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 D3.6**  
**Evolvability Final Report**  

Due date of deliverable: (T0+48)  
Actual submission date: 1 March 2013  

---  

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

**Organisation name of lead contractor for this deliverable:** KTH  

---  

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

**Integrated Project supported by the 7th Framework Programme of the EC**
Executive Summary:
Evolvability Final Report

This document summarises deliverable D3.6 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.

In this final deliverable of HATS WP 3 we present three new work items that in various ways address the problem of dynamically evolvable software, we report on a journal special issue in progress on the topic of evolvable software, and we briefly summarize and discuss the achievements of HATS WP 3 more broadly.

As a main dissemination activity of WP 3 a special issue of the Science of Computer Programming journal is currently in progress, with a planned publication date in the second half of 2014. For the special issue six papers have been submitted, all from the HATS project, on dynamically evolvable software with focus on evolvability aspects of ABS, the Abstract Behavioral Specification language, and on software product lines, both of which are central topics of HATS.

The journal issue is currently under preparation, edited by D. Clarke and M. Dam; the call for papers closed on Jan 10, and the submissions are at the time of writing under review. The intention is to have the table of contents settled by the time of the final review of the HATS project.

Three of the submissions for the SCP special issue, on the semantics of dynamic product lines, and on deployment variability in delta-oriented models, are reported here. The remaining three submissions are reported in other HATS deliverables.

List of Authors

Mads Dam (KTH)
Einar Broch Johnsen (UIO)
Michiel Helvensteijn (CWI)
Radu Muschevici (KUL)
Rudolf Schlatte (UIO)
Lizeth Tapia (UIO)
Contents

1 Introduction 4
  1.1 Deviations from the DoW 6
  1.2 List of Papers Submitted to the SCP Special Issue 6
  1.3 Organization of the Deliverable 7

2 Deployment Variability in Delta-Oriented Models 8
  2.1 Aspects of Variability in an ABS Product Line 9

3 An Abstract Operational Semantics for Dynamic Product Lines 11
  3.1 Abstract Delta Modeling 11
  3.2 Dynamic Product Lines 11
  3.3 The Case Study: Delta Profiles 12
  3.4 The Problem 13
  3.5 The Operational Semantics 13
  3.6 Cost and Optimization 15
  3.7 Discussion 15

4 MetaABS and Dynamic Model Updates 16
  4.1 Introduction 16
  4.2 The MetaABS Interface 16
  4.3 Applications 16
    4.3.1 User-Defined Process Schedulers and Real-time Support 16
    4.3.2 Dynamic software product reconfiguration 17
  4.4 Related Work 18
  4.5 Conclusion 18

5 Conclusion 19

Bibliography 20

Glossary 25

A Paper 2: MetaABS and Dynamic Model Updates 27

B Paper 5: An Abstract Operational Semantics for Dynamic Product Lines 50

C Paper 6: Deployment Variability in Delta-Oriented Models 51
Chapter 1

Introduction

In this deliverable we summarize the work of HATS WP 3 on evolvable systems. The main contribution reported here is a special issue of the Science of Computer Programming (SCP) journal on evolvable and adaptable systems, with a planned publication date in the second half of 2014. For the special issue six papers have been submitted, representing original work within HATS on dynamically evolvable software with focus on evolvability aspects of ABS, the abstract Behavioral Specification language, and of software product lines, both of which are key contributions of the HATS project. The journal issue is currently under preparation, edited by D. Clarke and M. Dam; the call for papers closed on Jan 10, and the submissions are at the time of writing under review. The intention is to have the table of contents settled by the time of the final review of the HATS project.

In this deliverable we briefly summarize the contributions of HATS WP 3 and the contributions submitted to the SCP journal. Three of the submissions to the special issue correspond closely to contributions reported in other deliverables, so we shall only summarize these submissions very briefly here. The remaining three submissions are summarized in more depth in the main body of the present document.

Evolvable Software

The need to evolve and adapt software systems arises for many different reasons:

- Changing requirements. As users need change over time, old functionality may need to be discarded, and new functionality may need to be added.

- Changes in operating conditions. This sort of change can occur, for instance, due to changing performance requirements, changes in application load, or changing operating configurations, for instance due to mobility, changing power requirements, failures, or attacks.

- Changes in execution platforms, for instance because of hardware or API's with better performance or new functionality is becoming available, or because of changes in OS support.

- Changes in the code base, for instance because of patching, code refactoring, or addition or removal of functionality.

To support these different types of changes different approaches are needed, and the type of support that can be provided varies greatly depending of the conditions under which evolvability is to take place.

The Full Information Case

For instance, in a product line setting it may be reasonable to assume that the entire product line is available and in principle known at the time evolution is to take place. This is the static variability case studied in HATS WP 2. In the full information case, runtime evolution is greatly aided, and the problem becomes one of checking that product lines do not break under evolution (i.e., by adding delta’s), and that products can be gracefully upgraded with minimal interruption. Within HATS WP 3 we have for instance developed a dynamic product line formalism and a complementary conflict detection type system, both based on delta-modelling, which expresses and checks possible evolution paths of software...
product lines. We have also developed the MetaABS meta-language to support adapting the runtime of the ABS language from within the language itself, and we have studied semantics and optimization techniques for dynamic deltas. These contributions have been reported in Tasks 3.3 and 3.5, and the paper is reported in this deliverable.

The Partial Information Case However, it is not always the case that full information is available, or indeed desired, on the systems that are subject to evolution. In this case, evolution is a much more difficult problem. The problem has many dimensions:

- If information to perform evolution steps is missing, are there ways to build models for it that does not involve full manual inspection?
- How can evolution be constrained in a way such that overall consistency properties can be derived and maintained during evolution?
- Is it possible to build “pluggable” systems, with clearly defined components and component interfaces, that ensure that evolution steps are possible, and safe?
- Is it possible to guide the evolution towards a specific objective, in terms of functionality, security, or performance?

The HATS project has made important contributions in all these areas.

Model Mining One way of overcoming the problem of partial information for modeling and development in a software product line context is to use some form of model mining. One option is to use automaton learning. In this case a learner performs experiments on some systems component as a black box, in order to iteratively build a model of the component that is adequate for use in a product line context. Theory and tools for black box automaton learning have been developed in Task 3.2 and applied to the Fredhopper case study in Task 5.4. This is the subject of one of the submissions to the SCP special issue.

In another line of work we have used model mining from source code to extract variability models. These can then be used as ABS specifications in a product line context, or they can be used for various analyses, e.g., for resource consumption or timing analysis. This work is reported in Task 3.2 as well.

Components, and Dynamic Software Updates An important problem is to develop techniques that support the safe evolution of source code over time, for instance to avoid breaking type safety. At a fundamental level we may ask under which conditions it is possible to replace, at runtime, one class with another, without breaking well-typedness, and in Task 3.1 we developed conditions based on denotational semantics to identify sufficient and necessary conditions for this type of class compatibility. A central contribution of the HATS project is the notion of “delta”, as a device to add specific new functionality to a class, or a component. Supporting deltas for dynamic evolution has been an important theme in HATS WP 3. In Task 3.1 and 3.3 we examined type-based conditions for dynamic delta updates in an ABS context to be sound, i.e., not lead to runtime errors. We also examined methods to ensure the typing consistency of a running system, under possible update of its classes. This work was extended in Task 3.3 where we study several closely related issues:

- Evolving components: We have developed a concept of reconfigurable components. These are special objects that can be reconfigured at runtime, by special ports. In a series of works we have formalized such types of components and studied conditions for safe component based evolution.
- Runtime evolution of object groups. We have shown how objects can be moved between object groups in a type safe manner, and how service discovery can be used to dynamically bind objects using this mechanism.
• The MetaABS tool \cite{MetaABS} has been produced which allows introspection and manipulation of running code.

**Goal-Directed Evolution** One basic and difficult problem in systems evolution concerns performance adaptation: How to dynamically change the configuration of a running system such that it meets given performance goals, for instance in terms of capacity and latency. In Task 3.5 we have introduced a novel object mobility model ABS-NET capable of transparently and effectively migrating objects between physical processors in order to adapt to given network constraints \cite{ABS-NET}. We have proved that the model is sound and fully abstract with respect to a network unaware reference semantics \cite{ABS-NET_soundness}, and we have shown how adaptation can be performed in such a model with respect to load and latency-related measures \cite{ABS-NET_load_latency}.

In order to support adaptation also with respect to functionality and security we have examined different forms of goal-directed adaptation. For instance, in Task 3.1 we proposed the use of preprogrammed code modification rules \cite{preprogrammed_modification}, and in Task 3.4 \cite{monitor_inlining} we studied monitor inlining as a way of modifying a program given in the form of multithreaded Java bytecode in order to enforce a given security policy.

### 1.1 Deviations from the DoW

Any deviations from the DoW are reported in the final deliverables of each Task 3.1–3.5.

### 1.2 List of Papers Submitted to the SCP Special Issue

As explained, the main outcome reported in this deliverable is a special issue of the Science of Computer Programming journal, as of Feb. 2013 in preparation. The special issue has received the six submissions listed below, all from the HATS project. Three of the submissions do not directly correspond to a paper reported as part of some other HATS deliverable. Those appear as papers 2, 5, and 6 in the listing below. The remaining submissions, 1, 3, and 4, appear as parts of other HATS deliverables, as detailed below.

**Paper 1: Component Model for Concurrent Object Groups: Theory and Practice**

The paper \cite{component_model} presents a new component model for the ABS language, and shows its applicability on an industrial case study (the Fredhopper case study).

The paper is written by Michael Lienhardt, Mario Bravetti, and Peter Wong, and is submitted to the special issue of SCP. The paper is reported as part of D5.4 \cite{D5.4}.

(Download [Paper 1](#))

**Paper 2: MetaABS and Dynamic Model Updates**

The paper \cite{MetaABS} presents dynamic ABS: ABS extended with a new reflective layer that allows introspection and manipulation of running code.

The paper is written by Radu Muschevici, Jose Proenca, and Dave Clarke, and is submitted to the special issue of SCP.

(Download [Paper 2](#))

**Paper 3: Resource-Aware Configuration in Software Product Lines**

The paper \cite{resource_aware_config} investigates resource-aware product configuration for ABS guided by an off-the-shelf resource analysis tool.

The paper is written by Elvira Albert, Taslim Arif, Karina Villela, and Damiano Zanardini, and is submitted to the special issue of SCP. The paper is reported as part of D4.4 \cite{D4.4}.

(Download [Paper 3](#))
Paper 4: Testing Abstract Behavioral Specifications

The paper [56] presents a range of testing techniques for ABS and applies them to an industrial case study. The paper is written by Peter Wong, Richard Bubel, Frank S. de Boer, Miguel Gomez Zamalloa, Stijn de Gouw, Reiner Hähnle, Karl Meinke, and Mudassar A Sindhu, and is submitted to the special issue of SCP. The paper is reported as part of D2.7 [20].

(Download Paper 4.)

Paper 5: An Abstract Operational Semantics for Dynamic Product Lines

The paper [34] presents an operational semantics to support reasoning about the behaviour of product lines based on abstract delta modeling in a dynamic setting.

The paper is written by Michiel Helvensteijn and is submitted to the special issue of SCP.

(Download Paper 5.)

Paper 6: Deployment Variability in Delta-Oriented Models

The paper [39] combines deployment models with the variability of concepts of ABS, in order to model deployment choices as features when designing a family of products.

The paper is written by Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa, and is submitted to the special issue of SCP.

(Download Paper 6.)

1.3 Organization of the Deliverable

In the remainder of the deliverable we summarize each the contributions of papers 2, 5, and 6 above, all related to the topic of dynamic variability, ABS, and delta models. We conclude by briefly summarizing the work of the work package, what has been achieved, and some challenges remaining to be solved in the area of evolvable systems.

As requested by the reviewers, the papers 2, 5, and 6 are not directly attached to deliverable 3.6. A version of this deliverable with the papers attached is available on the HATS web site at the following url:

Chapter 2

Deployment Variability in Delta-Oriented Models

Variability modeling has been extensively investigated within the case studies of the HATS project, but the focus has been on the functionality and logical structure of the model. In the following, we briefly summarize the results of an investigation into modeling multiple deployment scenarios combining various techniques developed within HATS.

This approach to describing deployment architectures is based on a separation of concerns between the application model, which requires resources, and the deployment scenario, which reflects the virtualized computing environment which provides heterogeneous amounts of resources. For example, the functional features which can be selected for the different products in a cell phone SPL, depend on the physical capacity of the different cellphones; e.g., a cell phone with limited processing capacity may require a simpler camera application than a very powerful cell phone. In a virtualized setting, such as cloud deployment, an application model may be analyzed with respect to deployments on virtual machines with varying features: the amount of allocated computing or memory resources, the choice of application-level scheduling policies for client requests, or the distribution over different virtual machines with fixed bandwidth constraints.

Figure 2.1 shows how functional variability modeling in ABS extends Core ABS, and how time and deployment models in Real-Time ABS extend Core ABS. Although these extensions coexist for the same modeling language, these two aspects of ABS had so far never been combined. We combined these two extensions in order to model deployment variability, corresponding to the dotted area in Figure 2.1.

Some main results are:

- We integrated delta models with deployment components in the ABS modeling language;
- Our integration allows orthogonality between functional variability and deployment variability;
- The integration was illustrated by variability patterns for MapReduce [16], a programming model for highly parallelizable programs; and
- The integration allows ABS tools to be used to analyze functional features with respect to a deployment scenario during the early design stage of an SPL.

We modeled a product line consisting of a range of services which inspect a set of documents. The individual products may implement, among others, Wordcount, which counts the number of occurrences of words in the given documents, and Wordsearch, which searches for documents in which a given word occurs. Each product was implemented on (a model of) a cluster of computers, using MapReduce as the common underlying infrastructure. In addition to the different product functionalities, we modeled deployment scenarios differing in the number of servers (“demo” vs. “full” version) and cost models for each MapReduce product (see Figure 2.2).

Note that the approach outlined in this chapter does not utilize the dynamic reconfigurability aspects of model behavior as described in Chapters 3 and 4 yet. A semantics of dynamic model reconfigurability
where deltas modify physical aspects of the system can likely be formalized using previous work on object mobility \cite{10} and dynamic resource reallocation \cite{38}.

2.1 Aspects of Variability in an ABS Product Line

The variation points in the SPL needed to alter functional and deployment aspects turned out to be orthogonal and could be modified independently of each other in the example. We theorize that this is a common occurrence in distributed systems having a well-designed architecture – witness the fact that introducing deployment aspects into the Fredhopper case study developed within HATS entailed very local code changes as well.

Furthermore, the methods to be modified by deltas are not public; i.e., they are not part of the published interface of the classes comprising the base model. This appears to be a recurring pattern: public methods interact with the outside world, gather and decompose data for computation and returning. If the modeler factors out computation into private methods with only one single task to perform, these methods can be cleanly replaced in deltas, without imposing constraints on the implementation. This suggests that clean object-oriented code will in general be likely to be amenable to delta-oriented modification.

Figure 2.3 shows simulation results of a model with two different deployment features applied. Similar qualitative investigations can be performed regarding the influence of varying cost models (e.g., worst-case vs. average cost) and more involved deployment strategies.
Figure 2.3: Simulation results: varying deployment model, cost and functional model constant.
Chapter 3

An Abstract Operational Semantics for Dynamic Product Lines

This section summarizes the theory from the article entitled “An Abstract Operational Semantics for Dynamic Product Lines” [34], submitted for publication to the SCP HATS special issue.

3.1 Abstract Delta Modeling

Delta Modeling [53, 51, 52] is designed as a technique for implementing software product lines [50]: a way to optimally reuse code between software products which differ only by which features they support. Given a desired feature configuration we can automatically derive the corresponding product by the incremental application of a selected set of deltas to a core product, which contains only the bare basics. The set of legal feature-combinations is expressed through a feature model [2].

Clarke et al. [5] described delta modeling in an abstract algebraic setting known as the Abstract Delta Modeling (ADM) approach. In that work, delta modeling is not restricted to software product lines per se, but rather product lines of any domain. It gives a formal description of deltas and how they can be combined and linked to the feature model. This approach has since been extended and refined in several directions [31, 33, 28, 15, 30, 32, 6].

3.2 Dynamic Product Lines

Traditionally, a feature configuration is chosen once at build-time. Its corresponding product is then generated and cannot change at runtime. That is sometimes limiting, as it could be advantageous for products to be able to adapt to dynamic conditions. Dynamic product lines [29] 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, after which the product should adapt accordingly.

Our article, submitted to the SCP special HATS issue [34] explores dynamic product lines in the abstract context of ADM; a context we call Dynamic Delta Modeling (DDM). We introduce an operational semantics [53, 48, 35, 49] in order to reason about the behavior of product lines in a dynamic setting. We develop models to represent dynamic product lines based on their static counterparts as well as their formal specifications. We then explore different strategies for ‘running’ them—with an eye on both flexibility and efficiency.

We define such strategies in terms of a Mealy machine [45]. The input symbols of the machine correspond to features that have been turned on or off by the environment and the output symbols correspond to the deltas that have to be applied to the current product in order to subsequently bring it up to date.

Finally, we introduce a cost model. We assume that monitoring specific features for change has a certain cost and that some features are more costly than others. We can then optimize dynamic product lines by
disregarding costly features until they become relevant. This is modeled by selectively removing transitions from the Mealy machine.

The SCP article is loosely based on previous work [33] (mentioned also in [21]) which introduced the idea of using a Mealy machine to model ADM-based dynamic product lines. Compared to that work we now capture the notion of ‘strategy’ with an operational semantics, we drop several restrictions and make the approach more general. The SCP article means to subsume the earlier publication.

### 3.3 The Case Study: Delta Profiles

We present a novel case-study to serve as an example. It refines the formalism to the concrete domain of automated profile management for smartphones and has lead to the development of an Android application. By monitoring personal data such as time, location and schedule, a smartphone can automatically adjust its internal 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”. We show that delta modeling and, by extension, dynamic delta modeling, are a natural fit for modeling such rules.

We chose not to use a more traditional software-based product line primarily for one reason: This would require us to address several specific issues that would distract from the article’s main contribution. In particular, the article is not about control flow or heap management. The profile management example, however, is based on a relatively simple key-value mapping, making it an ideal case study. In Section 3.7 we briefly discuss the possibilities for extending DDM to a programming language context.

The idea behind the profile manager application is that the user manually inputs a set of rules using the app’s graphical interface (Figures 3.1 and 3.2), which we interpret as a product line. We can then deploy it as a ‘dynamic product line’, regulating the devices profiles.
3.4 The Problem

The problem is as follows: Say we are supposed to conform to feature configuration \( F_e \) and we are currently running product \( p \). Assume also that \( p \) is correct with regard to that environmental feature configuration, so we are in a stable state.

The environment could then request a new feature configuration \( F'_e \). At that time, we need to update our product to \( p' \) so that it is again correct.

Our goal is to find the best possible strategy for doing so while staying in the abstract setting of ADM (Section 3.1). Preferably one that is (potentially) efficient, since we are in a runtime setting, where time and space matter.

For the profile manager example (Section 3.3), the environmental feature configuration would change whenever the truth value of a constraint is ‘flipped’ by an environmental quantity receiving a new value.

3.5 The Operational Semantics

In order to reason about different strategies for updating product \( p \), we introduce an operational semantics, which describes the progress of a dynamic system by defining a transition relation \( \to \) over a set of configurations:

\[
\langle cn_1 \rangle \to \langle cn_2 \rangle \to \langle cn_3 \rangle \to \langle cn_4 \rangle \to \cdots
\]

This allows us to both visualize our system moving from state to state and to reason about whether it could reach certain desirable or undesirable states.

The nature of the configuration space determines what kind of information we can find in these configurations and, consequently, which properties we can express about our dynamic system.

In the SCP article we first explore a type of minimal configuration space

\[
\langle F_e, p \rangle \in \Phi \times P
\]

where \( \Phi \) is the set of valid feature configurations and \( P \) is the set of all possible products.

We see the system as existing in one of two phases: the environmental phase, in which the environment imposes a new feature configuration, or the local phase, in which we update the product to match:

\[
\langle F_e, p \rangle \to \langle F'_e, p \rangle \to^* \langle F''_e, p' \rangle \to^* \langle F'''_e, p'' \rangle \to \cdots
\]

But we soon discover that this is not enough information to describe anything but the most naive strategy:

- **LOC-PRD** Generate the correct product from scratch in the familiar ‘static’ way.

This strategy is quite inefficient. It would be better to transform only the part of the product that corresponds to the change in the environmental feature configuration. To deduce that change, we need to keep track of the ‘local feature configuration’ \( F_l \). The type of configuration we will be working with is

\[
\langle F_e, F_l, p \rangle \in \Phi \times \Phi \times P.
\]

The difference between two feature configurations is expressed through the set operation of symmetric difference. For feature configurations \( F, G \in \Phi \):

\[
F \oplus G \overset{\text{def}}{=} (F \cup G)/(F \cap G)
\]

A product transformation is encoded in a delta. We use a Mealy machine to decide which delta to apply for a given feature configuration difference from a given state. A Mealy machine is a finite state machine with an input symbol and an output symbol on each transition. See Figure 3.3 for a simple diagrammatic example of one.

In our case, the input symbols are feature-sets and the output symbols are deltas, which are used to transform the product. The first strategy we attempt is the following:
When our product is out of date, apply a delta for the entire feature difference \( F_e \ominus F_l \).

This strategy is somewhat more efficient than \( \text{loc-prd} \), but requires storing an exponential number of deltas. So we need something a bit more clever:

\[
\text{loc-diff-nd}(MM(T_s)) \quad \text{When our product is out of date, nondeterministically choose a delta for some feature } \{ f \} \subseteq F_e \ominus F_l.
\]

See Figure 3.4 for the visual representation of an example Mealy machine corresponding to this strategy. Each node is annotated with its feature configuration and each arrow with a \( f/d \) pair where \( f \) is the feature that changed and \( d \) is the delta that is used to update the product. The marked states represent an example situation wherein we are currently occupying state \( F_l = \{ t \} \) (shaded) and we need to generate a product consistent with state \( F_l = \{ t, l, m \} \) (circled). We can either apply delta \( \epsilon \cdot y_x \) or delta \( y_x \cdot x_x \) to achieve that.

But this strategy suffers from the limitation that the feature model cannot be ‘restricted’: all feature combinations have to be valid. Otherwise we might be missing necessary ‘intermediate states’ in our Mealy machine. For example, what could we do if \( \{ t, m \} \) and \( \{ t, l \} \) were both invalid feature configurations? We adapt the strategy:

\[
\text{loc-diff-nd}(MM(T_m)) \quad \text{When our product is out of date, nondeterministically choose a delta for a minimal feature-set } \{ f \} \subseteq F_e \ominus F_l \text{ that allows us to reach an intermediate state.}
\]

For specific details and proofs, we refer you to the journal article [34].
3.6 Cost and Optimization

Finally, we assume that occupying a state in a Mealy machine has a cost: the cost of monitoring the features from the accepted input-symbols for change. We posit that monitoring some features can be more expensive than monitoring others.

For example, it will be more draining to the battery of a smartphone to constantly monitor GPS location than it will to intermittently check the calendar for meetings, since the GPS receiver needs to receive signals and the calendar is internal. But checking the calendar is still more costly than keeping track of the time. The operating system does that anyway, and can notify our app through an alarm-subscription service.

The trick to optimizing this cost is that we don’t really need to reach a configuration where $F_i = F_c$. It is sufficient to have a correct product. This allows us to remove certain transitions from our Mealy machine (in particular, transitions that would not have modified the product) and thus reduce the cost of running it. Which transitions to remove can be seen as the solution to a constrained optimization problem—one which cannot be solved in an abstract setting.

3.7 Discussion

Hallsteinsen et al. [29] introduce several properties they believe constitute a dynamic software product line. Our approach 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 environmental feature configuration can change during runtime, the set of available feature configurations is still fixed at ‘build time’.

The case study from Section 3.3 cannot really be called a ‘software product line’, since the generated products are not software. The profile manager bears greater resemblance to a Self Adaptive System or a Context-aware Program. It is true that, even though ADM was designed from a software product line engineering perspective, its abstract nature makes its dynamic counterpart quite suitable for modeling systems of either of those descriptions.

Damiani et al. [14,13] explore dynamic delta modeling in the concrete context of object oriented programs and their focus is on the specific issues faced in that context. For example: How do we manage memory when a dynamic reconfiguration extends or reduces data-types? When do we allow such a reconfiguration in the first place so that it does not break normal flow of control? As mentioned before, we could not discuss such issues in detail, since they would have distracted from our intended contribution.

But the work of Damiani et al. [14, 13] complements our work in this regard. They introduce the concept of a reconfiguration automaton which can reconfigure existing objects in the heap to be consistent with the change in code. Further, they introduce a reconfigure statement that, when reached, allows a dynamic software product line to reconfigure without breaking control flow.

It would be quite possible to unify their approach with a concrete object oriented version of the abstract framework presented in our SCP article.

The Mealy Machine generated by our techniques may be enriched by reconfiguration translations, essentially embedding a reconfiguration automaton into it in order to take care of heap consistency. Further, our operational semantics could readily be joined with the semantics of a proper programming language and coordinate with it by the addition of a number of simple inference rules around the reconfigure statement.
Chapter 4

MetaABS and Dynamic Model Updates

4.1 Introduction

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. A new, dynamic compiler backend and the MetaABS language provide a framework for tightly integrating activities that rely on dynamic aspects of ABS.

4.2 The MetaABS Interface

MetaABS is a largely object-oriented reflective interface to the ABS language that provides an abstraction of the underlying ABS runtime. MetaABS comprises a set of operations 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. Figure 4.1 shows the operations provided by the MetaABS interface.

The purpose of MetaABS is to provide a unified interface for various runtime model analysis tasks. By adding meta-programming capabilities to ABS, such tasks can be encoded directly in ABS and performed at runtime. Analysis tasks of particular interest within HATS include the scheduling of tasks inside concurrent object groups and the dynamic reconfiguration of software products.

4.3 Applications

Meta-programming opens the possibility of modifying program aspects that were initially intended to be static, such as the execution semantics of the language or the code itself. We explore two such applications, which are presented respectively in Sections 4.3.1 and 4.3.2. The first application investigates how to dynamically change the order of execution of processes in a single COG, while the second application concerns the reconfiguration of a model’s structure and behaviour by applying delta modules at runtime. Both these problems arise naturally in ABS due to (1) the existence of non-determinism for selecting tasks within a COG and (2) the presence of a mechanism that allows the production of a family of software products based on transformations to a core code base.

4.3.1 User-Defined Process Schedulers and Real-time Support

While task scheduling in ABS is non-deterministic by default, the ABS Java compiler backend provides a flexible configuration mechanism to statically define the scheduling strategies that are used during the execution of an ABS system [19]. We make this configuration mechanism accessible at runtime for ABS models that are compiled to Java.
User-defined scheduling of concurrent objects has already been studied in the context of ABS for real-time systems [4]. In that context, ABS code annotations are used to define functions that implement specific scheduling algorithms, and manage real-time aspects such as duration and deadlines of tasks.

MetaABS provides operations to access and modify the COG of a particular object. The COG 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. Following Bjørk et al.’s work [4] we augment processes with a set of standard real-time attributes that can play a role in the scheduling of tasks.

Two main advantages emerge from this approach. First, configurable task scheduling can be embedded in a more general framework for meta-programming problems. Second, MetaABS is connected to a Java backend that compiles ABS code into executable Java programs, contrary to the previous approach which was based on a simulator running on top of the Maude rewriting engine. Bringing real-time issues to the Java platform allows us to use JVM’s time measuring capabilities instead of the more artificial notion of time used in the Maude interpreter.

4.3.2 Dynamic software product reconfiguration

Dynamic software product reconfiguration is understood as the ability to reconfigure products at runtime, that is, the transformation of a product into another valid product defined by the SPL, all without the need to re-compile and deploy the system. To support this kind of dynamic adaptation, ABS models need to accommodate dynamic changes in their structure and behaviour. Adding this facility to ABS complements the static SPL modelling capability of ABS. 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 product specifications and delta modules and by 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.

While static and dynamic product configuration are related concepts, they differ in one key aspect. Static product configuration always starts with the base product (represented by a core ABS model) and applies a sequence of modifications until obtaining any of the products specified by the product line. Dynamic product reconfiguration starts with any product already configured using the above process, and applies a set of modifications to obtain a new product (out of the set of specified products). The set of products that are configurable from a given product at runtime is constrained in the sense that they have to be explicitly listed in the ABS product selection. Figure 4.2 illustrates this aspect.

Figure 4.2: Static product configuration transforms a core into any product of the SPL (left). With dynamic product reconfiguration (right), only certain transitions between products are allowed at runtime.

4.4 Related Work

The work presented here focuses on the design of a reflective API suitable for analysing and changing various aspects of ABS models at runtime. This includes but is not limited to dynamically exploiting the variability of SPL systems modeled in ABS. For more details about this line of work we refer to Chapter 4 of Deliverable 3.5 [23]. Chapter 3 of this deliverable explores dynamic SPL in an abstract context and provides an operational semantics for reasoning about the behavior of product lines in a dynamic setting. Chapter 2 tackles ABS model variability with a focus on variable deployment scenarios.

4.5 Conclusion

This paper presents MetaABS, a reflective interface for the ABS language, and a dynamic backend, both designed together to facilitate tasks concerned with runtime model analysis and adaptation. Examples of such tasks are user-controlled scheduling policies and dynamic software product lines that enable dynamic product reconfiguration.
Chapter 5

Conclusion

The objective of HATS WP3 was to develop the theory, algorithms, and core mechanisms needed to build software systems that can dynamically reconfigure themselves—without service interruptions—to adapt to changes that were not anticipated at the time the components which make up the running system were initially constructed. Changes to be considered in the project included:

- The availability of new components implementing new features
- Changes in the code base itself
- Changes in security policy or performance requirements
- Changes in execution platform properties

Also, the goal was to consider both safety and consistency of evolution, as well as goal-directed evolution: Evolution which is guided by the need to meet some objective in terms of functionality, security, or performance.

These workpackage objectives have all been met.

Among the centerpiece achievements we highlight the following:

- A rich collection of theories, languages, and tools to support runtime adaptation within the ABS framework, including dynamic delta-oriented programming for functional variability, and deployment components to support variability in the underlying computational resources.
- Techniques based on black box learning and bytecode rewriting to manage software system evolution when little is known a priori about the systems internal structure.
- A new object mobility model ABS-NET with promising properties for automating the evolution process, particularly regarding performance properties.

We finally highlight some of the main lessons learned, and challenges identified, during the work on HATS WP3:

- Building on theories and approaches existing today, and compatible with mainstream trends in software engineering, it is possible to build rich and semantically well-founded tools for a very wide range of dynamic evolvability concerns related to safety and consistency.
- Scalability of these tools is a research challenge. The dominant approach based on delta flattening is fundamentally unscalable, and new techniques are needed to overcome this problem.
- Reflection can be fruitfully used to account for many performance aspects, allowing the object language itself to be used to guide and program various types of adaptation strategies.
• In some restricted cases such as monitor inlining it is possible to evolve a system even if very little is known about it. Generalizing this to richer systems objectives with reliable results is a challenge.

• Semantics driven component discovery remains a significant challenge. Program analysis techniques, e.g., as developed in other HATS workpackages may help, but it is not clear how to scale this to large and complex components. Delta-aware specification and verification techniques as described in Deliverable D4.3 [24], Section 2.3 could help to solve the scaling problems.

• Taking in emerging results on e.g., compact routing it appears possible to build large scale message passing systems that exploit object mobility for load balancing and performance adaptation, at least in some cases. Many challenges remain, however, to handle faults, secure routing, and multi-objective resource allocation.
Bibliography


[47] Radu Muschevici, José Proença, and Dave Clarke. MetaABS and dynamic model updates. Submitted for publication.


Glossary

Terms and Abbreviations

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

**ABS-NET** A model for transparently executing ABS programs on a static network of processors.

**Adaptation** The process modifying a system or a piece of software to make it function better, faster, more safely, more securely, or with less resource consumption.

**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.

**Network semantics** A semantics of a programming language given as an execution model on a message passing network of processing nodes.

**Routing** The process of determining a path for a message to take in order to find its way from one node in a network to another.

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

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

**Software product** A software systems with a well-defined set of features.
Software product reconfiguration  The process of adding and removing features from a software system at runtime.

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

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

Paper 2: MetaABS and Dynamic Model Updates
MetaABS and Dynamic Model Updates

Radu Muschevici, José Proença and Dave Clarke
DistriNet & IBBT, Dept. Computer Science
KU Leuven, Belgium

Abstract

Long-lived, dependable systems evolve during their life time: new functionality is added, errors are discovered, new requirements appear, etc. It is often infeasible to stop and redeploy a system in order to upgrade it, due to the associated downtime. A system that can adapt to changes dynamically is more dependable. In this paper we present 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 implementing and analysing evolving software in ABS.

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.

Several approaches for meta-programming exist, explained better in the related work section below. This paper describes a meta-programming mechanism for the ABS language [15], introducing MetaABS. The nature of ABS brought several challenges in the development of MetaABS.

- The ABS language was designed as a balanced compromise between a simplistic model that can be easily verified and formally analysed, and a language rich enough to model real world and complex scenarios. Hence MetaABS should also strive for both objectives: have a small core that can be easily analysed and still be practical enough to be used in larger examples, and to have an executable engine.

- There is an explicit concurrency model in ABS, where remote objects communicate only asynchronously, and concurrent local objects have a cooperative scheduling in a single thread. Meta-objects in MetaABS also need to fit this concurrency model.

- Some static meta-programming capabilities already exist in ABS: it has a core language extended with other small languages that describe how to generate a family of software products by applying program transformations, wrapped in delta modules. MetaABS needs to make sure the same static functionality is made available at runtime.

The main contributions of this paper are (1) MetaABS, a meta-programming facility for the ABS language, and (2) a dynamic Java back end that supports it. Further, we illustrate their application in adapting ABS models at runtime. The first application shows how user-defined process scheduling algorithms are defined and dynamically attached to groups of concurrent objects. Secondly, we show how a model is transformed based on the software product line at its core. Executable ABS models can include a collection of related software products. While only one product can run at any time, MetaABS allows us to adapt the current product to a different one at runtime.

The purpose of MetaABS is to provide a unified interface for various runtime models, either developed and under development, for the ABS language. By adding meta-programming capabilities to ABS we allow some model analysis tasks to be encoded directly in ABS and performed at runtime.
1.1 Abstract Behavioural Specification (ABS) language

ABS is a concurrent, multi-paradigm modelling language [15]. Syntax-wise, ABS resembles standard programming languages like Java. Nevertheless ABS is more a modelling than a programming language, because the design of ABS is strongly focused on providing a language that is easy to analyse. High execution performance, for example, is not a design goal of ABS.

ABS supports first-order functional programming with algebraic data types. Functional code is guaranteed to be free of side effects. One consequence of this is that functional code may not use object-oriented features [1, 15]. ABS also supports class-based, object-oriented programming with standard imperative constructs. ABS is especially designed for modelling concurrent and distributed systems. The concurrency model of ABS is based on the concept of concurrent object groups (cogs). A typical ABS system consists of multiple, concurrently running cogs at runtime. Cogs can be regarded as autonomous runtime components that are executed concurrently, share no state and communicate via method calls. Cogs can reference objects of other cogs, however, these far references can only be used as targets for asynchronous method calls.

ABS has a nominal type system with interface-based subtyping. ABS does not support class inheritance and overloading, and instead code reuse can be achieved in ABS by using deltas. Delta-oriented programming [23] is an approach that aims at developing a set of programs simultaneously from a single code base, following the software product line (SPL) engineering approach [21]. In delta-oriented programming, features defined by a feature model are associated with code modules that describe modifications to a core model. These modules are called delta modules (or deltas for short).

1.2 MetaABS

MetaABS comprises a set of operations (a meta-object protocol [19]) 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. Some research tracks being currently pursued with respect to the ABS language include the scheduling of tasks inside concurrent object groups; the dynamic reconfiguration of software products; deployment component configuration; and runtime method dispatch.

We designed MetaABS based on requirements provided by several runtime analysis use cases within the HATS European project,\(^1\) such as the two

\(^1\)http://www.hats-project.eu
applications mentioned in the next subsection. For example, the scheduling of tasks within a cog for real-time systems has been investigated by Bjork et al. [3], but the authors could execute only within the abstract ABS interpreter implemented in the Maude [7] 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.

Other uses for MetaABS include the implementation of the ABS component model developed by Lienhardt et al. [20] and resource analysis using deployment components by Johnsen et al. [17]. 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.

1.3 Applications

Meta-programming opens the possibility of modifying program aspects that were initially intended to be static, such as the execution semantics of the language or the code itself. This paper explores two example of such applications, which are presented respectively in Sections 4 and 5. They investigate how to dynamically change the order of evaluation of parallel tasks in a single cog, and how to reconfigure a model’s structure and behaviour by applying delta modules at runtime. Both these problems arise naturally in ABS due to (1) the existence of non-determinism for selecting tasks within a cog and to (2) the presence of a core mechanism that allows the production of a family of software products based on code transformations.

We now focus on the former application to motivate MetaABS. Using user-defined schedulers of concurrent objects has already been study in the context of ABS for real-time systems [3]. The authors include special annotations to ABS code defining functions that select the task that should be executed next, and manage aspects such as duration and deadlines of tasks.

```plaintext
// A scheduler which switches strategy based on the length of the queue
def Process lengthSensitive(Int limit, List<Process> l) =
    if (length(l) < limit) then shortestProc(l) else earliestProc(l);

[Scheduler: lengthSensitive(limit,queue)]
class ServerImp (Int limit) implements Server { ... }
```

The example above defines a function lengthSensitive that acts as a scheduler for the cogs of instances of ServerImp. Annotations are written within square brackets, and the keyword queue acts as an argument that will bind to the
queue of tasks during execution. Our meta-programming approach avoids
the usage of special annotations, using reflection instead.

```java
class LengthSensitive(Int limit) implements Scheduler {
    Process schedule(List<Process> l) { ... }
}
Cog g = ...
g.setScheduler(new LengthSensitive(limit));
```

Two main advantages emerge from this approach. On one hand the scheduling mechanism can be included in a more general framework for meta-
programming problems. On the other hand MetaABS provides a Java back end that compiles into executable Java programs, contrary to the previous approach that has only a hand-crafted simulator implemented in the Maude rewriting system [7]. Bringing real-time issues the Java platform allows us to use JVM’s time measuring capabilities instead of the more artificial notion of time in Maude, which is implemented by assigning a cost to every expression.

### 1.4 Organisation of the paper

We provide an overview of related work in Section 2. Section 3 details the MetaABS API. Sections 4 and 5 present two applications of MetaABS for run-
time model analysis: user-configurable process schedulers and the dynamic reconfiguration of software products. 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. Section 7 concludes this paper.

### 2 Related Work

Meta-programming consists of writing programs that can write or transform other programs (or themselves), and has appeared in a multitude of flavours. Different classifications for such programs exist, including: generative vs. intensional (creating or analysing programs), compile-time vs. run-time (when the programs are written or transformed), heterogeneous vs. homogeneous (what programs are transformed: others or themselves), and lexical vs. syntactical (over strings or over abstract syntax trees). MetaABS can be categorised as an intensional, run-time, homogeneous, and syntactical approach, and is based on reflection – possibly the most common mechanism for this
category, supported by mainstream languages such as Java, C#, and Python. In this section we present a brief overview over some meta-programming approaches and describe existing research tracks in the context of ABS language that deal with some forms of meta-programming.

Observe that the delta mechanism of ABS provides per se meta-level functionality. Similarly to aspect oriented programming [18], a collection of delta modules transforms a core program by applying a sequence of transformations. A higher-level heterogeneous meta-programming approach has been explored for ABS (cf. Chapter 4 of Deliverable 1.3 [11]), using the meta-programming language is Rascal\(^2\) to generate and transform full ABS code, including delta modules and descriptions of the product lines. The homogeneous approach by MetaABS allows manipulating ABS programs at run-time, opening new possibilities for evolvability.

Lisp was one of the first languages to support homogeneous meta-programming, using expressions based on macros. Macros provide a generative approach for meta-programming, allowing macros to be expanded at compile-time. Macros in Scheme benefit also from being fully hygienic, that is, they avoid name clashes during macro expansion. Later other generative homogeneous approaches improved the macro expansion mechanism of Lisp using multi-stage programming, starting with the introduction of MetaML [27], reaching more modern and rich languages. Multi-stage programming provides a finer control of the temporal aspects, allowing multiple levels of programs generated inside programs to co-exist, wrapped in so-called quasi-quotations. Other powerful and (type) safe languages borrowed these ideas, such as MetaKlaim [12], MetaOCaml [26], Template Haskell [25] (which fully supports code compilation at runtime), and also in Scala macros are being experimented.\(^3\)

In the context of object-oriented programming, a common way to write meta-programs is based on \textit{reflection}. For example, the Java reflection API\(^4\) allows the reification of program elements such as classes into special objects that can support reflective operations. Bracha and Ungar [5] expose some of the limitations of Java’s core reflection mechanism and propose the use of a new level of indirection which they call \textit{mirrors}. The authors claim that reflective APIs built around mirrors adhere to the design principles of encapsulation, stratification, and ontological correspondence. This means the behaviour of mirrors is encapsulated, the meta-level facilities are separated

\(^2\)http://www.rascal-mpl.org
\(^3\)http://docs.scala-lang.org/overviews/macros/overview.html
\(^4\)http://docs.oracle.com/javase/tutorial/reflect

6
from the base-level functionality, and the ontologies of the meta-level and the language that is manipulated have a correspondence. The MetaABS follows this approach and uses mirrors to encapsulate the meta-level transformations. Furthermore, we developed a new Java back end that supports these mirrors, that is, we compile ABS programs with the MetaABS layer into executable Java code with meta-level functionality that goes beyond Java-reflection. Currently we do not know of any realisation of mirrors in Java. Delta modules in MetaABS are seen as a sequence of meta-level operations, and thus can be applied dynamically.

A challenge addressed by MetaABS that is not present in most object-oriented languages is concurrency. Objects in ABS live in cogs, and communication between objects from different cogs must be asynchronous. This principal is extended to the mirror objects that must also belong to a given cog, and modification of remote objects must be done asynchronously. Meta-programming of concurrent languages has been less study than for sequential languages. For example, Schneider and Lumpe propose a meta-level approach for concurrent object-oriented programs based on the Form-calculus, a variant of the $\pi$-calculus introduced by the authors [24], and Neendorfer explored in his PhD dissertation meta-programming in actor-oriented languages [2], focusing on the reconfiguration of actors at run-time.

With respect to the application of MetaABS, this paper uses two examples to illustrate different aspects of this meta-programming language. The first application was initially presented Bjork et al., where they introduce real-time with user defined schedulers as a means to control the scheduling policy at the level of active objects; 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. Their work relies on annotations. We use reflective operations provided by MetaABS to configure real-time parameters and schedulers.

MetaABS is also designed to enable models of dynamic software product lines in ABS. Based on the delta mechanism to build a family of software products, Damiani and Schaefer [8] described a model to dynamically modify the existing software product to another with different features. Later Helvensteijn also explored the modelling of delta models that evolve dynamically [14]. In our second example of an application of MetaABS we further explore dynamic evolution of software product lines, where the feature model changes and code is removed or added to a model at runtime. The usage of MetaABS allows the treatment of such problems under a single meta-level framework, without requiring the development of specialised tools to man-
age the redeployment of software products whenever it needs to be modified.

Finally, safe dynamic software updates remain a very interesting application of MetaABS that still needs to be investigated carefully for the context of ABS. Several studies have been made in this direction, such as the work on incremental dynamic updates by Wernli et al. [28], where concurrent threads can share memory, and the work on dynamic classes by Johnsen et al. [16], where communication between objects is exclusively asynchronously.

3 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 1.

3.1 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 1). We opted for a mirror based design [5] in order to achieve a separation between an object’s regular interface, determined by its type, and its reflective interface.

Future Mirrors  Asynchronous messages to objects in remote cogs return a future to the caller and create a new process in the remote cog. The future is used to obtain the message’s result after the message has been processed. A FutureMirror reflects on the future. It provides a set of operations that configure real-time attributes of the created process, such as setting the deadline and cost of the process that will supply the result to the future.

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().

**Product Line** A separate interface is dedicated to configuring the SPL attached to an ABS model. ABS supports dynamic SPLs, meaning that a running product can be reconfigured into another product dynamically. All a user needs to do is to obtain a handle to a Productline object by calling the getPL() function, and call configureProduct() with the name of the target as an argument. The user also needs to ensure that the system is in a “safe” state when reconfiguration occurs.

Figure 1: MetaABS interface. Labels on the dotted arrows denote the MetaABS functions that return an object of the type they point to.
3.2 Usage Example

The following example illustrates how reflective operations are accessed from an object mirror.

```java
import * from ABS.Meta;
class C implements I { Unit foo() {...} }
{
  I obj = new C();
  ObjectMirror mir = reflect(obj);
  Class cls = mir.getClass();
  cls.removeMethod("foo");
}
```

4 User-Defined Process Schedulers and Real-time support

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 [9]. We make this configuration mechanism accessible at runtime for ABS models that are compiled to Java.

MetaABS provides operations to access and modify the cog of a particular object. The cog 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.

Real-time ABS provided support for analysing real-time properties of computations in ABS on the Maude platform. Following (author?) [4]’s work we augment processes with a set of standard real-time attributes.

- **Arrival time** is the time when the process is created in response to an asynchronous method call.
- **Cost** is the processor time needed to complete the task without interruption.
- **Deadline** is the time by which the process should complete, relative to the arrival time.
- **Criticality** is a boolean parameter related to the consequence of missing the deadline.
- **Start time** is the time when the process starts running.
class MyScheduler(Int x) implements Scheduler {
    Process schedule(List<Process> l) {
        return head(l);
    }
    Unit setX(Int x) {
        this.x = x;
    }
}

interface C {
    Unit m();
}

class CImpl implements C {
    Unit m() {
        ...
    }
}

class CImpl implements C {
    { 
        ObjectMirror om = reflect(this);
        Cog g = om.getCog();
        Scheduler s = new MyScheduler();
        ObjectMirror sm = reflect(s); sm.setCog(g);
        g.setScheduler(s);
    }
    Unit m() {
        ...
    }
}

Fut<Unit> f = o!m();
FutureMirror fm = reflect(f);
fm.setDeadline(1/40);
fm.setCost(1/100);
fm.setCritical(false);
}

Figure 2: Implementation of a user-defined scheduler in MetaABS.

**Finishing time** is the time when the process is completed.

Arrival time, as well as start and finishing time are set automatically to the system’s time at process creation, start and completion. *Cost, Deadline* and *Criticality* are process parameters settable by the user. To set these attributes, the MetaABS future mirror is used.

Implementing support for real-time processes in the ABS Java back end means we can measure time using Java’s standard clock. Previous implementations of Real-time ABS execute on the Maude rewrite engine and uses a clock that runs from 0 to 100 by default and only advances (from the point of view of a process) when the process suspends, awaits, executes a get expression or a statement that has a cost.

The ABS code example in Figure 2 shows two ways how the scheduling strategy of a cog can be customised. The *MyScheduler* class implements a *schedule* method, which returns the *Process* to be scheduled for execution. The state of the program at runtime can also play a role in scheduling through class parameters, represented here by the parameter *Int x*.

To assign a *Scheduler* object to a cog, one has to obtain a handle on the
Cog object from an object that is defined inside that cog. Then one uses to the setScheduler operation on that cog object to change the scheduler.

In the variant on the left side, the MyScheduler object is created outside the cog containing object o. This would cause a runtime error when synchronously calling schedule(). Therefore it is necessary to move the scheduler object s to the cog whose processes it should schedule. MetaABS allows to change the cog of an object, effectively moving the object from one processor to another. Alternatively, a scheduler can be defined directly inside class C’s init block, effectively placing it inside the same cog as any instance of class C. This implies that the scheduler is tied to the class inside which it is defined. In effect, it sets the scheduler of any cog inside which an instance of the class is created. In the first variant, the scheduler can be assigned to any object (of any class). This variant is shown in the right column.

Since the object o is in a remote cog, its method is invoked asynchronously. This creates a process that will be scheduled using the user-defined scheduling algorithm. To set the real-time parameters of the process, we reflect on the future and use its mirror to set deadline, cost and criticality.

5 Dynamic Software Product Re-configuration

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 SPL modelling capability of ABS. 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 product specifications and delta modules and by 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.

A runtime product specification includes a set of adaptations that describe how the product can evolve at runtime. An adaptation contains the names of the products that the current product can be transformed into, and a state update specification, which describes the transformation of the model’s state when the product is adapted.
The conceptual difference to static product configuration is that, while with static product configuration we always start with the base product (represented by a core ABS model) and apply a sequence of deltas until we obtain any of the products specified by the product line, dynamic product reconfiguration starts with any product and applies a set of deltas to obtain a different product from a subset of the set of specified products. The premise is that any particular product can be only reconfigured at runtime into a different product from a well-defined set of products. Figure 3 illustrates this idea.

Figure 3: Static product configuration transforms a core into any product of the SPL (left). With dynamic product reconfiguration (right), only certain transitions between products are allowed at runtime.

MetaABS provide a Productline interface that gives access to operations that adapt the running software product by applying a sequence of delta modules. A Productline represents the set of products $P$ that are available at runtime. Each of these products can be reconfigured into another product from a subset of $P' \subseteq P$. $P'$ is defined statically in the product configuration.

Figure 4: Product specification example for dynamically re-configurable product.

Figure 4 shows an example of a product specification. A product $P_1$ contains features $F_1$ and $F_2$. This product can be reconfigured at runtime.
into products $P_2$ or $P_3$ by applying a set of delta modules as specified by delta clauses in the product line configuration (line 8), and the updates $\text{Upd1}_2$ or $\text{Upd1}_3$ (lines 2–3) respectively.

With compile-time product configuration we always start with a program core and apply a sequence of delta modules. At run-time, products can be re-configured, meaning that we start with a product and obtain another product. The syntax of delta clauses has been extended to allow two application conditions: one that is evaluated against the current feature selection (the product before reconfiguration), introduced by the from keyword and one which specifies the feature selection after reconfiguration, introduced by the to or when keywords (lines 10–11).

The by keyword introduces a state update, which specifies how to transfer the objects’ state when transitioning from a product to another product, and also specifies synchronisation conditions that define when objects can be updated safely.

6 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 [10]) 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.

6.1 Design

Figure 5 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 1) are mapped to this interface.

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

### 6.2 Usage

To generate Java bytecode using the dynamic ABS Java back end, the ABS compiler is invoked using the `-dynamic` switch. For example, the following command generates Java code for the ABS program `PeerToPeer.abs` (provided as an example in Deliverable 1.2 [9]) into the `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 ABS plugin installed.

### 6.3 Code Generation

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

```java
class C(Int x) implements I {
    Int getX() { return x; }
}
```

```java
public final class C_c extends ABSObject implements ABSClass, I_i {
    private ABSInteger x;
    public C_c(ABSInteger) {...}
    public final ABSInteger getX() {...}
}
```

```java
public final class C_c {
    private static ABSDynamicClass c;
    public static ABSDynamicClass singleton() {
        if (c == null) {
            c = new ABSDynamicClass();
            c.addField("x", new ABSField() { /* Override init */ });
            c.addMethod("getX", new ABClosure() { /* Override exec */ });
        }
        return c;
    }
}
```

**Figure 6:** Code generation example: class declaration.

The example in Figure 6 shows the code generation process for class declarations. The generated static Java code is very similar to the original ABS code: the generated Java class `C_c` corresponds to ABS class `C`, and the generated method `ABSInteger getX()` corresponds to `Int getX()`. In the dynamic setting, ABS classes are represented as singleton instances [13] of class `ABSDynamicClass`. The static method `C_c.singleton` (line 3) 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).

```plaintext
C object = new C(42);
Int x = object.getX();

C_c object = new C_c(42);
ABSInteger x = object.getX();
```

**Figure 7:** Code generation example: class instantiation and method call.

The example in Figure 7 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 Code generation for dynamic software product lines

The **ABS** compiler, which translates **ABS** source code to either Java code or Maude rewriting rules, supports static SPL deployment. At compile time, the user chooses a specific product, which is defined in the model as a set of features together with specific values assigned to feature attributes. Upon code generation, the set of delta modules that are applicable for this product are applied to the core **ABS** model in a process known as **flattening** of the model. Flattening removes all variability from a model, producing a runnable software product out of the set of products represented by the product line.

The goal of the dynamic **ABS** back end is to allow the modification of a large number of model elements and aspects at runtime. **Dynamic SPLs** (DSPLs) are executable models of SPLs that can transform their structure.
and behaviour from one product to another at runtime. DSPLs are a popular application of dynamically adaptable software and since ABS already has extensive support for static SPLs, extending that support to the dynamic version was a logical development. To that end, a runtime representation of product specifications and delta modules is needed.

The code generated process for ABS product specifications is illustrated in Figure 8 using the example from Figure 4.

```java
package products;
public class P1_prod {
    private static ABS DynamicProduct prod;
    public static ABS DynamicProduct singleton() {
        if (prod == null) {
            ABS DynamicProduct prod = new ABS DynamicProduct();
            prod.setName("P1");
            prod.setFeatures(Arrays.asList("F1", "F2", "F3"));
            prod.setUpdate("P2", "Upd1_2");
            prod.setUpdate("P3", "Upd1_3");
            prod.setDeltas("P2", Arrays.asList("AddF3"));
            prod.setDeltas("P3", Arrays.asList("AddF4RemoveF2"));
        }
        return prod;
    }
}
```

Figure 8: Product specification example for dynamically re-configurable product.

The list of deltas that is passed to `setDeltas` when creating the runtime instance of product `P1` is determined at compile time from the product configuration in the same manner as it is done when generating a static product. When generating a static model, these delta modules are applied already at compile time in a flattening process, whereas for the dynamic model, a valid sequence of deltas is simply recorded so that it can be applied at runtime.

The configuration of an SPL product – whether it occurs at compile-time or at run-time – follows the delta-oriented programming approach [22]. This means that in order to obtain the code for a specific product, a list of delta modules is applied in sequence to a program. Delta modules prescribe code modifications such as adding or removing classes, methods, fields, functions and data types. In ABS, delta modules are associated with features in delta clauses using application conditions, as part of the product line configuration [6]). Figure 9 illustrates the representation of delta modules in generated dynamic Java. Note that there is no equivalent representation in generated
delta D;
modifies class C {
    adds Int y;
    adds method Int getY() {...}
}

package delta.D;
public class C_mod {
    public static void apply() {
        ABSDynamicClass cls = C.c.singleton();
        clsaddField("y", new ABSField() { /* Override init */ });
        cls.addMethod("getY", new ABSClosure() { /* Override exec */ });
    }
}

Figure 9: Code generation example: delta modules.

static Java code, as the static Java back end applies and then discards deltas prior to code generation. Each class modifier defined by a delta is represented as a class with a static apply method that uses the API provided by the dynamic Java back end to apply the modifications to the current representation of the respective class.

7 Conclusion

This paper presents MetaABS, a reflective interface for the ABS language, and a dynamic back end, both designed together to facilitate tasks concerned with runtime model analysis and adaptation. Examples of such tasks, which are detailed in the paper, are user-controlled scheduling policies and dynamic software product lines that enable dynamic product reconfiguration. Other tasks, including the development of safe concurrent and dynamic software updates, are currently being investigated and left as future work.

References


Appendix B

Paper 5: An Abstract Operational Semantics for Dynamic Product Lines
Appendix C

Paper 6: Deployment Variability in Delta-Oriented Models
Deployment Variability in Delta-Oriented Models *

Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa

Department of Informatics, University of Oslo, Norway
{einarj,rudi,sltarifa}@ifi.uio.no

Abstract

Software engineering today increasingly emphasizes variability by developing families of software products together to satisfy a range of application contexts or user requirements, where the different products vary in the features they support. In feature modeling, much focus has been on selecting functional features for different products and on the software quality attributes for features and products. However, the quality of the selected functional features of a product also depend on how the product is deployed.

ABS is a modeling language which supports variability in the formal modeling of software by using feature selection to transform a delta-oriented base model into a concrete product model. Recently, ABS has been extended for the timed modeling of deployment architectures, based on a separation of concerns between execution cost and the server capacity in the model. This allows the effect of deployment choices for a product to be observed on its quality of service. This paper combines deployment models with the variability concepts of ABS, in order to model deployment choices as features when designing a family of products.

1 Introduction

Variability is prevalent in modern software systems in order to satisfy a range of application contexts or user requirements [34]. A software product line (SPL) realizes this variability by providing a family of product variants, see, e.g., [29]. A specific product is obtained by selecting features from a feature model, which typically focuses on the functional features of different products and on the software quality attributes of different features and products. Many variants of feature-based variability modeling exist; for a survey on different feature model languages [36].

To express variability in the design of systems, features typically take the form of architectural models, behavioral models, and test suites [35]. Architectural variability focuses on the presence of component variants, and can be described using, e.g., the Variability Modeling Language [27], UML stereotypes [13], or (hierarchical) component models such as Koala [37]. Delta modeling [8,30,31] is a variability modeling concept in which a set of system deltas specify modifications to a core product in order to obtain other products. Δ-MontiArch applies delta modeling to architectural description languages [14]; a delta can add or remove components, ports, and connections between components.

Complementing architectural models, which are concerned with describing the logical organization of a system in terms of components and their connections, we are interested in the physical organization of software units on physical (or virtual) machines; we call this physical organization the deployment architecture. Our approach to describing deployment architectures is based on a separation of concerns between the application model, which requires resources, and the deployment scenario, which reflects the virtualized computing environment which provides heterogeneous amounts of resources. For example, the functional features which can be

---

*Partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (http://www.hats-project.eu).
selected for the different products in a cellphone SPL, depend on the physical capacity of the
different cellphones; e.g., a cell phone with limited processing capacity may require a simpler
camera application than a very powerful cell phone. In the virtualized setting, an application
model may be analyzed with respect to deployments on virtual machines with varying features:
the amount of allocated computing or memory resources, the choice of application-level scheduling
policies for client requests, or the distribution over different virtual machines with fixed
bandwidth constraints.

In this paper, we are interested in how deployment variability can be integrated in SPL
models. By introducing deployment variability at the design stage of SPL engineering, the
different targeted deployment architectures of the SPL may be taken into account for different
feature configurations early in the design of the SPL. Our starting point for this work is the
recently developed abstract behavioral specification language ABS, which adds support for
variability to models in the kernel modeling language Core ABS [18]. ABS is object-oriented
to stay close to high-level programming languages and to be easily usable as well as accessible
to software developers, it is executable to support full code generation and (timed) validation
of models, and it has a formal semantics which enables the static analysis of models; e.g., the
worst-case resource consumption can be derived for a model. ABS is particularly suitable for
our objective because (1) ABS supports SPL modeling based on deltas [7, 9], and (2) ABS
supports the modeling of deployment decisions based on the modeling concept of deployment
and their dynamic management to be leveraged to the abstraction level of software models.
Figure 1 shows how functional variability modeling in ABS extends Core ABS, and how time
and deployment models in Real-Time ABS extend Core ABS. Although these extensions coexist
for the same modeling language, these two aspects of ABS have so far never been combined.
The purpose of this paper is to combine these two extensions in order to model deployment
variability, corresponding to the dotted area in Figure 1.

This paper proposes a way to introduce deployment variability into models of SPLs. The
main contributions of the paper are:

- we integrate delta models with deployment components in the ABS modeling language;
- our integration allows orthogonality between functional variability and deployment vari-
  ability;
- the integration is illustrated by variability patterns for MapReduce [11], a programming
  model for highly parallelizable programs; and
- the integration allows ABS tools to be used to analyze functional features with respect to
  a deployment scenario during the early design stage of an SPLs.
Figure 2: A family of products sharing an underlying MapReduce structure.

Paper overview The paper is organized as follows. Section 2 motivates our work by introducing a running example of deployment variability. Section 3 introduces the basic modeling level of the abstract behavioral specification languages ABS. Section 4 introduces delta-oriented variability and its realization in the ABS modeling language. Section 5 explains the approach to deployment modeling taken in ABS. Section 6 combines delta-oriented variability with deployment modeling, and discusses how to extend a feature model with deployment variability. Section 7 presents the model and simulations of our case study. Section 8 discusses related work and Section 9 concludes the paper.

2 Motivating Example

MapReduce [11] is a programming pattern for processing large data sets in two stages; first the Map stage separates highly parallelizable jobs on distinct parts of the data to produce intermediate results, then the Reduce stage merges the intermediate data into a final result. The initial and intermediate data are on the form of key/value pairs, and the final result is a list of values per key. MapReduce as such does not specify the computations done by the two stages or the distribution of workloads across machines, making it a good abstract base model for software product lines.

We use MapReduce to model a product line consisting of a range of services which inspect a set of documents. The individual products may implement, among others, Wordcount, which counts the number of occurrences of words in the given documents, and Wordsearch, which searches for documents in which a given word occurs. For simplicity, we will assume that a service only provides one of the Wordcount or the Wordsearch feature. The services are implemented on a cluster of computers, using MapReduce.

For attracting clients to the word count and word search services, there are freely available demo versions of the services, which offer the same functionality as the full versions, albeit with a lower quality of service. In the implementation of the services, the demo versions will run on a few machines, whereas the full versions have access to the full power of the cluster. In our model, we therefore have three versions of each service: the purely functional model, the model with full access to the cluster, and a model with restricted access to the cluster. We will use this product line, depicted in Figure 2, as a running example in the paper.
3 Behavioral Modeling in ABS

The kernel of the abstract behavioral specification language ABS is a formally defined object-oriented language called Core ABS [18], targeting the executable design of distributed systems. ABS is based on concurrent object groups (COGs), akin to concurrent objects [6,19], Actors [1], and Erlang processes [4]. COGs in Core ABS support interleaved concurrency based on guarded commands. This allows active and reactive behavior to be easily combined by means of a cooperative scheduling of processes, which stem from method calls. Core ABS models have a functional and an imperative layer, combined with a Java-like syntax. Real-Time ABS extends Core ABS models with implicit time; the execution time is not specified directly in terms of durations (as in, e.g., UPPAAL [26]), but rather observed by measurements of the executing model. For the formal definition of the syntax, type system, and semantics of Core ABS, we refer the reader to [18]. For the formal definition of Real-Time ABS, we similarly refer the reader to [5].

3.1 The Functional Layer

The purpose of the functional layer of Core ABS is to describe computation succinctly in a representation independent way. The modeler may abstract from the details of low-level imperative implementations of data structures while maintaining an overall object-oriented design close to the target system. The functional layer of Core ABS consists of algebraic data types such as the empty type Unit, booleans Bool, integers Int; parametric data types such as sets Set<A> and maps Map<A, B> (for type parameters A and B); and functions over values of these data types, with support for pattern matching.

Example 1. To illustrate the definition of data types and functions, polymorphic sets can be defined using a type variable A and two constructors EmptySet and Insert. To illustrate function definitions in Core ABS, we show some standard functions on sets: emptySet(xs) checks whether xs is an empty set; insertElement only inserts an element e if the set xs does not already contain e (in fact, our definition of the set does allow multiple occurrences); remove removes an element e from a set xs (assuming single occurrences in the set); and take returns some element of the set xs.

```java
data Set<A> = EmptySet | Insert(A, Set<A>);
def Bool emptySet<A>(Set<A> xs) = (xs == EmptySet);
def Set<A> insertElement<A>(Set<A> xs, A e) = case contains(xs, e) {True => xs; False => Insert(e, xs); };
def Set<A> remove<A>(Set<A> xs, A e) = case xs { EmptySet => EmptySet;
               Insert(e, ss) => ss;
               Insert(s, ss) => Insert(s, remove(ss,e)); };
def A take<A>(Set<A> ss) = case ss { Insert(e, _) => e; };
```

3.2 The Imperative Layer

The purpose of the imperative layer of Core ABS is to describe concurrency, communication, and synchronization. This is done at the level of objects, and defines interfaces, classes, and methods. Core ABS objects are active in the sense that their run method, if defined, gets called upon creation. Communication and synchronization are decoupled in Core ABS. Communication is based on asynchronous method calls, denoted by assignments f=ocl(e) where f is a future variable, o an object expression, and e are (data value or object) expressions. After calling
f = o!m(e), the caller may proceed with its execution without blocking while m(e) executes. Two operations on future variables control synchronization in ABS. First, the statement `await f? suspends the active process` unless a return value from the call associated with f has arrived, allowing other processes in the same COG to execute. Second, the return value is retrieved by the expression `f.get`, which blocks all execution in the COG until the return value is available.

Inside a COG, Core ABS also supports standard synchronous method calls o.m(e).

A COG can have at most one active process, executing in one of the objects of the COG. This active process can be unconditionally suspended by the statement `suspend`, adding this process to the queue of the COG, from which an enabled process is then selected for execution. The guards g in `await g` control suspension of the active process and consist of Boolean conditions conjoined with return tests f? on future variables f. Just like functional expressions, guards g are side-effect free. The remaining statements of Real-Time ABS are standard; e.g., sequential composition s1; s2, assignment x=rhs, and `skip`, `if`, `while`, and `return` constructs. Right hand side expressions rhs include the creation of an object group `new cog C(e)`, object creation in the group of the creator `new C(e)`, method calls, and future dereferencing f.get, in addition to the functional expressions e.

**Example 2.** To illustrate how the functional and imperative layers of Core ABS are typically combined, let Worker be an interface and `workers` a set of objects typed by `Worker`. Sets are defined in Example 1. The method `getWorker` will only create a new Worker (by instantiating the class `Worker`) if no Worker is available in the set `workers`, otherwise it will take one worker which it removes from the set. The method `finished` inserts w into the set if it is not already there.

```java
Worker getWorker() {
    Worker w = null;
    if (emptySet(workers)) {
        w = new cog Worker(this); nWorkers = nWorkers + 1;
    } else {
        w = take(workers); workers = remove(workers, w);
    }
    return w;
}

Unit finished(Worker w) {
    workers = insertElement(workers, w);
}
```

4 Delta-Oriented Variability in ABS

This section describes how software product lines (SPLs) are modeled in ABS. Variability is used to specify that multiple similar models can be created by selecting different instances of features and applied them to a variable source code. ABS includes a delta-oriented framework for variability [7, 9]. Figure 3 depicts a delta-oriented variability model where a feature model F with orthogonal variability [16] is represented as two trees that hierarchically structure the set of features of this model; it is also possible to observe variability at the level of code where a common base model P can be modified by applying delta modifications from the delta model ∆. Sets of features from the feature model F are linked to sets of delta modifications from the delta model ∆, which apply to the common base model P to produce different product line configurations C, C' and C'', and finally a specific product ρ is extracted from the product line configuration C.

**Feature model** A feature model in ABS is represented textually as a forest of nested features where each tree structures the hierarchical dependencies between related features, and each feature in a tree may have a collection of Boolean or integer attributes. The ABS feature model
can also express other cross-tree dependencies, such as mandatory and optional sub-features, and mutually exclusive features. The \textbf{group} keyword is used to specify the sub-features of a feature; the \textbf{oneof} keyword means that exactly one of the sub-features must be selected in the created product line, the range of values associated to an attribute specify the values in which an attribute can be instantiated when an specify product is generated. For the full details, we refer the reader to Clarke et al. [7, 9].

\textbf{Example 3.} In the functional feature model of the MapReduce example from Section 2, a tree with a root \textbf{Calculations} offers two alternative and mutually exclusive features that can be selected to express that a specific product supports counting words or searching for words.

```
root Calculations {
  group oneof {
    Wordcount,
    Wordsearch
  }
}
```

In addition ABS allows a feature model with multiple roots (hence, multiple trees) to describe orthogonal variability [16], which is useful for expressing unrelated functional and other features (e.g., features related to quality of service).

\textbf{Delta model} The concept of delta modeling was introduced by Schaefer et al. [31–33] as a modeling and programming language approach for software-based product lines. This approach aims at automatically generating software products for a given valid collection of features, providing flexible and modular techniques to build different products that share functionality or code. In delta-oriented programming, application conditions over the set of features and their attributes, are associated with units of program modifications called delta modules. These delta modules may add, remove, or otherwise modify code. The implementation of an SPL in delta-oriented programming is divided into a common core module and a set of delta modules. The core module consists of classes that implement a complete product of the SPL. Delta modules
describe how to change the core module to obtain new products. The choice of which delta modules to apply is based on the selection of desired features for the final product.

Technically, delta modules have a unique identifier, a list of parameters, and a body containing a sequence of class and interface modifiers. Such a modification can add a class or interface declaration, modify an existing class or interface, or remove a class or interface. The modifications can occur within a class or interface body, and modifier code can refer to the original method by using the `original()` keyword. Delta modules in ABS can be parametrized by attribute values to enable the application of a single delta in more than one context.

**Product line configuration** The product line configuration links feature models with delta modules to provide a complete specification of the variability in an ABS product line. A product line configuration consists of the set of features of the product line and a set of delta clauses. Each delta clause names a delta module and specifies the conditions required for its application, called application conditions. A partial ordering on delta modules constrains the order in which delta modules can be applied to the core module.

**Specific product** A product selection clause generates a specific product from an ABS product line. It states which features are to be included in the product and specifies concrete values for their attributes. A product selection is checked against the feature model for validity. The product selection clause is used by the product line configuration to guide the application of the delta modules during the generation of the final product.

**Generated final product** Given a Core ABS program \( P \), a set of delta modules \( \Delta \), a product line configuration \( C \), a feature model \( F \), and a product selection \( \rho \) (as depicted in Figure 3), the final product, which will be a Core ABS program, is derived as follows: First check that the product selection \( \rho \) satisfies the constraints imposed by the feature model \( F \); then select the delta modules from \( \Delta \) with a valid application condition with respect to \( \rho \); and finally apply the delta modules to the core program \( P \) in some order respecting the partial order described in \( C \), replacing delta parameters in the code with the literal values supplied by the feature.

5 Deployment Modeling in ABS

Real-Time ABS extends Core ABS with primitives to describe deployment architectures. The purpose of describing deployment in a modeling language is to differentiate execution time based on where the execution takes place. With implicit time, no assumptions about execution times are hard-coded into the models. Instead, we want to model how the execution time of a statement varies with the available capacity of the chosen deployment architecture and on synchronization with (slower) objects. For example, the response time to a request in a distributed system depends not only on the size of the job requested, but also on the amount of available resources and on the usage policy for these resources, which are scattered around the deployment architecture of the distributed system. The execution time of a method call depends on how quickly the call is effectuated by the server object; in fact, *similar calls to the same method do not always take the same amount of time.*

Deployment architectures describe how distributed systems are mapped on physical and/or virtual media with many locations. The planning and validation of a deployment architecture to optimize performance implies determining the amount of necessary resources at the different locations as well as an optimal usage of these resources, such that the system fulfills its performance requirements. Real-Time ABS lifts deployment architectures to the abstraction level of
the modeling language, where the physical or virtual media are represented as deployment components [20]. In a Real-Time ABS model, different deployment components may have different bounds on the locally available resources.

Real-Time ABS introduces a separation of concerns between the resource cost of performing a computation and the resource capacity of a given deployment component. This separation of concerns between resource cost and resource capacity aids to model and validate different deployment scenarios at an early stage during the software development process. The focus in this article is on CPU resources in virtualized media; we use resource cost annotations to express resource consumption during computation.

5.1 Deployment Components

A deployment component in Real-Time ABS is part of the deployment architecture of the model, on which a number of COGs are deployed. Each deployment component has an execution capacity, which is specified as the amount of resources which is available per accounting period; for simplicity, this accounting period is fixed in the semantics of Real-Time ABS and corresponds to the interval between integer values in the dense time domain of the language. The main block of a model executes in a root COG located on a default deployment component environment, with unrestricted processing capacity. To capture different deployment architectures, a model may create other deployment components with different resource capacities. When COGs are created, they are by default allocated to the same deployment component as their creator, but they may also be allocated to a different deployment component. Thus, in a model without explicit deployment components all objects execute in the default environment, which places no restrictions on the processing capacity of the model.

Deployment components are first-class citizens of Real-Time ABS. Syntactically, deployment components in Real-Time ABS are manipulated in a way similar to objects. Variables which refer to deployment components are typed by an interface DC. They may be passed around as arguments to method calls and they support a number of methods for load monitoring and load balancing purposes (further details may be found in [20]). Deployment components are dynamically created as instances of class DeploymentComponent, which implements DC. Deployment components may be created dynamically, depending on control flow, or statically in the main block of the model. Deployment components are created by the expression new cog DeploymentComponent(d,c). Here, the parameter c of type Resource specifies the initial CPU capacity of the deployment component. The parameter d of type String is a descriptor mainly used for monitoring purposes; i.e., it provides a user-defined name for the deployment component which facilitates querying the run-time state but that has no semantic effect. COGs are deployed on deployment components upon creation. By default a COG is deployed on the same deployment component as its creator. However, a different deployment component may be selected by means of an optional deployment annotation [DC: e] to the object creation statement, where e is an expression of type DC. Note that deployment annotations can only occur associated with the creation of COGs, not with objects.

The available resource capacity of a deployment component determines the amount of computation which may occur in the objects deployed on that deployment component. Objects allocated to the deployment component compete for the shared resources in order to execute, and they may execute until the deployment component runs out of resources or they are otherwise blocked. For the case of CPU resources, the resources of the deployment component define its capacity inside an accounting period, after which the resources are renewed.
5.2 Resource Consumption

The resource consumption of executing statements in the Real-Time ABS model is determined by a default cost value which can be set as a compiler option (e.g., `−defaultcost=10`); the default is for statements to have no execution cost. However, this default cost does not discriminate between statements, so a more refined cost model will often be desirable. For example, in a realistic model the assignment `x=e` should have a significantly higher cost for a complex expression `e` than for a constant. For this reason, more fine-grained costs can be inserted into Real-Time ABS models by means of adding a *cost annotation* \[\text{Cost: } e\] to any statement.

It is the responsibility of the modeler to specify appropriate resource costs. A behavioral model with default costs may be gradually transformed to provide more realistic resource-sensitive behavior by inserting such cost annotations. The manual estimation of resource cost is time consuming and error-prone. Therefore, it is desirable to have tool support for this activity. COSTABS [2] is an automated static analysis tool that is able to compute a worst-case approximation of the resource consumption of the non-virtualized programs, based on static analysis techniques.

However, the modeler may also want to capture *normative* constraints on resource consumption, such as resource limitations, at an abstract level; these can be made explicit in the model during the very early stages of the system design. To this end, cost annotations may be used by the modeler to abstractly represent the cost of some computation which is not fully specified.

6 Deployment Variability in ABS

This section lifts deployment to the level of feature models for SPLs. As deployment decisions may be introduced in the SPL in a number of ways, the main guideline for our approach is to keep a reasonable orthogonality between the functional variability and the deployment variability in the SPL model. Moreover, the separation of concerns between cost and capacity in the deployment models of ABS will be reflected in the feature models. Thus, deployment variability introduces two new variation points in the feature models of ABS:

- **Resource cost variability**: These features determine the choice of cost model for different parts of the model’s logical artifacts, which determines how the resource cost is estimated during execution in the model; and

- **Deployment architecture variability**: These features determine how the logical artifacts of the model are mapped to a specific deployment architecture, which determines the execution capacity of the different locations on which the logical artefacts execute.

The resulting variability space is depicted in Figure 4. To keep the separation of concerns between the cost and capacity of resources, the features expressing resource cost and the features expressing deployment architectures are kept in different trees in the feature model expressed in ABS.

**Resource cost model variability** The basic feature in this tree is the *no cost* feature, typically selected for the functional analysis of the SPL model. In many cases, it is relevant to introduce fixed costs for selected jobs, similar to costs in a basic queuing network or simulation model (see, e.g., [17]). ABS allows data-sensitive costs to be expressed for selected jobs. The modeler may be interested in either measured, real cost for selected jobs, or in worst-case approximations (which may depend on data flow as well as control flow), both of which can be expressed via cost annotations.
Deployment architecture variability  The basic feature in this tree is the *undeployed* feature which does not impose any capacity restrictions on the execution. This feature is typically selected for the functional analysis of the SPL model. In more refined variants, selected parts of the logical architecture can be deployed on deployment components with restricted capacity. For physical deployment, the deployment architecture is typically configured before the product execution starts. For virtualized deployment, the deployment architecture may evolve over time, modeling the startup and shutdown of virtual machines in a cloud computing scenario.

For both of these variation points, a refined feature model will typically allow to select or deselect resource-sensitive features for different parts of the model, expressed by the feature hierarchy and by cross-tree dependencies.

**Example 4.** We extend the feature model of the running example with a tree for resource costs, with root `Resources`, and another tree for deployment architecture, with root `Deployments`. Observe that this will increase the number of products in the SPL. The `Resources` root has the basic cost model `NoCost`, the model `FixedCost` to express that the CPU cost of executing a piece of code from the specific product (i.e., the cost of searching a word will depend on the specific searching algorithm) is data-independent and can be specified in the attribute `cost`. Furthermore, the feature `WorstcaseCost` selects the worst-case cost model in terms of the size of the input files, and `MeasuredCost` monitors the execution and reports the actual incurred cost during execution of the model. The `Deployments` root has three alternative features related to the number of available machines in the physical deployment architecture and the capacity of each of them specified by the attribute `capacity`.

```
root Resources {
  group oneof {
    NoCost,
    FixedCost { Int cost in [ 0 .. 10000 ] ; },
    WorstcaseCost,
    MeasuredCost
  }
}

root Deployments {
  group oneof {
    NoDeploymentScenario,
    UnlimitedMachines { Int capacity in [ 0 .. 10000 ] ; },
    LimitedMachines { Int capacity in [ 0 .. 10000 ] ;
    Int machinelimit in [ 0 .. 100 ] ; }
  }
}
```
7 Example: Variability in an SPL based on MapReduce

This section describes the implementation of a generic MapReduce framework in ABS and its adaptation to different products in the SPL described in Section 2. It will become apparent that a product that is implemented according to best practices for object-oriented software (i.e., decomposing functionality, methods implementing one task only, and the careful definition of datatypes) also makes the product well-suited as a base product for a software product line.

7.1 Commonalities in the ABS Base Product

Figure 5 shows the interfaces for the main MapReduce object and for the Worker objects which will carry out the computations in parallel. The computation is started by calling the mapReduce method with a list of (key, value) pairs. The main object will then create a number of worker objects, call invokeMap on these objects, gather and collate the results of the mapping phase, and then call invokeReduce on the workers and return the final results.

The base product in our example implements a word count function; it takes a list of files and their contents and returns a list of words and their total number of occurrences across all files. It does not implement resource or cost management. The MapReduce object reuses Worker objects from a pool and creates new workers when the pool is empty, as shown in Figure 6. Workers add themselves back to the pool by calling finished.

Figure 7 shows part of the worker implementation of the base product (i.e., a Wordcount product without any cost model). The invokeReduce method sets up the result, calls a private method reduce which emits intermediate results using the method emitReduceResult. The reduce method in Figure 7 is equivalent to the one shown in the original MapReduce paper [11]. The mapping functions of the worker objects are implemented in the same way.

7.2 Variability in the ABS Product Line

To change the functional feature of the model from computing word counts to computing word search, some parts of the model need to be altered via delta application. The same applies when
null.

```
... Worker w = this.getWorker();
...
```

```java
class MapReduce implements MapReduce {
    int nWorkers = 0;
    List<Pair<OutKeyType, List<OutValueType>>>
        mapReduce(List<Pair<InKeyType, InValueType>> items) {
        while (!isEmpty(items)) {
            Worker w = this.getWorker();
            ...
        }
        Worker getWorker() {
            Worker w = null;
            if (emptySet(workers)) {
                w = new cog Worker(this);
                nWorkers = nWorkers + 1;
            } else {
                w = take(workers);
                workers = remove(workers, w);
            }
            return w;
        }
    }
}
```

Figure 6: Fragments of code in the `MapReduce` class from the base model of the MapReduce example in ABS.

varying the deployment and cost model, as explained in Section 6. These variation points in the SPL turn out to be orthogonal and can be modified independently of each other.

In the example, the methods to be modified by deltas are not public; i.e., they are not part of the published interface of the classes comprising the base model. This appears to be a recurring pattern: public methods like `invokeReduce` of Figure 7 interact with the outside world, gather and decompose data for computation and returning. If the modeler factors out computation into private methods with only one single task to perform (like `reduce` in Figure 7), these methods can be cleanly replaced in deltas, without imposing constraints on the implementation. This suggests that clean object-oriented code will in general be likely to be amenable to delta-oriented modification.

**Functional Variability** Figure 8 shows a delta fragment that modifies the functionality of the base model. Since the types of the input data and result can change, the base model uses the type synonyms `InKeyType`, `InValueType`, `OutKeyType`, `OutValueType`, which can be altered in a delta. This means that the signatures of the `MapReduce` class do not need to be adapted. To change the computation itself, one modifies the methods `map` and `reduce` of the `Worker` class. The implementer of the new `map` and `reduce` methods can use `emitMapResult` and `emitReduceResult` as in the base model, and does not need to care about invocation or return value handling protocols.

**Resource Cost Variability** Costs are incurred during (and because of) computational activity, hence the cost model seems to be linked to the computation. However, in MapReduce one can associate cost with two events: invoking a mapping or reduction step, and producing an intermediate result. Both of these events occur outside the `map` and `reduce` methods that implement the computation, therefore the cost model can be modified independently.

Figure 9 shows a delta implementing the `FixedCost` feature, which assigns a cost given as a feature attribute to each computation of an intermediate result; the feature attribute is passed in as a delta parameter. In general, costs are introduced by modifying the methods `onMapStart` and
class Worker(MapReduce master) implements Worker {
    List<OutValueType> reduceResults = Nil;
    ...
    List<OutValueType> invokeReduce(OutKeyType key, List<OutValueType> value) {
        reduceResults = Nil;
        this.onReduceStart(key, value);
        this.reduce(key, value);
        master!finished(this);
        List<OutValueType> result = reduceResults;
        reduceResults = Nil;
        return result;
    }
    Unit reduce(OutKeyType key, List<OutValueType> value) {
        List<Int> numlist = value;
        Int result = 0;
        while (!~(numlist == Nil)) {
            result = result + head(numlist);
            numlist = tail(numlist);
        }
        this.emitReduceResult(result);
    }
    Unit emitReduceResult(OutValueType value) {
        this.onReduceEmit(value);
        reduceResults = Cons(value, reduceResults);
    }
    Unit onReduceEmit(OutValueType value) { skip; }
}

Figure 7: Fragments of code in the Worker class from the base model of the MapReduce example in ABS: implementing the reduce part of the Wordcount example.

onReduceStart for assigning costs to starting a computation step, and by modifying onMapEmit and onReduceEmit for assigning costs to the production of a result.

Deployment Architecture Variability Deployment architecture, i.e., decisions on how many workers to create and how many resources to supply them with, is implemented in the method getWorker of the MapReduce class. Figure 6 in Section 7.1 shows an implementation that simply creates any worker objects needed, in an environment disregarding any resource constraints. To change this behavior, the modeler implements a delta that overrides getWorker (and also the method finished should the new getWorker method not use the resource pool of the base model); an example can be seen in Figure 10.

This delta changes the deployment model as follows:

- Workers execute in deployment components with a specified processing capacity capacity.
- The system will use a maximum number of workers maxWorkers.

The Product Line Configuration The feature model presented in Section 6 extends the SPL of Section 2 with resource cost variability, resulting in an SPL with 14 different products. Figure 11 shows part of the product line configuration for the SPL and Figure 12 shows the specification of some of the derivable products.

7.3 Results

In the deployment components of the deployment architecture features, capacity is defined by the amount of resource costs that can be processed per accounting period (in terms of the dense time semantics of execution in Real-Time ABS). When the base model is extended with features for
deployment architecture and resource cost, the load on the individual deployment components, defined as the actual incurred cost per accounting period, can be recorded and visualized.

We illustrate how deployment variability for products can be validated using the simulation tool of ABS, by comparing the performance of two different deployments of the Wordcount product, varying the number of available machines between 5 (the “Demo” version) and 20 (the “Full” version), but keeping the cost model, input data and computation model constant. The graphs in Figure 13 shows the total load of all machines over simulated time for the two products. The figure shows two typical instances of a typical MapReduce workload; first, the map processes execute until they are finished, then the reduce processes execute. The start of the reduce phase can be observed in the graph of Figure 13 as the second spike in processing activity. It can be seen that the demo version takes over twice as much simulated time to complete its execution, while the full version completes its execution earlier by incurring a load that is higher than for the demo version (while still decreasing as the map processes terminate).

Similar qualitative investigations can be performed regarding the influence of varying cost...
productline MapReduceSPL;

features
  Wordcount, Wordsearch; // Functional features
  NoCost, FixedCost, WorstCaseCost, MeasuredCost,
  NoDeploymentScenario, UnlimitedMachines, LimitedMachines, // Resource cost features
  DeltaDOccurrences when Wordsearch;
  DeltaDFixedCost(Cost.cost) when Cost;
  DeltaDUnboundedDeployment(UnlimitedMachines.capacity) when UnlimitedMachines;
  DeltaDBoundedDeployment(LimitedMachines.capacity, LimitedMachines.machinelimit) when LimitedMachines;

...
Architecture approach. We follow a similar approach based on the extending a purely functional model with deployment features, but our framework is based on simpler concepts which does not introduce the overhead of description logic. In the context of QoS variability, Kattepur et al. [24] study a modeling and analysis framework for testing the QoS of an orchestration before deployment to determine realistic Service Level Agreement contracts; their analysis uses probabilistic model of QoS. Our work similarly allows the model-based comparison of QoS variability, but focuses on deployment architecture and processing capacity rather than orchestration.

The MapReduce programming pattern which is the basis for the example of this paper, has been formalized and studied from different perspectives. Yang et al. [40] develop a CSP model of MapReduce, with a focus on the correctness of the communication between the processes. Lämmerl [25] develops a rigorous description of MapReduce using Haskell, resulting in an executable specification of MapReduce. Ono et al. [28] formalize an abstract model of MapReduce using the proof assistant Coq, and use this formalization to verify JML annotations of MapReduce applications. However, none of these works focus on deployment strategies or relate MapReduce to deployment variability in SPLs.

9 Conclusion

Software today is increasingly often developed as a range of products for embedded devices with restricted resource capacity or for virtualized utility computing. For a software product line (SPL) targeting such platforms, the deployment of different products in the range should also be considered as a variation point in the SPL.

In this paper, we have integrated an approach to describing explicit deployment scenarios by resource restricted deployment components into a formal modeling language for SPL engineering. This integration is based on delta models to systematize the derivation of product variants, and demonstrated in the ABS modeling language. The integration proposed in this paper emphasizes orthogonality between functional features, resource cost features, and deployment architecture features, to facilitate finding the best match between functional features and a target deployment architecture for a specific product. The analysis supported by ABS allows the validation of deployment decisions for specific products in the SPL, which may entail a refinement of the feature model. Resource cost variability can be exploited in this context to compare product
performance under different cost models such as fixed cost, measured simulation cost, and worst-case cost.

The approach has been demonstrated on an example of a SPL using the highly parallelizable MapReduce programming pattern as its common base product, and used to compare the performance of full versions to restricted demo versions of products. A restriction of the current work is the concrete semantics used for simulation, which necessitates a per-product trial and failure approach to validation. An interesting extension of the work presented in this paper is to use a symbolic semantics and apply symbolic execution techniques to analyze the deployment sensitive SPL models. This could allow the analysis to be lifted from concrete deployment scenarios for specific products to a more generalized analysis.

References


[22] E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. A formal model of user-defined resources in resource-restricted deployment scenarios. In B. Beckert, F. Damiani, and D. Gurov,


