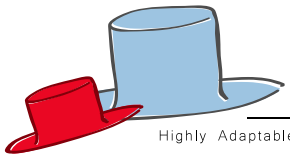

Project N°: **FP7-231620**

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 D3.1.b
Final Report on Evolvable Systems

Due date of deliverable: (T24)

Actual submission date: 1st March 2011

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

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

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:
Final Report on Evolvable Systems

This document summarizes deliverable D3.1.b of project FP7-231620 (HATS), an Integrated Project supported by the 7th Framework Programme of the EC within the FET (Future and Emerging Technologies) scheme. Full information on this project, including the contents of this deliverable, is available online at `http://www.hats-project.eu`.

This deliverable reports on the investigations pursued in the setting of Task 3.1. The goal of this task is to analyze and model fundamental aspects of system and component evolution, formally pin down and compare the concepts, construct scenarios, and build formal execution models that can be subjected to evaluation, theoretical examination, comparison, and simulation.

From the different research lines investigated in the first year of the project in Task 3.1, which have been reported in the previous deliverable D3.1.a., a subset was selected to be further studied and developed in the second year. In particular, we continued work on (a) a framework for behavioral equivalence and compatibility of classes to support modular refactorings (see Chapter 2) and (b) a component model for specifying dynamic evolution steps on the ABS model layer (see Chapter 4).

List of Authors

Mads Dam (KTH)
Einar Broch Johnsen (UIO)
Ivan Lanese (BOL)
Michael Lienhardt (BOL)
Arnd Poetzsch-Heffter (UKL)
Davide Sangiorgi (BOL)
Yannick Welsch (UKL)

Contents

1 Introduction: HATS Support for Model-based Evolution of Software Families 4
- 1.1 Introduction to the HATS Approach for Evolution . . . 5
- 1.2 Changing Requirements . . . 7
- 1.3 Evolving Implementations . . . 8
- 1.4 Dynamic Evolution . . . 8
- 1.5 Discussion . . . 9

2 Behavioral Compatibility of Classes 10
- 2.1 Introduction . . . 10
- 2.2 Related Work . . . 10
- 2.3 Testing Compatibility . . . 11
- 2.4 Fully Abstract Trace Semantics . . . 13
- 2.5 Proving Compatibility/Equivalence . . . 15
- 2.6 Results and Outlook . . . 15

3 Towards Dynamic Evolution of Features 16
- 3.1 Introduction . . . 16
- 3.2 Dynamic Evolution of Features: The Approach . . . 18
- 3.3 Results and Outlook . . . 20

4 Component Model with Evolvability Constructs 21
- 4.1 Introduction . . . 21
- 4.2 Dynamically Evolvable Components . . . 21
- 4.3 Component Model . . . 22
- 4.4 Formalization . . . 24
- 4.5 Examples . . . 25
- 4.6 Related Work . . . 28
- 4.7 Results and Outlook . . . 29

5 Rule-Based Dynamic Evolution 30
- 5.1 Introduction . . . 30
- 5.2 Results . . . 30
- 5.3 Related Work . . . 31
- 5.4 Outlook . . . 31

6 Conclusion 32

Bibliography 32

Glossary 37

Chapter 1

Introduction: HATS Support for Model-based Evolution of Software Families

What's the use of a good quotation if you can't change it?

— Doctor Who

Long-living software evolves over time. A central aspect of HATS is to support the evolution of individual software systems and of families of software systems. This deliverable reports on the studies in Task 3.1 and extends the work already presented in Deliverable D3.1.a [16] (in particular, we refer to D3.1.a for an introduction to the challenges of evolution in HATS and to the deliverable D1.2 [18] of the EternalS Coordination Action for a broader survey of evolvable systems). Task 3.1 is of an explorative nature. Its central goal is to analyze and model fundamental aspects of evolution and to explain how they are addressed by HATS. The Description of Work for Task 3.1 especially mentions the following three questions as relevant challenges for the HATS approach:

1. How can evolution be constrained in a way such that overall consistency properties can be derived and maintained during evolution?
2. What does correctness mean for evolution steps that are intentionally changing the behavior of individual components?
3. What are the fundamental dynamic mechanisms needed to model system evolution at runtime?

The deliverable is structured as follows: This chapter provides an introduction to the HATS support for model-based evolution of software families. It describes the parts of the HATS framework that are relevant for linking the work in this task to the project as a whole. Furthermore, it discusses the advantages of the HATS approach both in general and with respect to the questions above. The remaining four chapters present specific aspects that were investigated as part of this task in more detail:

- Chapter 2 analyzes techniques to guarantee source compatibility where a source is a model description or a program. The underlying goal is to control source evolution.
- Chapter 3 is about techniques for runtime updates of deployed object systems, i.e., the evolution of installed code bases.
- Chapter 4 presents a component abstraction for evolution that bridges the model and implementation layer and allows studying dynamic evolution of systems on both layers.
- Chapter 5 investigates techniques to describe automated system adaptations and evolutions by an abstract rule-based approach.

In the next sections we describe the relation of the sketched topics of Chapters 2–5 to the three questions above and to the HATS approach in general. Earlier work of this task that is not covered in this deliverable was concerned with autonomous system adaptation and test system evolution (see Chapters 5 and 6 of Deliverable D3.1.a [16] "First report on Evolvable Systems"). The aspects about the reuse of proofs or other analysis information under changes in components and specifications (mentioned in the Description of Work of Task 3.1) will be studied by CTH as part of Task 4.3 ("Correctness") where it was decided to fit better.

1.1 Introduction to the HATS Approach for Evolution

In every general model-based approach to software evolution of distributed systems, we can distinguish three different layers:

- models, a more or less consistent collection of descriptions;
- programs that implement the behavior as described by the models;
- installations/deployments that configure executable programs together with data sources on distributed platforms.

Models play different roles in the evolution of software, depending on the reasons underlying the evolution steps. Essentially, there are four reasons why software systems and installations evolve over time:

1. *Changing requirements*: Changing functional or non-functional requirements lead to extensions or modifications of the software system behavior; e.g., new features or a higher degree of availability could be required.
2. *Code maintenance*: Implementations are corrected or improved (e.g., code is refactored, bugs are fixed).
3. *Technology changes*: The technologies (the languages used, external components, libraries, APIs, etc.) or platforms (e.g., computers, new OS versions, GUI frameworks, etc.) have changed and the implementations must be adapted accordingly.
4. *Changing environments*: System installations may need to *dynamically* react to changes in the environment of the system (e.g., increasing work load, context changes of mobile systems); these changes might result in short-time adaptations, but can also cause long-living changes in the runtime configuration[1].

Each of these kinds of evolution addresses different stakeholders. Changing requirements are related to users or customers of the software. They occur because the original specifications are incomplete or ambiguous or because new needs arise that had not been predicted at design time. Code maintenance is the task of the developers. Platform issues establish dependencies between the developers and third-party providers of the platform components. Dynamic evolution steps and adaptations are usually managed by system administrators or by the self-adaptation capabilities of the system.

Using a model-based approach allows expressing and analyzing the requirement changes on the layer of the models. This is an advantage as it simplifies the analysis of the evolution steps. Furthermore, the evolved models can be used for evolution of the code and the installations either as a reference for checking the correctness of code or as a source for generative techniques. Because evolution may happen on different abstraction layers of software systems and software families, it is usually expressed by different, often only loosely related, models or descriptions. The HATS approach[2] extends and refines a general model-based approach to evolution in three ways:

[1] Of course, if the system is not capable of handling the changes in the environment dynamically, a static evolution step is necessary)

[2] When we speak of the HATS approach in this section, we refer to the capabilities of the completed framework and not to the state of development at the time this deliverable is written. This coincides with the explorative nature of Task 3.1.

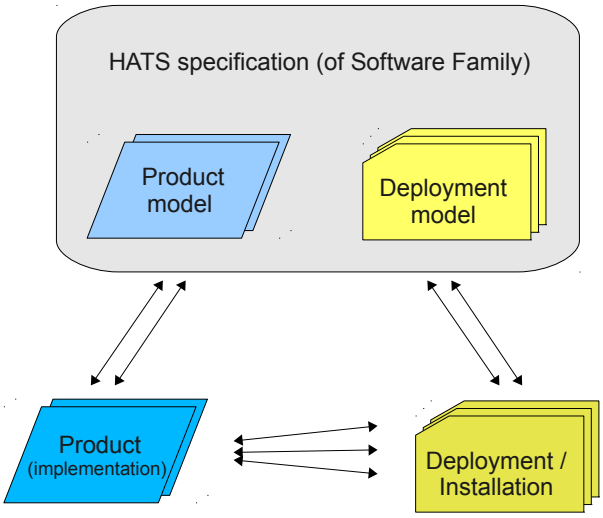

Figure 1.1: Overview of artifacts involved in the HATS approach.

1. It supports modeling of software families (see in particular Tasks 1.2 "Feature Modeling, Platform Models, and Configuration" and 2.2 "Feature integration"). This is important because most software systems come in many variants. In addition, the support for software families allows to express evolution caused by changing requirements as *family extension and modification (variability in time)*; i.e., such evolution steps can be expressed within the framework using the feature models.

2. It provides a behavioral specification with an *integrated semantics* as a referential backbone that allows relating evolving requirements, implementations, system variants, and platforms in a formal and consistent way. The integrated semantics is in particular necessary for tool-based checking.

3. The use of object- and component-orientation at both the model and implementation layers with compatible interfacing concepts allows the development and support of dynamic installation/deployment updates at the model layer.

The relevant artifacts involved in the HATS approach are sketched in Figure 1.1. The HATS specifications formally describe a family of software products (product models) together with deployment models. It may additionally express properties of the family and the product models. The product models allow to partially generate and check product implementations. Each product implementation can be used in different deployments/installations. In the following three subsections, we explain the role of HATS models for

1. software evolution with respect to changing requirements,

2. evolving implementations and

3. dynamic evolution, i.e., adaptations of running systems.

We assume an evolution scenario in which a HATS specification of a software family is available and maintained (*HATS approach*). At the end of this chapter, we compare the *HATS approach* to a *traditional approach* that only uses informal descriptions or different models without an integrated semantics.

1.2 Changing Requirements

Changing requirements lead to the evolution of the software family. We distinguish three kinds of evolution steps:

- modifying the product set,
- modifying individual products and
- modifying components.

As the HATS approach provides a semantically integrated specification of a software family, evolution steps can be formally analyzed at an abstraction layer well above the program and configuration code of a distributed implementation. In the following, we explain the different evolution steps based on changing requirements and discuss how HATS addresses the questions mentioned at the beginning of this chapter.

Modifying the software family. Modifying the product set can simply mean changing the feature model. In general, however, adding new features and thus new products requires changing parts of the model descriptions (called *deltas* in the HATS approach) and/or adding new deltas (analogously for deletion of products).

As the HATS approach provides formal descriptions of the products and their behavior in the *original* family, i.e., the family before the evolution step, and descriptions of the *evolved* family (after the evolution step), the approach allows the evolved family to be checked formally against the following properties:

1. All products of the original family are also part of the evolved family and behave the same way.
2. The new features do not cause (undesired) feature interactions between features of the original products.
3. Global properties of the family remain valid for the evolved family.

These properties are strong criteria to verify the consistency and adequacy of the software family evolution. It should be noted that they can as well be used in evolution scenarios in which not all original products or global properties should remain unchanged. For example, an evolution step from an original family with a hundred global properties might maintain only 95 of them and add several new properties. Even in such scenarios, the framework provides crucial support to maintain consistency and correctness in the software family, because most of the properties can be reused for checking.

Modifying individual products and components. The products of a family are built from components. Several products can share components. The HATS approach supports a behavioral semantics for components and allows us to verify behavioral relationships between different versions of components. For evolution, the following relationships are of particular interest:

- Equivalence: the behavior of the evolved component is equivalent to the original component, i.e., the evolution step is a refactoring.
- Refinement: the evolved component refines the behavior of the original component in the sense that it constrains nondeterminism.
- Extension: the evolved component shows the same behavior if used in contexts of the original family, but also provides additional functionality for the contexts in the evolved family.
- Projection: if only a part of the component interface is used, the evolved component behaves as the original component. The behavior of the rest of the interface might change.

The support for these notions in the ABS modeling language is based on the input/output traces of components. In evolution scenarios, the traces provide important feedback for checking whether evolution steps maintain those component properties that should be stable under the evolution step. Projection allows checking aspects of components that are partially modified. The stable behavior can be verified against the original component. The new behavior has to be verified against newly specified properties, in the context of a configured product, or in the contexts of the software family.

Products are either components themselves or are configured from components. In the latter case, evolution steps can also pertain to the configuration. The HATS approach allows checking whether invariants of the original product are also valid in the evolved configuration.

In Chapter 2 we present foundational work for analyzing and proving source code components equivalent. The basic contribution is a fully abstract semantics that is needed to check the above relationships in a description/implementation-independent way. It will be further developed and applied to ABS in Task 2.6 "Refinement and Abstraction".

1.3 Evolving Implementations

To simplify the following presentation, we use the term *implementation* for program code, installation, and platform aspects. In principle, implementations evolve for the same reasons that were mentioned for software systems at the beginning of this chapter. However, model-based approaches support an intermediate model layer between requirements and implementations. The models are central for generation and analysis of implementations and their parts. In particular, in the case of changing requirements, the evolved models precisely specify the goal of the evolution steps. In the case of code maintenance and technology changes, the original models provide the point of reference against which the new implementation can be checked.

Whereas all model-based approaches provide a helpful point of reference for evolution (with varying degree), the specific strength of the HATS approach for generation results from (1) the support for software families together with product derivation and (2) the semantics-based integration of the specification concepts of ABS around an object-oriented language. The specific strength of the HATS approach for analysis follows from the techniques for feature-oriented verification and from the behavioral semantics supporting modular verification and checking. Contributions to feature-oriented verification were made in Task 2.5 "Verification of General Behaviour Properties".

1.4 Dynamic Evolution

Maybe the most complex kind of evolution is dynamic evolution. By *dynamic evolution* of a system we refer to the possibility that the functionalities offered by the system may change over time while the system is running. This may involve reconfiguring and updating applications to meet new requirements and new operating conditions, which were unexpected when the application was developed and deployed. In this task, we considered three important topics for dynamic evolution:

1. runtime updates of deployed object systems at the level of classes (Chapter 3),
2. component abstractions to express dynamic evolution at the model level (Chapter 4) and
3. techniques to describe automated rule-based evolution (Chapter 5).

This work addresses the third question for this task: "What are the fundamental dynamic mechanisms needed to model system evolution at runtime?". The project results on runtime updates (mainly developed at the beginning of the project) are important to realize evolution of installed systems. This experience is needed for Task 3.4 "Evolvability at Bytecode Level". On the other hand, it showed that a higher-level handling and modeling of runtime adaptations is important and this motivated an effort by BOL with the goal of isolating interesting constructs for expressing dynamic evolvability. Specifically, the goal was to identify constructs that offer enough expressiveness to model common patterns of evolvability for systems,

and at the same time are mathematically robust so to allow formal analysis of desired behavioral properties of the system; this is in accordance with the goal to constrain evolution in a way that overall consistency properties can be derived and maintained during dynamic evolution.

For this, two directions of work had been pursued. In the first direction, a core calculus of evolvable components was defined, following the process calculus tradition (see Chapter 4). The integration of this more heavy-weight component model[3] for evolution into ABS is under way and will be continued as part of Task 4.3 "Correctness". In the second direction, a new, language-independent, approach was studied which is based on the combination of *adaptation hooks* provided by the adaptable application, specifying where adaptation can be performed, and *adaptation rules* external to the application, specifying when and how adaptation can be performed (Chapter 5). This is an important step towards systems that are capable of adapting themselves to changes in the surrounding environment. Both calculi/models provide helpful background for designing autonomously evolving systems, which will be the topic of Task 3.5 "Autonomously Evolving Systems".

1.5 Discussion

Model-based approaches are important for evolution as they provide an intermediate layer between requirements and implementations. In the case of changing requirements, the evolution steps can be analyzed on the model layer, which is simpler when compared to an analysis directly on the implementation layer. In cases where only the implementation changes, the consistency and correctness of the evolution step can be formally checked with respect to the model.

Compared to *traditional model-based approaches* that use only informal descriptions or different models without an integrated semantics, the HATS approach aims to make three central contributions:

1. Extending model-based techniques to express variability and evolvability of software families within the model space.

2. Providing a semantics-based framework as backbone for the integration of different modeling and description techniques.

3. Explaining dynamic adaptations on the model layer for a better control.

Integrating techniques from family engineering into the models not only allows capturing variability of a system at a *point in time*. The same feature-based modeling techniques can be exploited to express evolution as part of the modeling approach by treating evolution as *variability over time*. A typical example of the latter aspect is an evolution step where a new additional feature is added or where an alternative feature is incorporated into the feature model.

[3]Compared to the light-weight concurrent object groups (COGs) of Core ABS.

Chapter 2

Behavioral Compatibility of Classes

2.1 Introduction

Classes are used as building blocks in OO design (e.g., ABS) to describe the behavior of systems. It is useful to compare different implementations of classes or sets of classes (e.g., libraries). One particular comparison is to check for conformity, i.e., whether a new version provides at least the same behavior as the old version. Every refactoring, i.e., behavior preserving evolution step, should guarantee conformity. Our central goal is to support *modular* refactorings, i.e., refactorings for single classes or components that are correct in all possible contexts or a well-defined set of contexts.

A standard way to compare classes is by using the notion of contextual equivalence: Two classes are equivalent if and only if, for every possible (class) context, they exhibit the same operational behavior. Proving equivalence using this definition directly is in general not possible. Even if one could deal with the in general infinite amount of possible contexts, the proof can not assume equal stores (heap/stack) as the compared classes may manipulate the store in a similar but not identical way.

In contrast, denotational methods give a meaning to program elements without quantifying over all possible contexts. A denotational semantics for classes is called *fully abstract* if classes that have the same denotation are exactly those that are contextually equivalent. Proving that two (sets of) classes are equivalent in the (fully abstract) denotational setting amounts to proving that they have the same denotation. For practical proofs, a simple denotational model, i.e., simple semantic domains, is needed. Furthermore the denotation should be defined in a constructive way. Giving fully abstract denotations to classes also helps to better understand OO languages as the denotations capture exactly the relevant information of a class.

2.2 Related Work

Previous work on reasoning about class equivalence can be roughly divided into two different categories.

One way to relate two program parts is by using bisimulations, which were first used by Hennessy and Milner [25] to reason about concurrent programs. Sumii and Pierce used bisimulations which are sound and complete with respect to contextual equivalence in a language with dynamic sealing [51] and in a language with type abstraction and recursion [52]. Koutavas and Wand, building on their earlier work [31] and the work of Sumii and Pierce, used bisimulations to reason about the equivalence of classes [32] in different Java subsets. The subset they considered includes inheritance and down-casting. Their language, however, neither considers interfaces nor accessibility of types.

Another way to relate program parts is by using denotational methods [14]. The denotational semantics according to these methods provide representations of program parts (e.g., classes) as mathematical objects describing how program parts modify the state and heap. However, these denotations are not *abstract* enough, i.e., they differentiate between classes that have the same behavior. Banerjee and Naumann [5] presented a method to reason about whole-program equivalence in a Java subset similar to the one described before. Under a notion of confinement for class tables, they prove equivalence between different

implementations of a class by relating their denotations by simulations. Silva et al. [49] extend this work to prove several refactoring laws for whole hierarchies of classes; this is similar to our goals, but we want to use a different reasoning framework. Jeffrey and Rathke [27] give a fully abstract trace semantics for a Java subset with a package-like construct. However, they do not consider inheritance and downcasting, and also do not apply their technique to prove the equivalence between different implementations. Similarly, Abraham et al. [1] give a fully abstract semantics for a concurrent class-based language and they also do not consider inheritance.

In these last techniques, the denotation of a class is characterized as the interaction of a class with its environment, i.e., by which messages a class responds to input messages. Our approach extends this to a setting where it is not evident any more what the interactions are, i.e., we slice objects into parts which are known to the context and parts which are known to the implementation under investigation. Our goal is also to provide better connections between the works on fully abstract trace-based semantics and the first-mentioned works using bisimulations.

2.3 Testing Compatibility

We developed our approach for an object-oriented language (LPJava, see Figure 2.1), which can be considered as a *sequential* subset of Core ABS plus modules. The language has interfaces, classes and subtyping, which directly correspond to ABS. To consider the additional challenge of inheritance and type hiding (also present in Full ABS due to a module system), we added subclassing and a package system to our language. Classes can extend other classes, and can be declared either package-local or public. We have not included data types, as they do not provide additional insight into our results. For simplicity, we assume that methods are public and fields are private. This is a bit more general as for ABS, where methods are also implicitly public but fields are private to the current object instance. Our language also allows explicit casting[1].

$$
\begin{array}{rcl}
K, X, Y & ::= & \overline{Q} \\
Q, R & ::= & \textbf{package}\ p\ \textbf{;}\ \overline{D} \\
D & ::= & [\textbf{public}]\ \textbf{class}\ c\ \textbf{extds}\ p.c\ \textbf{impls}\ \overline{p.i}\ \{\ \overline{F}\ \overline{M}\ \} \\
 & | & [\textbf{public}]\ \textbf{interface}\ i\ \textbf{extds}\ \overline{p.i}\ \{\ \overline{M}\ \} \\
F & ::= & \textbf{private}\ p.t\ f\ \textbf{;} \\
M & ::= & \textbf{public}\ p.t\ m(\ \overline{p.t\ v}\ )\ \left(\ \textbf{;}\ |\ \{\ e\ \}\ \right) \\
e & ::= & v \mid \textbf{null} \mid \textbf{new}\ p.c() \mid (p.t)\ e \mid e.f \\
 & | & e.f = e \mid \textbf{let}\ p.t\ v = e\ \textbf{in}\ e \mid e.m(\ \overline{e}\ ) \\
 & | & e == e\ ?\ e\ :\ e
\end{array}
\qquad
\begin{array}{rcl}
t & ::= & c \mid i \\
c & \in & \text{class names, including } \textit{Object} \\
i & \in & \text{interface names} \\
p & \in & \text{package names, including } \textit{lang} \\
f & \in & \text{field names} \\
m & \in & \text{method names} \\
v & \in & \text{variable names, including } \textbf{this}
\end{array}
$$

Figure 2.1: Abstract syntax of LPJava.

Example. As a simple example for behavioral compatibility of classes, we consider three different implementations of the Cell class in Figure 2.2. The Cell class models a storage unit that allows to get and set values. We provide different versions of the Cell class. The first version is a very simple one. It has a field which stores the last value that was set. The second version is equivalent to the first one, however, it stores the information redundantly. It is behaviorally equivalent to the first one. The third version preserves the behavior of the first two versions, but adds additional functionality, namely, it allows access to the second-last value that was stored.

Notions and notations. A *codebase*, which consists of a set of packages, is denoted by K, X or Y. If it satisfies all the well-formedness conditions of the language, i.e., well-formed type hierarchy, well-typedness

[1] This leads to more distinguishing power from class contexts.

```
public interface Val {}

public class Cell { // Version 1
  private Val v;
  public void set(Val nv) { v = nv; }
  public Val get() { return v; }
}

public class Cell { // Version 2
  private Val v1, v2;
  private boolean f;
  public void set(Val nv) {
    f = !f;
    v1 = nv;
    v2 = nv;
  }
  public Val get() { return f ? v1 : v2; }
}

public class Cell { // Version 3 (Caching)
  private Val v1, v2;
  private boolean f;
  public void set(Val nv) {
    f = !f ;
    if (f)
      v1 = nv;
    else
      v2 = nv;
  }
  public Val get() { return f ? v1 : v2; }
  public Val getPrevious() { return f ? v2 : v1; }
}
```

Figure 2.2: Example of different versions of a Cell class.

of all expressions, etc., we write $\vdash K$ (or $\vdash X$, $\vdash Y$) and call such a codebase a *component*. To join two different codebases into a larger codebase, we write them in juxtaposition (i.e., KX). If we join a codebase K and a component X, we often call K a *context* of X.

A prerequisite for two components to have the same behavior is that whenever the first component can be joined with a context into a larger component, then the second one can be joined as well using the same context. This property is called *source compatibility*:

Definition 2.3.1 (Source compatibility)**.** *A component Y is* source compatible *with a component X iff for any codebase K: $\vdash KX$ implies $\vdash KY$.*

It is important to notice that the definition does not allow automatically checking that a component Y is source compatible with X, because the definition quantifies over an infinite set of contexts. In Deliverable D3.1.a "First report on Evolvable Systems" [16], we studied how to derive a set of necessary and sufficient syntactic conditions that guarantee that Y is source compatible with X. In this deliverable, we focus on behavioral compatibility.

If K is a context of X and $p.c$ is a startup class of K, i.e. containing the main method, we call $K_{p.c}$ a *program context* of X. We write (X, ζ) for a runtime configuration where X represents the program and ζ represents the stack and heap of the computation. We assume there is some operational semantics defined by the reduction relation $\longrightarrow$ such that $(X, \zeta) \longrightarrow (X, \zeta')$ means that (X, ζ) can be reduced to (X, ζ'). We write $(X, \zeta) \downarrow$ to denote that there exists some terminal configuration (X, ζ_{term}) such that $(X, \zeta) \longrightarrow (X, \zeta_{term})$.

We can then use the standard notion of testing or contextual compatibility [42] to relate two components. This means that every program context which terminates with the first component must also terminate with the second component.

Definition 2.3.2 (Testing compatibility)**.** *A component Y is* testing compatible *with X iff Y is source compatible with X and for any program context K of X: $(KX, \zeta_{init}) \downarrow$ implies $(KY, \zeta_{init}) \downarrow$.*

The definition of testing compatibility again quantifies over all program contexts and cannot be used for proving that two components are compatible.

2.4 Fully Abstract Trace Semantics

In this section, we characterize the behavior of a component X in terms of its possible interaction traces with program contexts. We first define the interaction traces of X with a specific program context, then, we introduce nondeterministic expressions that allow defining a most general context (that exactly simulates all possible contexts).

We say that X *controls the execution* if the code of X is being executed; otherwise the context (K) controls the execution. An interaction is a change of control. Labels record changes of control. An interaction trace is a finite sequence of labels (see Figure 2.3). Interaction is considered from the viewpoint of the component. Input labels (marked by ?) express a change of control from the context to the component; output labels (marked by !) express a change from the component to the context. There are input and output labels for method invocation and return, as well as a label for error aborts. The labels for method invocation and return include the parameter and result values together with their (abstracted) types.

To compare traces of different components and with different contexts, we abstract from package local types and types declared in the context. Package local types should not appear in the labels, because different components might use different local types. Types declared in the context are abstracted so that the labels are independent of the types declared in possible contexts. The *abstracted* types are represented as intersections of public types declared in X that could have a common subtype in the context.

We enhance the reduction relation $\longrightarrow$ in such a way that at the points of interaction labels are generated. This new relation is denoted as $\stackrel{\bar{l}}{\Longrightarrow}$ where $\bar{l}$ represents the sequence of labels that is generated. This sequence of labels is also called a *trace*.

o	::=	$j{:}T_\cap$	object reference with abstracted type
v	::=	$o \mid \mathbf{null}$	values in labels
μ	::=	$\texttt{call}\ o.m(\overline{v})$	call message
	$\mid$	$\texttt{rtrn}\ v$	return message
l	::=	$\mu! \mid \mu? \mid \texttt{error}$	label

Figure 2.3: Labels.

Definition 2.4.1 (Traces)**.** *The* traces *of X in a program context K are:*

$$\mathit{Traces}(KX) = \{\bar{l} \mid \exists \zeta : (KX, \zeta_{init}) \stackrel{\bar{l}}{\Longrightarrow} (KX, \zeta)\}$$

Two traces t_1 and t_2 are equivalent iff they have the same length and there is a renaming ρ of object identifiers such that $t_1 \equiv_\rho t_2$.

Note that traces in $\mathit{Traces}(KX)$ only refer to types and methods in X. Also note that $\mathit{Traces}(KX)$ is prefix-closed.

Most General Context. For a component X, we construct a most general context κ_X that enables all possible interactions that X can engage in with any possible context. Compared to a concrete context, κ_X abstracts over types, objects, and operational steps. To represent κ_X, we extend LPJava by adding nondeterministic expressions. We do not explain the construction of κ_X from X here, but rather we assume that such a construction is possible.

We can then finally give the denotation of a component X as the interactions of the most general context with the component X.

Definition 2.4.2 (Denotation of a component)**.** *The* denotation of a component *X is defined as $\mathit{Traces}(\kappa_X X)$.*

Note that this definition does not quantify anymore over all possible contexts. We can then define a partial order on component denotations which we call *behavioral compatibility.*

Definition 2.4.3 (Behavioral compatibility)**.** *A component Y is* behaviorally compatible *with X iff Y is source compatible with X and $\mathit{Traces}(\kappa_X X) \subseteq \mathit{Traces}(\kappa_Y Y)$.*

Theorem 2.4.4 (Full abstraction)**.** *Two components are testing compatible if and only if they are behaviorally compatible.*

Proof. We give a short proof outline to show the amount of effort necessary to prove the theorem above. We have done most of the proofs so far on paper. The following main lemmata are needed:

- Component independency: This means that components execute independently of their context. If K_1 and K_2 are two contexts for a component X and $t \in \mathit{Traces}(K_1 X)$ and $t \in \mathit{Traces}(K_2 X)$ where the last label of t is an input label, then the component responds with the same output label in both cases. The proof for this lemma goes by establishing appropriate relations between the runtime configuration of $K_1 X$ and $K_2 X$ and proving their preservation.

- Context independency: This means that contexts execute independently of their components and it is the dual to component independency.

- Trace abstraction: This means that every concrete program context can be simulated by the most general context. The proof involves the construction of the most general context and relations between a run of this most general context and a concrete context.

- Trace concretization: This is again the dual to trace abstraction and means that for every trace generated by the most general context, there is exist some concrete context which generates the same trace. This is proved by giving a construction of such a concrete context.

□

Similar to compatibility, we can give a notion of equivalence for all the previous definitions.

Definition 2.4.5 (ψ equivalence)**.** *Components X and Y are* ψ equivalent*, where* $\psi \in$ {source, behavioral, testing}*, if X is ψ compatible with Y and vice versa.*

2.5 Proving Compatibility/Equivalence

Using the fully abstract denotations of components, we can prove compatibility between components. The proofs proceed by induction on the length of the traces and rely on the constructive definition of the component denotation. Different proof schemes are possible, however. We have not yet investigated the possibilities here but plan to do so in the future. One way is using coupling invariants (similar as in [5]). The presented proof scheme is state-based. Using the coupling invariants, the proof obligations are as follows: if an incoming message (input label) arrives, assuming the invariant holds, then both components respond in the same way. The coupling invariant only needs to talk about the component-local store and the local view on the (evaluation) stack, as the remaining store is identical for any run (a property of component independency). For the example depicted in Figure 2.2, we would need to define a coupling invariant between the values stored in the fields v, v1, v2 and f. The method for reasoning about the compatibility of components is sound and complete.

Theorem 2.5.1 (Soundness and Completeness of the Proof Method)**.** *There exists a coupling invariant iff two components are behaviorally compatible.*

Weaker equivalences. Sometimes we are not interested in preserving the full behavior of components in an evolution step. For example, we might be interested in checking that the new version of a component has the same behavior as the old one with respect to a subset of its interface methods. In this case, we can compare only those traces including these methods. This is a typical evolution scenario. Similarly, we can use our approach to prove that components behave the same in a restricted set of contexts. This is important w.r.t. evolution of software families, because the family defines the set of possible contexts of a component.

2.6 Results and Outlook

UKL has worked out the details of the fully abstraction proof and is currently preparing a publication of it. As part of Task 2.6 "Refinement and Abstraction", we plan to investigate in more depth how to formally prove compatibilities/equivalences/refinement using our denotations and how to handle additional language constructs of the ABS language.

Chapter 3

Towards Dynamic Evolution of Features

The following introduction recalls the basic motivations and notions of dynamic evolution of class-based software systems at runtime (already given in Deliverable 3.1.a). The next section sketches an approach to generalize the technique to dynamic feature evolution based on the ABS delta modules. The underlying investigation prepares Task 3.3.

3.1 Introduction

In long running (or eternal) systems, deployed software units often need to evolve over time, e.g., due to maintenance needs and bug fixes, changing user requirements, and changing environments. In systems with high availability requirements, the evolution of the systems must happen without disrupting regular functionality. In the object-oriented setting, it is natural to organize dynamic evolution of code in terms of object-oriented abstractions, i.e., methods, classes, and interfaces. An advantage of this approach is that the upgrade system becomes *modular*: all instances of the upgraded class will evolve by means of one upgrade operation. We illustrate the basic idea by the following example.

Motivating example. We adopt a separation of concerns between external service specifications, given as interfaces, and implementation code, organized in classes. Object pointers are typed by interfaces while objects are instances of classes. A type system is used to ensure that methods invoked on object pointers are supported by the objects. Consider a simple scenario with three classes $\mathtt{C}_1$, $\mathtt{C}_2$, and $\mathtt{C}_3$, where $\mathtt{C}_3$ inherits from $\mathtt{C}_2$ (the comment $V : n$ means version n of a class, whereas $U : m$ means upgrade m applied to that class):

```
class C1 {   //   V : 1, U : 0
  void run(){
    n(); run();
  }

  void n(){...}
}

class C2 {   //   V : 1, U : 0
}

class C3 extends C2 {   //   V : 1, U : 0
}
```

The example sketch is given in the Creol language [29, 30], a predecessor to ABS. In the example, *U:0* comments that a class has not (yet) been upgraded. Here, $\mathtt{C}_1$ objects are active as the `run` method is activated at object creation, with a nonterminating behavior consisting of repeated local calls to a method `n()`. The external functionality of each class is given by its interfaces. None are given here, so in this example only internal calls are possible in $\mathtt{C}_1$.

By *dynamically upgrading* the class C_2 with a new method m, this method will become available via objects of class C_2 and its subclass C_3. However, after the update the new method is only known internally in these classes. In order to *export* the new functionality, we dynamically add a new interface I providing a method m() with an appropriate signature, after which m() may be invoked on pointers typed by I. The interface I can be given as follows:

```
interface I {
  T m();
}
```

If we can type-check that C_3 implements I, it is type-safe to bind a pointer typed by I to an instance of C_3 and invoke the new method m() on this object. This may be achieved by dynamically redefining method n in class C_1 to create an appropriately typed instance of C_3 and invoke m() on this instance, for example by the code

```
I x; x:= new C3(); x.m();
```

These dynamic updates may be realized by four update messages added to the running system: introducing I, upgrading C_1 by the redefinition of n(), C_2 by a new method m(), and C_3 by the new interface I. After successful upgrades (*U:1*), the following classes replace the previous runtime class definitions:

```
class C1 {   //   V : 2, U : 1
 void run(){
  n(); run();
 }

 void n(){ I x; x := new C3(); x.m(); }
}

class C2 {   //   V : 2, U : 1
 void m(){...}
}

class C3 implements I extends C2 {    //   V : 3, U : 1
}
```

Furthermore, the active behavior of existing instances of C_1 now create instances of C_3 on which the new method m is invoked.

In the general setting of distributed systems with high availability requirements, the evolution of the running code should ideally not interfere intrusively with the normal operation of the system, but rather propagate local changes asynchronously. However, there may be dependencies between different upgrade operations, as illustrated by the example above. A type-safe introduction of these upgrades in such a distributed setting requires a combination of type checking and careful timing of upgrade operations at runtime. In particular, the redefinition of method n has an immediate effect on any instance of C_1. In order to avoid errors, this upgrade cannot be applied *before* C_3 implements the new interface I. However, the addition of the new interface requires the presence of method m(), which in turn requires that the application of the upgrade of C_2 has *already* occurred. In fact, C_3 has been upgraded twice, once directly and once indirectly through the upgrade of C_2. Our proposed mechanism of class updates formalizes an asynchronous runtime update mechanism which handles these dependencies, maintaining runtime type safety throughout the upgrade process.

Related work. A number of approaches to runtime evolution of software systems have been proposed. These approaches can be distinguished with respect to how they deal with the existing software units in the systems, which may be by

- keeping multiple co-existing versions of a class or schema [8, 7, 4, 22, 24, 26] or by
- applying a global update or "hot-swapping" operation to the system [46, 38, 2, 9].

The approaches also differ in how they address structures with *active behavior*, which may be

- disallowed [46, 38, 24, 9],
- delayed [2], or
- supported [50, 26].

For example, Hjálmtýsson and Gray [26] propose proxy classes and reference indirection for C++, retaining multiple versions of each class. Old instances are not upgraded, so their activity is not interrupted. Existing approaches for Java, using proxies [46] or modifying the Java virtual machine [38], use global upgrade and do not apply to active objects. Automatic upgrades by *lazy global update* have been proposed for distributed objects [2] and persistent object stores [9], in which instances of upgraded classes are upgraded, but inheritance and (nonterminating) active code are not addressed, limiting the effect and modularity of the class upgrade mechanism.

Formalizations of runtime upgrade mechanisms are less studied, but exist for both imperative [50], functional [7], and object-oriented [8] languages. In a recent upgrade system for (sequential) C [50], type-safe updates of type declarations and procedures may occur at annotated points identified by static analyses. However, the approach is synchronous as upgrades which cannot be applied immediately will fail. The object-oriented language UpgradeJ [8] uses an incremental type system in which class versions are only type-checked once. UpgradeJ is synchronous and uses explicit upgrade statements in programs (i.e., the programmer must accommodate for the upgrade during the development of previous versions of the program). Upgrades only affect the class hierarchy and future instances of the classes, but not the running objects. Multiple versions of a class will coexist and the programmer must explicitly refer to the different class versions in the code.

3.2 Dynamic Evolution of Features: The Approach

Above, we described a formal model for runtime evolution of classes in distributed object-oriented systems based on the concurrency model, synchronization constructs, and interface mechanism underlying the ABS modeling language. This model supports dynamic evolution in the form given by the example above. Since concurrent objects are typically persistent, it is essential that the upgrade system addresses existing objects and active behavior. However, in the distributed setting the upgrade system should not enforce a disruptive global update but rather *asynchronously evolve* the system. The described work was developed for the Creol language [29, 30], a predecessor to ABS which features class inheritance as a code structuring mechanism. In our model, runtime evolution of code is based on *dynamic classes*; i.e., the class definitions of the distributed object system can change at runtime. Such changes to a class affect both future and existing instances of the class and of its subclasses. The results have been published in [28].

Now that the full ABS language is available, we started to adapt our approach to the abstraction mechanisms introduced for feature variability and study how the dynamic replacement and modifications of features may be realized in terms of type-safe object-oriented upgrade mechanisms. This will mainly be done in Task 3.3. In particular, the runtime representation of deltas has been deliberately chosen to accomodate the technique for runtime evolution developed here, by maintaining the structure of the deltas at runtime. Thus, we intend to adapt the technique for runtime evolution by means of asynchronous class upgrades in order to support the runtime evolution of deltas in ABS. In particular, we plan to introduce a notion of *dynamic delta module* in order to perform runtime upgrades on a deployed product. A sketch of the intended approach is given below.

We also intend to study how a set of upgrade operations, as proposed in the present work, affects verified code. For this purpose, we consider adapting the approach of *lazy behavioral subtyping* [19, 20, 21] to the abstractions of the ABS language; i.e., classes, deltas, and runtime upgrade mechanisms.

Dynamic delta modules. In order to facilitate the modeling of temporal variability for high-level ABS models, it is crucial that evolution is expressed at the abstraction level of the modeling language. Therefore,

```
DynDeltaDecl ::= dyndelta DeltaDecl
DeltaDecl ::= TypeId [DeltaParamDecls]
                {ClassOrInterfaceModifier* }

DeltaModifier ::= adds DeltaDecl
                | modifies delta DeltaDecl

ClassOrInterfaceModifier ::= adds ClassDecl
                | adds InterfaceDecl
                | modifies class TypeId ImplementsModifier*
                    { Modifier* }
                | simplifies class TypeId
                    { Simplifier* }

ImplementsModifier ::= adds TypeId

Modifier ::= adds FieldDecl
           | adds MethDecl
           | modifies MethDecl

Simplifier ::= removes FieldDecl
             | removes MethDecl
```

Figure 3.1: Syntax for dynamic delta declarations in ABS. The *DeltaParamDecls* are as defined in Fig. 5.6 of HATS Deliverable D1.2.

temporal variability is captured by a *series of asynchronous changes* to the executing product, where each change addresses one of the structuring concepts of the modeling language and where the series of changes together bring about the desired overall modification of product behavior. In ABS the main structuring concept is the delta module, which contains modifier operations for the fields, method declarations, and interfaces of classes (see the grammar of the Delta Modules, given in Fig. 5.6 of HATS Deliverable D1.2). Temporal variability may be expressed in terms of *dynamic delta modules*. Remark that dynamic delta modules may also introduce new class and interface declarations into the executing product.

Syntax. A *dynamic delta module* in ABS is a set of changes to a product, given in terms of its delta modules, classes, and interfaces. The grammar for dynamic delta modules is given in Figure 3.1. Dynamic deltas allow us to add new class and interface declarations to the deployed product, and also to change existing class declarations in the following ways:

- add new fields and method definitions
- modify existing method definitions
- simplify the class by removing redundant field and method declarations

Thus, the dynamic delta modules are slightly more restrictive than those used for spatial variability. In particular, classes and interfaces cannot be removed, and the modifiers are distinguished from the simplifiers (in standard delta modules as defined in Deliverable D1.2, the simplifiers are in the same syntactic category as the modifiers).

Example. We refer to the *MultiLingualHelloWorld* product given in Example 5.2.2 of Deliverable D1.2. Let us assume that the product family has been deployed, and that we want to modify products to make a more personal greeting in English. The dynamic delta module *ExtendedGreeting* applies to products with the *English* feature. A new method is added to the *Greeter* class, and the delta module *Rpt* is modified to use the new method in its modification of the `say_hello` method.

```
dyndelta ExtendedGreeting [ hasFeature English ] {
  modifies class Greeter { // Adds a new method to the class Greeter
    adds String i_am_bob () { return ", I am Bob!"; }
    }

  modifies delta Rpt (Int times) { // Uses the new method in the loop
     modifies class Greeter {
         modifies String say_hello() {
           String result = " "; Int i = 0;
              while (i < times) {
                 result = result + original() + this.i_am_bob();
                 i = i + 1;
              }
              return result;
         }
     }
}
```

Restrictions on dynamic delta modules. The rationale for the syntactic restrictions introduced in the dynamic delta modules is to guarantee that the temporal evolution of the product's code does not give rise to runtime errors. This is done by a static analysis which identifies runtime applicability conditions for the different elements of the dynamic delta module. In order to apply a change to a class, other changes may be required to have taken place already. In the asynchronous setting of ABS, this cannot be statically controlled. For example, if new code in a change to a class D invokes a method on an instance of another class C, that method must be available. If the method is introduced as a change to C, the static analysis of the change to D will generate an applicability condition to require that this change to C has already occurred at runtime. The distinction between modifiers and simplifiers correspond to the distinction between runtime applicability conditions at the level of classes and at the level of instances of those classes [28]. The removal of classes, as well as the removal and modification of interfaces, are disallowed at runtime because they would require a full state space inspection to inspect all references of the involved interfaces (including references in messages), which is a very heavy operation in the asynchronous distributed setting targeted by ABS.

3.3 Results and Outlook

UIO has worked out the details of dynamic class updates and generalized this technique to an approach to dynamic feature evolution in the ABS framework. The plan is to further pursue this approach as part of Task 3.3.

Chapter 4

Component Model with Evolvability Constructs

4.1 Introduction

Evolution is an important issue in complex software systems. The needs and requirements on a system may change over time. This may happen because the original specification was incomplete or ambiguous, or because new needs arise that had not been predicted at design time. As designing and deploying a system is costly, it is important that the system is capable of adapting itself to changes in the surrounding environment.

By dynamic evolution of a system we refer to the possibility that the functionalities offered by the system may change over time. This may involve reconfiguring and updating applications to meet new requirements and new operating conditions, which were unexpected when the application has been developed and deployed. An effort has been initiated during the first year of the project by BOL with the goal of isolating interesting constructs for expressing dynamic evolvability. The goal was to identify constructs that offer enough expressiveness to model common patterns of evolvability for systems, and at the same time are mathematically robust so to allow formal analysis of desired behavioral properties of the system. For this, two directions of work had been pursued. In the first direction, we defined a core calculus of evolvable components, following the process calculus tradition. In the second direction, a new, language-independent, approach was studied based on the combination of adaptation hooks provided by the adaptable application, specifying where adaptation can happen, and adaptation rules external to the application, specifying when and how adaptation can be performed.

During the second project year we started to study the integration of these two directions with ABS. For this, we have revised the constructs and the techniques that we studied in isolation in the first year with the goal of finding the most suitable way of combining them with the core constructs of ABS. We outline here the technical work done, much of which represents ongoing work, and which will be exposed in more detail in this and the following chapter.

4.2 Dynamically Evolvable Components

During the first project year we formalized a calculus of evolvable components, called MECo (Model of Evolvable Components) [41]. We recall the main ingredients in MECo. The key construct in MECo is that of a *component*. MECo components have similarities with the *locations* studied in distributed process calculi (cf. EU project Sensoria). In contrast to locations, however, MECo components have an interface, comprising both input and output ports. Input ports show the functionalities that the components make available to its environment; output ports, in contrast, collect the dependencies of the components to its environment, that is, what the components need in the environment to work properly. The other important features of MECo are: a hierarchical structure of components; the possibility of stopping and capturing

components; a mechanism of channel interactions, orthogonal to the activity of components, with tunneling effects that bypass the component hierarchy. Interactions along channels may be triggered when a method in the input interface of a component is invoked. Channels can be used to implement sessions of interactions between components. The possibility of stopping a component is used to capture the current state of a component and possibly send it around; this is essential in MECo to modify the hierarchical structure of a system and model non-trivial forms of evolution.

The activity of components is local: when the body of the method of a component is executed, calls may only be issued to inner components or to components that are reachable via output port bindings. In particular, the environment surrounding a component c_1 may call c_1 but not components internal to c_1. Such components are only reachable if some input port of c_1 forwards messages to them.

For the integration of MECo with ABS, three main concerns have arisen. The first is about the presence of output ports. In contrast to input ports, which are very similar to the methods of an ABS object, output ports have no counterpart in objects. In MECo output ports appeared to be useful to be able to statically determine all dependencies and acquaintances of a component. However, they create complications in the semantics and, most importantly, they create complications in the integration with ABS, as their presence would create a striking contrast with the ABS computational units—the objects.

The second concern has to do with the scope and component visibility. In MECo, the hierarchical structure of a component system is rigid: the boundary of a component a hides all the inner components, which are unreachable from a's environment. This makes the model elegant and simplifies its presentation. However, it also makes it hard to model common patterns of distributed systems that involve sharing of resources.

Our third concern is about the stopping construct in MECo. This construct allows us, in MECo, to freeze a component and capture it; the component can then be sent around, for instance, with the possibility of duplicating it at will. As a consequence, MECo, is a higher-order calculus (as computational units such as components are first-order values), and it is well-known that the theory of higher-order calculi is complex. Also, the practical relevance of the act of copying a running component is dubious.

The language that we are currently designing, referred to in this document as Comp, addresses the three concerns above as follows:

- Components have an input interface, but no output interfaces; as a consequence, components can be used much in the same way as objects; indeed objects may be viewed as a special case of components that do not have, in particular, mobility capacities.
- We have added the possibility of opening and closing the scope of a component. Thus, while by default communication in Comp remains global to fit the communication capacities of objects, if needed, the boundary of a component c_1 can be closed to restrict access to specific components internal to c_1, say c_2; as a consequence, a component external to c_2 will not be able to directly access c_2 (provided that c_2 is known, of course).
- Mobility of components is allowed by means of movement primitives rather than by communication of components. These movements are inspired by the constructs for achieving mobility in the Ambient calculus. Thus in Comp a component may move in the tree structure of a component system. Components however are not first-order values; they may not be communicated, and may not be copied.

4.3 Component Model

Components can be useful for deployment, adaptation and evolution. A component model provides facilities for structuring a software system. Thus a component can be a containment for terms (objects, other components) that, for instance: share a physical location; share some computational resources; are within a given security perimeter; jointly implement some functionality. As adaptation/evolution are major concerns for us, a component model should also provide facilities for dynamically changing a structure so to match

the needs of adaptation/evolution (which are intended to includ needs of mobility). In this section, we describe in detail Comp, the component model that we developed for integration with ABS and show how to exploit it for simple evolution patterns. Note that at the moment ABS does not provide tools to this aim, since objects have a flat structure, apart from COGs, which are, however, only used for concurrency issues. The model that we present is inspired by MECo, adapted to facilitate the integration with the object model of ABS, as discussed in this chapter. As will be clear from the description of the language, Comp at the moment is defined in the process calculus style, but main design choices have been taken with the integration with ABS objects in mind. We stick to the process calculi style since process calculi are a convenient tool to experiment with different primitives and semantic solutions and compare them in a formal framework. The main idea for a full integration with ABS is that components are ABS objects extended with additional features, such as hierarchical structure and possibility of dynamic reconfiguration.

The challenge in the formalization of a component model is to isolate key aspects of component-based systems and reflect these into specific constructs. The success of the model will be measured by its ability to specify in a simple way adaptation and evolution patterns that will be needed during the lifetime of the system, also including needs for evolution that may emerge after deployment. A few small examples will be presented in this same chapter, while a more high-level approach will be described in Chapter 5. While developing the model we followed a general strategy called "the architect principle", which basically means that each component in the system is in charge of (and is allowed to) managing its children. We choose to follow this principle in our model because it improves its consistency, making it more easy to work with. Indeed, because of this principle, the one responsible for a modification is uniquely identified, which makes the behavior of programs clearer, and the modifications easier to code. Moreover, as the parent has the control over its children and their communications, it can help their integration into the rest of the system. We describe the main ingredients of our model.

Components. Components are a way to structure a program into a set of units, each of them having a clear boundary. Components inside the same boundary share some features, which can be of various type. For instance, they may share some computational resources, they may be in the same physical location or just inside the same security perimeter, or they may jointly implement some functionalities. In particular, here each component has its own data and its own execution space. In other words, components are in this chapter the unit of evolution. More specifically, evolution is obtained by adding, removing, replacing, wrapping or changing in another way the component structure. Notably, while a component is being, e.g., replaced, other components can continue to execute normally, thus minimizing service disruption.

Components give rise to a hierarchical structure, which allows for modular definition of complex software architectures. Nesting of components may have different meanings, corresponding to the meanings of components themselves. In our case the parent of a component is in charge of updating its children, according to the so-called "architect principle".

Another use for components inside HATS is for deployment. Components in fact may represent physical locations and available resources, and deployment can be done by specifying how to associate to each object its enclosing components, thus defining how to deploy it (see also Deliverable D1.2 "Full ABS Modelling Framework" [17, Chapter 6]). Notably, evolution constructs discussed later allow for dynamic redeployment of components to answer modifications of the underlying architecture. We will not consider this possibility in depth.

Methods. Each component is equipped with a set of methods, defining the functionalities it makes available. Having an explicit *input interface* is important, since this provides an abstract description of the functionalities of the component, to be used to ensure correctness of evolution steps. For instance, if an evolution step preserves the interface of the involved components, it will introduce no typing errors on runtime invocations. Also, having methods in a component matches the intuition that components are objects with extended capabilities, as needed for having them as a smooth extension of ABS. Note that we have no output interface, in contrast with most of the component models in the literature, since this has no

direct correspondence in objects.Moreover, the presence of output interfaces would make the semantics of the isolation mechanism more complex.

Isolation. A main effect of a component hierarchy is that the enclosing component may hide its internal components, or may make (some of) them available to the external world. Hiding is fundamental for encapsulation: hidden components cannot be reached directly from the outside. Also, isolation can be used to encode wrapping, where the parent component hides its children while providing updated functionalities. In this way, for instance, methods can be easily removed, added or redefined. On the other hand, having the possibility of making inner components visible is fundamental for defining shared resources.

According to the "architect principle" the parent of a component can decide when and whether making it available to the outside.

Mobility. Having a hierarchical structure in place, the immediate way for achieving evolution is to allow components to move along the hierarchy. Remember that this mobility can be logical or physical, according to whether components model the software architecture of the system or its physical distribution.

Clearly, different forms of mobility are possible. We decided to introduce two primitives for mobility, _ **in** _ and _ **out** _, inspired from the ambient calculus [12], that move a component inside or outside another one. The choice has been motivated by the desire to disallow direct mobility between far locations (unrealistic in many cases, e.g., for physical locations). Also, again following the architect principle, the parent is the one able to move its children.

The remaining chapter is structured as follows. We first present in Section 4.4 the calculus and explain the syntax. We illustrate this syntax with a number of examples in Section 4.5 that show in particular how various patterns of evolvability of components are captured.

4.4 Formalization

Figure 4.1 presents the syntax of Comp. The main construct of Comp is the component $a(S)\{M\}[P]$, where (i) a is the name of the component; (ii) S is the set of names of inner components that are hidden from the rest of the architecture; (iii) M is the set of the methods of the component; and (iv) P is its body, containing its currently running code and its sub-components. Note that we do not use classes to describe the fields and the methods of our components: this information is directly provided by the components themselves. We preferred this approach rather than to add classes because it simplifies the component model without changing the behavioral aspects we concentrate on here.[1]

$$
\begin{array}{rcll}
P & ::= & a(S)\{M\}[P] \mid P \mid P \mid A.P \mid 0 \mid x \mid \nu a P & \text{Process} \\
A & ::= & a\_m\langle J\rangle \mid \textbf{open}\ S \mid \textbf{close}\ S \mid a\ \textbf{in}\ b \mid a\ \textbf{out}\ b & \text{Action} \\
 & \mid & a(x) \mid a\langle J\rangle & \\
M & ::= & 0 \mid m(x).P \mid M \mid M & \text{Methods}
\end{array}
$$

Figure 4.1: Calculus Syntax

Methods are defined as usual: $m(x).P$ declares a method m with the formal parameter x and the code P. The code P is defined as a sequential and parallel composition of different actions. The main actions in

[1] Obviously, the integration of this component model into ABS will require the definition of component classes to fit with the object model, but these classes are not relevant for the description of the component's behavior.

our calculus are: (i) method call $a\_m\langle J\rangle$ that calls the method m of the component a with the parameter J (here J is either some code P that can be used for code injection, or a component name b); (ii) **close** S hides the child-components in the set S from the rest of the system; (iii) **open** S on the opposite reveals the components in S to the environment; (iv) a **in** b puts the component a inside the component b while (v) a **out** b takes the component a that is inside b, and puts it outside as shown in Figure 4.2. These different actions, plus the creation of new components (the command $a(S)\{M\}[P]$ creates a new component a), constitute the basis of adaptation and evolvability in our model.

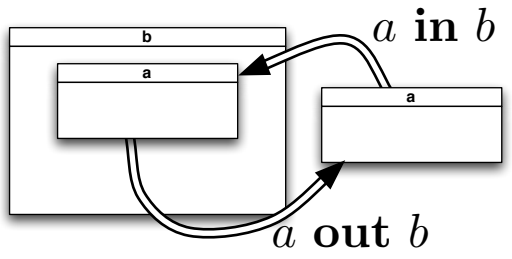

Figure 4.2: In and Out of a Component.

Finally, our calculus contains several other constructs to deal with less central concepts: (i) 0 corresponds to the **skip** command; (ii) x corresponds to the use of the formal parameter x in the definition of a method; (iii) νaP creates a new name a, usually used to create components with new names (we sometimes let ν bind sequences of names, to avoid repeating the ν for each name); and (iv) $a\langle J\rangle$ and $a(x)$ are communication actions on channels that encode the **return** statement and the use of futures. We preferred to use channels in this model rather than return statements and futures since while channel communication and futures are quite close semantically, the semantics of channels is a little simpler and far more common in the field of process calculi. Obviously, in the process of integrating this component model into ABS, channels will be replaced by the already defined future mechanism of ABS. But for simplicity reasons, we use channels here. Note that in the following examples we use several time channels, but always in a return/future style.

4.5 Examples

Adding and removing a component. The two most basic operations to manipulate a program structure, in addition to the **in** and **out** operators, are the addition and removal of components. The addition of a component is extremely easy to do in our calculus as it is equivalent to the creation of a new component:

$$\mathbf{Add}(a, S, M, P) \triangleq a(S)\{M\}[P]$$

The encoding of component removal is a little more subtle. Indeed, instead of destroying the component, we simply hide it with the insurance that it will never be accessible again:

$$\mathbf{Remove}(a, P) \triangleq \nu b.(b(a)\{0\}[0] \mid a \textbf{ in } b.P)$$

Here, we create a new component b that no one knows (and thus no one can modify it) and put the component a inside. After this, we execute the process P in parameter which is supposed to notify the environment that a is not available anymore. Note that b hides a from the rest of the architecture: it cannot be accessed ever again. Hence, as soon as the computation inside a finishes, it becomes behaviorally equivalent to 0 and can be safely removed by a garbage collector.

Our encoding of component removal does not directly destroy the component by design. Indeed, suppose we want to remove the component a to replace it with a newer version. The current version of a can be computing some result of some method calls. Directly destroying the component a would create some consistency issues inside the program as some other computations are waiting for the results being computed

by a to continue. Our encoding ensures that we can replace a with another component without creating any inconsistencies.

Hiding structure manipulation. In practice often the part of the program to be modified is isolated prior to its manipulation to prevent possible inconsistencies. In our case, inconsistencies can emerge because two components can have the same name: suppose we want to create a component b and put a component a inside it. The natural approach is to first create the component b and then move a, but, because after the creation of b there may be two different components named b, we cannot ensure that a is moved inside the correct one. A simple way to solve this problem is to create a temporary fresh component where we put merely the component part of the restructuring. Then, after the modification process finishes, we can put the result back in its place.

We propose here a simple encoding of this isolation process, where we hide the component a, apply the modifications P on it which results in the component b that needs to be put back in the environment. Finally, we execute P' that is supposed to make the environment aware of the modifications.

$$\mathbf{Hide}(a, b, \lambda ch.P, P') \triangleq \nu c, ack(c(a, b)\{0\}[P\{^{ack}/_{ch}\}] \mid a \textbf{ in } c.ack(x).b \textbf{ out } c.P')$$

Note that in our encoding, the process P in-parameter is annotated with λch: here, ch is a channel that is supposed to say when P is finished, so we know when it is possible to put b in the environment.

Renaming a component. We can encode the operation of renaming of a component provided that we know the interface of the component to be renamed. Suppose, for instance, we want to rename a into b: the idea is to create a new component with the name b, with the restriction $S = a$ and the same methods as a that forward the calls to a; then we put a inside b which basically removes a from the environment: only b is now available and acts like a. We thus have replaced a with b. In the following encoding, we use: (i) $m_1, \ldots, m_n$ that are supposed to be the methods of a; and (ii) a process P that is supposed to make the environment aware of the name modification. Let us remark that we use the isolation command **Hide** to ensure that there will be no inconsistencies.

$$\begin{aligned}&\mathbf{Rename}(a, b, P)\\ &\quad\triangleq \mathbf{Hide}(a, b, \lambda ack.\left(b(a)\left\{\begin{array}{c} m_1(x).a\_m_1\langle x\rangle \\ \ldots \\ m_n(x).a\_m_n\langle x\rangle \end{array}\right\}[0] \mid a \textbf{ in } b.ack\langle 0\rangle.0\right), P)\end{aligned}$$

Finally, let us also note that in this renaming function, we suppose that b is different from a, otherwise, inside the component c, we would not know whether we are moving the old component into the new one or vice versa.

Wrapping. Wrapping is a useful feature in evolvable architectures. The principle of this operation is to replace an old component (named for instance a) with another component (let it be named b) that needs the code of a to work. A very simple encoding of this wrapping operation is to modify a little bit the **Rename** operation so to make the code of the component b parametric:

$$\begin{aligned}&\mathbf{Wrapping}(a, b(S)\{M\}[P], P')\\ &\quad\triangleq \mathbf{Hide}(a, b, \lambda ack.(b(S, a)\{M\}[P] \mid a \textbf{ in } b.ack\langle 0\rangle.0), P')\end{aligned}$$

With this encoding, we wrap a into b and put in parallel the process P' that is supposed to make the environment aware of the wrapping.

As for the renaming operation, we suppose that we wrap a in a component with a different name. But it could be nice to wrap a inside a component with the same name too (so components using the component a would not notice the update).

However, the previous encoding needs to be refined to deal with this possibility. In fact, the code of the new component must be able to differentiate the old a component from itself, while neither the environment nor the old a should see any difference after the wrapping. We can solve this problem by combining renaming and the previous wrapping example. In this encoding, the process P of the new version of a is supposed to reference the old version with a name b that is specified in a lambda binder:

$$\begin{aligned}&\mathbf{Wrap}(\lambda b.(a(S)\{M\}[P]), P')\\ &\quad\triangleq \nu b, i(\mathbf{Hide}(a, a, \lambda ack.\left(\begin{array}{c}\mathbf{Rename}(a, b, i\langle 0\rangle.0)\\ i(x).\mathbf{Wrapping}(b, a(S)\{M\}[P], ack\langle 0\rangle.0)\end{array}\right), P'))\end{aligned}$$

In this encoding, we use the channel i to synchronize the renaming of the component a with the actual wrapping process.

Update. The update consists of adding some new features to existing components, either by means of new components, new processes, or new methods. In our current calculus, it is not possible to directly add methods to components. The only way to encode it here is wrapping. Adding code and components instead can be done in our calculus using higher order communications. Hence it requires from the updated component to implement some sort of protocol to ensure the validity of the update and to install it. Typically, if we want to add the process P into the component a, we can do:

$$\mathbf{Update}(a, P) \triangleq a\_m\langle P\rangle \quad \text{with the component } a \text{ being} \quad a(S)\{m(x).x\}[P']$$

Replacement. A very simple way to encode replacement is to remove the original component and create a new one with the same name, like in the following:

$$\mathbf{Replace}(a, S, M, P) \triangleq \nu ack(\mathbf{Remove}(a, ack\langle 0\rangle.0) \mid ack(x).a(S)\{M\}[P])$$

But in some cases we want to do more than just replace a component with a new version: we also want to transfer the state of the original component to the new one. To be able to do it, we suppose we have a method "state" that sends on the channel c the state of the original component and a method "up" that updates the new component with the state of the previous one. With this assumption, we can present our encoding:

$$\begin{aligned}&\mathbf{Replace}(a, S, M, P)\\ &\quad\triangleq \nu(b_1, b_2, b_3, c).\left(\begin{array}{l}b_1(a)\{0\}[a\_\text{state}\langle c\rangle.0]\\ \\ |a \textbf{ in } b.b_2(a)\{0\}\left[\begin{array}{l}b_3(\emptyset)\{0\}[a(S)\{M\}[P]]\\ |c(x).a\_\text{up}\langle x\rangle.a \textbf{ out } b_3.0\end{array}\right]\\ \\ |a \textbf{ out } b_2.0\end{array}\right)\end{aligned}$$

We need one component in this encoding to isolate the original a and two others to ensure that the new version of a is available to other components only when its state has been updated. Other encodings (using an acknowledgment, for instance) are possible.

Links. In many other component models [3, 10, 10, 54], components come with input ports, output ports and bindings connecting them. In our model, methods can be considered as input ports, and we choose not to include output ports and bindings in our calculus because their interaction with evolvability raises some consistency issues.

Nonetheless, output ports and bindings can be encoded in our model if programmers want to use them. The output ports of a component "a" correspond to the dependencies the component needs to work properly. Typically, output ports are in our model method calls to other components or methods. Bindings are a way to satisfy the dependencies a component "a" has by stating which input ports of an existing component

will answer the calls on each output port of "a". We encode bindings between the output port $p.m$ and the input port $p'.m'$ with a process acting as a forwarder

$$b(\emptyset)\{0\}[p(\emptyset)\{m(x).p'\_m'\langle x\rangle\}[0]]$$

This encoding exploits a container component, named here "b". We add this component to be able to name the bindings we create, so we can manipulate them. In particular, we want to be able, like in the other models, to delete them. Finally, we present our encoding for the creation and the deletion of a binding:

$$\begin{array}{ll}\textbf{Connect}(p, p', \lambda x.P) & \textbf{DisConnect}(b, P)\\ \quad \triangleq \nu b(b(\emptyset)\{0\}[p(\emptyset)\{m(x).p'\_m\langle x\rangle\}[0]] \mid P\{^b/_x\}) & \quad \triangleq \textbf{Remove}(b, P)\end{array}$$

In the function **Connect**, we use the continuation P to make the environment aware of the new binding "b".

Distribution. As already said, components can be used to model also physical distribution, e.g., by having each site being a different component inside a toplevel one representing the network. The network component may provide methods implementing different communication and/or reconfiguration protocols.

Here is, for instance, a simple configuration, with two sites $\texttt{Site}_1$ and $\texttt{Site}_2$, running programs P_1 and P_2, respectively. The global component `Net` acts like an interconnecting asynchronous network, relaying messages from one site to another: we suppose that $\texttt{Site}_1$ (resp., $\texttt{Site}_2$) listens for input messages with the method in and can be reached through the network using the method call $\texttt{Net}\_toS_1\langle x\rangle$ (resp., $\texttt{Net}\_toS_2\langle x\rangle$).

$$\begin{array}{l}\texttt{Network}(\emptyset)\{0\}[\\ \quad \texttt{Site}_1(All \setminus \{\texttt{Net}\})\{in(x).P_1'\}[P_1]\\ \quad \texttt{Site}_2(All \setminus \{\texttt{Net}\})\{in(x).P_2'\}[P_2]\\ \quad \texttt{Net}(\emptyset)\{toS_1(x).\texttt{Site}_1\_in\langle x\rangle \mid toS_2(x).\texttt{Site}_2\_in\langle x\rangle\}[0]\\ ]\end{array}$$

Note that in this example, we use the set of names $All \setminus \{\texttt{Net}\}$ to specify that both sites $\texttt{Site}_1$ and $\texttt{Site}_2$ can only communicate with the component `Net` which encodes the network.

Deployment at runtime. It is possible to extend the previous example to model redeployment at runtime. Indeed, suppose we have a running architecture, and that we want to move a component to another location, for instance for efficiency issues. We can encode such an operation by adding to the component `Network` some redeployment capacity in the form of a method d:

$$\texttt{Network}(\emptyset)\{d(c, l_1, l_2).c \textbf{ out } l_1.c \textbf{ in } l_2.0\}[\dots]$$

This method takes three parameters, c which is the component to re-deploy, l_1 being its current location and l_2 being its destination. With this, redeployment can easily be achieved by calling the method `Network_d` with the proper parameters.

4.6 Related Work

Components have been introduced as a new programming paradigm in the mid-nineties, as a means to solve several limitations in the object models [54]. Typically, one of the main limitations of objects that can be solved by component models is the lack of high level operators for adaptation/evolution, as it is motivated in [44, 45]. Nowadays, there exist many component models, distinguished by their definition of the structure of component and by the operations provided on them. For instance, the OSGi component model [3] developed by IBM defines its components as a set of objects and classes with some extra information, like which services the component provides, and on which services it depends on. This model thus allows the

addition of components at runtime, with a constraint solver that checks and solves the dependencies of the added component. Let us note that components in OSGi can only be assembled in a flat structure, while in Fractal [10], developed at INRIA and France-Telecom, they can be assembled into a tree structure. But in contrast to OSGi, in Fractal, the programmer must explicitly specify how the dependencies are solved by the use of *bindings*, similar to the links presented in our examples.

We rejected these component models, as well as many other, like COM [15], Java Beans [53], Appia [39], Click [43], Coyote [6] or Dream [34], because their integration with ABS would have been heavy complex; also we wanted a component model which clearly adresses evolvability. Further, these models are often informally defined or even have an ambiguous semantics. Moreover, even if most of these models are implemented on an object-oriented language, none of them describes the interaction between components and objects, thus raising more ambiguities and inconsistencies. For instance, the Fractal model states that communication between components can only occur by using the input and output ports.But because of its implementation in Java, it is possible to use the objects and their methods to make components communicate without an explicit use of ports.

A few models, defined in the process calculus style, bear resemblances to components, in that they allow the isolation and possibly the nesting of processes inside 'boxes'. We have for instance the M-calculus [47] and the kell-calculus [48] that are inspired from Fractal. Other calculi, like the different flavors of the Ambient calculus [12, 55, 36, 11], the join-calculus [23] or the seal calculus [13] use named boxes as a means to structure the program into a tree hierarchy. Moreover, some of these calculi use this structure to control communication and to enable adaptation through the modification of the program structure. Finally, in Oz/K [37], the authors propose a core programming language with components as a main feature. These component models, being formally defined, are a better fit for ABS than the previous ones, but still have several limitations. First, only Oz/K integrates objects in its model. The others do not provide any description of how components and objects interact. Moreover, Oz/K has quite a complicated communication pattern, and deals with adaptation via the use of *passivation*, which, as suggested by [35], is still a too high-level operation to hope for any tool to help proving behavioral properties.

4.7 Results and Outlook

For dynamic evolution on a coarse-grained and abstract level, BOL in collaboration with UKL developed a component model providing operations that support component updates in a hierarchically modeled system. In comparison to the light-weight component model realized as Concurrent Object Groups in ABS, this more heavy-weight component model allows formulating and analyzing dynamic updates of components in a systematic way. According to the explorative nature of this task, the work investigated the fundamental issue needed to support such a component model. The gained knowledge and expertise will be exploited in the follow-up tasks on evolution (Tasks 3.3 and 3.5).

Chapter 5

Rule-Based Dynamic Evolution

5.1 Introduction

In this chapter we consider dynamic evolution of products. More specifically, we assume to have a running ABS system and we want to make it evolve so to provide better functionalities and/or performances to the user, without causing service disruption and without the exact form of evolution being known when the application has been developed, deployed, or even started.

This work exploits the basic mechanisms for evolution based on components described in Chapter 4 and builds a framework based on dynamically created evolution rules following the principles described in Deliverable D3.1.a "First Report on Evolvable Systems" [16, Section 3.3]. The approach was applied therein to the service oriented language Jolie [40], since services are a suitable unit of evolution (i.e., a system can evolve by replacing an old service with a new one) and come equipped with mechanisms for runtime update. In the meantime, the component model Comp for the ABS language has been created. Components, like services, are a good unit of evolution, and can be dynamically replaced.

5.2 Results

The main idea underlying our rule-based approach is to minimize the amount of information about evolution that should be available at design time, so to make evolution able to proceed in unexpected directions. However, some information is needed at the very beginning, since it is very difficult to make an application evolve if the application itself does not provide some "evolution hooks".

The approach is based on the interaction between an *evolvable application* and an *evolution manager*. An evolvable application is for us an application that provides a basic support for evolution, but has no information on the particular evolution needed. An evolution manager instead is an application that manages a set of *evolution rules* defining specific evolution updates to be applied to other applications. The evolution rules can be created dynamically, and they can be applied to evolvable applications that were developed and started before the creation of the rules themselves. The application of each such rule may result in replacing a component of the system with an improved version. To this end one such rule should include information on which component should be replaced, under which conditions the replacement should happen, and must include a new version of the component itself. We discuss the three aspects below. The presented approach is inspired by the one described in Deliverable D3.1.a [16] and in [33]. For this approach to be applicable in practice, the evolvable application and the evolution manager should be able to interact. We assume here that each of them is able to invoke some of the methods of the other one. At the implementation level this can be obtained with various technologies, such as Java RMI. In this deliverable we will not go into the implementation details, staying at a more abstract level.

In order to find out which component may be replaced (and to verify the conditions ensuring that replacement is actually needed) the evolution manager should interact with a candidate component itself. To this end the evolvable application should have a dedicated method to provide a list of references to its inner

components that may be updated. When trying to apply a rule, the evolution manager will interact with candidate components and try to match the description of the component itself with a description included in the rule of which components it can be applied to. This allows one to identify to which components the rule can be applied.

If one such a component A is identified, then the evolution manager has to check whether replacing the component A with its version A' included in the rule itself will be an improvement or not. To this end each component comes with a method describing its set of functional and non-functional properties. We assume that each property has a name and a value (possibly computed dynamically, for instance, monitoring the component itself or as a function of the properties of the inner components). Typical non-functional properties are ExecutionTime, PricePerRequest, etc. Functional properties can be, for instance, described by SupportedFeatures, CodeVersion, etc. The component A should also provide a method allowing to compare two sets of properties according to user preferences. For instance, some users may prefer a cheap component over a more efficient one, while other users may decide the other way around. Thus the sets of properties of the component A in the application and A' in the evolution rule are compared: if the one in the evolution rule is better then replacement should be done.

Following the "architect principle" only the parent of the component can replace it. For this reason we decide to move the whole evolution interface to the parent who will then forward the messages to/from its children. Thus, assume that a component $Travel$ is the container component for an adaptable application aiming at organizing travels for its user (e.g., booking trains and buses, telling the user to go to the right platform,...). This application includes a component $BookTrain$ for booking trains. Assume that a new high-speed train becomes available, requiring a new booking protocol. This will be managed by adding to the evolution manager (part, e.g., of the train station information system) a new rule specifying that all the components for booking trains should be updated to support the new protocol. The evolution manager will interact with the $Travel$ component, asking for the list of its children and of their descriptions by invoking a suitable method. Looking at the list, the evolution manager will find out that the component $BookTrain$ may require adaptation. Then it will invoke other methods to check whether the evolution is desired (the user may not want high-speed trains because of their price). Finally, if the evolution rule applies, it will send the updated component $BookTrain'$ to the $Travel$ application. The $Travel$ application finally replaces the component $BookTrain$ with the new component $BookTrain'$ using the replacement pattern described in Chapter 4.

5.3 Related Work

The analysis of evolution/adaptation approaches based on rules in the literature has been performed in Deliverable D3.1.a [16]. Interestingly, all the approaches described therein were based on the service oriented paradigm. We are not aware of similar work in an object-oriented setting. However object-oriented evolution has been considered in Chapter 2, and we refer to Chapter 2 for comparison with the related literature. The main aim of the work described there is, however, to ensure efficient update and consistency of different object versions, while here we concentrate on when and where evolution is useful and/or desired. Thus, the work reported in Chapter 2 can be used as underlying technology to apply the evolution patterns required by our rules.

5.4 Outlook

The work above shows how to exploit the rule-based approach to evolution in the context of ABS components, exploiting some of its basic features such as higher-order communication and component replacement. The approach could also be applied directly to objects instead of components. One could do it by exploiting the techniques for dynamic evolution of classes and features presented in Chapter 3. Thus the rule-based approach can be seen as a high-level approach that can be based on different low-level evolution mechanisms, with the added value of ensuring that evolution is performed only if it is useful according to user needs.

Chapter 6

Conclusion

In this task, we have investigated different forms of evolution and related it to the model-based approach of HATS. In particular, we looked at the following three questions set out by the Description of Work and analyzed what the HATS approach offers to solve them:

1. How can evolution be constrained in a way such that overall consistency properties can be derived and maintained during evolution?
2. What does correctness mean for evolution steps that are intentionally changing the behavior of individual components?
3. What are the fundamental dynamic mechanisms needed to model system evolution at runtime?

In the second phase of this task, we further developed

- a technique to reason about static evolution steps on the class and module level (Chapter 2),
- an approach to transfer techniques for dynamic class updates to dynamic feature updates (Chapter 3),
- a calculus for the systematic analysis of dynamic system evolution based on a hierarchical component model (Chapter 4) and
- a rule-based approach to control self-adaptation and self-evolution in response to changes of the system environment (Chapter 5).

In addition to the results described in the chapters above, the task clarified the notion of evolution in general and its role in the HATS approach in particular. The basic research results of this task will be further integrated in the HATS framework and tools in several upcoming tasks (cf. outlook section of previous chapters).

Bibliography

[1] Erika Abraham, Marcello M. Bonsangue, Frank S. de Boer, and Martin Steffen. Object connectivity and full abstraction for a concurrent calculus of classes. In *Theoretical Aspects of Computing – ICTAC 2004: First International Colloquium, LNCS*, volume 1, 2004.

[2] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Modular software upgrades for distributed systems. In Dave Thomas, editor, *Proc. 20th European Conf. on Object-Oriented Programming (ECOOP'06)*, volume 4067 of *LNCS*, pages 452–476. Springer, 2006.

[3] OSGi Alliance. *Osgi Service Platform, Release 3*. IOS Press, Inc., 2003.

[4] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[5] Anindya Banerjee and David A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *JACM: Journal of the ACM*, 52, 2005.

[6] Nina T. Bhatti, Matti A. Hiltunen, Richard D. Schlichting, and Wanda Chiu. Coyote: A system for constructing fine-grain configurable communication services. *ACM Trans. Comput. Syst.*, 16(4), 1998.

[7] Gavin Bierman, Michael Hicks, Peter Sewell, and Gareth Stoyle. Formalizing dynamic software updating. In *Proc. 2nd Intl. Workshop on Unanticipated Software Evolution (USE)*, April 2003.

[8] Gavin Bierman, Matthew Parkinson, and James Noble. UpgradeJ: Incremental typechecking for class upgrades. In *Proc. 22nd European Conf. on Object-Oriented Programming (ECOOP'08)*, volume 5142 of *LNCS*, pages 235–259. Springer, 2008.

[9] Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, and Steven Richman. Lazy modular upgrades in persistent object stores. In Ron Crocker and Guy L. Steele Jr., editors, *Proc. Object-Oriented Programming Systems, Languages and Applications (OOPSLA'03)*, pages 403–417. ACM Press, 2003.

[10] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quema, and Jean-Bernard Stefani. The Fractal Component Model and its Support in Java. *Software - Practice and Experience*, 36(11-12), 2006.

[11] Michele Bugliesi, Giuseppe Castagna, and Silvia Crafa. Access control for mobile agents: the calculus of boxed ambients. *ACM. Trans. Prog. Languages and Systems, vol. 26, no 1*, 2004.

[12] Lucas Cardelli and Andrew D. Gordon. Mobile Ambients. *Theoretical Computer Science, vol. 240, no 1*, 2000.

[13] Giuseppe Castagna, Jan Vitek, and Francesco Zappa Nardelli. The Seal calculus. *Inf. Comput.*, 201(1), 2005.

[14] William R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.

[15] Geoff Coulson, Gordon Blair, Paul Grace, Ackbar Joolia, Kevin Lee, and Jo Ueyama. OpenCOM v2: A Component Model for Building Systsms Software. In *Proceedings of IASTED Software Engineering and Applications (SEA '04)*, 2004.

[16] First report on evolvable systems, March 2010. Deliverable 3.1.a of project FP7-231 620 (HATS), available at `http://www.cse.chalmers.se/research/hats/sites/default/files/Deliverable31a_rev2.pdf`.

[17] Full ABS Modeling Framework, March 2011. Deliverable 1.2 of project FP7-231 620 (HATS), available at `http://www.hats-project.eu`.

[18] Survey on state of the art time awareness and management, March 2011. Deliverable 1.2 of project FP7-247758 (EternalS), available at `https://www.eternals.eu/sites/default/file/D1_2_TF2_stateOfTheArt.pdf`.

[19] Johan Dovland, Einar Broch Johnsen, Olaf Owe, and Martin Steffen. Lazy behavioral subtyping. In Jorge Cuellar and Tom Maibaum, editors, *Proc. 15th International Symposium on Formal Methods (FM'08)*, volume 5014 of *LNCS*, pages 52–67. Springer, May 2008.

[20] Johan Dovland, Einar Broch Johnsen, Olaf Owe, and Martin Steffen. Lazy behavioral subtyping. *Journal of Logic and Algebraic Programming*, 79(7):578–607, 2010.

[21] Johan Dovland, Einar Broch Johnsen, Olaf Owe, and Martin Steffen. Incremental reasoning with lazy behavioral subtyping for multiple inheritance. *Science of Computer Programming*, 2011. To appear.

[22] Dominic Duggan. Type-Based hot swapping of running modules. In Cindy Norris and Jr. James B. Fenwick, editors, *Proc. 6th Intl. Conf. on Functional Programming (ICFP'01)*, volume 36, 10 of *ACM SIGPLAN notices*, pages 62–73. ACM Press, September 2001.

[23] Cedric Fournet and Georges Gonthier. The join calculus: A language for distributed mobile programming. In *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal*, pages 268–332. Springer-Verlag, 2002.

[24] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A formal framework for on-line software version change. *IEEE Trans. Software Eng.*, 22(2):120–131, 1996.

[25] Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In *ICALP*, pages 299–309, 1980.

[26] Gísli Hjálmtýsson and Robert S. Gray. Dynamic C++ classes: A lightweight mechanism to update code in a running program. In *Proc. USENIX Tech. Conf. (USENIX '98)*, May 1998.

[27] Alan Jeffrey and Julian Rathke. Java Jr.: Fully abstract trace semantics for a core Java language. *Lecture Notes in Computer Science*, 3444:423–438, 2005.

[28] 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, *Proc. 16th International Symposium on Formal Methods (FM'09)*, volume 5850 of *LNCS*, pages 596–611. Springer, November 2009.

[29] Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, March 2007.

[30] Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, November 2006.

[31] Vasileios Koutavas and Mitchell Wand. Bisimulations for untyped imperative objects. In Peter Sestoft, editor, *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3924 of *Lecture Notes in Computer Science*, pages 146–161. Springer, 2006.

[32] Vasileios Koutavas and Mitchell Wand. Reasoning about class behavior. In *Informal Workshop Record of FOOL 2007*, January 2007.

[33] Ivan Lanese, Antonio Bucchiarone, and Fabrizio Montesi. A framework for rule-based dynamic adaptation. In *Proc. of TGC 2010*, volume 6084 of *Lecture Notes in Computer Science*, pages 284–300. Springer, 2010.

[34] Matthieu Leclercq, Vivien Quema, and Jean-Bernard Stefani. DREAM: a Component Framework for the Construction of Resource-Aware, Configurable MOMs. *IEEE Distributed Systems Online*, 6(9), 2005.

[35] Sergueï Lenglet, Alan Schmitt, and Jean-Bernard Stefani. Howe's Method for Calculi with Passivation. In Mario Bravetti and Gianluigi Zavattaro, editors, *CONCUR 2009 - Concurrency Theory*, volume 5710 of *Lecture Notes in Computer Science*, pages 448–462. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-04081-8_30.

[36] Francesca Levi and Davide Sangiorgi. Mobile safe ambients. *ACM. Trans. Prog. Languages and Systems, vol. 25, no 1*, 2003.

[37] Michael Lienhardt, Alan Schmitt, and Jean-Bernard Stefani. Oz/k: A kernel language for component-based open programming. In *GPCE'07: Proceedings of the 6th international conference on Generative Programming and Component Engineering*, pages 43–52, New York, NY, USA, 2007. ACM.

[38] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes. Runtime support for type-safe dynamic Java classes. In Elisa Bertino, editor, *Proc. 14th European Conf. on Object-Oriented Programming (ECOOP'00)*, volume 1850 of *LNCS*, pages 337–361. Springer, June 2000.

[39] Hugo Miranda, Alexandre S. Pinto, and Luis Rodrigues. Appia: A flexible protocol kernel supporting multiple coordinated channels. In *21st International Conference on Distributed Computing Systems (ICDCS 2001)*. IEEE Computer Society, 2001.

[40] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Composing services with JOLIE. In *Proc. of ECOWS'07*, pages 13–22. IEEE Computer Society Press, 2007.

[41] Fabrizio Montesi and Davide Sangiorgi. A model of evolvable components. In Martin Wirsing, Martin Hofmann, and Axel Rauschmayer, editors, *Trustworthly Global Computing*, volume 6084 of *Lecture Notes in Computer Science*, pages 153–171. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-15640-3_11.

[42] James H. Morris. Lambda-calculus models of programming languages. Technical Report 57, MIT Laboratory for Computer Science, 1968.

[43] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. In *ACM Symposium on Operating Systems Principles*, 1999.

[44] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th international conference on Software engineering*, ICSE '98, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

[45] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Runtime software adaptation: framework, approaches, and styles. In *Companion of the 30th international conference on Software engineering*, ICSE Companion '08, pages 899–910, New York, NY, USA, 2008. ACM.

[46] Alessandro Orso, Anup Rao, and Mary Jean Harrold. A technique for dynamic updating of Java software. In *Proc. Intl. Conf. on Software Maintenance (ICSM'02)*, pages 649–658. IEEE Computer Society Press, October 2002.

[47] Alan Schmitt and Jean-Bernard Stefani. The M-calculus: A Higher-Order Distributed Process Calculus. In *Proceedings 30th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2003.

[48] Alan Schmitt and Jean-Bernard Stefani. The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In *Global Computing*, volume 3267 of *Lecture Notes in Computer Science*. Springer, 2005.

[49] Leila Silva, David A. Naumann, and Augusto Sampaio. Refactoring and representation independence for class hierarchies: extended abstract. In *Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs*, FTFJP '10, pages 8:1–8:7, New York, NY, USA, 2010. ACM.

[50] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis mutandis: Safe and predictable dynamic software updating. *Transactions on Programming Languages and Systems*, 29(4):22, 2007.

[51] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for dynamic sealing. *Theoretical Computer Science*, 375, 2007.

[52] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for type abstraction and recursion. *Journal of the ACM*, 54, 2007.

[53] Sun Microsystems. JSR 220: Enterprise JavaBeans, Version 3.0 – EJB Core Contracts and Requirements, 2006.

[54] Clemens Szyperski. *Component Software, 2nd edition.* Addison-Wesley, 2002.

[55] David Teller, Pascal Zimmer, and Daniel Hirschkoff. Using Ambients to Control Resources. In *Proceedings CONCUR 02*, 2002.

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.

API Application programming interface, provided by a component to enable communication with other components.

Architect principle An architectural principle for designing component-based systems, specifying that each component in the system is in charge of (and is allowed to) managing its children.

Comp A calculus of component designed for integration with the ABS language.

Compatibility A relation between two versions of a component which shows whether a usage context can observe a difference in behavior.

Dynamic Evolution An evolution that deals with run-time modifications, such as rebinding and exchanging of components.

MECo A calculus of components which specializes in operators related to adaptability and evolvability. It stands for Model of Evolvable Components.

PLE Product Line Engineering, software engineering methods whose concern is creating a family of products with well-defined commonalities and variabilities.

Software Evolution The process of updating software to fix bugs, implement improvements, adapt new or changed requirements, platforms or technology.

Static Evolution An evolution that deals with the change of the program design, such as the change of API.