intf}(\text{export})$ 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].
(NEW-GROUP)
\[ \text{fresh}(g) \]
\[ o(a, m \{ l \mid x = \text{newgroup}; sr \}; \rho) \]
\[ \implies o(a, m \{ l \mid x = g; sr \}; \rho) \quad g(\emptyset) \]

(JOIN)
\[ (a \circ l)^V(x) = v \quad \ell(y) = (\text{Group}(S), g) \]
\[ T = \text{Group}(S \cup T) \quad \text{exports}' = \bigcup_{l \in T} \{ v : I \} \cup \text{exports} \]
\[ o(a, m \{ l \mid x \text{ joins } y \text{ as } T; sr \}; \rho) \quad g(\text{exports}) \]
\[ \implies o(a, m \{ l \mid y \to (T; g)[sr]; \rho \} \quad g(\text{exports}') \]

(Join)
\[ (a \circ l)^V(y) = g \quad (a \circ l)^V(x) = v \]
\[ \text{exports}' = \text{exports} \setminus \bigcup_{l \in T} \{ v : I \} \quad \text{inf}(\text{exports}) = \text{inf}(\text{exports}') \]
\[ o(a, m \{ l \mid x \text{ leaves } y \text{ as } T \{ s_1 \} \text{ else } \{ s_2 \}; sr \}; \rho) \quad g(\text{exports}) \]
\[ \implies o(a, m \{ l \mid s_1; sr \}; \rho) \quad g(\text{exports}') \]

(AQUIRE)
\[ (a \circ l)^V(g) = g \quad (v : J) \in \text{exports} \quad J \prec I \quad v \notin (a \circ l)^V(x) \]
\[ o(a, m \{ l \mid x = \text{acquire } I \text{ in } y \text{ except } x; sr \}; \rho) \quad g(\text{exports}) \]
\[ \implies o(a, m \{ l \mid x = v; sr \}; \rho) \quad g(\text{exports}) \]

(Leave1)
\[ (a \circ l)^V(y) = g \quad (a \circ l)^V(x) = v \]
\[ \text{exports}' = \text{exports} \setminus \bigcup_{l \in T} \{ v : I \} \quad \text{inf}(\text{exports}) \neq \text{inf}(\text{exports}') \]
\[ o(a, m \{ l \mid x \text{ leaves } y \text{ as } T \{ s_1 \} \text{ else } \{ s_2 \}; sr \}; \rho) \quad g(\text{exports}) \]
\[ \implies o(a, m \{ l \mid s_2; sr \}; \rho) \quad g(\text{exports}) \]

(Leave2)
\[ (a \circ l)^V(x) = g \quad (a \circ l)^V(x) = v \]
\[ \text{exports}' = \text{exports} \setminus \bigcup_{l \in T} \{ v : I \} \quad \text{inf}(\text{exports}) \neq \text{inf}(\text{exports}') \]
\[ o(a, m \{ l \mid x \text{ leaves } y \text{ as } T \{ s_1 \} \text{ else } \{ s_2 \}; sr \}; \rho) \quad g(\text{exports}) \]
\[ \implies o(a, m \{ l \mid s_2; sr \}; \rho) \quad g(\text{exports}) \]

(Query1)
\[ y \notin \text{dom}(a \circ l) \quad a \circ l(x) = (\text{Group}(S), g) \quad o' : J \in \text{exports} \quad J \prec I \]
\[ o(a, m \{ l \mid x \text{ subtypeOf } I \} y \{ s_1 \} \text{ else } \{ s_2 \}; sr \}; \rho) \quad g(\text{exports}) \]
\[ \implies o(a, m \{ l \mid y \to (\text{Group}(S \cup \{ I \}, g))[s_1; sr]; \rho \} \quad g(\text{exports}) \]

(Query2)
\[ (a \circ l)^V(x) = g \quad \text{Group}(\text{inf}(\text{exports})) \neq I \]
\[ o(a, m \{ l \mid x \text{ subtypeOf } I \} y \{ s_1 \} \text{ else } \{ s_2 \}; sr \}; \rho) \quad g(\text{exports}) \]
\[ \implies o(a, m \{ l \mid s_2; sr \}; \rho) \quad g(\text{exports}) \]

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 3).

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) [78, 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() {...}
    }
}
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{lienhardt2008} that addresses the first of these limitations using row polymorphism \cite{rowpolymorphism} 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{thaker2006} 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{lfj}. 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{ffj} presents a similar base language that also formalises Thaker et al.’s \cite{thaker2006} approach to safe composition, although for this system type checking occurs only on the generated product. Coloured Featherweight Java \cite{coloredffj}, 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{coloredffjrefinement} 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{kuhlemann2008} 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{schaefer2008} 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{heidenreich2008} 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{czarnecki2008} 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{apel2008} 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{apel2008}. Clarke et al. \cite{clarke2008} 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{padmanabhan2008} 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{aop} 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 $\text{delta } d \text{ after } DL \{ COL \}$, where $DL$ is used to construct the partial order between deltas and $COL$ is the body of $d$ 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.

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

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 $act$, 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 $s$, 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 $act(PL)$, 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 $\mathcal{A}$ denote the set of all actions $A$:

\[\]
\[
\begin{align*}
act_c(\text{adds class } c \{ML\}) & \triangleq \{c\} \cup \text{members}(c, ML) \rightarrow \text{add} \\
act_c(\text{removes class } c) & \triangleq \{c\} \cup \{c.m \mid m \in M\} \rightarrow \text{rem} \\
act_c(\text{modifies class } c \{MOL\}) & \triangleq \{c\} \rightarrow \text{mod} \triangleright \text{act}_m(c, MOL) \\
act_c(CO; COL) & \triangleq \text{act}_c(CO) \triangleright \text{act}_c(COL)
\end{align*}
\]

\[
\begin{align*}
\text{add} \triangleright \text{add} & \triangleq \text{add} & \text{add} \triangleright \text{rem} & \triangleq \perp & \text{add} \triangleright \text{mod} & \triangleq \text{add} & A \triangleright \perp &= A \\
\text{rem} \triangleright \text{add} & \triangleq \text{mod} & \text{rem} \triangleright \text{rem} & \triangleq \text{rem} & \text{rem} \triangleright \text{mod} & \triangleq \text{rem} & \perp \triangleright A &= A \\
\text{mod} \triangleright \text{add} & \triangleq \text{mod} & \text{mod} \triangleright \text{rem} & \triangleq \text{rem} & \text{mod} \triangleright \text{mod} & \triangleq \text{mod}
\end{align*}
\]

Figure 4.4: Actions of Operations

**Definition 4.4.1** Given a product line \(PL\), an element \(E\), a product \(\rho\) \(\{DL\} \in PL\), and two delta names \(d_1, d_2 \in DL\) that are not ordered.

Product line \(PL\) has a soft conflict iff \(E\) is a member \(c.m\) and \(\text{act}(PL)(d_1)(E) = \text{act}(PL)(d_2)(E) = \text{mod}\).

There is a hard conflict iff both deltas are acting on \(E\) and one of them is not doing a simple modification. That is,

\[
(\text{act}(PL)(d_1)(E), \text{act}(PL)(d_2)(E)) \notin (\{\perp\} \times A) \cup (A \times \{\perp\}) \cup \{(\text{mod}, \text{mod})\}
\]

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.

It is possible to resolve soft conflicts with another delta that acts on the element after the conflict. The details are found in the corresponding paper (Section 1.5).

### 4.5 Type System

The type system extends our previous work \([59]\) to capture conflicts. Its syntax is presented in Figure 4.5. Like in \([59]\), row types and row polymorphism \([59]\) capture the structure of products and classes, as well as the manipulations performed by the deltas. Annotations \(J\) are the mechanism we use to detect conflicts and will be presented later in the text. The type of a core program \(TP\) consists of a mapping between class names \(c\) and their types \(TC\). The type of each class consists of presence information that can either be \(\text{Pre}_J(TML^0)\), meaning that the class is present and has the structure \(TML^0\), or \(\text{Abs}_J(TML^0)\), meaning that the class is absent, i.e., not part of the product \((TML^0\) being the structure of the class before it was removed). The superscripts \(c\) and \(m\) of \(TCL^c\) and \(TML^m\), respectively, are used only to prevent duplicated class and member names, and are dropped in the examples below for readability. Row polymorphism is enabled with variables \(\rho\) that stands for an unknown mapping. The structure of the type \(TC\) for classes is similar to the type \(TM\) for members of classes, with the difference that it only records if a member is present or not, as members do not have an inner structure. Deltas are typed with functional types \(TD\), where \(\alpha\) can either be a row variable \(\rho\) or a conflict variable \(\gamma\), and modifying operators of members of classes are functional types \(TDC\) that are analogous to the types of deltas but over types of members \(TML\).

We illustrate this first part of the type syntax in Figure 4.6, which presents the type of the program and the type of the delta \(\text{Choco}\) from Figure 4.1. For now we ignore the annotations \(J\), which we will use to detect conflicts. As expected, the type of the program contains all the information concerning its structure: the class \(\text{Settings}\) 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 the core program’s type, and perform an addition on elements for the product. We illustrate in Figure 4.7 how this second step detects the soft conflict (resp. hard conflict) in the product.

Figure 4.5: Type Syntax

Figure 4.6: Types of Two elements from Figure 4.1

Here we observe that the delta expects to find the class **Settings** in the input program, which makes sense since this class will be modified. Its inner structure is not constrained (this is modeled by the variable \( \rho_J \)), except for the field **chocolate** which must be absent (as it will be added). Similarly, the class **CMachine** must be present in the input, and in this class the method **makeChoco** must be absent (it will be added) and the method **make** must be present (it will be modified).

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 \( \gamma \) 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 follows: for each product of the product line, we take 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 4.8, 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 \( (\text{choco}, \text{mod}) \) 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 \( p_b \) 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

\[
\text{make} : \text{Pre}_\bot; \quad \text{makeChoco} : \text{Abs}_\bot;
\]

2. Application of Choco

\[
\text{make} : \text{Pre}_\bot(\text{choco.mod}); \quad \text{makeChoco} : \text{Pre}_\bot(\text{choco.add});
\]

3. Application of Sugar

\[
\text{make} : \text{Pre}_\bot(\text{choco.mod}, \text{sugar.mod}); \quad \text{makeChoco} : \text{Pre}_\bot(\text{choco.add}, \text{colourPrint.mod});
\]

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

Conflict Detection in \(p_s\)

\[
\begin{align*}
\Phi = \bot & \Rightarrow (J = J' = \bot) & \Phi = d, \gamma & \Rightarrow (J = \gamma; (d, \text{add}) \land J' = \gamma) \\
\Phi \vdash T_1 \ f_1 \ \text{def}_1; \ldots; T_n \ f_n \ \text{def}_n : (f_1 : \text{Pre}_J; \ldots; f_n : \text{Pre}_J; \text{Abs}_\rho) & \\
\text{T:ML} & \\
\Phi \vdash M_{L_1} : TML_1 & \ i \in 1..n & \Phi = \bot & \Rightarrow J = \bot & \Phi = d, \gamma & \Rightarrow J = \gamma; (d, \text{add})
\end{align*}
\]

\[
\begin{align*}
\Phi \vdash \text{class} \ c_1 (\text{ML}_1) \ldots \text{class} \ c_n (\text{ML}_n) & \\
\vdash \langle c_1 : \text{Pre}_J(\text{TML}_1); \ldots; c_n : \text{Pre}_J(\text{TML}_n); \text{Abs} & \\
\text{T:CL} & \\
\end{align*}
\]

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

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 \(c : \text{Abs}_J(m_i : \text{Abs}_{J_i}(d, \text{rem}))\). The unification of the annotations \(J_i\) is achieved using \textit{local substitutions} [61, 63], 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

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

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 \(\langle c : \text{Pre}_J(m_1 : \text{Pre}_{\gamma'}; \ldots; m_n : \text{Pre}_{\gamma'}; \text{Abs}_{\gamma'}) \rangle\), 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 \(\text{T:ML}\) types the body of a class with a mapping stating that all the members of the class are present. The rule \(\text{T:CL}\) 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 \(\text{T:ADDMEM}\), 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 \( (d, \text{add}) \) 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 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 come 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 devices 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 straight-forward 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’ \((t)\) (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 \(\emptyset\), \(\{t\}\) 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 \(\epsilon\) output.

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

We would like to remove \(l/\epsilon\) transitions first, then \(m/\epsilon\) transitions, then \(t/\epsilon\) 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 \(\emptyset\) 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 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 \(\mathbf{cfc} = \mathbf{tfc} = \emptyset\). Say we reach specified GPS coordinates before 9:00. Transition \(l\) would be fired, if it were available. As it is, \(\mathbf{cfc} = \emptyset\) and \(\mathbf{tfc} = \{l\}\). If we then start a scheduled meeting, \(m\) is fired, and \(\mathbf{cfc} = \{m\}\) and \(\mathbf{tfc} = \{l,m\}\). If the meeting ends (bringing us back to the previous situation), and it becomes 9:00, \(\mathbf{cfc} = \mathbf{tfc} = \{t,l\}\), because \(t\) and \(l\) both fire. In all cases, we have \(\mathbf{cfc} \equiv \mathbf{tfc}\), 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 [50]) 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) 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 \texttt{ABSClosure} overriding \texttt{exec} needs to be provided. Fields are represented as objects of type \texttt{ABSField}. Concrete fields inherit from \texttt{ABSField} and provide a specific initialisation expression by overriding the \texttt{init} method. \texttt{ABS} objects are instances of \texttt{ABSDynamicObject}, which offers an interface through which it is possible to modify their class and \texttt{COG} associations, update their fields and call methods. \texttt{ABSCog} objects are associated to objects and mainly control the scheduling of tasks. By modifying a \texttt{COG}’s \texttt{TaskScheduler} one can configure its scheduling policy, implemented as a \texttt{TaskSchedulingStrategy}.

### 6.3.2 Usage

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

```
java -cp absfrontend.jar abs.backend.java.JavaBackend -d javagen -dynamic PeerToPeer.abs
```

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

```
java -cp javagen:absfrontend.jar PeerToPeer.Main
```

These tasks can be performed equally using the Eclipse IDE with the \texttt{ABS} plugin installed.

### 6.3.3 Code Generation

To illustrate code generation for the dynamic \texttt{ABS} Java back end, we show how a few simple \texttt{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 \texttt{ABS} code: the generated Java class \texttt{C_c} corresponds to \texttt{ABS} class \texttt{C}, and the generated method \texttt{ABSInteger \texttt{getX()} \texttt{corresponds to Int \texttt{getX()}}. In the dynamic setting, \texttt{ABS} classes are represented as singleton instances [38] of class \texttt{ABSDynamicClass}. The static method \texttt{c_c.singleton} (line 3)
Figure 6.3: Code generation example: class declaration

creates an ABSDynamicClass object (line 5) and adds class C’s fields and methods. These are represented as subclasses of ABSField and ABSClosure. For each method, a new class inheriting from ABSClosure is created, which, by overriding the exec method, encodes the method’s specific behaviour. Similarly, ABSField is subclassed for each field, with the overriding init method defining the field’s specific initialisation expression. Instances of these classes are passed as arguments to addField (line 6) and 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 ABSDynamicObject is created with a reference to our ABSDynamicClass object representing class C (line 1). Calling C’s method then amounts to calling the dispatch method on the ABSDynamicObject with the name of the method as an argument (line 2). The dispatch method returns a generic 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 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.


[33] ABS system derivation and code generation, September 2012. Deliverable 1.4 of project FP7-231620 (HATS), available at \url{http://www.hats-project.eu}.

[34] Analysis, September 2012. Deliverable 1.3 of project FP7-231620 (HATS), available at \url{http://www.hats-project.eu}.

[35] Types for variability, March 2012. Deliverable 2.4 of project FP7-231620 (HATS), available at \url{http://www.hats-project.eu}.


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


Glossary

Terms andAbbreviations

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.