Project N°: **FP7-231620**

Project Acronym: **HATS**

Project Title: **Highly Adaptable and Trustworthy Software using Formal Methods**

Instrument: **Integrated Project**

Scheme: **Information & Communication Technologies**

Future and Emerging Technologies

Deliverable D3.1.a
First Report on Evolvable Systems

Due date of deliverable: (T12)

Actual submission date: 1st March 2010

Revision date: 30th March 2010

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

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

Revised 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:
First Report on Evolvable Systems

This document summarizes deliverable D3.1.a 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.

The investigations of Task 3.1 form two research lines. In the first line, three different forms of software/system evolution are selected from the evolution space in order to develop a precise basis for these forms. In the second line, the effect of evolution on different concerns is investigated.

List of Authors

Mads Dam (KTH)
Nikolay Diakov (FRH)
Einar Broch Johnsen (UIO)
Ilham Kurnia (UKL)
Ivan Lanese (BOL)
Arnd Poetzsch-Heffter (UKL)
Davide Sangiorgi (BOL)
Yannick Welsch (UKL)
Peter Wong (FRH)

Contents

Chapter 1

Introduction

The reasonable man adapts himself to the world; the unreasonable one persists in trying to adapt the world to himself. Therefore all progress depends on the unreasonable man.
- George Bernard Shaw (1856 - 1950)

Today's long-living software systems are all but static monoliths. Embracing evolution is one of the key requirements for the longevity of such systems. In many cases, however, uncontrolled change (evolution) leads to unmanageable software or even to disaster. The typical example which demonstrates the consequences of non-proper evolution of a software system is the Ariane 5 crash for which the inadequate reuse of an older Ariane 4 software module was blamed. Only three of the seven integer variables in the module were checked against overflow conditions, the remaining four were proven not to overflow, however only for the Ariane 4 system. Unfortunately, these proofs were not valid anymore in the setting of Ariane 5.

Building the foundation for *adaptable* software is of upmost importance in the HATS project. Software evolution as a process is different from software construction. According to [39], "*the ability to change is now more important than the ability to create [e-commerce] systems in the first place*". In order to support the software/system evolution process in a precise way, a formal basis is needed.

Essentially, there are three driving forces for evolution:

1. The need to adapt the systems functionality to *new or changed requirements*

2. The need to adapt the systems implementation and deployment to *new or changed platforms or technology*

3. Necessary maintenance steps including improvements and bug fixes

Changed requirements could for instance result from changes in the application domain. Changes in the platform can be triggered on all levels of the software/hardware stack. For example, they can be caused by new operating system versions, new computers, new sensors and actors, and new component frameworks. Thus, a general need for change is a natural part of the lifecycle of a software system. According to the Merriam-Webster Online Dictionary, evolution is "*a process of continuous change from a lower, simpler, or worse to a higher, more complex, or better state*".

This document reports on the first phase of Task 3.1. The central goal was to characterize the different forms and concerns of *software evolution steps* that arise in practice and to analyze and model fundamental aspects of system and component evolution. The report is structured as follows. The rest of the introduction discusses relevant dimensions of evolution and summarizes the forms and concerns of evolution steps that we selected for investigation, how we approached the analysis of these steps, and how they are related to the goals of this task. The other sections summarize the work on the investigated specific forms and concerns of evolution.

Organization of work. In addition to bilateral visits, we had three meetings of the contributors of this task. The first one took place as part of the kickoff meeting in March. A separate one day workshop was held in Leuven in June. The third meeting was part of the annual HATS meeting in Gothenburg in September.

1.1 Dimensions of Evolution

Software and system evolution appears in various forms. In order to characterize these forms of evolution and to focus the analysis on relevant aspects, we propose to classify them according to different dimensions. The dimensions of evolution, which are of importance in the HATS project, are the following ones:

Software development phases. The first considered dimension of software/system evolution is the different phases of the software development. Evolution can happen in all these phases. The HATS methodology focuses on the design level and uses the models to formally state the system behavior. Changes to functional requirements and the system architecture affect the models and cause changes in the implementation, and possibly in the deployed systems. Usually evolution appears both at the inter-component and intra-component level. As a consequence, evolution carries on to the actual *component development* where the behavior of components is refined and implemented. At the *testing* phase, evolution has an influence on the tests as well as the testing methodology. In the *integration and deployment* phase, reconfiguration or deployment to new platforms can be seen as evolutionary steps. An important part of the software lifecycle is also the *maintenance and administration*, in which, for example, bugs in certain system parts are fixed.

Static/dynamic. Another dimension of evolution distinguishes between modification to system descriptions (models, implementations, installation scripts, etc.) and to installed and potentially running systems. In the first case, we speak of *static* evolution. In the second case, we speak of *dynamic* evolution. For example, if bug fixes are made to a large web application, the code is first rewritten in such a way that the bug does not manifest itself anymore (static evolution), then the application is patched with the bug fixes at runtime such that there is a minimal downtime. Static evolution steps are usually independent of persistent data, whereas dynamic evolution steps have to make sure that the existing data state is mapped to the new version. For example, the integration of multiple information systems requires to evolve the data model of at least one of the systems in order to realize a clean integration.

Features. In the context of software families and product-line engineering, evolution of single features as well as evolution of a feature model must be considered. In the former case, evolution of a feature might be easy to understand at the feature-local level. However, if this feature is mapped onto several components, it might not be trivial to understand in which fashion these components evolve. If a feature model evolves, existing constraints on the initial feature model might not hold anymore. An important problem in this context is the consistent evolution of the artifact base.

Properties. Evolution can be characterized according to the addressed properties. In particular, evolution may focus more on functional or non-functional aspects, or maybe both.

The above dimensions are not completely independent, but many combinations are possible. In particular, every evolution step can be done as a modification of the system description, i.e. statically, or as an update of an installed or even running system (dynamically).

1.2 Approach

We organized the investigations of this task into two research lines. In the first line, we selected three different forms of evolution from the evolution space sketched in the last section and started to develop a precise basis for software/system evolution for these forms. In the second line, we investigated the effect of evolution on different concerns. The following paragraphs introduce the research lines; the details are given in the subsequent chapters.

Forms of evolution. In the first phase of this task, we investigated three *forms of evolution*:

1. Static evolution of models and programs (Chapter 2)
2. Dynamic evolution of component systems (Chapter 3)
3. Dynamic evolution of code (Chapter 4)

In static evolution, an evolution step corresponds to the modification of a model description or of programs. The central question addressed in this area is how to prove that the modification of a module does not affect the overall system behavior. We investigated this question with respect to object-oriented program modules as the modeling language was not fixed when we started this work. The first result in the area is a compatibility criterion that allows to check whether a refactored module can be integrated into all possible contexts of the old module. We expect that the results can be transferred to the modeling language.

Dynamic evolution means investigating modifications to executing systems. Typical operations are re-binding and exchanging components. Two questions were investigated in this area:

1. How can we formally model evolving systems by a minimal calculus that allows to study the semantics of evolution in detail? (Section 3.2)
2. How can we realize high-level adaptation steps in distributed information systems? (Section 3.3)

Whereas the first question addresses overall consistency properties of dynamic evolution, the second question is concerned with dynamic mechanisms needed to model system evolution at runtime.

A special, but important case of system evolution at runtime is the modification of the code base of an executing system. The central concern here is not that the new code provides the desired behavior, but that the upgrade mechanism works correctly.

Concerns of evolution. In addition to identifying existing important forms of evolution and to answer the related consistency and correctness requirements, this task is as well about general concerns of software system evolution. We identified two important concerns for the HATS project:

1. Mechanisms for *autonomous* adaptation
2. Evolution of the software development environment

Autonomous adaptation is about the fundamental dynamic mechanisms needed to provide a system with the capability of self-adaptation. Chapter 5 addresses this concern and describes how the modeling techniques of HATS can be used to support autonomous system adaption, in particular with respect to non-functional properties (e.g., resource consumption).

During the evolution process of a software system the complete development infrastructure has to evolve as well. In this area, we focused so far on one aspect, namely the evolution of test systems (Chapter 6). In particular, concerns related to the development of *evolvable* test systems are considered. These concerns arise from the context that intersects the HATS methodology, the theoretical foundation of testing and industrial applications. Another concern in this area that is important for HATS is the evolution of proofs. This topic will be addressed in the next phase of Task 3.1.

Chapter 2

Static Interface Evolution

2.1 Introduction

Large systems are usually structured into smaller parts (modules), which communicate with each other through well-defined interfaces. These interfaces, very often also called application programming interfaces (APIs), may evolve over time due to several reasons. For example, additional functionality is provided in a new version of a module. Obsolete functionality may be removed or deficiencies of the old module version may be fixed. Interface evolution has an impact on both the API producer and consumer.

The maintainer of an API needs to consider whether the changes to the API guarantee backward compatibility. This is especially the case when the API of a library, which is used by a large number of projects, is evolved. As more than one API version now exists, the library developer has to maintain (e.g., bug fixes) multiple versions of the API (which may provide functionality under different APIs). This additional complexity has an impact on large parts of the software engineering process, for example testing and documentation. The API user has to make a choice which version of the API he wants to use. It might not always be useful to migrate a large part of his code to use the new API version if there is no clear benefit. Determining the benefits is a (not to be neglected) burden on the API consumer.

In the HATS project, the spatial software dimension of variability also interferes with the temporal dimension of evolution. Evolving a software architecture with multiple configurations (variable parts) is more complicated than in the single-system case. Evolution in the feature model impacts evolution of the APIs at the program or model level.

In context of the HATS project, UKL addresses the question what the API of a Java package actually is. We only consider a restricted form of API evolution of Java packages (no variability model), which is called compatibility.

Source Compatibility for Java Packages

Often library packages or packages used by third party clients have to be modified, extended, or refactored in such a way that client code is not affected. A prerequisite is that the client code at least compiles with the modified package version. For library packages and in scenarios where the client code is not known, the check has to be done only based on the old and new package versions.

If a new package version compiles with all possible client code of the old version, we say that the new version is *source compatible* with the old one. For languages like Java and C#, source compatibility is a complex property. As an example, we invite the reader to have a look at packages Q and R shown in Fig. 2.1.

Even for this very simple case, it takes some thoughts to prove or disprove that package R is source compatible with Q. We will show that this problem results from the complex interplay between encapsulation using access modifiers, inheritance and subtyping. To our knowledge, there are currently no tools to automatically check source compatibility. For the following reasons, we believe that a better understanding of source compatibility is needed:

Implementation Q

```
package p;

public abstract class c implements i {
   public i f;
   protected c g;
}

interface i {
   ...
}
```

Implementation R

```
package p;

public class c {
   public d f;
   public c g;
   protected Object m() { ... }
}

final class d {
   ...
}
```

Figure 2.1: An example for compatibility of Java packages

- *Software engineering*: With increased system size, separate development becomes more and more important. Thus, well-defined interfaces between package providers and clients are needed. In particular, compatible modifications and extensions of old code must be possible.
- *Modular refactorings*: Source compatibility is the basis for *modular* refactorings of packages that maintain behavior in all or a well-defined set of possible contexts.
- *Language design*: Future module systems for object-oriented language should simplify compatibility checks and the specification of module contracts.

Our general goal is to consider compatibility as a foundation to study the evolution of API's on the source level. In particular, we use Java packages to technically investigate the following questions:

What is encapsulated within a package? Java packages in combination with access modifiers provide encapsulation. When creating a library, a designer has to be careful to encapsulate all implementation details that should not be accessible from the outside. In Java and similar languages, however, it is not clear what the interface of a package is. For example, and as illustrated later, the relationship between non-public types can influence compatibility. Thus, it needs to be documented as part of the package API.

Can we simplify compatibility checking without restricting Java too much? In the following, we present techniques to simplify syntactic compatibility conditions. We propose small changes to Java's well-formedness conditions that make reasoning about compatibility a lot easier. We provide empirical evidence that this does not lead to practical limitations.

2.2 Results

The central result of our investigation is a technique to make source compatibility checkable. The result is based on the following technical contributions and discussed in more detail in [117]:

1. We define and discuss source compatibility for packages.
2. We provide a formal definition of the context conditions and typing rules for a Java subset with all modifiers covering access modifiers, as well as *abstract* and *final* program elements. For the relevant aspects, the definition goes beyond existing formalizations of Java's static semantics.
3. We present *syntactic* conditions for checking source compatibility that avoid the quantification over all contexts.
4. We prove that the syntactic conditions are *necessary* and *sufficient*. In particular, we develop a new proof technique for *static context abstraction*.

5. We discuss possible improvements to Java's context conditions (e.g., public methods can only have public types in their method signature) to simplify source compatibility checking and present an investigation how far such simplifications affect existing code. Furthermore, we illustrate the relation of source compatibility and modular refactoring.

Illustration

We now illustrate source compatibility by the previously given example assuming that the reader knows Java's access rules. We consider two implementations Q and R of a package with name p. Let K be some arbitrary *program context*, consisting of one or several packages, that uses Q such that the program KQ, consisting of K and Q, is well-formed according to the context conditions and typing rules of Java. Is KR well-formed as well? Or, to put it differently, is there a program context which compiles with Q but not R? The answer is: Yes. Both versions of class c define a field f of non-public type i and d, respectively. If we have a context which has a variable v of static type c, the cast expression $(Comparable)v.f$ is well-formed for Q (as there might be an object of an interface type *Comparable* referenced by f) but not for R as it can be statically ensured that the class can never have a subclass implementing the interface *Comparable*.

If we remove the final modifier on class d, does this ensure compatibility of R with Q? To answer this, let us consider the expression $this.f = this.g$. The expression is well-formed in a subclass of c when Q is used (as c is a subtype of i) but not when R is used (no subtyping between c and d). The same issue would still occur if both fields f and g have a non-public type (in the given example, only f has a non-public type). Thus, it can have an effect on the well-formedness of entities outside of a package whether two non-public types are in a subtype relationship. If we have, for example, two (accessible) fields of non-public types which are in a subtype relation in Q but not in R, we can easily construct an expression which types with Q but not R. The same is applicable if, instead of the field which we assign to, there is a method with a non-public parameter type, or if, instead of the field we assign from, there is a method with a non-public return type.

Another difference between Q and R is the access modifier of the field g. As it is more accessible in R, it should be accessible in all contexts of Q. Can we conclude that the difference does not lead to incompatibility of Q and R? And how does the additional method m in R and the missing abstract modifier of class c in R affect compatibility? In our investigation, we develop syntactic conditions to answer such questions automatically.

Related work

Dmitriev [29] investigates `make` technology for the Java language, in particular how a change to a class *may* affect other classes. Source incompatible changes (at the class, not package level) are listed in a semi-formal way, but neither proven necessary nor sufficient. To our knowledge there is no other work that makes source compatibility the focus of investigation. In the following, we relate our topic to work on behavioral or contextual equivalence for object-oriented components, binary compatibility, new OO module systems, language specification aspects, and refactoring techniques.

Behavioral equivalence. Two classes, two packages, or generally two components are called behavioral equivalent if they have the same interface behavior. Source compatibility is a prerequisite for behavioral equivalence: If two components are not source compatible, there is a context that compiles with one, but not with the other component and thus the components are not equivalent. Koutavas and Wand [70] present proof techniques to show that two classes are equivalent, but source compatibility is almost trivial in the language subset they consider, without packages and with only very restricted use of access modifiers. Closely related to our work is the notion of compatibility of Jeffrey and Rathke [58] who develop a fully abstract trace semantics for a Java-like core language with a package construct. They develop a syntactic characterization of compatibility[1] ([58], Sect. 3) as a prerequisite to a fully abstract semantics of packages. However, the setting they consider has a non-standard package concept and only two different access modifiers.

[1] Their work provided one of the starting points of our studies.

Binary compatibility. Chapter 13 of the Java Language Specification [47] defines properties for binary compatibility: a set of changes that developers are permitted to make to their packages, classes or interfaces. This set must guarantee that preexisting class files which linked with the previous (package, class or interface) versions still link with the current versions. As already mentioned in the JLS (Section 13.2), binary compatibility is different from source compatibility. For example (Section 13.4.6), introducing a new field, with the same name as an existing field, in a subclass of the class containing the existing field declaration, does not break binary compatibility with preexisting binaries. However, at the source code level, this may lead to source incompatibility (typing error). A new declaration is added, changing the meaning of a name in an unchanged part of the source code, while the preexisting binary for that unchanged part of the source code retains the fully qualified, previous meaning of the name.

In our opinion, this notion of (binary) compatibility is not the right one for our use. If we consider the case that a library developer has made binary compatible changes to his code, a client developer may not be able to recompile his (ongoing project) client code with the new version of the library (e.g., if he wants to make some fixes to his client code). Another important issue with the compatibility as defined by the JLS is that only sufficient conditions[2] are given (e.g., Section 13.3). Furthermore, different encapsulation boundaries are considered (e.g., packages, classes and interfaces) which complicate the presentation even more.

Forman et al. [43] have investigated binary compatibility for IBM's System Object Model. They provide a set of transformations which should guarantee compatibility, but do not provide any proofs. Drossopoulou et al. [32] analyzed binary compatibility as it is defined in the JLS, show that some of the transformations described in the JLS do not guarantee successful linking, and prove their own binary compatibility criteria correct for a Java subset. However, they do not consider whether the criteria they give are necessary conditions for compatibility.

Module systems and language design. New module systems for Java have been proposed, e.g., [110], and are under current consideration [64, 65] for the new Java 7 language. These module systems do not really solve the question, what the API of a module is. Very often, this is defined as the aggregation of the API of a set of packages or types. However, it remains unclear what the actual API of a package or type is. We hope that the compatibility conditions we present initiate further research on alternative definitions of modules and their interplay with compatibility.

The following work studies the Java accessibility modifiers. Müller and Poetzsch-Heffter [85] identify the changes that access modifiers in a program can have on the program semantics. Schirmer [102] gives a formalization of the access modifiers and shows interesting runtime properties with respect to access integrity. The Java subset we formalize further considers the modifiers *abstract* and *final*, which also have a huge impact on compatibility.

A setting similar to ours is the *fragile base class* problem [81]. It studies the effects that changes in a (base) class can have on its subclasses.

Refactoring. There exists a lot of work in the refactoring community to deal with API evolution. Most solutions work by creating compatibility layers for the libraries or adapting the client programs (in binary and source setting), for example [7, 22, 28, 45, 53], but this addresses a different issue. The question, what the API of a library actually is, is left unanswered or only partially answered for a fixed set of clients.

Steimann and Thies [106] describe a semantic-preserving refactoring technique to refactor programs under constrained accessibility. However, as for most of the work on refactoring, they operate on programs where all accesses to program entities are known, i.e., the usual closed-world view. Our goal is to have a refactoring technique which is modular in the sense that the parts of the library, which form the API, are only modified in a compatible way.

[2] Not even these hold true, as shown in [32].

2.3 Outlook

2.3.1 Static Compatibility Checking

The compatibility conditions were designed in a way which easier allows them to be proven necessary and sufficient. They are not ideal for automated checking. It would lead to checking many sub-conditions several times, but it provides the specification for better algorithms.

Besides looking for more efficient checking algorithms, future work should extend the approach to include larger subsets of the Java language. Especially static members, nested classes, nested packages and constructors with reduced accessibility and generics can affect the compatibility conditions. Furthermore, another simplification should be dealt with. We considered the set of package, class and variable identifiers to be disjoint in our formalization, which is not the case in Java. We also avoided issues of ambiguous type names, as we require all our type names to be fully qualified. To avoid problems resulting from ambiguous names, one might want to restrict the set of contexts (e.g., contexts should not use *import* wildcards).

Instead of defining the syntactic compatibility relation directly on package implementations, as it is done in [117], it is also possible to derive some (syntactic) package signatures from the implementations and then define compatibility based on these signatures. This is a two-step process which might lead to better module/package designs. Currently, the package signature is hidden in the definitions of compatibility. A possible application for this is modular typechecking at the package level, e.g., the compiler may not need to know about non-public types to typecheck other packages. It would also be interesting to apply the proof approach to other programming languages and module systems.

2.3.2 Modular Refactoring

Most semantic-preserving refactoring techniques assume that the complete program is available (the usual closed-world view). They allow to reason about semantic preservation of refactorings for one single context, the given program. This might fit well for developers of a single program, but not at all for library or component developers. While most of the existing work which tries to address this issue, e.g., [7, 28, 45, 53], tracks modifications of the library and creates compatibility layers or adapts the clients, we consider a far simpler setting where no such tracking is needed.

Let us consider the *Rename Variable* refactoring as described in [101]. The refactoring should work in such a way that all bindings are preserved, i.e., all accesses to a declaration should be preserved by the renaming. In a complete program, all accesses to a declaration are known. In the setting of refactoring a library, however, this assumption does not hold.

We see two ways to realize modular refactoring. One way would be to do the refactoring and then check if the new library version is (syntactically) compatible with the old one. Another way would be to statically prove that a certain class of refactorings guarantee (syntactic) compatibility. One could also restrict the set of possible contexts by describing acceptable contexts syntactically in the signature (contract) of the package. We plan on further investigating both scenarios in the future.

2.4 Conclusion

Every module system should be able to deal with the following simple question: What is the API of a module? We have shown that, in languages with complex encapsulation properties like Java, this question is far from trivial to answer. We have demonstrated how to derive necessary and sufficient syntactical conditions for compatibility. This syntactic characterization of compatibility gives new insight into the encapsulation of Java packages and provides a basis for further studies on module systems and modular refactoring techniques.

Chapter 3

Dynamic Component Evolution

3.1 Introduction

Evolvability 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 in BOL with the goal of isolating interesting constructs for expressing dynamic evolvability. The constructs should be expressible enough to model common patterns of evolvability for systems, and at the same time should be mathematically robust so to allow verification of desired behavioral properties of the system. Two directions of work have been pursued.

The first direction aims at isolating a core calculus of evolvable components, following the process calculus tradition. The main problem was isolating primitives that capture the relevant concepts of component-based systems and common evolvability patterns for them.

In the other direction, a new, language-independent, approach is 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. Specifically, the application publishes an interface including a list of activities that could be updated. An *adaptation manager* includes a list of rules specifying how and under which conditions an activity of an application has to be updated, keeping into account its state and the required non functional properties. If the rule applies, the new code for the activity is sent by the adaptation manager to the application and replaced for the old one. As a proof-of-concept implementation we have produced JoRBA (Jolie Rule-Based Adaptation framework) [63], based on the service oriented language Jolie [83], realizing the mechanisms described above.

3.2 A Model of Dynamically Evolvable Components

3.2.1 Results

In our first line of work, we have followed the process calculus approach. Process calculi have been successfully employed in the modeling, analysis, and verification of concurrent and distributed systems. They are particularly interesting when it comes to understanding linguistic constructs targeted to specific needs — in our case evolvability[1].

[1] The same also applies to sequential computations, for core calculi such as the λ-calculus; several extensions of the λ-calculus, targeted to specific needs, have been studied.

When trying to formalize a calculus of evolvable components, we first had to decide how components should be represented. In recent years, proposals of calculi for distributed systems have been put forward with explicit notions of *location* (also called *site*). While locations may allude to components, the differences between the two concepts remain noticeable. In particular locations do not have explicit notions of interface.

To represent components, we have used ideas from object-oriented languages of interest in HATS, as well as ideas from components as used in complex software systems and Software Families (cf. presentation by Frank van der Linden at the general HATS meeting, or the Fractal component architecture). Suggestions and discussions from Arnd Poetzsch-Heffter (during a visit to Bologna and during HATS meetings) as well as the work on CoBoxes [100] have also been very influential.

We have designed a basic calculus of components, called MECo (Model of Evolvable Components). We have experimented with a number of operators, especially related to adaptability and evolvability. Those retained for MECo seemed to us a reasonable compromise between practical component needs (as, e.g., in Fractal component systems) and conciseness. The main features of MECo are: a hierarchical structure of components; a prominent role to input/output interfaces; 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 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.

Other than through component methods, interactions can take place through channels. When a component calls another one, the first may pass a private channel to the second; this channel can then be used for further interactions between the two components (other components may actually get involved, if the channel is sent around). Creation and communication of channels can lead to tunneling effects: for instance a component c_1 calls a component c_2 that forwards the message to some inner component c_3. If the message contains a channel, then c_1 and c_3 can use the channel for direct interactions.

Components are stopped by means of a construct, `extract`, reminiscent of the passivation operator of calculi such as Kell [103] and Homer [54, 15]. The `extract` operator, when applied on a component, say c, intuitively allows us to stop c and extract its current state and functionalities (the methods); now c can be manipulated, for instance wrapped inside another component so to modify or hide certain methods of c to the external environment, or to restart c with a different output port binding so that c will use different resources from the environment. The `extract` operator is the only one that permits modifications of the structure of components which is otherwise static.

In the paper [84], we present the calculus and explain the syntax. We then formulate its operational semantics. We have equipped the calculus with a basic type system to avoid run-time errors. A number of examples how the calculus is used are presented. In particular, we show how various patterns of evolvability of components are captured; they include:

Rebinding. Rebinding is a technique for modifying the output port bindings of a component at runtime. This is done by extracting the component and putting it into execution with the new output port definitions.

Interceptors and wrappers. Both the *interceptor* and the *wrapper* patterns are about modifications of the functionality of a given legacy component. The two techniques are similar in their basic concepts but the structures resulting from their applications are different, and this may affect the interactions with other components, as commented at the end of the section.

There are two kinds of interceptors: input interceptors and output interceptors. Input interceptors are used to adapt the input interface of the legacy component by intercepting calls for it from other components, whereas output interceptors intercept calls coming out of the output ports of the legacy component.

Input interceptors can be used for making the same component available under multiple identities, and for exposing a different interface. There are three possible cases: offering a new method (the system may have more requirements than what the legacy component supports); hiding a method (for encapsulation or security purposes); changing the behavior of a method.

Output interceptors are supposed to capture outgoing calls issued by the legacy component and then trigger some actions.

While interceptors execute as siblings of the legacy component, a wrapper captures the legacy component (the *wrapped component*) and executes it as an inner component of another one (the *wrapper*), that is responsible for offering a modified view of the wrapped component.

There are important differences between interceptors and wrappers when adapting a legacy component. On one hand, a wrapper has tighter control on the legacy component, as the legacy component becomes an inner component; thereafter, wrapping and wrapper components can be treated as a single unit. On the other hand, a wrapped legacy component is not anymore reachable from the rest of the system other than through the wrapper itself, whereas with interceptors the legacy component remains reachable by those components that know its identity.

3.2.2 Related Work

The closest process calculi to ours are Kell [103] and Homer [54, 15]. These are calculi of mobile distributed processes in which computational entities may move in a dynamic hierarchy of locations. They have passivation operators that behave similarly to the `extract` of MECo. We may also see these calculi as calculi of components, thinking of locations as component boundaries. The main differences between Kell/Homer and MECo is the explicit use of interfaces in MECo (interfaces make MECo components look like objects, in fact, more than Kell/Homer locations). Another difference is the presence of channels in MECo; the resulting tunneling effects are not possible in Kell or Homer where communication is local. The relation of MECo with other process calculi with locations, e.g., Ambients [17] and Seal [116], is weaker.

3.2.3 Outlook

The study of MECo is, admittedly, in a preliminary stage; for instance, as discussed below, typing is very rigid, and behavioral equivalences remain unexplored. Also, and most important, more experimentation with examples is needed to test the constructs chosen in MECo.

For the future, we should also explore refinements of the basic type system, especially subtyping. Ideas from object-oriented languages should be useful here too, though output interfaces will require extra care. This may also lead to refining the present channel interactions of MECo into notions of session from Service-Oriented calculi, e.g., [16, 114].

In another direction, we would like to examine stronger forms of run time errors, whereby if a call to a component is made, then one is ensured that a component capable of consuming the message does indeed exist. As a prerequisite one would probably have to record the set of components that a process needs for its execution. This is non-trivial, as component identities may be communicated and components may be passivated.

Another issue to study in MECo may be behavioral equivalence; for instance, one may be able to establish behavioral properties on the evolvability patterns studied, so to be able to actually prove that an evolvability step is transparent to clients.

3.3 A Framework for Rule-Based Dynamic Adaptation and Evolution

3.3.1 Results

While experimenting with process calculi primitives, we also took a more pragmatic approach to dynamic adaptation and evolution. When developing an adaptable application, or an application that should evolve

over time, it is common to put most of the information concerning adaptation and evolution inside the application itself, via dedicated constructs [37] or architectures [41]. In case of adaptation this is called *built-in* adaptation.

However, in most of the cases, the exact ways in which the application should change are neither known at design time, nor at application deployment time. Thus we looked for approaches minimizing the information needed in the early stages. Clearly, however, an adaptable application must provide some *adaptation hooks*, to be exploited by the *adaptation manager* external to the application to change the application itself. In [57] we propose a rule-based approach following these principles. We concentrated on adaptation, leaving for the future the analysis of how to apply those mechanisms for evolvability. The approach is language-independent, but has also been put at work in practice using language Jolie [83, 62], a language for programming service oriented applications. A prototype of our implementation can be found at [93].

The main idea is that an adaptable application has an *adaptation interface*, i.e., it makes public and exposes via this interface information concerning some of its *domain activities* (or simply activities). These are the activities that can be updated and replaced at runtime. In our approach, in fact, it is necessary to know at design time where adaptation can happen (i.e., which are the involved activities, and a few information about them), but it is not necessary to know which conditions will trigger the adaptation and which updates will be performed on the application. These last pieces of information will be provided by a set of *adaptation rules*, which are external to the application, managed by a dedicated adaptation manager, and can be created and replaced at any time regardless of the state of the application. Each such rule specifies a possible adaptation, including a description of the activities that can be adapted by the rule (to be matched by the description of the activity which is part of the adaptation interface of the application), an applicability condition that checks whether adaptation is applicable, referring to both variables of the adaptation manager (containing information on the environment) and public variables of the application (again, included in the adaptation interface), the new code for the activity, and information about the non-functional properties provided by the new implementation, to be compared with the ones provided by the current implementation.

At runtime, those rules are matched with the adaptable activities in the applications in the range of the adaptation manager. If one of them applies, then adaptation of this activity is performed. In particular, the new code of the activity is sent to the adaptable application and replaced for the old one. All the information needed for enacting this process is part of the adaptation interface of the application. In particular, the new activity should be able to interact with the rest of the application via the set of public variables of the application.

We have validated our approach by implementing a prototype [93] of an adaptable application (a skeleton, to be extended with the user defined code of the application) and of adaptation manager in Jolie [83, 62]. We have chosen Jolie since a service oriented language is a good candidate to realize dynamic interactions between the different components involved in the adaptation process, e.g., the applications and the adaptation manager. Jolie, in particular, provides primitive mechanisms which were very helpful for our task. For instance, Jolie *dynamic embedding* has been used to dynamically install the new code for activities when adaptation is performed. The prototype also implements a simple traveling scenario. We describe one of the adaptation examples below.

Assume that Bob is traveling from Bologna to Trento for a working meeting. He has an application on his mobile phone taking care of the travel, e.g., buying the necessary tickets and instructing Bob about what to do. Bob was supposed to take an intercity train which takes 2h41m and costs 20 euros. When he arrives at Bologna's train station, the mobile phone enters in the range of Bologna's train station adaptation manager. Assume that a new FrecciaRossa connection (Italian high speed train) is available from Bologna to Trento, which takes 1h23m and costs 32 euros. This is reflected by a rule able to adapt all the activities related to taking trains which can be replaced by FrecciaRossa trains. Thus the new non-functional properties of the FrecciaRossa are compared with the ones of the old connection according to Bob's preferences, and if judged better the activity for booking and taking the new train is replaced for the old one. The new code will be used each time such an activity is executed, thus the application has actually evolved. Having an actual code update instead of a simple data update allows to change the logic of the application. For instance, the booking of the FrecciaRossa train may use a communication protocol that is different from the other trains.

Framework	Dynamic adaptation rules	Dynamic rule selection	Functional improvement	Non-functional optimization
Spanoudakis et al.[105]	−	+	+	+
Narendra et al.[86]	−	+	−	+
METEOR-S[115]	−	−	−	+
PAWS[5]	−	+	+	+
Our framework	+	+	+	+

Table 3.1: Features of frameworks for dynamic adaptation

Other scenarios for adaptation implemented in our prototype include renting a car if the train is cancelled and taking a taxi (instead of a bus) from Trento's train station to the meeting place if the train arrives late.

3.3.2 Related Work

Most of the approaches to adaptation found in the literature concern built-in adaptation, i.e., the adaptation logic is completely specified at design time. They concentrate on how to specify adaptation mechanisms and adaptable applications, exploiting different tools. For instance, the specification may be performed by extending standard notations (such as BPEL [87]) with adaptation-specific tools [66, 77], using event-condition-action like rules [9, 23], variability modeling [50] or aspect-oriented approaches [69]. Other work extends Software Architectures [90] to deal with adaptation, giving rise to *Dynamic Software Architectures* (DSAs) [71, 41]. Many research work focuses on the design and analysis of DSAs by applying formal methods [2, 14]. Other approaches to built-in adaptation instead define novel languages to specify structural reconfiguration aspects [46, 71, 112].

There is a main difference between the proposals listed above and ours: their adaptation logics are hard-wired into the application and defined at design-time, while we separate the running application from the adaptation logic, allowing to create and update the latter after application deployment (i.e., at runtime). Also, the proposals above are limited to a chosen language, while we propose a general approach applicable to different languages (although, clearly, we had to choose one for the implementation).

In the literature there are, however, proposals of frameworks for dynamic adaptation, all featuring an adaptation manager separated from the application. We make a comparison below, considering the following aspects: *(i)* whether the set of adaptation rules can be created and modified during the execution of the application; *(ii)* whether the choice of which rule to apply is static or dynamic; *(iii)* whether adaptation is aimed at changing the functionalities of the application or *(iv)* optimizing the non-functional properties. The results of the comparison are depicted in Table 3.1. Notably, all the listed approaches are in the service-oriented field.

In [105] the authors consider the problem of adapting the application by replacing malfunctioning services at runtime. The adaptation rule is fixed at design time, but it is dynamically applied by a *manager* component that monitors functional and non-functional properties, creates queries for discovering malfunctioning services and replaces them with dynamically discovered replacements.

[86] proposes an aspect-oriented approach for runtime optimization of non-functional QoS measures. Specifically, it allows to adapt non-functional properties of service compositions by changing the non-functional properties of their components. Here aspects replace our adaptation rules. They are statically defined, but dynamically selected.

The METEOR-S framework [115] supports dynamic reconfiguration of processes, based on constraints referring to several QoS dimensions. Reconfiguration is performed essentially at deployment-time.

PAWS [5] is a framework for flexible and adaptive execution of web service-based applications. At design-time, flexibility is achieved through a number of mechanisms, i.e., identifying a set of candidate services for each process task, negotiating QoS, specifying quality constraints, and identifying mapping rules for invoking services with different interfaces. The runtime engine exploits the design-time mechanisms to

support adaptation during process execution, in terms of selecting the best set of services to execute the process, reacting to a service failure, or preserving the execution when a context change occurs.

3.3.3 Outlook

The work above can be extended in many directions. First of all, a few aspects of the current approach need to be refined. One of those aspects is the description of the activity. For now, we just have syntactic matching of the descriptions, but one can consider more general descriptions, specifying the minimum requirements the activity should satisfy, e.g., exploiting some ontology. Another aspect that needs to be refined is the management of functional updates. At the moment, a functional update simply has the compulsory flag set to true. However, one needs to ensure that competing updates are managed in a safe way, avoiding, e.g., to replace old activities for newer ones. On the non-functional side instead, it would be interesting to relate the non-functional properties of the activities to the non-functional properties of the whole application, as to perform a more informed choice.

As a more long-term plan, we would like to target our approach towards stronger forms of evolvability, to allow deeper changes in the applications. An interesting possibility may be to combine the techniques studied in MECo, in order to obtain more complex transformations than just replacements of whole activities.

Chapter 4

Dynamic Evolution of Code

4.1 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. 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 [60, 61], 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 $\mathtt{C}_2$ with a new method `m`, this method will become available via objects of class $\mathtt{C}_2$ and its subclass $\mathtt{C}_3$. 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`. If we can type check that $\mathtt{C}_3$ implements `I`, it is type-safe to bind a pointer typed by `I` to an instance of $\mathtt{C}_3$ 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 instance 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 asynchronously and propagate local changes. 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 [12, 11, 6, 33, 49, 55] or by
- applying a global update or "hot-swapping" operation to the system [89, 76, 1, 13].

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

- disallowed [89, 76, 49, 13],
- delayed [1], or
- supported [109, 55].

For example, Hjálmtýsson and Gray [55] 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 [89] or modifying the Java virtual machine [76], use global upgrade

and do not apply to active objects. Automatic upgrades by *lazy global update* have been proposed for distributed objects [1] and persistent object stores [13], 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 [109], functional [11], and object-oriented [12] languages. In a recent upgrade system for (sequential) C [109], 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 [12] uses an incremental type system in which class versions are only typechecked 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.

4.2 Results

We developed a formal model for runtime evolution of code 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. Our work is based on the Creol language [60, 61], 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.

We propose a number of upgrade operations at the abstraction level of the modeling language in order to add and remove features from class definitions. A type system and an operational semantics have been defined for these operations. The developer of the original system does not need to accommodate later upgrades. Instead, upgrades propagate asynchronously through the distributed system. However, the asynchronous manner in which upgrades affect the system introduce some technical challenges to control the ordering of operations at runtime. For example, an object may try to invoke a method that has been introduced as part of an upgrade, before that upgrade has been applied. Another example is that fields can be accessed in old code after these fields have been removed from an object. In order to control this ordering, we have developed a type and effect system which derives dependencies between update operations as part of their type checking. These dependencies are used at runtime to constrain the local applicability of an update operation. Using this system, we show that the system with asynchronously evolving class definitions is type safe in the sense that *method not understood* errors cannot occur at runtime for well-typed programs. The type checking algorithm has been implemented in Ocaml and the operational semantics has been implemented in the Maude rewrite system. The results have been published in [59].

4.3 Outlook

As the ABS language gets settled, we intent 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. 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* [30] to the abstractions of the ABS language; i.e., classes, features and feature composition mechanisms.

Chapter 5

Autonomous System Adaptation

Software systems face ever increasing demands on complexity, volume, and speed of information processing. In light of the similarly increasing complexity and volume of software, and the transition to new concurrency-intensive approaches such as multicore/manycore, cloud computing, and ultra-large-scale (ULS) systems, current approaches to building, running, and managing software systems must be rethought [38].

Autonomous system adaptation, or self-adaptation [20], is a key component towards answering these challenges [21]. Self-adaptation concerns the problem of building systems with the ability to automatically adapt to changes in operating conditions, changes in the system itself such as the available hardware and software resources, and changes in requirements.

In this chapter we discuss the scope of self-adaptation in the context of the HATS project, present the results obtained so far in Task 3.1, mainly the identification of key scenarios and research issues towards the building of systems with self-adaptation capabilities, and finally we discuss their relations to planned and ongoing work in Task 3.1 on models, Task 3.4 on bytecode level evolvability and Task 3.5 on autonomous evolution, as well as to related European projects such as SecureChange and 4WARD.

5.1 Introduction

Jackson's WRSPM model [48] points to five main artifacts affecting software evolution:

- Domain knowledge (W), providing information about the environment in which the software product is to be executed.
- Requirements (R), reflecting users and customers needs and expectations.
- Specifications (S), the system specification prescribing design decisions and constraints such as the use of a specific product line architecture.
- Programs (P), the software product.
- The programming platform (M), including the abstract or physical machine platform on which the software is running.

Changes in any of the five artifacts may require evolution of the ultimate software product. Many such changes involve human intervention, e.g., to guide refactoring, to resolve ambiguities, or to make informed decisions affecting a redesign. Currently, there is no global consensus on the scope of automation in the software evolution process (cf. [20, 36]). In this chapter we take the first step to pin down the scope of automation in the software evolution process within the HATS framework, in particular to prepare the ground for future work on Task 3.5, on self-adaptation.

5.2 Results

In this section we present a state transition model of runtime software evolution, and develop a number of scenarios to help us determine the scope of self-adaptation within the HATS project.

To set the stage of discussion we assume we are working in the context of a large software system with multiple clients, application servers, and backend systems operating in an environment with constraints on performance, security, and robustness. The three case studies in HATS, presented in [95] (trading platform, virtual office, and the Fredhopper Access Server), all fit this picture very well.

We may assume that significant parts of the software are specified in ABS. Some analyses may have been performed of some parts of the software, but it is unrealistic to assume that anything even approaching complete, global specifications are statically available. Clients are outside our control. Probably they are web clients in practice, and the storage and backend systems are also to a large extent outside our control as well, governed by known, standardized or proprietary, possibly changeable interfaces.

Evolution of such a software system may take place at various levels. At one extreme, the system may be completely redesigned offline. At the other extreme, changes in the evolution artifacts are observed and acted upon at runtime, without developer or operator intervention as far as possible. It is the latter type of change which is at focus here.

5.2.1 A State Transition Model of Evolution

Runtime evolution is naturally modeled in terms of a state transition system. At any given time the state of the software system is determined by parameters such as the following:

Software state. Installed application programs, libraries, objects, classes, features, traits, monitors, aspects, and so forth, along with instance management information to handle versioning and feature updates.

Machine and network state. What processors are available, what network equipment is available, in which configurations?

Runtime state. How is the software configured, which threads are running on which physical machine, what is the state of each thread, which memory is allocated where, which objects reside in which cache, what is the state of queues, locks, and buffers, and so on.

Load state. What is the nature and volumes of data being received by the system? The data might for instance be valid application data, or it might represent an attack. Where is the data received? At which rate is the data received?

Requirements state. Which volumes of data are required to be produced by the system, where, and at which rate? Which authentication and access control mechanisms are available? Who is allowed to read and write what data, and who is allowed to invoke which operations?

Evolution steps are caused by changes in the software evolution artifacts such as:

- The software base changes because new features, traits, aspects, libraries, etc. become available, and old ones become defunct. This may happen to add new functionality, or in response to bug fixes.
- The machine and network state changes due to availability of new equipment, changing configurations, or breaking or unavailability of equipment, etc.
- Runtime state changes autonomously all the time due to operating systems, running code, and changes in load. This happens continuously. It also changes as a result of O&M (operations and management) functions changing configurations for some reason.
- Load state changes for external, not directly controllable reasons: The service becomes more popular, or less popular, is used only during Central European day time, or only during Christmas. The frequency and intensity of these changes is a bit harder to predict.

- Requirements change because of the customers needs, and due to the competition.

Except for load and run-time state changes which may happen at sub-second rates most of the above processes are slow, at the rate of hours or more.

Many of the above state changes are manual and can not easily be automated in the foreseeable future. This applies to many cases of software update, for instance where end users require some control over which features to select, to changes in requirements, and to some extent to the allocation and deallocation of hardware resources. Note, however, that progress in power aware computing and many-core computing makes it increasingly necessary to incorporate hardware management into the control loop.

Other state changes are not manual, but outside our control. This applies for instance to breaking equipment and changes in load.

Finally, some state changes are controllable, and this is where automation should focus:

- Automatic software updates, within a given product line, or relative to a given set of selected features.
- Selection, deselection, and configuration of software and equipment.
- Structure and management of the runtime state.

The overall goal is to devise provably correct, robust, and secure schemes which will maintain a rough equilibrium in the system over time, e.g., by automatically updating and reallocating the software base, or by allocating new hardware as appropriate to obtain for a given load state a runtime state which is sufficient to meet the requirements.

5.2.2 Scenarios for System Evolution

We elaborate a few scenarios for illustration.

Scenario 1. Adding a new simple, non-interfering knob
We add a new feature to the application which does not interfere with anything, only spends an extra amount of processor cycles. Once the knob is released it is probably safe to assume that it should be installed for all instances of the given product, or product line for which it is intended. Using a static time and resource analysis we may be able to determine in advance what the performance cost of the knob is. Since the knob does not interfere we can in principle add it and remove it freely, and tune its priority level, according to the current overall system performance.

The main support we would like from an autonomous evolution framework in this case would be:

- Automatic reallocation of tasks and resources based on static analyses, tests, or performance measurements, to meet performance objectives.
- Automatic monitoring of performance, for bottlenecks and overall systems utilization.
- Automatic adaptation and configuration to new available computational resources, if the existing resources are no longer sufficient to meet the performance objectives.

Scenario 2. Adding a more complex, interfering knob
Alternatively, the new feature interferes more deeply with the application, and may, for instance, introduce synchronization constraints, delays, and performance penalties that may be impossible to estimate reliably in advance. We may have analyzed and unit tested the new knob pretty exhaustively, and performed system level tests to some extent. Even so, global system behaviour is in this case unpredictable, and performance may be degraded in unpredictable ways.

The automated recovery actions we can foresee in this case – in addition to Scenario 1 – would principally be automated rollback to throw out features that cause the requirements to be violated. This, however, is not easily realized in general:

- Features depend on each other. Removing one feature may cause an avalanche of rollback operations in systems that depend on the feature being removed.
- Properly identifying a non-performing component requires a root cause analysis which may not be feasible.
- The standard approach to rollback uses versioning. But indiscriminate use of rollback risks doing more harm than good, unless directed by a root cause analysis.

Other scenarios related to performance and security management are listed below.

Scenario 3. Resource upgrades/downgrades
Operations and maintenance might impose various policies that affect the underlying network and processing resources available. For instance, a policy might state a goal of systems utilization averaged over some given time interval to stay in the interval 20-50%. An autonomous reconfiguration service might automatically determine that processor or network resources be switched in and out of operation, and the active part of the system automatically adapted, as a result of monitoring systems utilization.

Scenario 4. Changing requirements
A 10ms latency requirement (formulated precisely in some way) for a stock trading system might be sharpened to a 1ms latency requirement. An autonomous reconfiguration service might attempt to adapt to this by deploying new hardware and automatically redistribute load.

Scenario 5. Security patching by monitor inlining
A vulnerability is discovered in a subsystem. To ensure survivability until a patch is developed a monitor could be automatically be derived from the description of the vulnerability and inlined into the relevant applications. Once a patch arrives, the inlined monitor is deactivated.

5.3 Ongoing Work and Plans

We situate our study in an abstract setting based on a collection of decentralized ABS software units which communicate by message passing and execute on top of a network of processing nodes. Except for a fixed set of IO ports the units will be oblivious to the executing node, network addresses, and network topology. This allows the software units to be easily shipped between nodes in a transparent manner.

Our approach uses neighbor-neighbor interactions in the style of gossiping [67, 121, 119] to monitor performance and systems utilization at local, region, and system-wide levels, and to support inter-unit message routing. The key idea is to use performance and utilization statistics to compute aggregate load statistics which will be used, at appropriate timescales, to relocate software units between nodes, and to cause nodes to be powered up and down as required, e.g., by the current systems level performance/utilization ratio. The central issues to be explored are:

- Can such an approach be realized in a transparent manner?
- Which aggregates are needed/most useful, and how is aggregation to be done?
- How is routing to be done?

- How are load statistics to be computed?
- What can be said about the overall performance of such a system?
- What can be said about scalability?
- How well does it perform in practice?

5.3.1 Towards An Autonomous Adaptation Framework

The realization of the autonomous adaptation framework uses a layered approach involving the following components:

ABS execution engine. A basic abstract machine is needed for execution of the software units. The abstract machine must be able to handle scheduling, start, stop, serialization and transmission to neighboring nodes of the software units. The ABS execution engine relies on results on systems derivation and code generation from HATS WP1.

Performance/utilization monitoring.

Allocation of software units to nodes depend on runtime components for performance monitoring. Such a performance monitoring layer is responsible for monitoring per node and per unit loads, and for aggregating these loads. It is used by the load balancing layer for redistribution of software units and for starting and stopping of processing nodes.

We are currently designing a robust and scalable in-network monitoring layer in the context of the EU FP7 project 4WARD. Both tree-based and gossip-based approaches are explored. We refer to [94] for a state-of-art survey, and to [121, 120] for recent results on robustness and distributed threshold detection using gossiping. This work forms the basis for the performance/utilization monitoring layer in Task 3.5.

Behavior/security monitoring. To secure the interaction between trusted and untrusted code, as well as the interaction between trusted code and legacy code, a behavior/security monitoring layer will be used. This is essential, also, for adaptation to changes in security requirements, cf. [36]. The monitors will ensure that RPC and communication events follow the agreed protocol, and that only allowed events go through. In the context of Task 3.4 we are examining monitor inlining as an implementation strategy for security monitors and have improved our understanding of the potentials and fundamental limitations of inlining [24, 25] for multithreaded Java bytecode. This includes proof-carrying code as an approach to secure, trustless interaction in a multi-domain setting [26].

Load balancing. The load balancing layer maintains for each unit a collection of load estimates which determines when and in which direction an executing software unit should be shipped in order to maintain a good distribution of load. The development, modeling, and analysis of such a load balancing layer is one of the major challenges of Task 3.5. The theoretical starting point is recent progress on game-oriented models for load balancing in distributed systems, cf. [10].

Software unit redistribution. A redistribution layer will use the node level runtime to stop, serialize and transmit an executing software component to a neighboring node in a direction determined by the load balancer. To obtain good performance, transportation of software units must be transparent to the message passing layer, i.e. guarantee that messages are correctly received, at least with high probability, even if the receiving agent is being relocated.

Addressing, routing, message forwarding, and interunit communication. Since software units are free to travel over the network, mechanisms are needed for addressing, routing, and message forwarding, for inter-unit communication. In the long term, dedicated routing protocols will be needed for this purpose. Within the context of the HATS project, however, we use only standard IP-based communication for internode, link-level interactions.

Adaptation and fault management. Software units communicate by message passing and RPC. Adaptation and fault management is needed in the error handling component to manage faults that arise when a software unit is removed and messages no longer find a recipient, and when nodes go down and entire collections of software units need to be recovered and redistributed.

Power management. Power management is the key tool used by the runtime system to adapt to changes in utilization or performance requirements. This component powers nodes up or down as determined by output from the performance and utilization monitoring component.

5.3.2 HATS Realization

Within HATS WP3 the work towards such an autonomous adaptation framework focuses on algorithmics, modeling, and semantics in the following areas:

1. A collection of provably correct algorithms for load balancing, performance monitoring, and network and node management. This work will take place in the context of Task 3.5.

2. A formal, abstract model of the autonomous adaptation framework including abstract models of the components listed in section 5.3.1. Work on developing such a model is currently going on. The high level model accounts for software units in an abstract form which is oblivious to the network execution model. A more concrete model accounts for execution on top of an abstract network model. This network machine models basic machine execution, and includes associated network primitives for message passing, power management, and software unit mobility. The goal for Task 3.1 is to show the model correct with respect to the abstract semantics.

3. This model will provide the basis for a concrete instantiation, in Task 3.5, of the framework in terms of concrete representations of networks, nodes, and software units usable in the ABS framework.

4. Finally, we will develop a simulation framework to allow us to perform experimental evaluations of the algorithms and the architecture.

Chapter 6

Test System Evolution

6.1 Introduction

In this chapter we identify technical concerns which should be addressed when defining theories and techniques for test system evolution in the HATS methodology. This is achieved by first identifying key phases in the methodology that require testing. We then identify technical concerns pertaining to the development of an evolvable test system across several industrial contexts. Based on these general concerns we also identify specific technical concerns about the evolvability of various levels of testing: unit, integration, system and stress testing. For each level of testing we give an overview of related research in the context of model-based testing [27, 34, 92, 113].

6.1.1 HATS Methodology

The HATS methodology is the selected formal development method that supports software product line engineering processes [80, Figure 2]. It supports the specification, generation and quality assurance of reusable artifacts [95, Requirements MR3, MR5 and MR6, Section 2.1]. Since the HATS methodology has to support existing industrial application engineering approaches [95, Requirements MR2, Section 2.1], two complementary activities, verification and testing, are considered during validation phases. These activities provide quality assurance on artifacts in the product line as well as its family members. Specifically in this chapter we focus on the testing activities in the methodology.

Model Mining

In a software product line, it is often necessary to derive variabilities and construct generic artifacts from legacy systems. Model mining can support this process [52, Page 42]. However, model mining is extremely difficult to automate in practice, and thus requires human intervention. As a consequence, the mined model might contain errors. It is therefore important to carry out conformance testing as part of the development effort to ensure both the system and its model exhibit the same behavior. In the HATS methodology, model mining is carried out during the *Product Line Planning and Scoping* phase, in which high level requirements for the product line are harvested from existing software implementations.

Validation

Automatic code generation from formal models into concrete implementations has proven to be a challenging issue [92]. The process of generating a target implementation from formal models normally requires human intervention. Moreover, in industrial software development, software systems derived from the models are often deployed under various hardware and resource constraints as well as integrated with legacy and third party software components. Therefore testing is required as part of the development method. In particular, it is carried out in *Generic Component Validation* and *System Validation* phases.

Model-based Testing

In model-based testing (MBT) [27, 34, 113], one generally derives test cases from a formal as well as informal *test case specification* and a model of the Application Under Test (AUT). Models used for test case generation are constructed using a variety of modeling notations and formalisms. These include UML diagrams [18, 51], Kripke structure, Finite state machines [73, 107], process algebras [122] and state-based modeling languages such as Z [3, 108] and the B-Method [72, 99].

Similar to code generation, however, it is extremely difficult to automate test case generation. As a result the HATS methodology assumes human intervention in test case generation. It is therefore important to record generic artifacts about test cases alongside generic component artifacts in a product line so that they can be reused during application engineering processes.

Product line requirements often change in order to either keep up with the market demand or include new customer-specific requirements. Whenever there is a change in product line requirements, both the components and the associated test systems in the artifact repository as well as the family members may have to be updated correspondingly.

6.1.2 Evolvable Test System

In this section we concisely describe what a test system is and subsequently what an evolvable test system is. These are then used throughout the chapter.

A test system is a collection of tests to evaluate the Application Under Test (AUT)'s compliance with its specified requirements. Each test (or test case) in a test system is therefore used either to increase confidence that the AUT corresponds to the requirements, or to prove that it does not [92].

A test in a test system is either executed automatically or manually. An automatic test is a piece of program codes that automatically exercises the AUT's functionalities and makes assertions on their observed results. A manual test is usually a document describing a sequence of steps that a tester would follow to interact with the AUT. The tester evaluates the observed results based on either a defined set of expectations provided by the document or his/her general experience.

As a software system can change in the dimensions defined in Section 1.1, its test system must also evolve with respect to the dimensional changes. An evolvable test system should first and foremost respect the consistency properties maintained during the AUT's evolution. In software evolution, consistency is a general safety property about the evolution. For example an evolution of a software system is consistent if its components remain interoperable. Moreover, evolvable test systems help minimizing throw-away code and hence encourage code reusability.

Definition 6.1.1 (Evolvable Test System)**.** *Given an evolvable software system, a corresponding evolvable test system is a set of test specifications that evolves consistently with respect to the the evolution of the software system in the dimensions defined in Section 1.1.*

6.1.3 Goal

The goal of this chapter is to highlight concerns that help achieve an evolvable test system in both *single-system* engineering and *product line* engineering (PLE). We present use case scenarios based on the industrial software system from Fredhopper and its associated test system to illustrate these concerns.

6.1.4 Structure

The structure of this chapter is as follows. Section 6.2 briefly describes the structure of the test system used at Fredhopper. In Section 6.3 we identify technical concerns pertaining to the general development of an evolvable test system across several industrial contexts. In Section 6.4 we identify specific technical concerns concerning the evolution at each level of testing. When considering each level of testing we give an overview of related research about model-based testing. We provide a short summary and conclusion in Section 6.5. An overview of the Fredhopper product is given in Appendix A.

6.2 Fredhopper Test System

In this section we give an overview of the test system implemented and deployed by Fredhopper. We also summarize high-level requirements on software evolution harvested during the requirement elicitation process [95].

6.2.1 Fredhopper Test System: An Overview

At Fredhopper, the test system consists of the following testing procedures:

Automatic unit testing During the development of a component, unit tests are written to validate the correctness of the unit and to detect regressions. A unit test exercises a unit of functionality in a system and makes assertions about the state of that system after the unit's execution. Unit tests are written as programs. They are executed automatically when the component containing the units of code changes. To test units individually, unit tests have to make assumptions about the state of the system, that is, the preconditions of the tests.

Automatic integration testing After a unit is tested, it can be integrated with the rest of the system using some glue codes. An integration test exercises the functionalities of the integrated units on a system and makes assertions about the state of that system after the execution of those functionalities to ensure that the units still perform correctly. Similar to unit tests, integration tests are executed automatically when the code of one or more integrated units changes. Similar to unit testing, to test integrated units independently, integration tests have to make assumptions about the state of the system, that is, the preconditions of the tests.

System testing System testing evaluates the overall behavior of the system. System tests are usually driven by the use case scenarios (test cases) constructed during the functional analysis of the system. These tests may either be automated or executed manually. Manual tests are performed by testers, who follow detailed test cases while testing the various functionality of the system. Automatic tests are machine-readable and are performed by a program, which executes the sequence of steps described by the test cases. Note that the test criteria at this level consists of user expectations.

Stress testing Stress tests evaluate the robustness of a system in environments that are more demanding than the system would normally experience. Stress tests collect measurements on the latency and throughput of the system during the tests and check if they fall within expectations.

6.2.2 Selections of Requirements

During the HATS project's requirement elicitation process, we have established three concerns that are relevant to test system provisioning [95, Concerns FP-C8, FP-C9 and FP-C10, Section 5.2.3]. This section gives a brief overview of these requirements. For more detailed descriptions and illustrations of the these requirements, we refer readers to the full report on the requirement elicitation process [95].

Test case generation (FP-C8) We have identified, via concrete examples, that the development of test systems should be based on a formal specification of the AUT. A formal specification of the AUT provides an unambiguous description of the behavior of individual components and their interactions, and it also serves as the model of the AUT for model-based testing (MBT) [34]. A model-based approach allows for the definition of precise test criteria and can hence produce tests that have better coverage and less false positive results. Note that we envisage cases in which automatic test generation might not be possible. In this case the HATS framework should incorporate usable techniques and tools to *guide* manual test generation. The corresponding concern is

Modeling behavioral changes (FP-C9) We have identified the importance of capturing component-wise both structural and behavioral changes of the AUT. To capture these changes, we believe it is necessary to develop a formal compositional technique that can systematically capture both structural and behavioral changes of (the model of) the AUT, and detect *inconsistencies* of changes.

Test system evolution (FP-C10) While the ideal situation is to be able to prove *consistency* of the AUT's component's structural and behavioral changes with respect to its formal specification, this might not be tractable in practice. One possible solution is to provide a formal compositional technique to change the AUT such that consistency is guaranteed. This requires *a theory of software evolution.*

Note that Requirement 1 gives rise to a formal approach towards MBT. A model-based approach encourages the systematic development of test systems that are consistent with the model of the AUT. We believe this is an essential prerequisite to test system evolution. Requirements 2 and 3, on the other hand, concern the evolution of the AUT rather than test systems. This is intentional as we believe a tractable and systematic approach to the evolution of (the model of) the AUT provides a formal basis to investigate evolution of test systems.

6.3 General Concerns

6.3.1 Introduction

This section identifies technical concerns that are fundamental to the development of a formal engineering method for producing an evolvable test system and which are applicable to industrial software development. We first consider test system evolution in a general software development context. In Section 6.3.2, we consider test system evolution in the context of industrial applications. The evolution of individual levels of testing are considered in Section 6.4. Note that some of the concerns raised in these two sections may be overly ambitious, and may not be addressed in this project. However, we envisage that most concerns are addressed by technical contributions from Work Package 3 of the HATS project.

Throughout this section we use *use cases* and their corresponding *scenarios* to illustrate the areas of concerns. We first relate use cases and scenarios using the following definitions adopted from Pohl et al.'s [91] and Rumbaugh et al.'s definitions [98].

Definition 6.3.1 (Scenario)**.** *A* scenario *is a sequence of actions that illustrate the interaction with a system. A scenario may be used to illustrate an interaction with the system and/or an execution of a use case instance.*

Definition 6.3.2 (Use Case)**.** *A* use case *is a specification of behavior of a system illustrating all possible sequences of interactions with the system. A use case may be instantiated as scenarios for more concrete descriptions.*

The following use case and scenario, identified during the requirement elicitation process [95, Step 3 of Scenario FP3 in Section 5.2.3], illustrate issues surrounding the evolution of the AUT as well as its test system.

Use Case 6.3.1. When adding a new feature to a component, new tests should be written to specify and validate the correctness of the feature. Similar to existing unit tests, new tests are implemented based on informal documentation of the component. Existing test systems may also be changed depending on whether or not the new feature changes other existing behavior of the component. After the test system has been changed, the component with the new feature must pass through the test system pipeline before being considered as an official component of the Fredhopper Access Server (FAS).

Scenario 6.3.1. An update is included in the forthcoming release of the Targeting Diagnostic Tool[1]. This update adds a graph visualization feature to the tool so that the relationship between different rules submitted

[1] A description of the diagnostic tool and its relation to the Fredhopper product is included in Appendix A

to the rule engine may be viewed as a graph. This feature should not change the existing behavior of the diagnostic tool. Therefore it is necessary for the diagnostic tool to pass all existing unit tests before this update is approved.

Assuming that we have a machine-readable specification of the component (diagnostic tool) and the AUT, and we also have the associated test system derived using a model-based approach, we have to deal with the following question:

> How can this approach be extended to systematically capture and reason about both the structural and the behavioral changes to the AUT due to an update (addition or modification) of one of its components?

This has an impact on the evolution of test systems, especially in a MBT approach in which the test system is heavily depended on the model of the AUT. We formulate the following general concern.

Concern (TSE-C1)**.** ***Model Evolution*** *How does the HATS methodology facilitate a MBT approach for evolving test systems?*

6.3.2 Industrial Applications

The HATS project aims at using industrial case studies to drive the development of techniques, policies, and tools. Therefore, it is important to address test system evolution in the context of industrial *best practices*. Specifically we highlight concerns about the development of evolvable test systems in the following technical areas.

Product Line Engineering (PLE) The HATS methodology aims at providing formal techniques and tool support for software product line engineering processes. As a result, test system evolution should be considered in the context of PLE.

Integrating Third Party Libraries In industrial software development environments, third party libraries are integrated into software products to both enhance functional features as well as reduce the cost of implementation. For example, the FAS uses many open source libraries from the Apache project group[2]. It is therefore essential to consider test system evolution in the context of integrating third party libraries. This concern has been identified during the requirement elicitation process [95, Concerns FP-C13, FP-C14, FP-C15, FP-C16, FP-C17, FP-C18 and FP-C19, Section 5.2.5].

Industrial Tools Support In order for any development methods and approaches to be incorporated into standard industrial software engineering practices, the associated technology must be made more accessible to software engineers and developers. It is therefore necessary to consider the development of tool support for test system evolution. This requirement has been identified during the requirement elicitation process [95, Requirement MR18, Section 2.3.4].

Product Line Engineering

While a single specification of AUT might be sufficient for single system engineering, the HATS framework aims to extend MBT to software product line engineering. This means that the specification of the AUT must be component-wise decomposed such that each component is associated with a *reference* specification[3] that contains both commonalities and variabilities of the component. The variabilities in the specification are then resolved by binding consistent variants across the entire AUT. This difference between constructing specifications at the family and application engineering level in a product line has an impact on how a test system can be generated and evolved using a MBT approach.

[2] `http://www.apache.org`

[3] A reference architecture is delivered during the design phase in the family engineering process [91, Chapter 11]. In a model-based approach, one expects a corresponding reference specification, from which a complete specification can be derived for each family member during application engineering.

Concern (TSE-C2)**.** ***Product Line Engineering*** *How does the HATS framework achieve MBT in product line engineering? In particular is it possible to derive a reference test system (artifacts) from the reference specification, which contains variabilities that are only resolved during application engineering? How does the HATS framework ensure that the variability binding between the reference test system and the family member test system is* consistent *with the variability binding between the reference specification and the family member specification?*

When considering the evolution of test systems in PLE, it is important to understand that variabilities of both the AUT and the test system are both temporal and spatial. Specifically it is common in PLE for each family member to use a different variation of a component. This could potentially implicate a different variation of the component's test system. The evolution of the component's domain artifacts would then implicate evolution across all variations of the component's test system.

Concern (TSE-C3)**.** ***Product Line Engineering: Test System Evolution*** *How does the HATS methodology facilitate a MBT approach for evolving test systems in PLE?*

Another concern that should be taken into account when considering a model-based approach to an evolving test system for product line engineering is the choice of testing strategy. In contrast to single system engineering, testing in product line engineering has to consider variability of the specification as well as the differentiation between family and application engineering processes. For existing strategies, readers may refer to Pohl et al.'s book [91, Pages 272 - 280]. Here we highlight the need to consider the effect of different strategies to MBT in the HATS framework.

Concern (TSE-C4)**.** ***Testing Strategy*** *How does the HATS framework cater different testing strategies in product line engineering? And how does the framework characterize the effect of each strategy on MBT and developing evolvable test systems? Essentially, what effects do different strategies have on how family member test systems are derived from the reference test system? Furthermore, how can changes to the test systems at both levels be reconciled?*

Third Party Libraries

During the development of the FAS, third party libraries are used to provide new features at a lower cost. For example, the FAS uses a third party rule engine to support business rule inferences[4]. We illustrate concerns of applying and testing third party libraries using the following use case and scenario, which has been identified during the requirement elicitation process [95, Step 1 of Scenario 5 in Section 5.2.5].

Use Case 6.3.2. We first identify specific information about the third party library that would be used: its version, the public interface methods it exposes and the (informal) documentation on the range of values and types of the input arguments and return values of its methods. We then implement the logic for transforming between the internal FAS data structure and the one which is used by the library. We finally construct a test system for both the library as well as the transformation logic (glue code).

Scenario 6.3.2. We use a third party library for visualizing the graph structure of FAS business rules in the rule engine during diagnosis. We identify the version of the library, the library's API for laying out nodes of the graph and for adding nodes and edges to the graph. We then implement the logic for transforming FAS business rules into nodes of the graph as well as for rendering the graph after construction. We finally implement unit tests to assert the correct transformation and rendering.

The following concern highlights the need to consider the integration of third party libraries during the development of the test system.

Concern (TSE-C5)**.** ***Third Party Libraries*** *What features does the HATS framework provide for extracting usable (partial) specifications from third party libraries? And how does these specifications, together with a theory of software evolution, help inferring the consistent evolution of the libraries as well as the (model-based) test systems generated from the specification?*

[4]FAS uses JBoss Drools Expert (`http://www.jboss.org/drools/`) as the implementation of the Rete network [42].

Possible Solution: Model Mining To generate test cases and test specifications for third party libraries, one can consider using existing model mining techniques. Model mining from legacy and black box components is an emerging subject. At present there are two main strategies for mining specifications: For a legacy component, its source code can be analyzed [118], while for a black box component, one can analyze observations about the run-time behavior of the component [4]. Existing work on model mining include Flanagan et al.'s [40] and Logozzo et al.'s [75]. The former provide the mechanism for assisting users in generating JML specifications from existing JAVA programs; the latter use the Clousot analyzer to infer pre-/postconditions for methods given in bytecode for the .NET framework.

Tool Support

For a formal engineering methodology to be applicable to traditional software engineering at industrial level, there must exist tool support to assist the formal development throughout the software life-cycle. In terms of unit test generation, for example, it would be beneficial to provide the necessary tooling for assisting the mechanical generation of unit tests [35]. Furthermore, these tools must be compatible with the current technology as well as easily adopted by traditional software engineers in the industry. These conditions are in accordance with the methodological requirements harvested during the elicitation process [95, Section 2.3.4]. The following concern highlights these requirements.

Concern (TSE-C6)**.** ***Tool Support*** *Does the HATS framework provide tool support for evolving test systems that can be easily integrated with existing technology such as integrated development environments (IDE)?*

One of the problems a company might face when transitioning to a model-based approach is how to derive high-level specifications from legacy systems.

Concern (TSE-C7)**.** ***Model Mining*** *If (part of) the AUT is a legacy system, how does the HATS framework use the current code base of the AUT as well as its associated test system to generate component-wise machine-readable specifications? Moreover, with such specification, how does the HATS framework facilitate the necessary tool support for evolving the test systems of the legacy system?*

Possible Solution: Versioning When developing tool support for both code and test system generation, one can also consider the reusability of the code base of the AUT and the test systems as well as traceability of changes that have been made to them. This is especially important when considering a methodology that is in line with PLE. Current research suggests the use of versioning systems as the starting point [96, 97]. However, the information contained in versioning systems such as Subversion[5] is not detailed enough. In particular, the historical information in versioning systems is only recorded at the explicit request of the programmers. Furthermore, the *semantics* information, essential for reasoning about software evolution [97], is not recorded. As a consequence, Robbes et al. [97] recommend a change-based approach to the software evolution analysis, in which they leverage existing IDEs to record all semantic changes at both method and class level performed on the application.

6.4 Specific Concerns

In this section we highlight technical concerns with respect to the evolution at different levels of testing. Moreover, for each level of testing we provide an overview of related research in model-based testing.

6.4.1 Unit Testing

Related Research

A lot of research work in model-based testing (MBT) has focused on application testing in which entire systems are tested as a single unit. According to Dias Neto et al.'s study in 2007 [27], 48 out of 72 approaches

[5] http://subversion.tigris.org/

are concerned with application testing, and only 8 of them are concerned with unit or component testing. This is mainly because units are generally considered as white boxes and hence are not situated at the usual level of abstraction for MBT. Moreover, while there exist other approaches that focus on structural testing of source code, it has been theoretically shown that white-box testing based on path exploration on the structure of the source code does not scale [79]. Nevertheless there are MBT approaches focusing on unit testing, including Kim et al.'s [68] and Offutt et al.'s [88].

Kim et al. [68] used UML State Diagrams to specify both control and data flow coverage criteria for unit testing of classes. They devised a method to generate test cases to satisfy these coverage criteria. This method includes transforming State Diagrams into Extended Finite State Machines (EFSMs). From EFSMs, flow graphs are identified, which are then used to generate test cases satisfying the coverage criteria of the state diagrams.

Offutt et al. [88] defined a method for generating tests from formal specifications using the modeling language SOFL [74]. In particular, they applied mutation analysis to evaluate coverage criteria of the test cases generated using their method for unit testing. Note that neither Kim et al.'s nor Offutt et al.'s approaches offer any tool support.

Industrial Applications

In this section we identify high-level concerns specific to unit test evolution from Fredhopper's perspective.

Concern (TSE-C8)**.** ***Test-unit Relationship*** *What is the relationship between an evolutionary step at the model level of the unit and the corresponding step for its unit test?*

Concern (TSE-C9)**.** ***Reusability*** *How can one maximize code reusability between an evolutionary step?*

Concern (TSE-C10)**.** ***Complete Specification*** *In a white box scenario, can one assume the possibility that unit tests may be determined completely by the specification of the unit? If so, is it possible to solely consider the evolution of the specification and not properties of the tests themselves such as code reusability since the codes in unit tests are automatically generated?*

Concern (TSE-C11)**.** ***Partial Specification*** *If a unit's specification is partial, such as those from a third party library, then how can the evolution of its unit test be characterized with respect to changes to the library?*

6.4.2 Integration Testing

Related Research

According to Dias Neto et al.'s study [27], 17 out of 72 approaches are concerned with integration testing. Many of these approaches consider the concerns of test selection and test oracle determination for integration tests.

While generating test cases for unit testing, the MBT approach can exploit contract-based specifications of units to generate test oracles. However it is extremely difficult to compose unit specifications to derive usable contracts for generating test oracles [19] for integration testing. Chen et al. [19] propose a possible solution to this. They consider using SOFL to model a given integration and applying data flow path coverage techniques to generate test oracles. Another MBT approach related to test selection for integrated testing is Sokenou's [104], whose proposal combines the message sequences information from UML Sequence Diagrams of the integrated components with states information from UML State Diagrams of individual components to derive test cases and oracles. Note that neither approaches offer any tool support, nor do they consider scalability issues with respect to integration testing.

Industrial Applications

In this section we identify high-level concerns from Fredhopper's perspective which are specific to integration test evolution.

Concern (TSE-C12). ***Compositionality*** *Is it possible to compositionally characterize the relationship between an integration test and the unit tests of the units that are being integrated? For example, if operation A has unit test T_A and operation B has unit test T_B, then what is the relationship between $T_{A||B}$ and T_A and T_B given a system defined by some composition $A||B$[6]?*

Concern (TSE-C13). ***Bounds of Compositions*** *What is the bound on component composition for integration testing? For example, it might be very difficult to characterize integration testing of components that are composed in parallel, e.g., when components are executed in separate threads of execution in a multithreaded system.*

Concern (TSE-C14). ***Incremental Construction*** *To establish the relationship above, what is the preferred approach to allow the incremental construction of integration tests?*

Concern (TSE-C15). ***Code Coverage*** *Assume we can specify the coverage criteria and the test specifications of A and B, how do we calculate the coverage criteria of the composition $A||B$? How do we generate test cases that satisfy these coverage criteria?*

Concern (TSE-C16). ***Characterization*** *Assuming it is possible to characterize the evolution of $A||B$ in terms of the evolution of A and B independently, can the same characterization be extended to cover the evolution of $T_{A||B}$?*

Concern (TSE-C17). ***Methodology*** *Assuming we could characterize the relationship above, does the HATS framework offer a methodology to guide the construction of integration tests and the tool support to assist such construction?*

6.4.3 System Testing

Unit testing and integration testing are usually considered as white box testing and are thus used to test both internal and external behavior of individual units as well as their composition. System testing, on the other hand, is designed to test external behavior of the complete AUT by interacting with it via its public (user) interface.

Related Research: Formal Foundations of Black Box Testing

Model-based testing focuses on generating test cases from abstract behavioral models. As consequence the majority of related research focuses on application/system testing. According to Dias Neto et al.'s study [27], 48 of 72 approaches are concerned with application testing. These approaches use models ranging from UML [82] to formal languages such as temporal logic [111] and Z [3] for specifying behaviors of the AUT.

Note that approaches, such as Tan et al.'s [111], apply specification techniques that have been extensively studied in the field of formal verification. In particular, Tan et al. use Linear Temporal Logic (LTL) for behavioral specification. Coupled with model checking facilities it has been shown to be possible to *automatically* generate test cases in the form of traces. As consequence a plethora of techniques have been developed in this direction [44].

Nonetheless, MBT approaches sketched so far assume certain types of information about the AUT. These include the number of states of the AUT [111] or order at which functions are invoked in the AUT [82]. These types of information can be captured with the appropriate level of abstraction using existing modeling languages and used for generating test cases. However, for *complete* black box testing such information is not available and so these assumptions cannot be made. Therefore, an alternative approach is required to determine test coverage and oracles.

One such approach is given by Meinke [78, 79]. Meinke provides a formalization of the coverage problem in black box testing [79]. Specifically he defines a stochastic calculus to calculate the probability that a

[6] We write $A||B$ to denote a general composition of A and B, which is not necessary parallel.

black box is correct with respect to a contract-based specification (Hoare triple [56]) after executing a finite number of (black box) tests on it. This calculus can also be applied with functional approximation [78] to provide a model for test cases generation.

Industrial Applications

At Fredhopper, application testing currently consists of the execution of test cases either manually by a tester or automatically by a testing program. Each test case consists of a sequence of interactions with the public interface of the AUT. Below we identify high-level concerns specific to application test evolution from Fredhopper's perspective.

Concern (TSE-C18)**.** ***Specification of Black-box Components*** *Is it possible to extract specifications of the external (reactive) behavior of the AUT from specifications of its components?*

Concern (TSE-C19)**.** ***Consistency*** *Given a specification of the external behavior of the AUT, is it possible to derive application tests that are consistent with both integration and unit tests?*

Concern (TSE-C20)**.** ***Compositional Evolution*** *Can the evolution in the unit and integration tests trigger the evolution of application tests? That is, can tests be constructed and evolved compositionally?*

6.4.4 Stress Testing

Stress testing is concerned with two major properties of the AUT: its robustness when running in environments that are more demanding than the AUT would normally experience, and its performance such as latency, throughput and resource usage. Specifically the former property deals with the issue whether the AUT performs functionally in a more demanding environment while the latter property is more concerned with the resource guarantees and service level agreements of the AUT. Note that stress tests for the former property usually also increase the code coverage of the test system as it accesses states of the system that are rarely reachable when operating in a normal environment. For example, certain states may only be reachable under the denial of service attack.

Related Research

Existing research in model-based testing mainly focus on functional correctness. A related but orthogonal area of research is model-based software performance prediction [8]. This area of research considers the development of suitable performance models of the AUT as well as the evaluation (testing) of the AUT against these models. Modeling languages used in software performance include queueing networks, stochastic Petri nets, stochastic process algebra, and simulation models.

Industrial Concerns

Below we highlight concerns specific to stress test evolution from Fredhopper's perspective.

Concern (TSE-C21)**.** ***Resource Specification*** *Given a resource specification of the AUT, is it possible to determine the specific conditions that should be covered by its stress tests?*

Concern (TSE-C22)**.** ***Code Coverage*** *Given a specification of the AUT, is it possible to determine a set of stress tests that achieve higher code coverage than those in integration and application testing?*

Concern (TSE-C23)**.** ***Resource Guarantees*** *Given a specification of the AUT, is it possible to determine the resource guarantees that stress tests should assert on the AUT?*

Concern (TSE-C24)**.** ***Compositionality*** *How does the evolution of unit, integration and application tests trigger the evolution of stress tests?*

6.5 Summary

By considering testing activities in the HATS methodology, existing research in model-based testing and industrial experience at Fredhopper, we have identified high-level technical concerns surrounding test system evolution in the HATS methodology. We first summed up the test system implemented and deployed by Fredhopper and the high-level requirements concerning both software and test system evolution. We then considered general issues surrounding test system evolution and their implications on industrial applications. We also considered individual levels of testing independently. For each level of testing, we studied related research in the area of model-based testing and raised specific technical concerns pertaining to the development of an evolvable test system in an industrial software development environment. Table 6.1 gives a summary of the concerns identified for test system evolution.

Concern Identifiers	**Concern Labels**	**Reference**
General Concern		
TSE-C1	Model Evolution	Page 32
Product Line Engineering		
TSE-C2	Product Line Engineering	Page 33
TSE-C3	Product Line Engineering: Test System Evolution	Page 33
TSE-C4	Testing Strategy	Page 33
Third Party Libraries		
TSE-C5	Third Party Libraries	Page 33
Tool Support		
TSE-C6	Tool Support	Page 34
TSE-C7	Model Mining	Page 34
Unit Testing		
TSE-C8	Test-unit Relationship	Page 35
TSE-C9	Reusability	Page 35
TSE-C10	Complete Specification	Page 35
TSE-C11	Partial Specification	Page 35
Integration Testing		
TSE-C12	Compositionality	Page 36
TSE-C13	Bounds of Compositions	Page 36
TSE-C14	Incremental Construction	Page 36
TSE-C15	Code Coverage	Page 36
TSE-C16	Characterization	Page 36
TSE-C17	Methodology	Page 36
System Testing		
TSE-C18	Specification of Black-box Components	Page 37
TSE-C19	Consistency	Page 37
TSE-C20	Compositional Evolution	Page 37
Stress Testing		
TSE-C21	Resource Specification	Page 37
TSE-C22	Code Coverage	Page 37
TSE-C23	Resource Guarantees	Page 37
TSE-C24	Compositionality	Page 37

Table 6.1: Concerns identified for test system evolution

Bibliography

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

[2] R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *Proc. of FASE'98*, volume 1382 of *LNCS*, pages 21–37. Springer, 1998.

[3] P. Ammann and J. Offutt. Using formal methods to derive test frames in category-partition testing. In *Proceedings of 9th Annual Conference on Computer Assurance*. IEEE Computer Society, 1994.

[4] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. *ACM SIGPLAN Notices*, 37(1), 2002.

[5] D. Ardagna, M. Comuzzi, E. Mussi, B. Pernici, and P. Plebani. PAWS: A framework for executing adaptive web-service processes. *IEEE Software*, 24(6):39–46, 2007.

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

[7] I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. *ACM SIGPLAN Notices*, 40(10):265–279, Oct. 2005.

[8] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: a survey. *IEEE Transactions on Software Engineering*, 30(5), May 2004.

[9] L. Baresi, S. Guinea, and L. Pasquale. Self-healing BPEL processes with dynamo and the jboss rule engine. In *Proc. of ESSPE'07*, pages 11–20. ACM Press, 2007.

[10] P. Berenbrink, T. Friedetzky, L. A. Goldberg, P. W. Goldberg, Z. Hu, and R. A. Martin. Distributed selfish load balancing. *SIAM J. Comput.*, 37(4):1163–1181, 2007.

[11] G. Bierman, M. Hicks, P. Sewell, and G. Stoyle. Formalizing dynamic software updating. In *Proc. 2nd Intl. Workshop on Unanticipated Software Evolution (USE)*, April 2003.

[12] G. Bierman, M. Parkinson, and J. 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.

[13] C. Boyapati, B. Liskov, L. Shrira, C.-H. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In R. Crocker and G. L. S. Jr., editors, *Proc. Object-Oriented Programming Systems, Languages and Applications (OOPSLA'03)*, pages 403–417. ACM Press, 2003.

[14] A. Bucchiarone, P. Pelliccione, C. Vattani, and O. Runge. Self-repairing systems modeling and verification using agg. In *Proc. of WICSA/ECSA'09*, pages 181–190. IEEE Press, 2009.

[15] M. Bundgaard, T. T. Hildebrandt, and J. C. Godskesen. A cps encoding of name-passing in higher-order mobile embedded resources. *Theor. Comput. Sci.*, 356(3):422–439, 2006.

[16] M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In R. D. Nicola, editor, *Proc. 16th Eur. Symp. on Programming Languages and Systems (ESOP 2007)*, volume 4421 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2007.

[17] L. Cardelli and A. Gordon. Mobile ambients. In N. M., editor, *Proc. FoSSaCS '98*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 1998.

[18] A. Cavarra, C. Crichton, and J. Davies. A method for the automatic generation of test suites from object models. *Information and Software Technology*, 46(5):309–314, 2004.

[19] Y. Chen, S. Liu, and F. Nagoya. An approach to integration testing based on data flow specifications. In *Theoretical Aspects of Computing - ICTAC 2004*, volume 3407 of *LNCS*. Springer, 2005.

[20] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee. Software engineering for self-adaptive systems: A research roadmap. In B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2009.

[21] S.-W. Cheng, D. Garlan, and B. R. Schmerl. Making self-adaptation an engineering reality. In O. Babaoglu, M. Jelasity, A. Montresor, C. Fetzer, S. Leonardi, A. P. A. van Moorsel, and M. van Steen, editors, *Self-star Properties in Complex Information Systems*, volume 3460 of *Lecture Notes in Computer Science*, pages 158–173. Springer, 2005.

[22] K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *ICSM*, pages 359–369. IEEE Computer Society Press, Nov. 4–8 1996.

[23] M. Colombo, E. Di Nitto, and M. Mauri. SCENE: A service composition execution environment supporting dynamic changes disciplined through rules. In *Proc. of ICSOC'06*, volume 4294 of *LNCS*, pages 191–202. Springer, 2006.

[24] M. Dam, B. J. 0002, A. Lundblad, and F. Piessens. Security monitor inlining for multithreaded Java. In Drossopoulou [31], pages 546–569.

[25] M. Dam, B. Jacobs, A. Lundblad, and F. Piessens. Provably correct inline monitoring for multi-threaded Java-like programs. *Journal of Computer Security*, 18(1):37–59, 2010.

[26] M. Dam and A. Lundblad. A proof-carrying code framework for inlined reference monitors in Java bytecode. KTH, unpublished, 2010.

[27] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos. Characterization of model-based software testing approaches. Technical Report ES-713/07, PESC-COPPE/UFRJ, 2007.

[28] D. Dig, S. Negara, V. Mohindra, and R. Johnson. ReBA: refactoring-aware binary adaptation of evolving libraries. In *ICSE*, pages 441–450. ACM, 2008.

[29] M. Dmitriev. Language-specific make technology for the Java programming language. In *OOPSLA-02*, volume 37, 11 of *ACM SIGPLAN Notices*, pages 373–385, New York, 2002. ACM Press.

[30] J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Lazy behavioral subtyping. In J. Cuellar and T. Maibaum, editors, *Proc. 15th International Symposium on Formal Methods (FM'08)*, volume 5014 of *LNCS*, pages 52–67. Springer, May 2008.

[31] S. Drossopoulou, editor. *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, volume 5653 of *Lecture Notes in Computer Science*. Springer, 2009.

[32] S. Drossopoulou, D. Wragg, and S. Eisenbach. What is Java binary compatibility? *ACM SIGPLAN Notices*, 33(10):341–358, 1998.

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

[34] I. K. El-Far and J. A. Whittaker. Model based software testing. In *Encyclopedia of Software Engineering*. John Wiley & Sons, Inc., 2002.

[35] C. Engel and R. Hähnle. Generating unit tests from formal proofs. In *Testing and Proofs*, volume 4454 of *LNCS*. Springer, Feb. 2007.

[36] N. A. Ernst, J. Mylopoulos, and Y. Wang. Requirements evolution and what (research) to do about it. In K. Lyytinen, P. Loucopoulos, J. Mylopoulos, and B. Robinson, editors, *Design Requirements Engineering: A Ten-Year Perspective*, volume 14 of *Lecture Notes in Business Information Processing*, pages 186–214. Springer, 2009.

[37] C. Escoffier and R. Hall. Dynamically adaptable applications with iPOJO service components. In *Proc. of Software Composition'07*, volume 4829 of *LNCS*, pages 113–128. Springer, 2007.

[38] P. Feiler, R. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, D. Schmidt, K. Sullivan, and K. Wallnau. *Ultra-large-scale Systems: The Software Challenge of the Future*. Technical Report. Software Engineering Institute (SEI), Carnegie-Mellon University, 2006.

[39] P. Fingar. Component-based frameworks for e-commerce. *Commun. ACM*, 43(10):61–67, 2000.

[40] C. Flanagan and K. Leino. Houdini, an annotation assistant for esc/java. In *FME 2001: Formal Methods for Increasing Software Productivity*, volume 2021 of *LNCS*. Springer, 2001.

[41] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using architecture models for runtime adaptability. *IEEE Software*, 23(2):62–70, 2006.

[42] C. Forgy. *On the efficient implementation of production systems*. PhD thesis, Carnegie-Mellon University, 1979.

[43] I. R. Forman, M. H. Conner, S. Danforth, and L. K. Raper. Release-to-release binary compatibility in SOM. In *OOPSLA*, pages 426–438, 1995.

[44] G. Fraser, F. Wotawa, and P. E. Ammann. Testing with model checkers: A survey. Technical Report SNA-TR-2007-P2-04, Competence Network Softnet Austria, Graz, Austria, Nov. 2007.

[45] T. Freese. Towards refactoring support in API evolution and team development. In *ICSE*, 2006.

[46] D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. In *Proc. of WOSS '02*, pages 27–32. ACM Press, 2002.

[47] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2005.

[48] C. A. Gunter, E. L. Gunter, M. Jackson, and P. Zave. A reference model for requirements and specifications. *IEEE Software*, 17(3):37 – 43, May 2000.

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

[50] A. Hallerbach, T. Bauer, and M. Reichert. Managing process variants in the process life cycle. In *Proc. of ICEIS (3-2)*, pages 154–161, 2008.

[51] A. Hartman and K. Nagin. The agedis tools for model based testing. *ACM SIGSOFT Software Engineering Notes*, 29(4), 2004.

[52] Highly Adaptable and Trustworthy Software using Formal Models. Project Proposal.

[53] J. Henkel and A. Diwan. CatchUp!: capturing and replaying refactorings to support API evolution. In *ICSE*, pages 274–283, 2005.

[54] T. Hildebrandt, J. C. Godskesen, and M. Bundgaard. Bisimulation congruences for homer, a calculus of higher order mobile embedded resources. Technical Report ITU-TR-2004-52, IT University of Copenhagen, 2004.

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

[56] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), 1969.

[57] A. B. Ivan Lanese and F. Montesi. A framework for rule-based dynamic adaptation. In *Proceedings of TGC 2010*, LNCS. Springer Verlag, 2010. To appear.

[58] A. Jeffrey and J. Rathke. Java Jr.: Fully abstract trace semantics for a core Java language. *LNCS*, 3444:423–438, 2005.

[59] E. B. Johnsen, M. Kyas, and I. C. Yu. Dynamic classes: Modular asynchronous evolution of distributed concurrent objects. In A. Cavalcanti and D. Dams, editors, *Proc. 16th International Symposium on Formal Methods (FM'09)*, volume 5850 of *LNCS*, pages 596–611. Springer, Nov. 2009.

[60] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.

[61] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, Nov. 2006.

[62] Jolie team. Jolie website. `http://www.jolie-lang.org/`.

[63] Jorba v0.1. Available at `http://www.jolie-lang.org/examples/tgc10/JoRBAv0.1.zip`.

[64] JSR 277: Java Module System. `http://jcp.org/en/jsr/detail?id=277`.

[65] JSR 294: Improved Modularity Support in the Java Programming Language. `http://jcp.org/en/jsr/detail?id=277`.

[66] D. Karastoyanova, A. Houspanossian, M. Cilia, F. Leymann, and A. Buchmann. Extending BPEL for run time adaptability. In *Proc. of EDOC'05*, pages 15–26. IEEE Press, 2005.

[67] D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *FOCS*, pages 482–491. IEEE Computer Society, 2003.

[68] Y. Kim, H. Hong, D. Bae, and S. Cha. Test cases generation from UML State Diagrams. *IEE Proceedings Software*, 146(4), 1999.

[69] W. Kongdenfha, R. Saint-Paul, B. Benatallah, and F. Casati. An aspect-oriented framework for service adaptation. In *Proc. of ICSOC'06*, volume 4294 of *LNCS*, pages 15–26. Springer, 2006.

[70] V. Koutavas and M. Wand. Reasoning about class behavior. In *Informal Workshop Record of FOOL*, 2007.

[71] J. Kramer and J. Magee. Self-managed systems: An architectural challenge. In *Proc. of FOSE'07*, pages 259–268, 2007.

[72] B. Legeard, F. Peureux, and M. Utting. Controlling test case explosion in test generation from b formal models. *Software Testing, Verification and Reliability*, 14(2), 2004.

[73] L. Li and Z. Qi. Test selection from uml statecharts. In *Proceedings of the 31st International Conference on Technology of Object-Oriented Language and Systems*. IEEE Computer Society, 1999.

[74] S. Liu. Developing quality software systems using the sofl formal engineering method. In *Proceedings of the 4th International Conference on Formal Engineering Methods*. Springer, 2002.

[75] F. Logozzo and M. Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In *Compiler Construction*, volume 4959 of *LNCS*. Springer, 2008.

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

[77] A. Marconi, M. Pistore, A. Sirbu, H. Eberle, F. Leymann, and T. Unger. Enabling adaptation of pervasive flows: Built-in contextual adaptation. In *Proc. of ICSOC'09*, volume 5900 of *LNCS*, pages 445–454. Springer, 2009.

[78] K. Meinke. Automated black-box testing of functional correctness using function approximation. *ACM SIGSOFT Software Engineering Notes*, 29(4), 2004.

[79] K. Meinke. A stochastic theory of black-box software testing. In *Algebra, Meaning, and Computation*, volume 4060 of *LNCS*. Springer, 2006.

[80] Part B: HATS Methodology, 2010. To be submitted as part of Deliverable 1.1 of project FP7-231620 (HATS).

[81] L. Mikhajlov and E. Sekerinski. A Study of the Fragile Base Class Problem. In *ECOOP*, number 1445 in Lecture Notes in Computer Science, pages 355–383. Springer-Verlag, 1998.

[82] C. Mingsong, Q. Xiaokang, and L. Xuandong. Automatic test case generation for UML Activity Diagrams. In *Proceedings of the 2006 international workshop on Automation of software test*. ACM, 2006.

[83] F. Montesi, C. Guidi, and G. Zavattaro. Composing services with JOLIE. In *Proc. of ECOWS'07*, pages 13–22. IEEE Press, 2007.

[84] F. Montesi and D. Sangiorgi. A model of evolvable components. In *Proceedings of TGC 2010*, LNCS. Springer Verlag, 2010. To appear.

[85] P. Müller and A. Poetzsch-Heffter. Kapselung und Methodenbindung: Javas Designprobleme und ihre Korrektur. In *Java-Informations-Tage*, pages 1–10, 1998.

[86] N. Narendra, K. Ponnalagu, J. Krishnamurthy, and R. Ramkumar. Run-time adaptation of non-functional properties of composite web services using aspect-oriented programming. In *Proc. of ICSOC'07*, volume 4749 of *LNCS*, pages 546–557. Springer, 2007.

[87] OASIS. *Web Services Business Process Execution Language Version 2.0*. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`.

[88] A. Offutt and L. S. Generating test data from SOFL specifications. *Journal of Systems and Software*, 49(1), 1999.

[89] A. Orso, A. Rao, and M. J. Harrold. A technique for dynamic updating of Java software. In *Proc. Intl. Conf. on Software Maintenance (ICSM'02)*, pages 649–658. IEEE Press, Oct. 2002.

[90] D. Perry and A. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992.

[91] K. Pohl, G. Böckle, and F. van der Linden. *Software product line engineering: Foundations, Principles and Techniques.* Springer, 2005.

[92] A. Pretschner and J. Philipps. Methodological issues in model-based testing. In *Model-Based Testing of Reactive Systems*, volume 3472 of *LNCS*. Springer, July 2005.

[93] Prototype in Jolie. Available at `http://www.jolie-lang.org/examples/tgc10/adaptationFramework_tgc10.zip`.

[94] D. Raz, R. Stadler, C. Elster, and M. Dam. In-network monitoring. In M. T. G. Cormode, editor, *Algorithms for Next Generation Networks.* Springer-Verlag, 2010. To appear.

[95] Requirement elicitation, Aug. 2009. Deliverable 5.1 of project FP7-231620 (HATS), available at `http://www.cse.chalmers.se/research/hats/sites/default/files/Deliverable5.1.pdf`.

[96] R. Robbes and M. Lanza. Versioning systems for evolution research. In *Proceedings of the Eighth International Workshop on Principles of Software Evolution.* IEEE Computer Society, 2005.

[97] R. Robbes, M. Lanza, and M. Lungu. An approach to software evolution based on semantic change. In *Fundamental Approaches to Software Engineering*, volume 4422 of *LNCS*. Springer, 2007.

[98] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The (2nd Edition).* Pearson Higher Education, 2004.

[99] M. Satpathy, M. Leuschel, and M. Butler. Protest: An automatic test environment for b specifications. In *International workshop on Model Based Testing*, volume 111 of *ENTCS*. Elsevier, 2005.

[100] J. Schäfer and A. Poetzsch-Heffter. Coboxes: Unifying active objects and structured heaps. In *10th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2008)*, LNCS, pages 201–219. Springer, 2008.

[101] M. Schäfer, T. Ekman, and O. de Moor. Sound and extensible renaming for Java. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 277–294. ACM, 2008.

[102] N. Schirmer. Analysing the Java package/access concepts in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 16(7):689–706, 2004.

[103] A. Schmitt and J.-B. Stefani. The kell calculus: A family of higher-order distributed process calculi. In C. Priami and P. Quaglia, editors, *Global Computing*, volume 3267 of *Lecture Notes in Computer Science*, pages 146–178. Springer, 2005.

[104] D. Sokenou. Generating test sequences from UML Sequence Diagrams and State Diagrams. In *INFORMATIK 2006*, volume P-94 of *Lecture Notes in Informatics*, 2006.

[105] G. Spanoudakis, A. Zisman, and A. Kozlenkov. A service discovery framework for service centric systems. In *Proc. of SCC'05*, pages 251–259. IEEE Press, 2005.

[106] F. Steimann and A. Thies. From public to private to absent: Refactoring Java programs under constrained accessibility. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, volume 5653 of *LNCS*, pages 419–443. Springer, 2009.

[107] K. Stobie. Model based testing in practice at Microsoft. In *Proceedings of the Workshop on Model Based Testing*, volume 111 of *ENTCS*. Elsevier, 2005.

[108] P. Stocks and D. Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11), Nov. 1996.

[109] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis mutandis: Safe and predictable dynamic software updating. *Transactions on Programming Languages and Systems*, 29(4):22, 2007.

[110] R. Strnisa, P. Sewell, and M. J. Parkinson. The Java module system: core design and semantic definition. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 499–514. ACM, 2007.

[111] L. Tan, O. Sokolsky, and I. Lee. Specification-based testing with Linear Temporal Logic. In *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration*. IEEE Computer Society, 2004.

[112] R. Taylor and A. van der Hoek. Software design and architecture: The once and future focus of software engineering. In *Proc. of FOSE'07*, pages 226–243, 2007.

[113] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing. Technical Report 04/2006, Department of Computer Science, University of Waikato, apr 2006.

[114] V. T. Vasconcelos. Fundamentals of session types. In *9th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Web Services (SFM 2009)*, volume 5569 of *LNCS*, pages 158–186. Springer, 2009.

[115] K. Verma, K. Gomadam, A. Sheth, J. Miller, and Z. Wu. The meteor-s approach for configuring and executing dynamic web processes. Technical report, Technical report, University of Georgia, Athens, 2005.

[116] J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In *Proc. Internet Programming Languages*, volume 1686 of *Lecture Notes in Computer Science*. Springer, 1999.

[117] Y. Welsch and A. Poetzsch-Heffter. Making source compatibility of Java packages checkable. Submitted, draft available at `http://softech.cs.uni-kl.de/Homepage/PublikationsDetail?id=138`.

[118] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. *ACM SIGSOFT Software Engineering Notes*, 27(4), July 2002.

[119] F. Wuhib, M. Dam, and R. Stadler. Decentralized detection of global threshold crossings using aggregation trees. *Computer Networks*, 52(9):1745–1761, 2008.

[120] F. Wuhib, M. Dam, and R. Stadler. A gossiping protocol for detecting global threshold crossings. *IEEE Transactions on Network and Service Management (TNSM)*, 2010. To appear March 2010.

[121] F. Wuhib, M. Dam, R. Stadler, and A. Clemm. Robust monitoring of network-wide aggregates through gossiping. *IEEE Transactions on Network and Service Management (TNSM)*, 6(2), 2009.

[122] Y. Yao and Y. Wang. A new approach to test case generation based on real-time process algebra (rtpa). In *Proceedings of Canadian Conference on Electrical and Computer Engineering*, volume 3. IEEE Computer Society, 2004.

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.

AUT Application Under Test

Binary Compatibility Compatibility based on the compiled version of the source codes.

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 run-time modifications, such as rebinding and exchanging components.

FAS Fredhopper Access Server

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

MBT Model-Based Testing

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.

QoS Quality of Service, refers to the ability to find resources.

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

Source Compatibility Compatibility based on the source codes.

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

Appendix A

Fredhopper Product: An Overview

This chapter provides an overview of Fredhopper's software system. The Fredhopper Access Server (FAS) is a component-based, service-oriented and server-based software system, which provides search and merchandising IT services to e-Commerce companies such as large catalog traders, travel booking, managers of classified, etc. For the purpose of the case study, we present the structure of the FAS in the following way: *query engine*, *business manager*, *indexer*, *data manager*, *search engine optimizer*, *targeting diagnostic tool* and *test system*. An architectural view of how these components are related in the FAS is shown in Figure A.1. We now provide an informal description of the functionalities of each component.

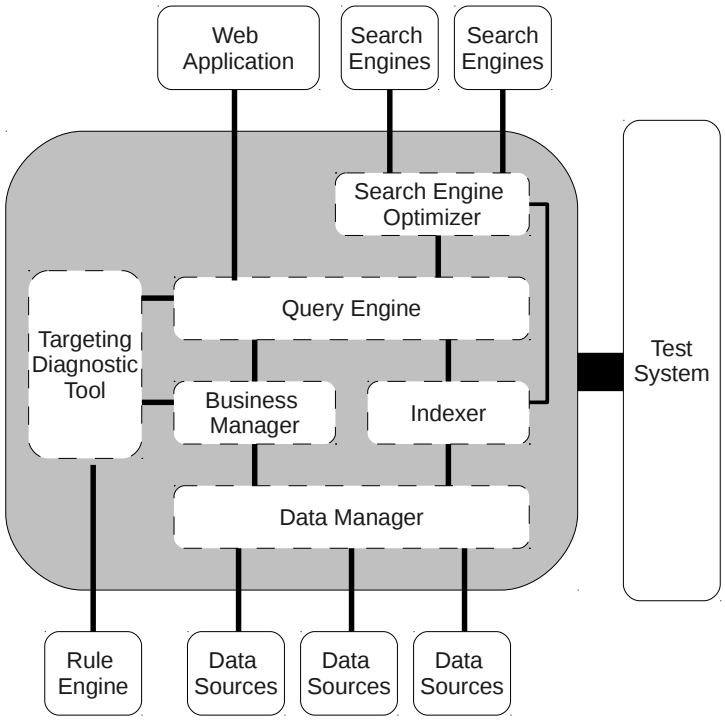

Figure A.1: An overview of the FAS architecture

Business Manager. The business manager provides to clients the management console for managing, monitoring and measuring searches, catalogs, navigations and promotions. There is also a graphical user interface, which allows non-IT business specialists to interact with the component.

Data Manager. The data manager is an Extract, Transform and Load (ETL) toolkit. It provides mechanisms

to extract data from a variety of data sources such as ERP systems, databases etc.; to transport extracted data and carry out transformations such as normalization and aggregation; and to save transformed data as input XML to the FAS.

Indexer. The indexer component is composed of three sub-components – XML loader, search indexer and tree builder. The XML loader takes a textual input description (XML) of operations on data items and performs those operations. Operations include adding items to the FAS index and annotating (enriching) items with more information. The search indexer is responsible for processing the loaded raw item into an index structure, allowing efficient search. The tree builder is responsible for constructing a tree model index from loaded items, which is then used for *faceted navigation* within catalogs of items.

Query Engine. The query engine provides the core query and response mechanisms. It serves request from both (web) search engines and customers.

Search Engine Optimizer. The search engine optimizer component implements a *white hat*[1] method to guide (web crawler) search engines when indexing web sites with faceted navigation. Faceted navigation provides users with the technique for accessing a collection of information represented using a *faceted classification*, allowing users to explore catalogs by progressively filtering available information.

Targeting Diagnostic Tool. The FAS provides a mechanism allowing end users to specify business rules that regulate what, how and where the FAS system retrieves and presents content. FAS integrates a third party rule engine that infers rules at run time. Nevertheless, employing a third party library has the limitation that the FAS system cannot easily track how the rule engine works. For this reason, we introduce a diagnostic tool (run time monitor) component to provide the capability to access information about particular inferencing and provide corresponding debug information such as if and why a particular business rule did/did not allow displaying of expected/unexpected information.

Test System. At Fredhopper, the test system is part of the FAS's architecture. It ensures the FAS's quality during its implementation and testing phases. A more detailed description of the test system has been provided in Section 6.2.

[1] `http://searchenginewatch.com/3483941`