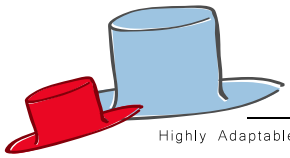


Project N°: **FP7-231620**

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**

Final version

| **Integrated Project supported by the 7th Framework Programme of the EC** | | |
|---|---|---|
| **Dissemination level** | | |
| 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) | |

# 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

# 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 [21, 43]. We have also developed the MetaABS meta-language to support adapting the runtime of the ABS language from within the language itself [47, 23], and we have studied semantics and optimization techniques for dynamic deltas [33]. These contributions have been reported in Tasks 3.3 and 3.5 [21, 23], and the paper [47] 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 [22, 46] and applied to the Fredhopper case study in Task 5.4 [25]. This is the subject of one of the submissions to the SCP special issue [56].

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 [27], 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 [18, 55]. 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 [18, 21]. We also examined methods to ensure the typing consistency of a running system, under possible update of its classes [37]. 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 [44, 41]
- 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 [3]

- The MetaABS tool [47] 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 [23]. We have proved that the model is sound and fully abstract with respect to a network unaware reference semantics [7, 12], and we have shown how adaptation can be performed in such a model with respect to load and latency-related measures [10].

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 [17, 36], and in Task 3.4 [8, 9, 11] 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 [42] 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 [25].

### Paper 2: MetaABS and Dynamic Model Updates

The paper [47] 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.

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

The paper [1] 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 [26].

### 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].

### 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.

### 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.

## 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:

`www.hats-project.eu/sites/default/files/Deliverable3.6-with-papers.pdf`.

# 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

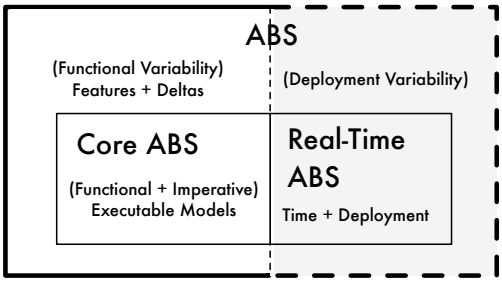


Figure 2.1: ABS language extension.

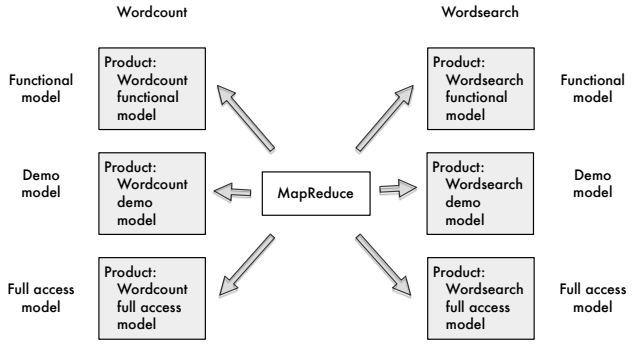


Figure 2.2: A family of products sharing an underlying MapReduce structure.

where deltas modify physical aspects of the system can likely be formalized using previous work on object mobility [40] and dynamic resource reallocation [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.

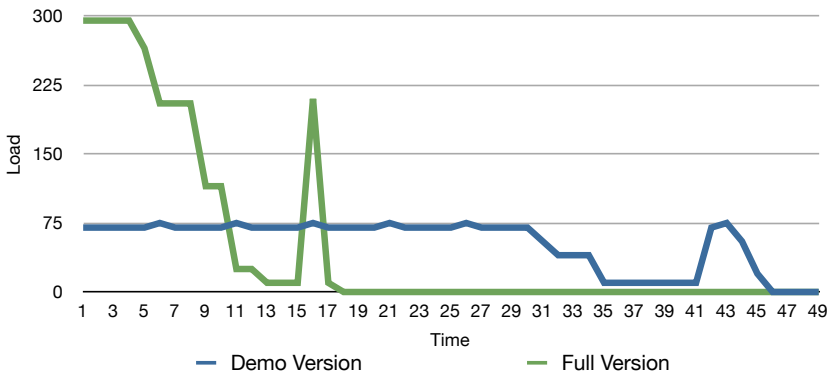


Figure 2.3: Simulation results: varying deployment model, cost and functional model constant.

# 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* [54, 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

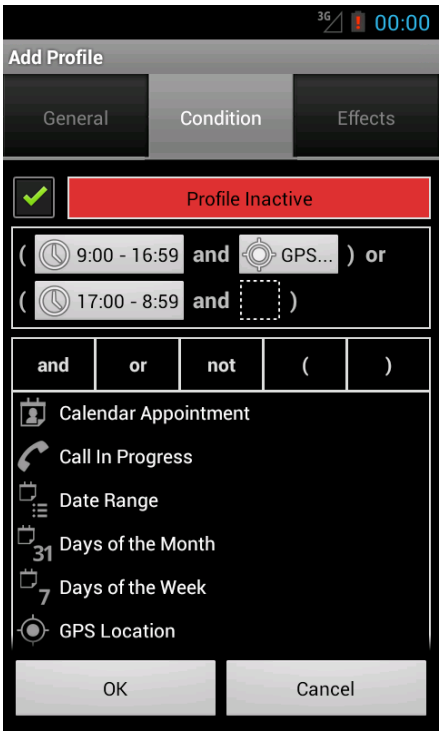


Figure 3.1: The Android interface for entering a condition (the dashed box is a drop-target for a new constraint)

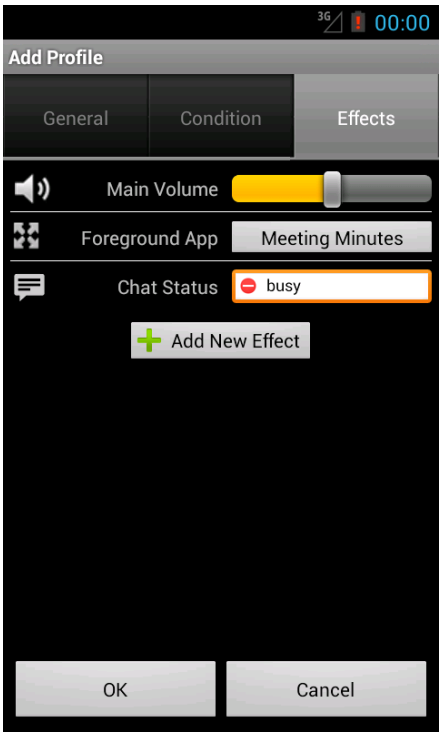


Figure 3.2: The Android interface for specifying the desired effects of a rule; this translates to a profile delta

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 $F_\mathrm{e}$ 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 $F'_\mathrm{e}$. 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* $\longrightarrow$ over a set of *configurations*:

$$\langle\, cn_1 \,\rangle \longrightarrow \langle\, cn_2 \,\rangle \longrightarrow \langle\, cn_3 \,\rangle \longrightarrow \langle\, cn_4 \,\rangle \longrightarrow \cdots$$

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

$$\langle\, F_\mathrm{e}, p \,\rangle \quad \in \quad \Phi \times \mathcal{P}$$

where $\Phi$ is the set of valid feature configurations and $\mathcal{P}$ 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:

$$\overbrace{\langle\, F_\mathrm{e}, p \,\rangle \longrightarrow \underbrace{\langle\, F'_\mathrm{e}, p \,\rangle \longrightarrow^* \langle\, F'_\mathrm{e}, p' \,\rangle}_{\text{local}} \longrightarrow \langle\, F''_\mathrm{e}, p' \,\rangle \longrightarrow^* \langle\, F''_\mathrm{e}, p'' \,\rangle \longrightarrow \cdots}^{\text{environmental}}$$

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' $F_\mathrm{l}$. The type of configuration we will be working with is

$$\langle\, F_\mathrm{e}, F_\mathrm{l}, p \,\rangle \quad \in \quad \Phi \times \Phi \times \mathcal{P}.$$

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

$$F \ominus G \quad \stackrel{\text{def}}{=} \quad (F \cup G)/(F \cap G)$$

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:

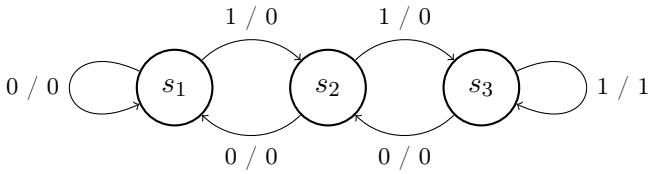


Figure 3.3: An example of an Mealy machine with input and output symbols $\{0, 1\}$

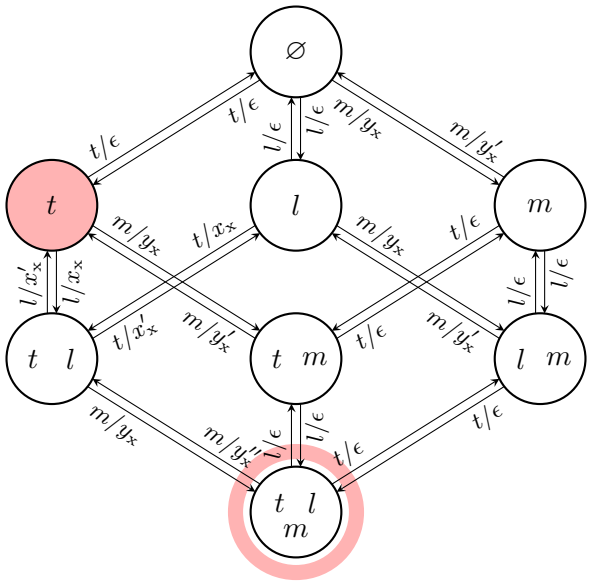


Figure 3.4: The DPL Mealy machine for the example product line from the SPC article [34]

LOC-DIFF-FULL 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 LOC-PRD, but requires storing an exponential number of deltas. So we need something a bit more clever:

LOC-DIFF-ND($MM(T_S)$) 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:

LOC-DIFF-ND($MM(T_M)$) When our product is out of date, nondeterministically choose a delta for a minimal feature-set $F_\Delta \subseteq F_e \ominus F_l$ 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 $F_{\mathrm{l}} = F_{\mathrm{e}}$. 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.

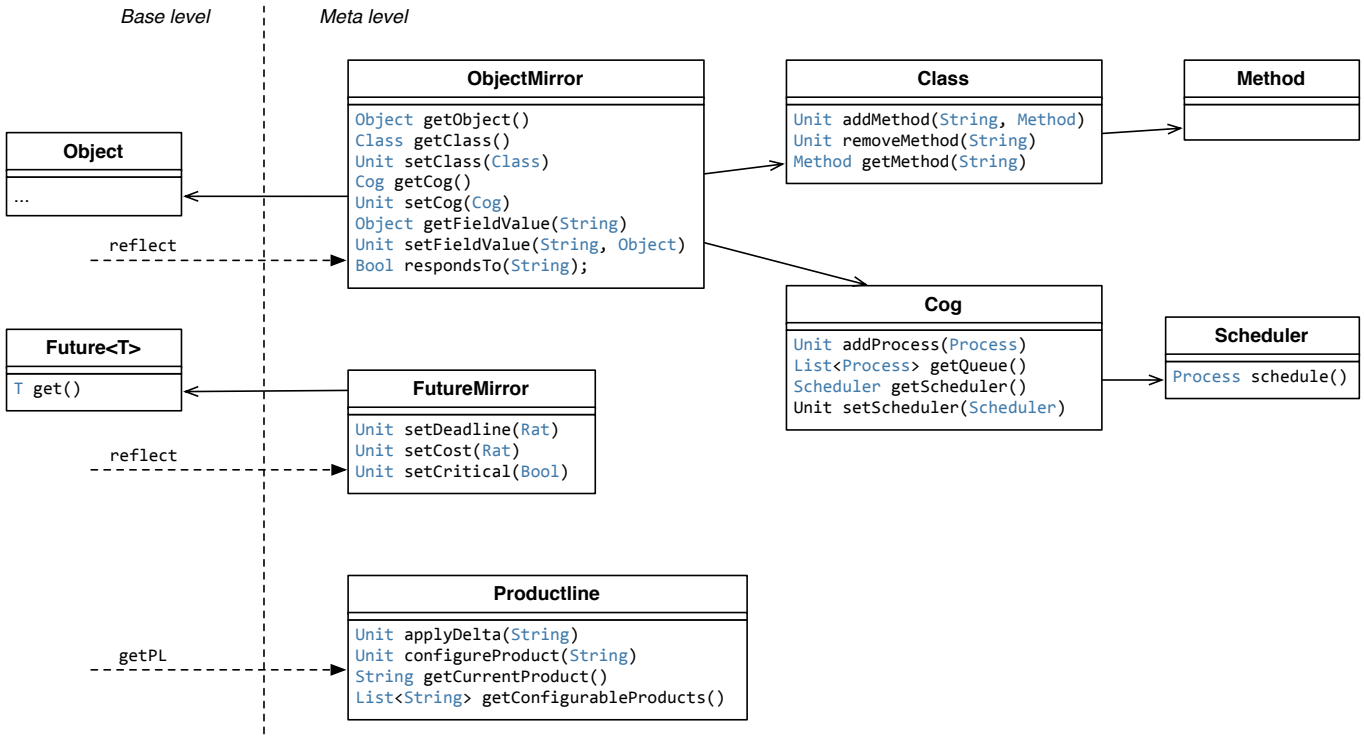


Figure 4.1: MetaABS interface. Labels on the dotted arrows denote the MetaABS functions that return an object of the type they point to.

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


[1] Elvira Albert, Taslim Arif, Karin Villela, and Damiano Zanardini. Resource-aware configuration in software product lines. Submitted for publication.

[2] Don S. Batory. Feature models, grammars, and propositional formulas. In J. Henk Obbink and Klaus Pohl, editors, *SPLC*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.

[3] Joakim Bjørk, Dave Clarke, Einar Broch Johnsen, and Olaf Owe. A type-safe model of adaptive object groups. In Natallia Kokash and António Ravara, editors, *FOCLASA*, volume 91 of *EPTCS*, pages 1–15, 2012.

[4] Joakim Bjørk, Frank S. de Boer, Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering*, 2012. Available online: `http://dx.doi.org/10.1007/s11334-012-0184-5`. To appear.

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

[6] Dave Clarke, Radu Muschevici, José Proença, Ina Schaefer, and Rudolf Schlatte. Variability modelling in the ABS language. In Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*. Springer-Verlag, 2011.

[7] Mads Dam. Location independent routing in process network overlays. Manuscript, 2013.

[8] Mads Dam, Bart Jacobs, Andreas Lundblad, and Frank Piessens. Provably correct inline monitoring for multithreaded Java-like programs. *Journal of Computer Security*, 21(1):37–59, 2010.

[9] Mads Dam, Bart Jacobs, Andreas Lundblad, and Frank Piessens. Security monitor inlining for multi-threaded Java. Accepted to Special Issue of MSCS, 2012.

[10] Mads Dam, Ali Jafari, Andreas Lundblad, and Karl Palmskog. ABS-NET: Fully decentralized runtime adaptation for distributed objects. Manuscript, 2013.

[11] Mads Dam, Gurvan Le Guernic, and Andreas Lundblad. Treedroid: a tree automaton based approach to enforcing data processing policies. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 894–905, New York, NY, USA, 2012. ACM.

[12] Mads Dam and Karl Palmskog. Efficient and fully abstract routing of futures in object network overlays. Manuscript, 2013.

[13] Ferruccio Damiani, Luca Padovani, and Ina Schaefer. A formal foundation for dynamic delta-oriented software product lines. In *Proc. GPCE'12*, pages 1–10. ACM Press, 2012.

[14] Ferruccio Damiani and Ina Schaefer. Dynamic delta-oriented programming. In Ina Schaefer, Isabel John, and Klaus Schmid, editors, *Software Product Lines, 15th International Conference, Munich, Germany, Workshop Proceedings (Volume 2)*, page 34. ACM, 2011.

[15] F.S. de Boer, M. Helvensteijn, and J. Winter. A Modal Logic for Abstract Delta Modeling. In *Workshop Proceedings of SPLC 2012*, 2012.

[16] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[17] First Report on Evolvable Systems, March 2010. Deliverable 3.1.a of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[18] Final Report on Evolvable Systems, March 2011. Deliverable 3.1.b of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[19] Full ABS Modeling Framework, March 2011. Deliverable 1.2 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[20] Analysis Final Report, December 2012. Deliverable 2.7 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[21] Hybrid Analysis for Evolvability, December 2012. Deliverable 3.3 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[22] Model Mining, March 2012. Deliverable 3.2 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[23] Autonomous Evolving Systems, March 2013. Deliverable 3.5 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[24] Correctness, March 2013. Deliverable 4.3 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[25] Evaluation of Tools and Techniques, March 2013. Deliverable 5.4 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[26] Trustworthiness, March 2013. Deliverable 4.4 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[27] Dilian Gurov, B.M. Østvold, and I. Schaefer. A hierarchical variability vodel for software product lines. In *Leveraging Applications of Formal Methods, Verification, and Validation : ISoLA 2011, Vienna, Austria, October 17-18, 2011*, number 336 in Communications in Computer and Information Science, pages 181–199, 2012.

[28] Reiner Hähnle, Michiel Helvensteijn, Einar Broch Johnsen, Michael Lienhardt, Davide Sangiorgi, Ina Schaefer, and Peter Y. H. Wong. HATS abstract behavioral specification: the architectural view. In Bernhard Beckert, Ferruccio Damiani, Frank de Boer, and Marcello M. Bonsangue, editors, *Proc. 10th International Symposium on Formal Methods for Components and Objects (FMCO 2011), Torino, Italy*, volume 7542 of *Lecture Notes in Computer Science*, pages 109–132. Springer-Verlag, 2013.

[29] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic software product lines. *Computer*, 41(4):93–95, 2008.

[30] M. Helvensteijn, R. Muschevici, and P.Y.H. Wong. Delta Modeling in Practice, a Fredhopper Case Study. In *Proceedings of the 6th International Workshop on Variability Modelling of Software-intensive Systems, Leipzig, Germany, January 25-27 2012*, ACM International Conference Proceedings Series. ACM, 2012.

[31] Michiel Helvensteijn. Abstract Delta Modeling: My Research Plan. In *Proceedings of the 16th International Software Product Line Conference - Volume 2*, pages 217–224. ACM, 2012.

[32] Michiel Helvensteijn. Delta Modeling Workflow. In *Proceedings of the 6th International Workshop on Variability Modelling of Software-intensive Systems, Leipzig, Germany, January 25-27 2012*, ACM International Conference Proceedings Series. ACM, 2012.

[33] Michiel Helvensteijn. Dynamic delta modeling. In Eduardo Santana de Almeida, Christa Schwanninger, and David Benavides, editors, *16th International Software Product Line Conference, SPLC, Salvador, Brazil, Volume 2*, pages 127–134. ACM, 2012.

[34] Michiel Helvensteijn. An Operational Semantics for Dynamic Product Lines. 2013.

[35] M. Hennessy. *The semantics of programming languages: an elementary introduction using structural operational semantics*. John Wiley & Sons, 1990.

[36] Antonio Bucchiarone Ivan Lanese and Fabrizio Montesi. A framework for rule-based dynamic adaptation. In *Proceedings of TGC 2010*, Lecture Notes in Computer Science. Springer-Verlag, 2010.

[37] Einar Broch Johnsen, Marcel Kyas, and Ingrid Chieh Yu. Dynamic classes: Modular asynchronous evolution of distributed concurrent objects. In Ana Cavalcanti and Dennis Dams, editors, *FM*, volume 5850 of *LNCS*, pages 596–611. Springer, 2009.

[38] Einar Broch Johnsen, Olaf Owe, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. Dynamic resource reallocation between deployment components. In J. S. Dong and H. Zhu, editors, *Proc. International Conference on Formal Engineering Methods (ICFEM'10)*, volume 6447 of *Lecture Notes in Computer Science*, pages 646–661. Springer-Verlag, November 2010.

[39] Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. Deployment variability in delta-oriented model. Submitted for publication.

[40] Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. A formal model of object mobility in resource-restricted deployment scenarios. In Farhad Arbab and Peter Ölveczky, editors, *Proc. 8th International Symposium on Formal Aspects of Component Software (FACS 2011)*, volume 7253 of *Lecture Notes in Computer Science*, pages 187–. Springer-Verlag, 2012.

[41] Michael Lienhardt, Mario Bravetti, and Davide Sangiorgi. An object group-based component model. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece*, volume 7609 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2012.

[42] Michael Lienhardt, Mario Bravetti, and Peter Wong. Component model for concurrent object groups: Theory and practice. Submitted for publication.

[43] Michael Lienhardt and Dave Clarke. Conflict detection in delta-oriented programming. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece*, volume 7609 of *Lecture Notes in Computer Science*, pages 178–192. Springer, 2012.

[44] Michaël Lienhardt, Ivan Lanese, Mario Bravetti, Davide Sangiorgi, Gianluigi Zavattaro, Yannick Welsch, Jan Schäfer, , and Arnd Poetzsch-Heffter. A component model for the abs language. In *Formal Methods for Components and Objects (FMCO) 2010*, volume 6957 of *Lecture Notes in Computer Science*, pages 165–185. Springer Berlin / Heidelberg, 2010.

[45] G.H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[46] K. Meinke. Learning-based software testing: a tutorial. In *Proc. Int. ISoLA workshop on Machine Learning for Software Construction*, CCIS. Springer-Verlag, 2012.

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

[48] G.D. Plotkin. A structural approach to operational semantics. 1981.

[49] G.D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60:3–15, 2004.

[50] Klaus Pohl, Günter Böckle, and Frank Van Der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.

[51] Ina Schaefer. Variability Modelling for Model-Driven Development of Software Product Lines. In *Proc. of 4th Intl. Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010)*, 2010.

[52] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *Proc. of 14th Software Product Line Conference (SPLC 2010)*, September 2010.

[53] Ina Schaefer, Alexander Worret, and Arnd Poetzsch-Heffter. A Model-Based Framework for Automated Product Derivation. In *Proc. of Workshop in Model-based Approaches for Product Line Engineering (MAPLE 2009)*, 2009.

[54] D. Scott. *Outline of a Mathematical Theory of Computation*. Oxford University Computing Laboratory, Programming Research Group, 1970.

[55] Yannick Welsch and Arnd Poetzsch-Heffter. Full abstraction at package boundaries of object-oriented languages. In Adenilso Simao and Carroll Morgan, editors, *Formal Methods, Foundations and Applications*, volume 7021 of *LNCS*, pages 28–43. Springer-Verlag, 2011.

[56] Peter Wong, Richard Bubel, Frank S. de Boer, Miguel Gomez Zamalloa, Stijn de Gouw, Reiner Hähnle, Karl Meinke, and Mudassar A. Sundhu. Testing abstract behavioral specifications. 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.