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 D5.1
Requirements Elicitation

Due date of deliverable: (T6)

Actual submission date: 31 August 2009

Revision date: 30th March 2010

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

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

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:
Requirements Elicitation

This document summarises deliverable D5.1 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`.

Deliverable D5.1 presents the requirements for the HATS method elicited in Task 5.1. The requirements are divided into methodological requirements and high level concerns. The methodological requirements stem from different audiences and different perspectives and provide a high-level requirements view to the HATS methodology. The high level concerns are the result of analyzing different scenarios of three selected case studies of different profile, abstraction, size, and application area. Besides the description of the concerns, the possible HATS support is already envisioned and mapped to the responsible work tasks. This will simplify the validation in the next steps of Work Package 5.

List of Authors

Pablo Antonino (FRG)
Ralf Carbon (FRG)
Nikolay Diakov (FRH)
Jan Schäfer (UKL)
Yannick Welsch (UKL)
Peter Wong (FRH)

Contents

Chapter 1

Introduction

As pointed out in the project proposal, in HATS [17] we aim to develop a methodological and tool framework achieving not merely far-reaching automation in maintaining dynamically evolving software, but an unprecedented level of trust by replacing informal processes with rigorous analyzes based on formal methods. Furthermore, the HATS framework and methodology aim at enabling organizations to produce trustworthy software that supports adaptation. The methodology takes an empirically successful, yet informal software development paradigm and puts it on a formal basis. In HATS, we plan to turn the software product family (SWPF) development into a rigorous approach. Specifically, HATS focuses on product line engineering (PLE) [25], i.e. a development approach to produce products as part of a product family based on strategic and pro-active reuse of available components.

The technical core of the framework consists of an Abstract Behavioral Specification language which allows precise description of SWPF features and components and their instances. Such level of description allows analysis of various system properties related to trustworthiness, for instance, security or performance [6], and enable new opportunities in code generation, automatic product derivation etc. The framework also provides extensive tool and method automation support to its users.

The project's work packages one, two, three and four focus on developing various aspects of the HATS framework and methodology. The fifth work package (WP5) focuses on producing input to the other packages to steer their results in the right direction. The WP5 includes execution of case studies to provide the final evaluation of the degree of success of the HATS framework and methodology.

Within the HATS project, we opt for a research method based on continuous empirical evaluation of the project results throughout the project execution. Specifically we use an iterative approach, such that at every step of the process, work package members would consider scenarios and requirements as starting points to develop the framework core, extensions and tooling. Furthermore, at every step of the process we challenge the intermediate results by evaluating them against a criteria of requirements, preferably via controlled experimentations. This allows for early feedback and continuous improvement. Therefore, WP5 has the important role to provide input that directs the work of other work packages, and the important role to validate results. In order to fulfill this goal, WP5 defines the following working tasks: Requirements Elicitation (Task 5.1), Evaluation of Core Framework (Task 5.2), Evaluation of Modeling (Task 5.3), and Evaluation of Tools and Techniques (Task 5.4). In particular, this document presents results from *Requirements Elicitation*.

1.1 Goal of the deliverable

This document presents deliverable D5.1, which contains the results from initial Requirements Elicitation (Task 5.1). In Task 5.1 we gather requirements in the form of detailed scenarios from selected case studies, and identify general methodological concerns. Requirements Engineering in D5.1 is driven by the owners of the example case studies used in HATS. In particular, FRH and FRG bring in requirements from their industrial perspectives.

The main goals of this document are as follows:

1. We study the general requirements of the HATS methodology. Specifically we collect high-level methodological requirements from both industrial and research perspectives. In particular from the industrial perspective, we study how the HATS methodology may be integrated to existing development and support processes, while from the research perspective, we investigate how existing methodological challenges, which arise in the scientific areas in which the HATS consortium is specialized, may be addressed.

2. We have interviewed members of the *end user panel* of the HATS project. End user panel is composed of representatives of external companies interested in the HATS technology. In this document we present an evaluation of these interviews in the context of HATS methodology.

3. We provide three case studies, which differ in profile, abstraction, size and application area. Specifically we have chosen an academic case study of a trading system for handling sales in supermarkets [27, 10]; a case study of the Virtual Office of the Future [30] for supporting seamless execution of office tasks independent of the office workers' physical location, and an industrial case study on Fredhopper's server-based software systems [15] providing search and merchandising IT services to e-Commerce companies.

4. Through each case study, we investigate several scenarios, and for each scenario, we present a concrete example detailing the typical steps in the development and support process of the software system described in the case studies. We then identify technical concerns that arise from the concrete examples and associate them to corresponding tasks from technical work packages 1 to 4, the result of which would address these concerns[1].

We now give details on how requirements and concerns identified in this deliverable are used throughout the HATS project.

The methodological requirements are used to guide the development of the HATS methodology in Task 1.1 as well as to guide the integration of different modeling techniques and tool support into the development method. These requirements will also be used to evaluate on the effectiveness of the methodology. Consequently these requirements will help the validation process (Tasks 5.2, 5.3 and 5.4) to provide constructive feedback to the technical work tasks for further improvements.

In Task 5.2 we will study the methodological requirements in more detail and in particular we will investigate suitable evaluation strategies with these requirements. These strategies will then be applied throughout the validation process, starting with the validation of milestone M1 of the HATS project as part of Task 5.2.

Through various scenarios from the case studies this deliverable identifies high-level concerns that should be addressed by the technologies delivered by the project's work tasks. We will carry out extensive analysis to further refine these concerns in Task 5.2. The analysis will be carried out in close cooperation with partners of individual work tasks. In Task 5.2 we aim also to define suitable evaluation methods and criteria for each concern. These results will then be used to evaluate results of each work task as well as to guide the validation process in WP5, starting with the validation of milestone M1 of the HATS project as part of Task 5.2.

1.2 Structure of document

In this section we present the structure of this deliverable and provide the necessary detail to assist readers navigating through this deliverable; we also highlight the scope of the requirements and concerns identified in this deliverable. A summary of this deliverable is provided in Chapter 6.

[1] Detailed requirements analysis of scenarios will be conducted in Task 5.2 together with members of the technical work packages

1.2.1 Methodological requirements

In Chapter 2 we describe requirements of the HATS methodology in the context of *industry* and *research*.

Industry

In the context of industry, we used three main sources for gathering methodological requirements – First is through the understanding of stakeholders' needs in the development and support process of software systems. Stakeholders in this process includes the managers that would have to decide on the adoption of the HATS method as well as the users such as software developers and support specialists, which would apply the HATS method to their daily work. Second is through examination of the most distinctive characteristics of the software production method employed by the Fredhopper consortium member. Third is through the evaluation of the interviews with members of the end user panel.

Research

In the context of research, we focus on methodological requirements from product line engineering [25]. Specifically we study the principle and foundation of product line engineering methodology and derive requirements of the HATS methodology in the context of product line engineering.

1.2.2 Case studies

In Chapters 3, 4 and 5, we study the cases chosen as the sources of requirement elicitation of the HATS project as well as the target where the evaluation of results produced from the project would be carried out. Specifically Chapter 3 presents the case study of the trading system for handling sales in supermarkets, Chapter 4 presents the case study of the Virtual Office of the Future, and Chapter 5 presents the industrial case study on Fredhopper's server-based software systems. Here we describe the structure of each case study in these chapters.

Structure of case studies

We format each case study using the following structure:

1. We informally introduce the functionality and the architecture of the software systems in question. This includes description of individual components of the systems and their interaction relationship. We then briefly overview each scenario and the technical concerns the scenario would identify.
2. We divide each scenario section into subsections. In each subsection we present the content in the following format:
   (a) Description of the scenario, its relationship to the software system in question and the type of concerns it aims to identify;
   (b) A concrete example of the scenario. These are procedural steps, which reflect or portray the way in which the software system is developed, used and maintained in the context of the scenario. For each step, a single instance of the procedure may be provided to help work package members to understand the scenario.
   (c) Concerns identified from the scenario. These are technical issues, which may be addressed by the HATS framework for improving qualities of the software system, which include correctness, evolvability, trustworthiness and effective resource consumption, as well as the efficiency of the development and support process of the software system. We associate each concern with one or more work tasks in the HATS project when possible.
3. We provide a summary of our presentation of the case study and tabulate the association of concerns with work tasks in the HATS project.

1.2.3 Labeling

Methodological requirements

Each requirement harvested in Chapter 2 has a label consisting of a unique identifier prefixed with *MR*, as well as a descriptive name. Table 2.1 at the end of Chapter 2 shows an overview of the methodological requirements.

Case studies

Scenarios and concerns in the case studies chapters are also uniquely identified. For the trading system case study in Chapter 3, each scenario has a unique identifier prefixed by *TS* and a descriptive name, while each concern has a unique identifier prefixed by *TS-C* and also a descriptive name. Table 3.1 shows an overview of the scenarios and concerns considered in this case study, while Table 3.2 maps each HATS project's work task to the concerns that the task could help to address. Both tables can be found at the end of Chapter 3.

For the Virtual Office of the Future case study in Chapter 4, each scenario has a unique identifier prefixed by *VF* and a descriptive name, while each concern has a unique identifier prefixed by *VF-C* and also a descriptive name. Table 4.1 shows an overview of the scenarios and concerns considered in this case study, while Table 4.2 maps each HATS project's work task to the concerns that the task could help to address. Both tables can be found at the end of Chapter 4.

For the Fredhopper product case study in Chapter 5, each scenario has a unique identifier prefixed by *FP* and a descriptive name, while each concern has a unique identifier prefixed by *FP-C* and also a descriptive name. Table 5.1 shows an overview of the scenarios and concerns considered in this case study, while Table 5.2 maps each HATS project's work task to the concerns that the task could help to address. Both tables can be found at the end of Chapter 5.

We have included Table 1.2.4 which maps task numbers to their task names.

1.2.4 Scope and structure

This deliverable defines high level requirements and concerns that the HATS framework should satisfy. Here we consider the scope of the deliverable in terms of the methodological requirements and the concerns identified through case studies. We stress that some methodological requirements and technical concerns identified through case studies might not be achievable within a basic research project such as HATS. An achievable scope for the validation process will be identified in Task 5.2 and become part of deliverable D5.2.

Moreover, in Task 5.2 we will carry out extensive analysis on the high level requirements and concerns. The result will give a structured classification to the analyzed requirements. The requirement analysis in Task 5.2 will also refine the requirements and concerns identified in this deliverable. This will minimize ambiguity and maximize verifiability of the requirements and concerns throughout the validation process in WP5.

Remark: *Deliverable D5.1 is not meant to prescribe a comprehensive set of requirements for the technical work packages 1–4. It provides initial input and needs to be complemented by requirements and concerns provided by the work package teams themselves. Although this is not the standard approach to elicit requirements in a commercial software project, it is appropriate and from our point of view even necessary in a basic research project to bootstrap development of a method extending the state of the art. The work package and task leaders are leading experts in their respective fields that will provide additional requirements in the HATS method as well as the ABS language as soon as they know the case studies and scenarios introduced in Task 5.1. The work of the work packages in HATS as a basic research project should not be over-constrained. The requirements provided in D5.1 will be extended and detailed in T5.2 in close collaboration with all task leaders based on D5.1.*

Task	**Task Name**
WP 1: Framework	
1.1	Core ABS Language
1.2	Feature Modeling, Platform Models, and Configuration
1.3	Analysis
1.4	System Derivation and Code Generation
1.5	Integrated Tool Platform
WP 2: Variability	
2.1	A configurable deployment architecture
2.2	Feature integration
2.3	Testing, debugging, and visualization
2.4	Types for variability
2.5	Verification of General Behavior Properties
2.6	Refinement and Abstraction
WP 3: Evolvability	
3.1	Evolvable Systems: Modeling and Specification
3.2	Model Mining
3.3	Hybrid Analysis for Evolvability
3.4	Evolvability at Bytecode Level
3.5	Autonomously Evolving Systems
WP 4: Trustworthiness	
4.1	Security
4.2	Resource Guarantees
4.3	Correctness
4.4	Auto Configuration and Quality Variability

Table 1.1: A table relating work task numbers to their names

Chapter 2

Methodological Requirements

This chapter collects general requirements on the HATS methodology. Thereby, we distinguish different perspectives. As product line engineering (PLE) provides the methodological basis of HATS we identify requirements from the point of view of a PLE researcher. Furthermore, requirements from an industrial perspective are presented. Fraunhofer IESE as an applied research organization focusing on technology transfer provides general requirements in the HATS methodology to be able to introduce the HATS results in industry during, but mainly after, the project. Fredhopper as the industrial partner in the HATS project provides requirements from their perspective. Finally, we present requirements from members of the HATS end-user panel elicited in telephone interviews. Some of the requirements described in this chapter will automatically be fulfilled if the project is conducted according to its proposal, for instance, the need for empirical evaluation of the HATS methodology is already manifested in work package five (WP5) on validation. Others, like the tailorability of the HATS method to different organizational contexts, are not yet explicitly mentioned in the proposal. Several requirements in this chapter, especially the ones presented in the section on the Fraunhofer perspective (Section 2.2) are very ambitious to be fulfilled in a basic research project like HATS. Consequently, their complete fulfillment is not in the scope of the HATS project. Nevertheless, we list them here as researchers of the HATS project should always keep them in mind to increase the likelihood that the HATS results can be transferred to industry after the project has been finished.

2.1 Product line engineering

This section provides general requirements from the product line engineering perspective. Thereby, we cover topics like the overall product-line life cycle, application engineering, architecture, regression testing, and evolution in general. We selected such topics based on our experience in Product Line Engineering.

Product line engineering (PLE) splits the overall development life-cycle into application engineering (AE) and family engineering (FE) (see Figure 2.1) [24]. FE builds reusable artifacts that are stored in a product line artifact base. Thereby, scoping provides the requirements to be fulfilled by FE. The product line artifacts provided by FE are generic, i.e. they contain variation points. AE builds products based on reuse from the product line artifact base. Variation points in reusable artifacts are resolved and product specific extensions are made to come up with the final customer specific products. Product line engineering methods in the end need to contribute to such essential activities in the product line life cycle. Either they support FE, i.e. they need to be able to deal with generic artifacts containing variation points, or they address AE, i.e. they need to deal with reuse and reusable artifacts that need to be instantiated, evolved, tested etc. for a specific customer's product. Product line methods like Fraunhofer PuLSE or the product line practice framework of the Software Engineering Institute [9] list essential product line activities that need to be conducted in the context of the product line life cycle. The HATS methodology should be aligned with such existing comprehensive product line approaches. Hence, we can formulate the following requirement.

Requirement (MR1)**.** ***Integrating Product Line Engineering****: The HATS methodology should be integrated with the Product Line Engineering (PLE) life-cycle as it is established in typical product line organizations and should be aligned with existing product line approaches.*

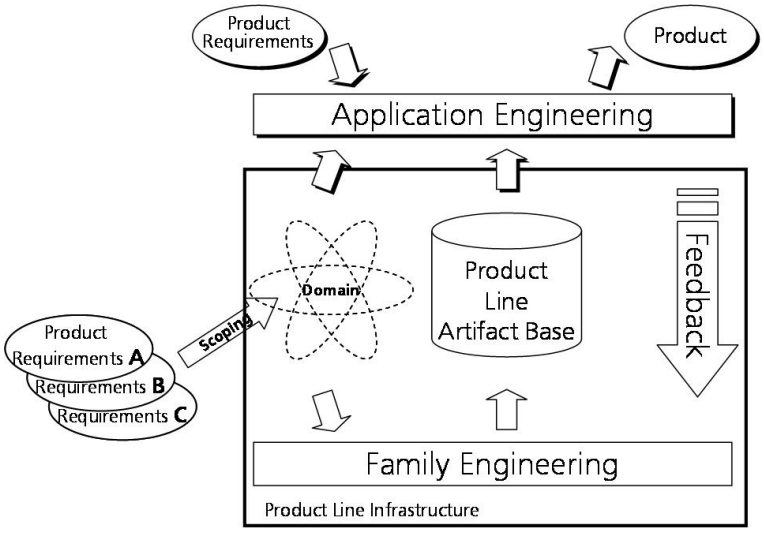

Figure 2.1: Product Line Life-Cycle

The ultimate goal of every product line organization is to produce products during AE. AE can be done in various fashions depending on the characteristics of a certain application domain. In general, one can distinguish approaches involving engineering activities during AE, i.e., for instance, design and implementation activities are part of AE [5] and approaches that try to automate AE, i.e. all engineering is done in FE and products are derived automatically based on a feature configuration selected by the customer or a provided specification [11]. HATS specifically looks at automatic product derivation but should keep other engineering approaches used in practice in mind as well.

Requirement (MR2)**.** ***Integrating application engineering****: The HATS methodology should be integratable with different application engineering approaches.*

AE typically involves testing activities, especially if customer specific requirements have been realized. Besides effort reduction during engineering a new product by means of reuse, effort reduction is expected in the AE testing step as well. It is expected that test artifacts (e.g. test cases) can be reused. Furthermore, it is expected that overall less testing is necessary during AE, since reusable artifacts are employed that have been already tested during FE.

When reusable artifacts are being changed in a product line, it must be possible to effectively run regression tests at all places where that reusable artifacts are actually being reused to assure the change did not introduce any flaws into the system. Consequently, regression test cases should be provided by FE and being reused in AE. The regression tests should be evolvable besides the system itself. This leads us to the following requirement.

Requirement (MR3)**.** ***Testing reusable artifacts****: The HATS methodology should allow the automated generation of regression tests in FE that ensure that changes on a reusable artifact do not violate properties of product line members that reuse that artifact.*

One of the product line practice areas mentioned above that should specifically be pointed out is architecture. The product line architecture is referred to as the core artifact of a product line. According to [20] software architecture is defined as follows: “A software architecture is a set of concepts and design decisions

about the structure and texture of software that must be made prior to concurrent engineering to enable effective satisfaction of architecturally significant explicit functional and quality requirements and implicit requirements of the product family, the problem, and the solution domains." A product line architecture is a generic artifact and defines the architecture for all the members of a product line. Compliance of a product architecture with the product line architecture facilitates high reuse rates of product line components. The product line architecture implies certain constraints to the product line components to be able to realize system wide properties like performance, security, availability, etc. During component design the constraints imposed by the architecture, for instance, regarding performance or how to realize security need to be considered. To be able to reason about the overall quality of the system on the architecture level, components should explicitly specify their quality properties, i.e. languages to specify components like the envisioned ABS language should provide the possibility to express quality properties of components. Besides the possibility to reason about the overall system quality on an architecture level this enables to specify evolution requirements. If in the context of an evolution a component is to be replaced, for instance, the specification of the component's quality properties can be used to select and evaluate the new component. Hence, we can formulate the following requirement.

Requirement (MR4). ***Providing language support for PLE****: The HATS methodology should take the role of the product line architecture as the core artifact of a product line explicitly into account. Consequently, the ABS language, should provide language constructs to specify quality properties of components.*

Product Line Engineering involves three types of artifacts: (a) reusable artifacts, (b) customized instances of reusable artifacts and (c) product-specific artifacts. Reusable artifacts are the result of FE. The goal of FE is to provide reusable artifacts that can be easily employed in the context of a product line member for solving a concrete problem. Customized instances of reusable artifacts are the result of AE. In this case, already existing reusable artifacts are either reused as-is (i.e. without any modifications) or configured and then adapted to the needs of a product line member. Finally product-specific artifacts are also the result of AE. Those artifacts emerge in the context of a particular product line member without reusing any artifacts provided by FE.

Product line evolution scenarios can hence be organized along the above three categories. A change request may either involve a reusable component, a customized instance or a product-specific component. The first two categories are of particular interest. When a reusable component evolves the change might need propagation to all customized instances that originate from it. On the other hand, when a customized component instance evolves, it should be easy to track that change so that reusable components the instance has been derived from can be revised. The ABS language could support evolution explicitly as follows.

Requirement (MR5). ***Specifying reusable contracts****: The ABS language should allow product line engineers to specify reuse contracts. A reuse contract describes how a component can be reused and specifically how it is actually being reused within a given product line member.*

For facilitating software impact analysis it is necessary that constituent parts of a software system have clearly specified interrelations. In the FE phase of Product Line Engineering reusable artifacts as, for instance, components are developed. Those components are reusable by containing variation points that capture what may vary in the component and how it may vary. For enabling impact analysis on reusable components it is necessary to have clear relations between the components but also between the variation points within the components.

Requirement (MR6). ***Defining reusable artifacts and variation points****: The HATS methodology should allow defining reusable artifacts and variation points contained therein as well as the according interdependencies.*

2.2 Organization: Fraunhofer IESE perspective

In this section the requirements from an organization perspective are elaborated in detail. As mentioned above we distinguish between managers deciding on the adoption of the HATS methodology and users as well

as Fredhopper as a member of the HATS consortium and members of the HATS end-user panel. Industrial organizations are mainly interested into how the HATS methodology can be adopted in their specific context and into the expected benefits. In the industrial context several stakeholders can be identified in the HATS context: managers that have to decide on the adoption of the HATS method in their project or even the overall organization and users or developers applying the HATS method in their daily work.

2.2.1 Managers' perspective

This section covers the requirements in the HATS methodology from the point of view of a manager in an industrial context. They have been derived by the authors of this deliverable and have been confirmed in the interviews with members of the end-user panel.

Typically, organizations have established development processes that should be improved but not replaced. Organizations want to keep development practices that have proven to work well in their context and select new practices to solve existing issues. Let's assume the following example: An organization already uses component-oriented development, for instance, according to KobrA [4], and models components using the Unified Modeling Language(UML). Developers have several years of experience with UML and UML models are successfully used for code generation. Introducing the HATS ABS Language and related code generation techniques in this context requires their adaptation to the described context. The usage of ABS and the HATS code generation technique needs to be combined with UML modeling and UML based code generation. Otherwise the acceptance rate of the HATS approach in such an organization will be low.

Requirement (MR7)**.** ***Tailorability****: The HATS methodology must support its tailoring to a specific organizational context. The HATS methodology needs to provide certain adjustment parameters that enable its tailoring to, for instance, the existing development practices in an organization, the application domain, the structure of an organization, or the experience of the developers.*

A complete introduction of the HATS methodology in an organization in a big-bang fashion leads to large-scale changes that may compromise the ongoing development activities. Furthermore, certain aspects of the HATS methodology are not relevant in a specific organizational context. Hence, an incremental and partial introduction of the HATS methodology needs to be possible which can be achieved by a modular structure of the HATS methodology. Different adoption strategies should be sketched that illustrate in which order certain modules of the HATS methodology should be introduced depending on the current situation in an organization.

Requirement (MR8)**.** ***Incremental adoptability****: The HATS methodology must be designed modularly to support its incremental introduction as well as partial adoption.*

Managers demand a solid basis for decision making in their organization whether the HATS methodology is appropriate or not. Case studies and controlled experiments should show the pros and cons of the adoption of the overall HATS methodology but also of single modules of the HATS methodology that can be applied in isolation.

Requirement (MR9)**.** ***Empirical evaluation****: The HATS methodology must be empirically evaluated to demonstrate that the expected benefits like higher quality in terms of trustworthiness, lower effort and cost to achieve such quality, and less time required can be achieved. The benefits need to be quantified by means of concrete measures and the context in which the empirical evaluation has been conducted needs to be thoroughly documented.*

The HATS methodology aims at providing support for small, medium, and large systems and organizations. In the case of large systems it should be applicable to system parts first and then be extended to cover the overall system. This requires the HATS methodology to be applicable to small systems or system parts without causing significant overhead in terms of effort.

Requirement (MR10)**.** ***Scalability****: The HATS methodology must be scalable, i.e. it needs to provide high quality results by using reasonable effort in the case of small, medium, and large scale systems. Applying it to larger systems parts or systems with higher complexity should not cause more than linear increase of effort and constantly deliver high quality.*

2.2.2 Users' perspective

This section covers the requirements in the HATS methodology from the point of view of a user from industry, i.e. a developer, tester, architect, designer. The requirements have been derived by the authors of this deliverable and have been confirmed in the interviews with members of the end-user panel.

Requirement (MR11)**.** ***Learnability****: The HATS methodology must be easy to learn. The new concepts to be applied by developers in an organization introducing the HATS methodology should be kept simple to be understandable by developers with average skills and experience. This implies offering of good documentation, tutorials, incremental examples.*

Requirement (MR12)**.** ***Usability****: The HATS methodology must be easy to use. Developers with average skills and experience should be able to apply the HATS methodology with reasonable effort. For example, this implies tools designed for ease of use.*

Requirement (MR13)**.** ***Reducing manual effort****: The HATS methodology must reduce manual effort as well as the error-proneness of manually defined code.*

If such requirements are fulfilled, the HATS methodology will find broad acceptance by developers as their satisfaction at work will be increased.

2.3 Industrial applicability: Fredhopper's perspective

In this section we examine the most distinctive characteristics of the software production method of the Fredhopper consortium member as a provider of a service-oriented server software product. We aim to collect methodological requirements to the HATS framework, so that it can easily support the Fredhopper production method out of the box.

Fredhopper offers services targeted at improving catalog-based sales over the Internet by assisting users to find relevant items quickly. The services consist of different functional blocks - searching of sale items, fast navigation of categorized items, promotional items, integration with third party online advertising systems (such as Google ADs), user context-based search, alternatives suggestions, and so on. Each block addresses different areas of the main mission – to help clients increase their sales over the Internet.

2.3.1 Developing an evolving product

Fredhopper develops and maintains its own server software. Fredhopper regularly enhances its product with new features to constantly offer to its clients a competitive edge. Therefore, Fredhopper employs a development cycle that includes large iterations that regularly produce quality versions of the product. In this way, the product provides more values to clients at each evolutionary step.

Requirement (MR14)**.** ***Iterative formal modeling****: The HATS framework plans to apply formal methods for enhancing software production in many aspects. In iterative development, we want to allow iterative formal modeling as well. For example, if the framework employs code generation, we have to consider model mining out of code too. This will allow the developer to integrate the changes in the code into the model.*

Since most software companies produce their software using many informal methods to produce the code of their products, these companies have to attest the quality of their products by doing extensive checking on whether the software does what one designs it to do. One typical approach includes testing by a team

of Quality Assurance (QA) engineers. The test plans typically contain test cases harvested from design specifications, solving of previous bugs, user requirements, and so on. This kind of testing is expensive, especially with products that evolve. Each iteration over the product code base may introduce changes to it. Every change brings the risk of regression - functionality attested in a previous iteration, suddenly becomes wrong or inaccessible.

Fredhopper maintains a test system as a satellite to the product code base. The test system consists of automated testing facilities and tests cases to perform. The test system contains tests which if passed altogether, attest the quality of the product version. The automated test system changes or grows as the product changes or grows with each product version. Naturally, Fredhopper uses the test system to quickly detect regression due to changes in the code base. The test system does continuous integration to rerun tests based on several events: weekly, nightly, and based on individual updates made to the code base.

Requirement (MR15)**.** ***Test system evolution****: The HATS framework has to support iterative updates of the test system. For example, if we have test generation, tests may change following a change of constraints in the model that the framework generates. The tools in the HATS system have to update the test system with the new or changed tests. The updated test system would subsequently rerun the changed tests.*

We point out that software development companies that contract one-time pieces of software, may not need to invest in such an incremental system, as clients may not agree to pay the additional investment to create and maintain it. Still, in the case of a necessary change at a later point in time, the client will pay the increased cost of a complete retest, instead of the initial investment in a test system that captures and automates the test of the previous iteration.

2.3.2 Server software

Fredhopper backs its service to its clients with server software that it deploys for them. This server software runs long time and has to meet the Service Level Agreement (SLA) with the client. The SLA includes, among others, functionality, data interoperability, support and performance guarantees. For example, the performance guarantees include:

- Stable memory usage guarantee;
- Number of concurrent user requests guarantee;
- Statistical average response time guarantee;
- Up time guarantee.

In order to make sure Fredhopper maintains the SLA between versions of its product, Fredhopper's iterative development process includes a phase which measures and compares various aspects of performance. For example, for memory guarantees, one measures the memory footprint of changed modules over different data sets. Measuring CPU utilization under concurrent user stress shows the capacity of the current code base to process concurrent requests. Measuring average response time on simulated user requests makes sure that the current code base covers the response time guarantee.

Requirement (MR16)**.** ***Resource guarantees****: The HATS framework will address resource guarantees. In this context, it would prove useful if the framework tooling allows performance metrics to be broken down using structural information from the models. This resembles 'code profiling' which uses structural information mined from the code to decompose the overall performance measuring down to smaller pieces, In this way developers may focus on improving the biggest contributors to delays. The same applies to memory and concurrency bottlenecks (high lock contention, etc).*

2.3.3 Software as a service

Fredhopper offers its software services in two different manners:

- Software product;
- Software as a service.

As a software product, Fredhopper integrates its own product with the electronic commerce (typically a web shop) software of the client by installing and integrating it in the client's IT infrastructure. As a software service, Fredhopper deploys a product instance in-house and leases to its clients a software service over the Internet according to some SLA. The SLA for software service includes extended guarantees compared to the product SLA:

- Additional connection guarantees;
- Interface guarantees.

Since Fredhopper hosts the service explicitly for its client, connection guarantees refer to additional requirements for connection throughput between the client infrastructure and Fredhopper infrastructure.

Interface guarantees define the messaging formats and protocols that Fredhopper service has to adhere to so that the client can have continuous service. This requirement becomes important, because when hosting, Fredhopper may update the server version transparently to allow clients to get improvements for performance, bug fixing and new features. In order to make sure the client does not get any undesired functional degradations, Fredhopper has to strictly adhere to the same protocol.

In contrast, a product hosted at a client requires to fix the product version. In this case, software updates at a client happen explicitly and the client typically plans testing phases. For service-oriented deployment, Fredhopper benefits from supporting transparent updates with no-protocol-degradation guarantees.

Requirement (MR17)**.** ***Protocol analysis****: The HATS framework addresses verification and protocol checking issues. From the methodological point of view, it would prove useful to provide an analytical tool that Fredhopper can integrate in its service provisioning process. The tool can support the detection of any protocol violations. Violations may be originated from unanticipated regressions in the Fredhopper product or unanticipated client misuses of the protocol. Furthermore, if such a tool can operate as a runtime component, Fredhopper may provide a proper default handling of the issue automatically during runtime. Note that the possibility for such errors comes from the informal process of producing the Fredhopper product. In addition, Fredhopper does not have control over the remote client, which may abuse the protocol due to own regressions or incorrect implementations.*

2.3.4 Integrated tools development environment

Fredhopper uses a modern development environment. Such an environment has the characteristic that it integrates all necessary tools. The integrated development environment (IDE) manages all resources that constitute the coding representation of the final product. These include models, text sources, tests, and so on. The IDE also integrates all activities in a development process - modeling, writing code, testing, collaboration with other team members, and so on.

Requirement (MR18)**.** ***Integrated environment support****: The tools of the HATS framework have to support usage in an integrated environment. As such these tools have to have:*

- *Interoperable formats. HATS tools that we expect to use in a sequence, have to speak common formats.*
- *Common visual representation. HATS tools may have different cores, working with different formalisms. Nevertheless they have to operate in the common visual and easy to use control environment of the IDE.*

- *Common resources. An IDE manages project resources by building a heterogeneous repository of resources, which one may share among teams and team members. The tools have to take into account versioning and collaborative work.*
- *Ability to work with large projects. HATS tools have to scale. This may even mean that individual tools and formalisms may have to introduce partitioning of models and splitting and merging of models just for the sake of utilizing work across the team.*

2.4 End-user panel perspective

In this section we present the requirements that the end-user panel stated in interviews that we conducted with their representatives.

Many organizations in practice already use model-based development approaches. They invested considerable effort in the past to construct models that they use in development. Using the HATS methodology in the future including the use of the ABS language may not lead to a loss of such efforts, i.e. organizations want to reuse their models in the context of the HATS methodology. This leads to the following requirement.

Requirement (MR19)**.** ***Existing modeling techniques support****: The HATS methodology must be able to cope with existing models in organizations. To allow the reuse of models from languages typically used in practice today, for instance, UML activity diagrams with Petri-net semantics, the HATS methodology should provide model-transformations from those models into ABS models.*

Note that to address this requirement is probably out of the scope of the research agenda of the HATS project and should be pursued in follow-up activities. The requirement has also to be relativized to the existence of a formal semantics of source models. While activity diagrams have a fairly clear semantics, in practice many companies use ad-hoc profiles that have at best informal meaning.

Organizations that are customizing their products for specific customers often try to restrict the allowed changes. They define extension points in their applications and prevent modifications of the core application. In doing so, they keep control over the evolution of the core application. One of the key benefits of such an approach is that the core application can be evolved without interfering with customer specific extensions.

Requirement (MR20)**.** ***ABS extensibility****: The ABS language should provide the possibility to on the one hand specify extension points of an ABS model to be used, for instance, for customizing activities and on the other hand specify a core model that should only be evolved by developers of the application core.*

Large information systems today are often based on service-orientation. Services support the alignment of business and IT by providing a means to map business related services to IT services. Sometimes services are composed to higher-level services to better align with the workflows of certain customers.

Requirement (MR21)**.** ***Service orientation****: The HATS methodology, and in particular the ABS language, should be able to deal with service-orientation. Ideally, the ABS language provides constructs to specify services and their properties. Additionally, the composition of services should be considered.*

Another major characteristic of today's information systems is that they are surrounded by lots of middleware realized in different technologies. Often, the behavior of such middleware is not specified which can lead to unexpected reactions of the respective software.

Requirement (MR22)**.** ***Middleware abstraction****: The ABS language should be able to abstract from concrete middleware technologies and provide means to describe the expected behavior of middleware solutions.*

The architectural style of a system determines the component types and connector types that are used to model a concrete system according to that style. Component and connector types have specific properties that need to be expressible by the language used to model components and connectors, in the case of HATS the ABS language.

Requirement (MR23)**.** ***Architectural style****: The ABS must be able to specify the properties imposed by a specific architectural style that is used for a concrete system to be modeled. Ideally, the ABS language would provide different variants according to the architectural style used in the architecture of a concrete system and provide the possibility to easily define new variants of it to address specific architectural styles.*

2.5 Summary

Table 2.1 summarises all methodological requirements for short reference. We have presented requirements grouped by the different sources. We have selected representative source that guarantee that HATS produces an efficient and usable methodology for applying formal methods to software development. The four categories of requirements emphasize the following issues:

Product line engineering This category focuses on support for PLE with formal methods.

Organization This category focuses on adoption, integration of the HATS new method into an existing infrastructure for software production, including large operations.

Industrial applicability This category focuses on modern professional product-based development that includes evolving products with some performance guarantees.

End-user panel This category focuses on modern trends in productivity and knowledge management such as links to different existing modeling languages and architectural styles for using them, middleware support, and service orientations. Requirements in this category were harvested by interviewing members of the end-user panel of the HATS project.

Requirements in the categories "product line engineering" and "industrial applicability" are more specific than those in the categories "organization" and "end-user panel". In particular we envisage the HATS methodology will meet some of these requirements (e.g. MR1 and MR14) by design, while other requirements (e.g. MR3 and MR16) in these two categories will be met by delivering corresponding technical contributions from respective work tasks.

For more general requirements in the categories "organization" and "end-user panel", we will apply further scoping to them so that HATS could contribute within the project time frame and resources. In some cases, for example, MR19 this could mean to defer addressing requirements in follow-up projects. Scoping will be carried out in Task 5.2. In addition, Task 5.2 will further analyze all requirements to clarify and finalize the mapping of work from requirements to work tasks.

Requirement Identifiers	Requirement Labels	Reference
Product Line Engineering		
MR1	Integrating Product Line Engineering	Page 11
MR2	Integrating application engineering	Page 11
MR3	Testing reusable artifacts	Page 11
MR4	Providing language support for PLE	Page 12
MR5	Specifying reusable contracts	Page 12
MR6	Defining reusable artifacts and variation points	Page 12
Organization		
MR7	Tailorability	Page 13
MR8	Incremental adoptability	Page 13
MR9	Empirical evaluation	Page 13
MR10	Scalability	Page 14
MR11	Learnability	Page 14
MR12	Usability	Page 14
MR13	Reducing manual effort	Page 14
Industrial Applicability		
MR14	Iterative formal modeling	Page 14
MR15	Test system evolution	Page 15
MR16	Resource guarantees	Page 15
MR17	Protocol analysis	Page 16
MR18	Integrated environment support	Page 16
End-User Panel		
MR19	Existing modeling techniques support	Page 17
MR20	ABS extensibility	Page 17
MR21	Service orientation	Page 17
MR22	Middleware abstraction	Page 17
MR23	Architectural style	Page 18

Table 2.1: Methodological requirements

Chapter 3

Trading System Case Study

3.1 Overview

The following example describes a Trading System as it can be observed in a supermarket handling sales. The Trading System is a typical example for a distributed component-based information system. It includes the processes at a single cash desk like scanning products using a bar code scanner or paying by credit card or cash as well as administrative tasks like ordering of running out products or generating reports. The following section gives a brief overview of such a Trading System and its hardware parts. The Trading System example was also used in the CoCoME modeling contest [27, 10], which was based on an idea of Larman [23].

Figures 3.1, 3.2, and 3.3 depict the basic structures of a single cash desk with the connected peripheral equipment such as bar code scanners, credit card readers, etc., as well as the store and enterprise server infrastructure. A store consists of an arbitrary number of these cash desks. Each of those is connected to the store server, holding store-local product data such as prices and inventory stock for each store. The store client, which is also connected to the server, allows manipulation and analysis of store-local product data. Additionally, all stores are connected to an enterprise server, which holds global product information, such as product descriptions and bar codes. An enterprise client is connected to the enterprise server, which allows to modify the global product data, e.g., adding or removing product types, and also allows to generate a set of statistics.

The cash desk is the place where the cashier scans the goods the customer wants to buy and where the paying (either by credit card or cash) is executed. Furthermore, it is possible to switch into an express checkout mode which allows only customers with few goods and cash payment to speed up the clearing. To manage the processes at a cash desk, a lot of hardware devices are necessary (see Figure 3.1). Using the cash box available at each cash desk, a sale is started and finished. Also the cash payment is handled by the cash box. To manage payments by credit card, a card reader is used. In order to identify all goods the customer wants to buy, the cashier uses the bar code scanner. During each transaction the cashier enters respective product IDs using a keyboard. Entering the price of a product manually is not allowed, as this would exclude automatic inventory management. After entering all products, the customer pays the final product value either by cash or credit card. At the end of the paying process, a bill is produced using a printer. Each cash desk is also equipped with a light display to let the customer know if this cash desk is in the express checkout mode or not. The central unit of each cash desk is the cash desk PC which wires all other components with each other. Also the software which is responsible for handling the sale process and amongst others for the communication with the bank is running on that machine.

For a more detailed description of the Trading System, we refer to [27].

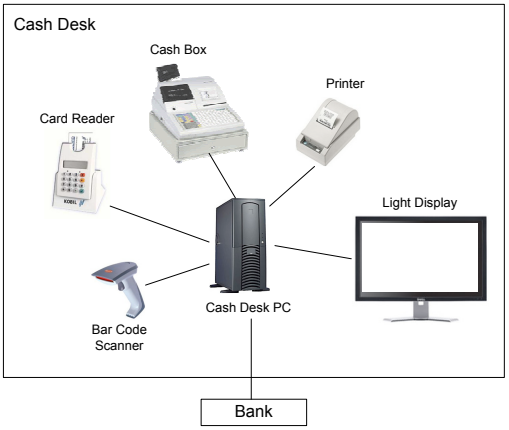

Figure 3.1: The hardware components of a single cash desk. Image taken from [27].

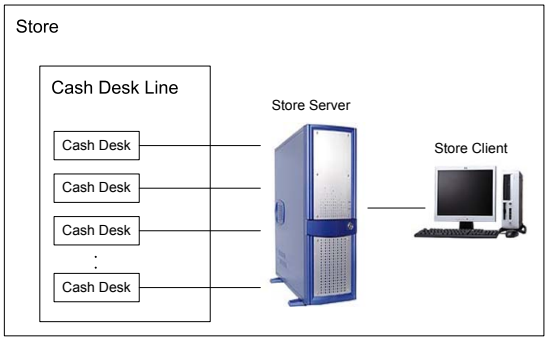

Figure 3.2: An overview of entities in a store which are relevant for the Trading System. Image taken from [27].

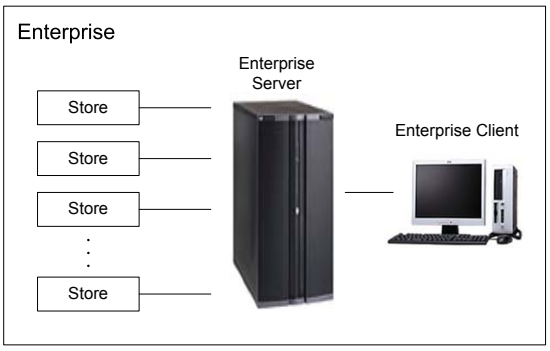

Figure 3.3: The enterprise consists of several stores, an enterprise server and an enterprise client. Image taken from [27].

3.2 Scenarios

3.2.1 Scenario TS1: Coupon handling feature

Description

This scenario covers the case of adding a new feature to the Trading System. As a concrete example, coupon handling is added to the Trading System.

Concrete example

The current trading system implementation does not have the possibility of handling coupons. Coupons allow customers to get a discount on a purchase. The discount may either be fixed to a certain amount of money, e.g., 5 Euro, or relative to the actual amount of a purchase, e.g., 5%. At each purchase at most one coupon can be redeemed, i.e., multiple coupons cannot be combined. Coupons are *customer agnostic*, which means that they are not bound to a certain customer.

Coupons have a *validity period*, which exactly defines the days when a coupon can be redeemed by the customer. This includes a start date and an end date.

Coupons can either be *unique* or *common*. A unique coupon can only be redeemed once, which is ensured by the system. A common coupon can be redeemed multiple times, the system only has to ensure the validity period. There may also exist multiple copies of a common coupon.

Coupons can be generated in different ways. For example, a shop may have certain discount days, on which each customer gets a coupon on each purchase together with the bill. Such a coupon could, for example, be redeemed on the next purchase on a different day. In that case, each coupon will be unique. Coupons could also be printed in newspapers as part of an advertisement. Such coupons would be common, and the cashier has to invalidate these coupons manually, by shredding them, for example.

Concerns and envisioned support by HATS

Concern (TS-C1)**. Correctness**: It should only be possible to redeem a coupon within its validity period. In addition, a unique coupon should only be redeemed once. The system must guarantee both properties.

HATS should provide mechanisms to ensure these properties. Ideally HATS would allow for a full verification (T2.5, T4.3) of this property on the model level. On the implementation level, either code generation (T1.4) should be applicable or a correctness verification of the implementation with respect to the model (T2.6, T4.3). At least, it should be able to generate test cases to validate the implementation (T2.3).

Concern (TS-C2)**. Failure isolation from standard sales process**: The coupon functionality should be implemented in a way that allows the standard sales process to work, even if the coupon functionality is not working at runtime. In that case, it should be possible by the cashier to manually apply the discount of the coupon to the purchase price.

HATS should provide support for guaranteeing such failure isolation. Ideally, the coupon handling would be modeled as a separate feature (T1.2) and HATS would be able to verify failure isolation on the ABS model level (T1.3, T2.2, T2.5, T2.6). If a complete feature separation is not possible, HATS should be able to ensure that the evolution from the old version to the new version preserves the mentioned property (T3.1, T3.3). It might also be the case that such a property can only be verified on the final product (T4.3). Finally, self-monitoring and self-correction techniques could be applied (T3.5).

Concern (TS-C3)**. Security**: It should not be possible for third parties to create fresh valid coupons. A coupon is valid if it is accepted by the system and results in a purchase discount. A coupon is fresh if it was not generated by the system itself. However, the system does not have to guarantee that existing coupons, i.e., coupons which have been generated by the system, cannot be copied.

HATS should provide support for ensuring these security properties. Most likely, many techniques must be applied, ranging from the correctness verification of the component, which validates coupons (as described above), to guaranteeing confidentially and privacy policies (T4.1).

Concern (TS-C4)**. Resource**: The system response time may not be noticeably slowed down during the sales process by the additional coupon functionality.

HATS could help here by allowing the specification of such resource properties on the model level and ensuring this property on the implementation level (T4.2).

Concern (TS-C5)**. System upgrade**: Existing cash desks need to be easily upgraded by the new software functionality. Ideally this would be done at runtime, while the cash desks are running.

HATS should allow to guarantee the safeness of such upgrades, ideally at runtime (T3.1, T3.3, T3.4).

3.2.2 Scenario TS2: Cash desk variability

Description

The trading system was originally designed as one possible configuration of a product line. As there is an increasing demand by shop managers to tailor the shop application to specific needs, we want to set up our system as a product line to realize several shop scenarios.

Concrete example

A cash desk system should always allow a cashier to perform payment transactions. There are, however, multiple ways to realize this. The payment options which can be available are cash, credit card, prepaid card, or electronic cash system like Maestro. Depending on the payment options, there is a need for several additional devices which allow to do credit card transactions. There can also be multiple ways for a cashier to enter bought goods into the system. For example, a barcode scanner, an RFID scanner or a keyboard can be provided which allows the cashier to enter the respective product number. Some cash desks may furthermore provide a scale to weigh goods whose prices are not calculated on a per piece basis. Shop managers want to select payment options which are available to their clients and only need the required devices to realize these payment options. The same applies for input options to the cash desk system.

Concerns and envisioned support by HATS

Concern (TS-C6)**. Variability modeling**: The HATS approach should provide support for modeling the variability at the language level (T1.1, T1.2). An adaption of the ABS model to incorporate feature modeling, different platform models and configurations should enable modular description of the presented scenario using the HATS modeling approach (T1.2). It should furthermore be possible to generate different configurations of the cash desks (T1.4).

Concern (TS-C7)**. Correctness**: The HATS approach should guarantee that all possible product configurations are correct for some definition of correctness, e.g., can be compiled, or satisfy certain invariants (T1.5). Also, a description of valid configurations should be possible (T2.1). To support these configurations, there is also a need for tools which allow for a simple deployment process of different configurations (T2.2). Visualization of variants, test generation and debugging for single features (T2.3) and their relations is also of great importance. This can furthermore be supported by type systems and specification aspects which consider variability (T2.4).

3.2.3 Scenario TS3: Dependability property

Description

This scenario covers the case of formulating and verifying dependability properties [19]. As a concrete example, one of the most important properties of a sale system, *exact payment*, is analyzed.

Concrete example: Exact payment

For the customer of a shop, it is essential that he or she pays not more than the amount of the complete purchase costs. It is acceptable for a customer to pay less, though. From the shop owners' perspective, a customer should pay at least the price that a purchase costs, but may also be interested in not displeasing a customer by charging too much. So, essentially, a shop owner is interested that a customer pays exactly the amount of money that the purchase costs. When the customer pays by cash, the system cannot guarantee much, because it is in the responsibility of the cashier to correctly count the money. However, when the customer pays by an electronic payment system, e.g., a credit card, the system should guarantee that the correct amount of money is withdrawn from the corresponding account. Two properties must be ensured by the system, namely that the purchase price is calculated correctly, i.e., it must exactly reflect the sum of all bought products, and that the calculated price is correctly withdrawn from the electronic account. Here again, the system must rely on the cashier to correctly scan all products.

Concerns and envisioned support by HATS

Concern (TS-C8)**. Correct price calculation**: The component that calculates the price of a purchase must do this correctly.

HATS should provide mechanisms to verify the correctness of the price calculation on the model level (T2.5, T4.3). On the implementation level, correctness should be ensured by either generating correct code from the model (T1.4), showing implementation correctness with respect to the model (T2.6), or at least by generating test cases (T2.3).

Concern (TS-C9)**. Transactional behavior**: The electronic payment process must be done in a transactional way, i.e., it should either completely be done or not at all. In the latter case, the payment process must either be retried (for a fixed number of times) or fall back to cash payment. The system may only finish the payment process if the correct amount of money was either been withdrawn from an electronic account or paid by cash. If the customer has not enough cash to pay the purchase and the electronic payment did not work, the payment process has to be canceled by the cashier. The system must also guarantee that an electronic account is only charged once, or not at all, but never more than once for a single purchase.

HATS should provide mechanisms to guarantee the transactional behavior of the overall payment process. This property should be verifiable on the model level (T2.5, T4.3). Correctness on the implementation level should either be guaranteed (T1.4, T2.6) or at least, test cases should be generated, which cover the possible failure cases (T2.3).

Concern (TS-C10)**. Secure transactions**: It should not be possible by third parties to interfere with the payment process.

HATS could provide mechanisms to ensure this (T4.1).

3.2.4 Scenario TS4: Authentication policies

Description

So far, we have not considered security aspects in our shop application. To introduce the application into a real-life scenario, we have to take authentication and security mechanisms into account.

Concrete example

We want to distinguish between a set of roles in our application model. First of all, we have customers, cashiers and shop managers. Then, there are system administrators and enterprise management staff. Shop managers should only be allowed to access inventory information for their own shop and the distribution center. Cashiers should not be able to directly modify the inventory (only by selling goods). Neither shop managers nor cashiers should be able to access the credit card information of customers.

Concerns and envisioned support by HATS

Concern (TS-C11)**. Security modeling and analysis**: HATS should support modeling of authentication and security features in the ABS language. Furthermore, analysis and verification of security policies at the model and at the implementation level should be possible (T1.3, T2.5, T4.1, T4.3). Another important property would be to show that the security model is implemented in a correct way. The HATS approach should also provide the possibility to model security aspects on top of existing system specifications in a modular way.

3.2.5 Scenario TS5: Feature evolution - Loyalty system

Description

This scenario covers the case of a feature evolution. As a concrete example, the coupon handling feature of Scenario TS1 is extended to a full loyalty system.

Concrete example

The coupon handling feature described in Scenario TS1 only allows anonymous coupons to get a discount. To create an even stronger binding of a customer to a certain company or shop, loyalty programs can be used. In a loyalty program, customers can have a special loyalty card, which they can use for each purchase. Customers can then get discounts, which can be based on their individual shopping behavior. For example, it is possible to give a customer a discount of 10 Euros for every 100 Euros the customer spent when using his or her loyalty card. This total amount can be based on the sum of recent purchases of the customer and is not restricted to one single purchase.

Detailed description In order to implement the loyalty program, individual customers must be managed by the system. To participate in the loyalty program a customer has to fill out a form with his or her name and address. Each new customer gets a unique *customer id*. The customer is then given out a *loyalty card*. In general, the customer will have to sign the card, and must agree that his or her private information and shopping behavior can be stored in the system and can be used for calculating the discount.

A loyalty card is a plastic card with a unique *card id* and a corresponding barcode[1], which allows for an automatic scanning of the card id. The card id is also printed on the plastic card to allow for a manual input by the cashier. It is possible to have multiple loyalty cards assigned to the same customer, e.g., to allow partner cards.

Whenever the customer does a purchase, he or she is asked for the loyalty card, which is then scanned by the system. After the payment of the purchase has finished, the corresponding information is stored in the system. In a first simplified version, only the amount of the purchase may be stored, while in a more advanced system, the complete product list of the purchase could be stored.

The discount, which is provided to the customer, can be given in various different ways. For example, there could be a fixed discount of 1% for every purchase, which is directly applied to the current purchase. Another possibility is to print out an *individual coupon* for the customer, which he or she can redeem on the next purchase. The individual coupon is bound to the customer and can only be redeemed together with a corresponding loyalty card. Such individual coupons can also be sent by mail to the customer's address, either if the customer reached a certain purchase amount, or as part of an advertisement. Like the other coupons described above, these coupons also have a validity period.

Concerns and envisioned support by HATS

All concerns, which are already mentioned in the coupon handling scenario, also apply to this scenario, so these concerns are not repeated here. Only those concerns which are new to this scenario are mentioned

[1] Other technologies like magnetic strips, smart cards, or RFID chips are left out for simplicity.

here.

Concern (TS-C12)**. Feature evolution**: The most important aspect of this scenario is that of a feature evolution. The existing coupon handling feature is extended by individual coupons, which are bound to a certain customer. This additional functionality should not influence the existing coupon handling functionality for anonymous coupons.

HATS should be able to model this new functionality. Ideally, it could be possible to completely factor out the additional functionality in a separate feature, which is based on the coupon handling feature and allow for different product configurations (T1.2, T2.2). It should then be possible to verify that the existing coupon handling functionality is not affected by the loyalty extension. In particular, proofs done on the original feature should be reusable (T3.1).

Concern (TS-C13)**. Privacy**: One important aspect of the loyalty feature is to ensure that the private purchase information of the customer can only be used by the loyalty system to calculate the discount. It is therefore important to save the customer information separately from product or inventory information. In addition, the system must guarantee that this information can only be used for the loyalty functionality. Certain authenticated staff members could also be allowed to access the information, building on the authentication feature described in Scenario TS4.

HATS should provide mechanisms to ensure the isolation of the customer information, from unrelated system functionality. Several different techniques have to be applied here, namely property verification on the model level (T2.5, T4.3) and corresponding implementation correctness by code generation (T1.4) or by refinement (T2.6), and enforcement of security requirements on the model level and on the implementation level, either by static analysis or by runtime checks (T4.1).

Concern (TS-C14)**. Security**: As with the anonymous coupons, it should also not be possible to create new valid individual coupons. In addition, it must not be possible for third-parties to modify the purchase sum of a customer in other ways except what has been described previously, i.e., by actually purchasing products.

HATS should provide similar mechanisms as described in Concern TS-C3.

Concern (TS-C15)**. Correctness**: As with the other scenarios, HATS should allow to formally describe the properties of this scenario and be able to verify them to a certain extend on the model and implementation level (T1.4, T2.3, T2.5, T2.6, T4.3).

3.3 Summary

The Trading System is an academic case study of moderate size. The focus of the scenarios lies on additional features and the variability of the system. Because of its size it should be possible to give a complete ABS model of the whole System. The model should be based on features, to be able to create different versions of the system. Table 3.1 gives a summary of the concerns harvested from this case study and Table 3.2 shows the relationship between concerns identified in this case study and tasks to be carried out in the HATS project.

Req. Identifiers	Req. Labels	Tasks	Reference
TS1: Coupon handling feature			
TS-C1	Correctness	1.4, 2.3, 2.5, 2.6, 4.3	Page 22
TS-C2	Failure isolation from standard sales process	1.2, 1.3, 2.2, 2.5, 2.6, 3.1, 3.3, 3.5, 4.3	Page 22
TS-C3	Security	4.1	Page 22
TS-C4	Resource	4.2	Page 23
TS-C5	System upgrade	3.1, 3.3, 3.4	Page 23
TS2: Cash desk variability			
TS-C6	Variability modeling	1.1, 1.2, 1.4	Page 23
TS-C7	Correctness	1.5, 2.1, 2.3, 2.4	Page 23
TS3: Dependability property			
TS-C8	Correct price calculation	1.4, 2.3, 2.5, 2.6, 4.3	Page 24
TS-C9	Transactional behavior	1.4, 2.3, 2.5, 2.6, 4.3	Page 24
TS-C10	Secure transactions	4.1	Page 24
TS4: Authentication policies			
TS-C11	Security modeling and analysis	1.3, 2.5, 4.1, 4.3	Page 25
TS5: Feature evolution - loyalty system			
TS-C12	Feature evolution	1.2, 2.2, 3.1	Page 26
TS-C13	Privacy	1.4, 2.5, 2.6, 4.1, 4.3	Page 26
TS-C14	Security	4.1	Page 26
TS-C15	Correctness	1.4, 2.3, 2.5, 2.6, 4.3	Page 26

Table 3.1: High level requirements harvested from Trading system case study

Work-packages	*Concerns harvested from Trading system case study*														
	TS1					TS2		TS3			TS4	TS5			
and Tasks	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1 Framework															
Task 1.1						•									
Task 1.2		•				•						•			
Task 1.3		•									•				
Task 1.4	•					•		•	•				•		•
Task 1.5							•								
2 Variability															
Task 2.1							•								
Task 2.2		•										•			
Task 2.3	•						•	•	•						•
Task 2.4							•								
Task 2.5	•	•						•	•		•		•		•
Task 2.6	•	•						•	•				•		•
3 Evolvability															
Task 3.1		•			•							•			
Task 3.2															
Task 3.3		•			•										
Task 3.4					•										
Task 3.5		•													
4 Trustworthiness															
Task 4.1			•							•	•		•	•	
Task 4.2				•											
Task 4.3	•	•						•	•		•		•		•
Task 4.4															

Table 3.2: Overview of the scenarios and concerns of the Trading system case study and related tasks

Chapter 4

Virtual Office of the Future Case Study

4.1 Overview

In a Virtual Office of the Future (VOF), office workers are enabled to perform their office tasks seamlessly independent of their current location. They are supported by a highly distributed, decentralized office infrastructure that is made up by the devices that are accessible via network by an office worker at a given point in time based on his current location. Office workers are able to use, for instance, devices like printers at their current location without knowing and installing them up-front or are able to securely exchange documents with colleagues in a meeting without using a central infrastructure. Fraunhofer IESE developed a prototype of an infrastructure aiming to support the vision of the VOF as sketched above. Following are the main characteristics of the VOF infrastructure prototype:

Peer-to-peer (P2P) style The VOF infrastructure is a distributed system according to the P2P style. The nodes in the VOF P2P network are called VOF nodes. VOF nodes are devices like laptops, mobile devices, printers, scanners, fax machines etc. that are connected to the VOF P2P network. The P2P style enables the decentralization of the VOF infrastructure and thus facilitates mobile working.

Service-orientation VOF nodes provide their functionality in terms of services. Printers, for instance, provide print services describing the features they are capable of in a service description.

Virtualization Services related to concrete devices can be virtualized. The resulting VOF service provides functionality to office workers independent of a concrete device. A VOF print service, for instance, accepts print jobs and maps them to an appropriate device based on the user's settings and context information like the user's current location. Hence, users do not need to deal with, for instance, specifics of single devices but are provided with higher-level VOF services.

Distributed workflow execution Workflows are executed in a distributed fashion. This is achieved by means of a concept called VOF documents. A VOF document is a document with attached workflow, data, and user interface. VOF documents are interpreted by VOF nodes, for instance, an office worker laptop and thus executed. Local interpretation of VOF documents on office workers laptops enables offline working.

Integration with standard office applications Office workers typically use office applications like Microsoft Office or Open Office. Hence, the VOF infrastructure is integrated with standard office applications. Microsoft Outlook, for instance, has been integrated with the VOF infrastructure as a user interface to enable its users to receive and send VOF documents.

The VOF infrastructure prototype is a framework that can be leveraged to realize support for office workflows on an underlying VOF P2P environment. A set of reusable artifacts is provided and a related production process describes how a VOF P2P environment can be set up and how new workflows can be realized and deployed on it.

4.2 Reusable artifacts

According to the product line idea, the VOF infrastructure prototype comes up with a set of reusable artifacts and a production process that can be used to develop VOF infrastructures. The following reusable artifacts are available:

VOF P2P platform The VOF P2P platform is one of the core components of the VOF infrastructure prototype. It offers functionality to provide, discover, and execute VOF services in a P2P environment. Furthermore, it offers the ability to connect existing office applications of a customer like Microsoft Outlook to the VOF infrastructure in order to use them as a UI. It is provided as a component that is ready to use and can be installed on end devices that are planned to collaborate within a P2P network.

VOF documents A VOF document is an architectural concept developed by Fraunhofer IESE to realize workflows in a Virtual Office environment. In general, a VOF document is an entity consisting of a workflow specification, files to be handled in a workflow, UIs to be used during the execution of a workflow by a user, and data related to the execution of a workflow.

XML language to specify workflows This is a language developed by Fraunhofer IESE to describe the workflows that are part of VOF Documents. Each workflow description contains the definition of activities, related roles, and data. It is based on XML and can be interpreted by a workflow engine that is part of the VOF P2P platform.

Framework for VOF document user interfaces VOF Documents contain user interfaces used during workflow execution by the respective user. The framework for VOF document user interfaces consists of libraries, APIs, and tools, and supports developers during the implementation of user interfaces in form of wizards and wizard pages.

Plug-in for MS Outlook This plug-in enables MS Outlook to be used in the context of a VOF infrastructure. MS Outlook is extended to participate in workflows that are executed on the underlying VOF P2P platform. VOF documents realizing workflows can be received, edited, and sent via MS Outlook.

VOF services A VOF Service provides functionality to be used in a Virtual Office environment. Each service can be deployed on the VOF P2P platforms to make it available and executable. The VOF services are used during the execution of workflows.

VOF service API This API is implemented in Java and has to be used to implement new VOF Services to become compliant with the general service specification used in the VOF P2P platform.

4.3 Scenarios

In this following section we describe concrete scenarios related to the VOF infrastructure prototype as evaluation candidates in the HATS project. The set of scenarios is not meant to be complete. It is the starting point for discussion on the VOF infrastructure prototype in the context of the work packages. During the HATS project, further scenarios should be elaborated depending on the needs implied by the ongoing work in HATS. If necessary, the scenarios will be elaborated in more detail before using them in a respective case study or experiment.

4.3.1 Scenario VF1: Realize a new workflow

One key requirement in the Virtual Office is the workflow support for office workers. For that, it must be easy to introduce new workflows into the system. Hence, FRG developed the VOF document concept, which offers the ability to realize new workflows by developing a new VOF document.

Concrete example

The concrete example we use here is an organization that wants to support requests for vacation electronically. For the realization of these examples, a VOF document has to be implemented by following the steps below:

1. **Modeling the workflow with VOF workflow language**

   For the vacation workflow, the **activities** are:

   - Filling out the vacation request form;
   - Approve or Not approve the request by the department head;
   - Forward the document to the administration;
   - Send the document back to the requester.

   The **roles** are:

   - Requester;
   - Department head;
   - Administration.

   The **data** involved are, for example:

   - Requester personal data;
   - Time period for vacation;
   - Contact in vacation period;
   - Substitute for the requester during the vacation period;
   - Name of the department head.

2. **Implementation of user interfaces**

   For the vacation workflow, four user interfaces are required:

   - Request form;
   - Approval or disapproval form;
   - Administration form;
   - Summarize form.

3. **Identify services**

   For the vacation workflow, two services are required:

   - Retrieve the name of the requester's department head;
   - Check the remaining vacation days of the requester.

4. **Compiling the VOF document**

   For the vacation workflow, there are basically two files to be compressed:

   - The XML workflow file containing the workflow description and the data bindings;
   - Four user interface e files (Dynamic-Link Libraries - DLL's);

5. **Deployment of the VOF document**

   The VOF document is now available in the peer network by copying it in each node in a specific folder which is accessible to all the related services.

   The services are addressed to specific nodes according to the preference of the network manager.

6. **Testing**

   A system test is performed in order to find errors in any component of the VOF document and fix them. Once fixed, the entire VOF document is recompiled.

Concerns and envisioned support by HATS

Concern (VF-C1)**. Testability:** The functionality of the workflow has to be guaranteed by testing all the possible paths (T2.5).

Concern (VF-C2)**. Availability of document-related services:** When deploying the VOF document, it has to be guaranteed that its related services will be available in the network (T2.5).

Concern (VF-C3)**. Services security:** Concerning security, one point to be considered is how to ensure that a user has access only to the service that it is supposed to. Another security issue is that the new workflow should not cause significant delay in the network, caused by an unsecured implementation of the VOF Document. For instance, the VOF Document splits up itself and sends several times to each member of the VOF (T4.1 and T4.2).

Concern (VF-C4)**. Analysis:** Models of workflows can reach high complexity with respect to the control flow and the use data types. Hence, workflow and related data models are source of inconsistencies and defects that could be prevented by analysis techniques applied to the respective models (T1.3).

Concern (VF-C5)**. Code generation for workflow realization:** Today the realization of a new workflow is concerned with many manual development activities. Code realizing the behavior of a VOF document cannot be generated so far (T1.4).

4.3.2 Scenario VF2: Add a new device type to the VOF infrastructure

Office workers use different devices during their daily work in the Virtual Office. The probability that a new device type that has not yet been used within the VOF infrastructure before is very high. A concrete example could be the integration of an iPhone as a new device type in the environment.

Concrete example

A reference architecture describes the required software components for end devices. The identified end device type (iPhone) has to be analyzed with respect to the operating system, available programming platforms (Objective C, Java, etc) and all constraints related to those issues. For instance, memory and CPU restrictions and library restrictions for programming platforms.

For adding a new device on the network, the following steps are required:

1. Implementation of software components concerning the end device type, if there is no concrete implementation available. During the development of each software component, module tests are performed for checking the functionality of each module.

2. The new implemented software product has to be installed on the end device (iPhone). Afterwards, the device has to join the P2P infrastructure for performing a system test.

Concerns and envisioned support by HATS

Concern (VF-C6)**. Portability:** We have to generate and offer different platform applications for different end devices (T1.2).

Concern (VF-C7)**. Device security:** The overall system has to know if the new node is trustable or not in reference to the provided services and its behavior in the network (T4.1).

Concern (VF-C8)**. Hybrid analysis for evolvability:** New devices are supposed to be integrated to a VOF infrastructure at runtime. This bears the risk that the integration in a partially unknown environment leads to runtime errors (T3.3).

4.3.3 Scenario VF3: Realization of a new VOF service

The realization of a workflow for a certain organization can require new functionality to be provided by a new service.

Concrete example

One concrete example can be a service for the authentication of users in the P2P network. For this, the following steps are required:

1. First of all, the specification of the service interface has to be done by documenting the method signature of the service. Concerning the example:

   ```
   authenticate(user, password) : boolean
   ```

2. A concrete implementation of the specification related to existing technologies is required. For instance an authentication mechanism for LDAP- or database servers, etc.

3. Testing the new implementation of the VOF Service.

Concerns and envisioned support by HATS

Concern (VF-C9)**. Availability of services:** The functionality of the VOF is based on service consumption. It is necessary therefore to guarantee that a high availability of each service (T2.4, T4.3 and T4.4).

Concern (VF-C10)**. Variability:** Based a on a service specification, we should have a variability point to be able to select the different implementations (T2.4).

Concern (VF-C11)**. Correctness:** If the implementation of a service specification changes, the functionality of the service has to be guaranteed (T4.3).

4.3.4 Scenario VF4: Integration of a new virtual office application

The VOF P2P platform is capable to communicate with local running applications and vice versa, by providing corresponding interfaces. Based on this, each application can be adapted to connect and use the VOF P2P platform. The workflow support based on VOF documents is integrated into office applications like Microsoft Outlook.

Concrete example

As a concrete example, consider the integration of a Thunderbird client instead of Microsoft Outlook.

1. Analyze existing APIs and programming tools for extending the Thunderbird application.

2. Integrate a software component (proxy) in the Thunderbird client to encapsulate the communication between the mail application and the VOF P2P platform.

Concerns and envisioned support by HATS

Concern (VF-C12)**. Test case generation:** Automatic generation of test cases in order to verify the functionality between the new application (Thunderbird) and the platform (T1.4).

Concern (VF-C13)**. Model mining:** The integration of a new office application like Thunderbird bears the risk that because of incomplete information about the new applications' realization the integration fails or is at least error-prone. Model-mining can provide additional information on an application to be integrated (T3.2).

4.3.5 Scenario VF5: Integration of new functionality in the VOF P2P platform

The architecture of the VOF Middleware consists of a combination of different architectural styles. The most important one is the layered- and event-driven style to address maintainability and customizability requirements. Thus, changing or integrating new functionality can be done easily. But the adaptation of event flows can cause a high effort during implementation and testing phases, because the collaboration between components via events is hard to understand.

Concrete example

In this example, we integrate a security component for encrypting and decrypting messages between peers. The new functionality is optional and can be configured during runtime. For this, the following steps are required:

1. Definition of component interfaces, events, and configurability of the new security component.
2. Identify and adapt affected components related to new event flows.
3. Implementation of a security component prescribed by requirements, architecture and design.
4. Integrate configuration functionality in the UI.
5. Definition and implementation of test cases related to the updated event flow.
6. Build a new release of the VOF Middleware and deliver it to all clients.

Concerns and envisioned support by HATS

Concern (VF-C14)**. Feature modeling:** Interfaces should be defined for the new component, as well as the related events to describe the behavior (T1.2).

Concern (VF-C15)**. Variability:** Adapt variation points of existing components to guarantee seamless integration of the new security component (T2.4).

Concern (VF-C16)**. Verification of general behavior properties:** In order to check the correctness of the updated event flow, a couple of test cases should be generated to guarantee the integration of it with the overall system (T2.5).

4.3.6 Scenario VF6: Replacement of technologies

To achieve new upcoming requirements, for instance the improvement of communication performance or becoming standard compliant, it makes sense to replace used technologies in the overall system.

Concrete example

In this example we replace the proprietary implementation of VOF Services by introducing the web-service standard. For this, the following steps are required:

1. Change the communication protocol used between peers, to discover VOF Services. For instance, the unique identification for discovering services has to be changed. Today, a service can be identified by "peername::servicename". Using web-services causes a change to: http://peername:port/servicename.
2. Implement and provide components for the realization of the required environment for web-services (HTTP-Server with web-service container, etc).
3. Adapt all VOF Service implementations to become technology compliant.
4. Generate and replace all proxy components that encapsulate the communication technology, which is used for service execution.
5. Build a new release of the VOF Middleware and deliver it to all clients.
6. Rebuild and deliver all service implementations.

Concerns and envisioned support by HATS

Concern (VF-C17)**. Code generation:** To keep the compliance among the services in the network, it should be specified with ABS how each service should behavior with the introduction of a new technology. It means that parts of the source code should be replaced by code generated using ABS specification (T4.3).

Concern (VF-C18)**. System derivation:** Based on the new introduced communication technology, all the proxy components should be replaced in order to achieve the compatibility among the VOF applications and the middleware (T1.4).

Concern (VF-C19)**. Configurable deployment architecture:** Once a new version of the VOF middleware was built, it has to be guaranteed that the version of the VOF middleware was deployed for all the nodes to avoid incompatibilities (T2.1).

Note that Fraunhofer IESE aims at providing tool-support for the development activities required to perform each of the scenarios.

4.4 Summary

This chapter introduced the Virtual office of the future case study to be used as validation example in the HATS project. The case study is an example for a software product line in the information systems domain. We introduced the available reusable artifacts, for instance, the VOF P2P platform, VOF documents, or VOF services and in particular 6 scenarios that can be leveraged for the validation of HATS results. The scenarios describe typical situations we have experienced in working with the Virtual Office prototype and its evolution over time. The case study provides input to WP1-WP4. Table 4.1 gives a summary of the concerns harvested from this case study and Table 4.2 shows the relationship between concerns identified in this case study and tasks to be carried out in the HATS project.

Req. Identifiers	Req. Labels	Tasks	Reference
VF1: Realize a new workflow			
VF-C1	Testability	2.5	Page 32
VF-C2	Availability of document-related services	2.5	Page 32
VF-C3	Services security	4.1, 4.2	Page 32
VF-C4	Analysis	1.3	Page 32
VF-C5	Code generation for workflow realization	1.4	Page 32
VF2: Add a new device type to the VOF infrastructure			
VF-C6	Portability	1.2	Page 33
VF-C7	Device security	4.1	Page 33
VF-C8	Hybrid analysis for evolvability	3.3	Page 33
VF3: Realization of a new VOF service			
VF-C9	Availability of services	2.3, 4.3, 4.4	Page 33
VF-C10	Variability	2.4	Page 33
VF-C11	Correctness	4.3	Page 33
VF4: Integration of a new virtual office application			
VF-C12	Test case generation	1.4	Page 34
VF-C13	Model mining	3.2	Page 34
VF5: Integration of new functionality in the VOF P2P platform			
VF-C14	Feature modeling	1.2	Page 34
VF-C15	Variability	2.4	Page 34
VF-C16	Verification of general behavior properties	2.5	Page 34
VF6: Replacement of technologies			
VF-C17	Code generation	4.3	Page 35
VF-C18	System derivation	1.4	Page 35
VF-C19	Configurable deployment architecture	2.1	Page 35

Table 4.1: High level requirements harvested from the VOF case study

Work-packages and Tasks	*Concerns harvested from VOF case study*																		
	VF1					VF2			VF3			VF4		VF5			VF6		
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1 Framework																			
Task 1.1																			
Task 1.2						•								•					
Task 1.3				•															
Task 1.4					•							•						•	
Task 1.5																			
2 Variability																			
Task 2.1																			•
Task 2.2																			
Task 2.3									•										
Task 2.4										•					•				
Task 2.5	•	•														•			
Task 2.6																			
3 Evolvability																			
Task 3.1																			
Task 3.2													•						
Task 3.3								•											
Task 3.4																			
Task 3.5																			
4 Trustworthiness																			
Task 4.1			•				•												
Task 4.2			•																
Task 4.3									•		•						•		
Task 4.4									•										

Table 4.2: Overview of the scenarios and concerns of the VOF case study and related tasks

Chapter 5

Fredhopper Case Study

In this chapter we present the industrial software system from Fredhopper and study general scenarios based on this software system. For each scenario we identify corresponding concerns and hint where in the HATS project these concerns may be addressed.

Specifically in Section 5.1 we provide an overview of Fredhopper's software system; in Section 5.2 we present several general scenarios that are typical during the development and support process of the Fredhopper software system. For each scenario, we detail concerns identified, which may be addressed in the HATS project.

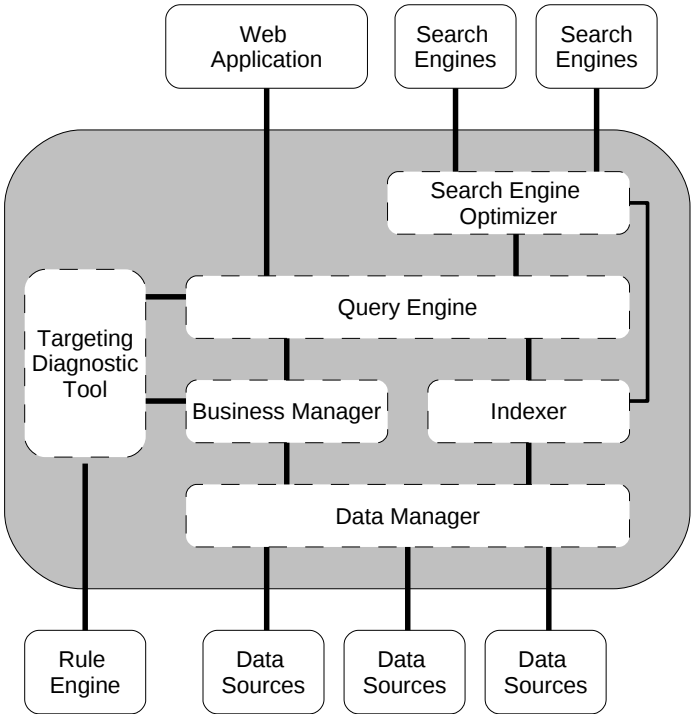

Figure 5.1: Architecture of Fredhopper Access Server

5.1 Overview

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 FAS in the following way: *query engine*, *business manager*, *indexer*, *data manager*, *search engine optimizer* and *targeting diagnostic tool*. An architectural view of how these components are related in FAS is shown in Figure 5.1. In the following paragraphs we describe informally the functionalities of each component:

- *Business Manager* The business manager component 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 specialist 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 transformation such as normalization and aggregation etc, and to save transformed data as FAS input XML.

- *Indexer* The indexer component is composed of three subcomponents – XML loader, search indexer and tree builder. The XML loader takes an input textual (XML) description 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 index structure, allowing efficient search. The tree builder is responsible for constructing 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* method to guide (web crawler) search engine when indexing web sites with faceted navigation. Faceted navigation provides users 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* 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.

We fix some basic terminologies when referring to stakeholders to facilitate presentation of the case study. Figure 5.2 shows a basic view of stakeholders interactions. Specifically there are two levels of interactions – between FAS and business *clients* and their web applications, and between clients and their *customers*. A *user* in this case may be either a client or a customer.

In the following section we describe general scenarios in FAS as evaluation candidate in the HATS project. The set of scenarios is not meant to be complete. It is the starting point for discussion on FAS in the context of the work packages. During the HATS project further scenarios should be elaborated depending on the needs implied by the ongoing work in HATS. If necessary, the scenarios will be elaborated in more detail before using them in a respective case study or experiment.

5.2 Scenarios

In this section we select several scenarios from the development and support process of FAS, and using which we details the technical concerns that the HATS framework should address. Below we briefly overview the scenarios:

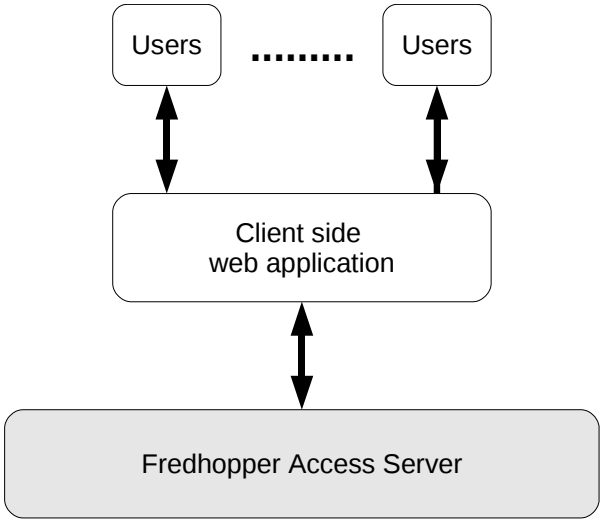

Figure 5.2: Basic view of stakeholders interactions

1. In Scenario FP1 we study the need to establish the correctness in sequential programming. This is described in Section 5.2.1.
2. In Scenario FP2 we study how a new component is integrated to FAS. Unlike Scenario FP1, this scenario concerns with ensuring correct and safe interaction between components, each of which contain many classes and methods. This is described in Section 5.2.2.
3. In Scenario FP3, we study how testing systems are provided when a new feature is added to an existing component in FAS. In particular we focus the provision of test system for the targeting diagnostic tool described in Section 5.1. This scenario is described in Section 5.2.3.
4. While Scenario FP1 concerns with the correctness of sequential behavior of FAS, Scenario FP4 focuses on concurrency issues in multithreaded code. This scenario is described in Section 5.2.4.
5. In Scenario FP5 we study the correctness and integration issues that arise from utilizing third party libraries. This is described in Section 5.2.5.
6. While scenarios FP1 to FP5 concern with functional correctness of FAS, Scenario FP6 focuses on the performance issues that must be addressed during the development process of FAS. This is described in Section 5.2.6.

5.2.1 Scenario FP1: Correctness of sequential programs

In this section we study a small fragment of programming code for a critical region of FAS. A critical region is a fragment that is executed most frequently and which other parts of the programming code highly depend on. Therefore the correctness of a critical region's behavior is particularly important. In FAS, behavioral correctness is checked by performing extensive testing. Testing includes a mixture of unit tests, automated integration tests, and manual testing. However, software testing is a technique, which can only show the presence of errors and not their absence. The guarantee provided by software testing hence is not sufficient when the code in question belongs to a critical region of a software system.

In the following section we consider a small fragment of code from the targeting diagnostic tool. We first give an overview of the diagnostic tool before describing the particular fragment.

```
1 public interface InferenceAnalyser {
2   public void queryTrial(Object query);
3 }
```

Listing 5.1: An excerpt of the public interface `InferenceAnalyser`

In FAS we use a business rule inferencing engine to define complete business configurations that regulate how FAS performs queries and how FAS structures its response information. To implement the inferencing engine, we use a third party implementation of a rule engine, in order to reduce the cost of developing and maintaining such a complex piece of technology ourselves. However the inferencing process carried out by the rule engine can be complex. This mean it would be very difficult for business clients to know exactly how their business rules are evaluated. We therefore provide a targeting diagnostic tool to collect information about the business rules during the inferencing process. The tool has the goal to display this information in a usable to the clients manner.

```
1 public class InferenceAnalyserImpl implements InferenceAnalyser {
2   private RuleEngine ruleEngine = RuleEngines.createEngine();
3   public void queryTrial(Object query) {
4     ruleEngine.publish(query);
5     for (Information stat : CollectorHolder.getCollector().getStats())
6               ServletFactory.getPage().displayInformation(stat);
7   }
8 }
```

Listing 5.2: An excerpt of an implementation of the interface `InferenceAnalyser`

During diagnosis, a business client submits a query input in the form of a URL to the diagnostic tool. The tool then publishes input to a third party rule engine for inferencing. While the inferencing process is being carried out, the targeting diagnostic tool would collect information about each matching business rule. This information includes the identification of the business rule as well as a break down of the parts of a business rule that are satisfied and parts that are not.

Listing 5.2 shows the code fragment to be considered in the following section. It is an excerpt of the class `InferenceAnalyserImpl`, which is responsible for accepting the query input from the business client. To encapsulate our implementation we expose this feature via the `InterfaceAnalyser`. An excerpt of the interface `InterfaceAnalyser` is shown in Listing 5.1. Specifically the method `queryTrial()` takes an URL and publishes it to the rule engine, which would carry out the inferencing procedure on this URL against a set of business rules. During the inferencing, a `Collector` object is responsible for storing information about evaluated business rules. The interface `Collector` is shown in Listing 5.3. After the inferencing, the method `queryTrial()` retrieves the information collected during the inferencing session and renders this information on a web page to the business client. To retrieve the information collected during inferencing the method `queryTrial()` calls the static method `CollectorHolder.getCollector()` in line 5 from Listing 5.2.

```
1 public interface Collector {
2   public List<Information> getStats();
3 }
```

Listing 5.3: An excerpt of the public interface `Collector`

Concrete example

In this section we use the code fragments introduced above to study the development and support process of a critical section of a component in FAS. In particular we consider on the current method for guaranteeing the correctness of the critical section in the following steps:

1. Before a component is integrated to FAS, corresponding unit tests must be passed. Each unit test is designed to test a unit of functionality in the component and this includes method level tests. In general given a class `A` containing the method `X(a1,..,an)`, which returns an object of type `B`, a corresponding `ATest` is implemented with the method `testX()`, which runs method `X(a1,..,an)` and make assertion on `B`. For each input argument `a1,..,an`, either a mock or a fake object is provided by the test.

   Step example: *Assuming the set of business rules in the rule engine is nonempty, we must ensure that the diagnostic tool component displays at least static information about those rules. This means the call to `getStats()` (line 5 of Listing 5.2) in the method body of `queryTrial()` must return a nonempty list. A corresponding unit test is written and is shown in Listing 5.4. Lines 2 and 3 of the listing show declaration of data (fake) objects that are used for these tests. These are predefined URLs, for testing the diagnostic tool. Note that the `ruleEngine` contains default nonempty set of business rules.*

2. Once both unit tests and integration tests have been carried out for the component, it would be passed on for performance analysis. Concerns related to performance analysis are discussed in Section 5.2.6. After testing it performs okay, the component is committed to the next release of FAS. If an error is discovered after the release, the error, along with a test case (instruction) for reproducing the error in the system, will be reported back to developers, who would attempt to identify the cause of the error by debugging the system using the test case given. This usually involves setting breakpoints in the code and stepping through them until the cause of error can be identified. Once the error has been corrected, new unit tests would be implemented to allow automatic detection any future regressions.

   Step example: *After a release of FAS including the diagnostic tool, an error was reported by the business clients. Specifically, there was a null pointer exception during the execution of the method `queryTrial()`. By debugging this entire component, it emerges that there was possibility an integer counter in the `Counter` object could some time fail to be reset. This has resulted in `getStats()` returning a null object. Subsequently the method for collecting information (not shown in the listings) has been changed and additional data objects were added to the unit test, which could reproduce the error, to ensure that the updated component does not regress.*

```
1  public class InferenceAnalyserImplTest {
2    @DataPoint public Object testquery1 = ...
3    @DataPoint public Object testquery2 = ...
4
5    @Theory
6    public void testqueryTrial(Object query) {
7      RuleEngine ruleEngine = RuleEngines.createDefaultEngine();
8      ruleEngine.publish(query);
9      assertTrue(! CollectorHolder.getCollector().getStats().isEmpty());
10   }
11 }
```

Listing 5.4: An excerpt of the unit test for `InferenceAnalyserImpl`

Concerns and envisioned support by HATS

In the example above we have identified a situation in which it becomes necessary to give complete guarantee that the call to the get method `getStats()` in `queryTrial()` does not return a null object. This requires method and technique to make formal behavioral assertions (properties) about units of (sequential) code; a unit here may be a class, an interface, a method or even several lines of code within a method. These properties may range from more general ones such as termination or to more specific ones about safety (something bad cannot happen) and liveness (something good must happen). One way of specifying formal assertions about sequential programs is via *preconditions and postconditions*. For example, in the concrete example of Scenario FP1 we would require the ability to assert the postcondition `\result != null` to the method `getStats()`, that is, the returned value cannot be null. Also we would require to assert the following invariant in between line 4 and line 5 of Listing 5.2.

```
CollectorHolder.getCollector().getStats().size() > 0
```

These assertions must then be formally *proved* to ensure that these assertions are met at all times. To this end we envisage HATS framework would address two major issues – *specification* and *verification*

Concern (FP-C1)**. Specification of sequential programs:** The HATS framework should provide the mechanism for specifying behavior of units of code precisely, as identified in the concrete example above. One way of providing this is to allow the specification of pre/postconditions of units of code. Traditionally these conditions are asserted as predicate or temporal logic expressions, but this requires knowledge, which cannot be assumed to be accessible to software developers and testers. Therefore HATS should deliver usable techniques and tool support to assist the generation of behavioral properties so that formal specification might become more amenable. In addition HATS should provide the tool support for harvesting behavioral properties from existing unit tests. (T1.1, T1.3, T1.5 and T4.3)

Concern (FP-C2)**. Verification of sequential programs:** The HATS framework should provide the mechanism to breach the gap between specifying and verifying program correctness. The need for verification of behavioral properties has been identified in the concrete example above, where pre and postconditions of method invocation should be formally proved. Similar to specification, HATS should incorporate usable techniques and tools (e.g. a model checker) to assist verification processes. However it is known to be too expensive to carry out verification directly at implementation level. We therefore would expect HATS to provide the mechanisms to (a) move both verification and respective specification processes up to an abstract behavirol (ABS) model level, and (b) consequently transfer verified properties back to the code level. (T1.1, T1.4, T1.5, T3.2 and T4.3)

5.2.2 Scenario FP2: Integration of a new feature component in FAS

In this section we consider a scenario about defining strict protocols for interaction between two components. Fredhopper defines behavioral interaction between components by using *glue code*. A piece of glue code of a component serves three purposes:

- Wiring – invoking functionalities of another component, through class instantiation, method invocation and interface implementation;
- Listening – reacting to calls such as method invocation and class instantiation from another component;
- Coordinating – defining the behavior in between invocations and reactions.

Very often the interaction of two components would require multiple pieces of glue code and they would not all be situated in one place in the system but scattered around different components. As a result, it becomes very difficult to manage the glue code and any changes to any one of the components such as addition and modification of functionalities would require careful analysis on the corresponding changes to the glue code.

In the following section, we study how to integrate the new targeting diagnostic tool into the FAS business manager application. In Section 5.1, we have described the purpose of the targeting diagnostic tool. Specifically we have developed a diagnostic tool to collect information, such as the internal rule structure of each inferencing session; a view of the inferencing session is important as we need to provide useful information to clients about when and why a rule's action is not executed. Moreover we do not manage the implementation of the third party rule engine, and we used an aspect-oriented approach [22] to collect information about when and why a rule's action is not executed.

Concrete example

This section looks at the process of integrating a new component to FAS in the following steps:

1. Component interaction identification. Before a new component is integrated to FAS, we must understand which existing components from the current system would interact (both invoking and reacting) with the new component. This must be addressed from the point of view of the new component as well as the components currently in FAS.

   Step example: *We must ensure that information collected by the diagnostic tool is properly shown to the business clients clients. Specifically we want the information collected to be useful and easily accessible. Furthermore, we want to easily control how the information is presented. For this a new functionality has been added to the business manager component. This functionality is exposed as a JavaServer page in the management console.*

2. Management of stateful components. If a new component integrated to FAS contains stateful information, we must be able to identify when to update this information. This is so that the component would always process new information and output correct results. The knowledge of when to update this information is often based on the informal documentation of the new component.

   Step example: *During an inferencing session, the rule engine evaluates the business rules against the query supplied by the busienss client. Each evaluation of a business rule would trigger the diagnostic tool to record the rule. Note that each business rule might be accessed multiple times by the rule engine within one inferencing session. Note that information about a business rule does not change within the same inference session. For diagnostic purposes it is therefore sufficient to only record an evaluated busines rule once. This means any subsequent evaluation of the rule within the same inference session would therefore not be monitored. Conversely, in between any two inferencing sessions, business rules may be added to and deleted from the rule engine. Moreover, each inferencing session might evaluate different business rules. Therefore the diagnostic tool must not retain the records of business rules evaluated in between inferencing sessions. This ensures that our diagnostic tool always contains correct information about each inferencing session.*

3. Semantics of glue code. We define the glue code for enforcing the interaction between the new component and existing ones. The semantics of this glue code is derived from the informal documentation and prototype source code.

   Step example: *To ensure that the correct information about an inferencing session is displayed to the business clients, we provide the glue code to coordinate the process. This requires the knowledge of what and how data is stored in the diagnostic tool. We also need to have knowledge about the types of data stored in the tool so that data is retrieved and displayed properly to the business clients.*

4. Software evolution. If there is a change to a component in FAS, the glue code that defines the interaction between this component and the rest of the system must be updated so that the change is incorporated without introducing regression to the component. In FAS, we derive the semantics of changes in the glue code based on informal documentations and prototype source code.

   Step example: *During each inferencing session, the diagnostic tool provides information about the business rules evaluated. However, as the number and the complexity of business rules increases, the*

complexity of the information provided by the diagnostic tool also increases. Therefore it becomes very difficult for business client utilize the collected information. For example, it becomes difficult to correlate different business rules evaluated in an inferencing session. To assist this analysis, a graph visualization feature is added to the diagnostic tool. This feature allows evaluated business rules to be viewed as a graph[1]*. To cater for this new feature, the glue code between the diagnostic tool and the rest of the system must be updated. The new glue code must take into consideration how the graph object should be retrieved and displayed to the business client.*

Concerns and envisioned support by HATS

Currently interactions of components are specified and enforced by hard-wiring glue code to components. Moreover, the glue code is scattered amongst the components in question. Since the definition of the interactive behavior between these components are based on the intuition of the developers, who have implemented the components, it is very difficult to enforce any notion of correctness to it. Moreover, a change to any one of the components in questions by adding new feature or modifying existing one could result in multiple changes to the glue code. Again implementation of these changes are based on the intuition of the developers, who have made the changes to the components. From the above scenario and concrete example we have therefore raised the important question of *correctness* and *maintainability*.

Correctness There is a lack of precise understanding of both behavior of individual components and their interaction. As such it becomes notoriously difficult or even impossible to provide guarantee to the correctness of the interaction. While testing, discussed in detail in Section 5.2.3, may provide certain level of validation, it is not exhaustive and it is only based on an informal and intuitive understanding of the intended behavior implemented by the interactions. Therefore, we envisage that the HATS framework would provide the supports for generation of interaction glue code, such that the following information about the glue code could be ascertained:

Concern (FP-C3)**. Correctness of interactions:** Whether the interaction behavior is correct with respect to the *intended behavior*; This requires a formal specification about the intended behavior. In formal engineering methods the specification of the intended interaction behavior is called a *behavioral contract* and has been studied as a formal notion extensively in the context of service-oriented architecture [8]. Similar notion of contractual behavior could be introduced in the HATS framework to provide the mechanics for specifying intended interaction behavior. This specification may then be used to ascertain correctness of the implementation glue code. Orthogonally while specification is carried out at the model level, HATS framework should also provide facility to guide the generation of glue code from the model's specification. (T1.1, T1.3, T1.4, T1.5 and T4.3)

Concern (FP-C4)**. Behavioral compatibility:** Whether the interaction behavior between components would invalidate the behavior of individual component; To ascertain this information, it must be possible to produce a high-level specification of behavior of individual components as well as a formal specification of the *intended* interaction behavior. In the area of formal engineering methods correctness of interaction behavior is known as *behavioral compatibility* [7], which is a binary relation such that a component P is said to be compatible with another component Q if and only if P does not cause Q to deadlock where deadlock freedom is assumed to be a desired property. Using results from studying behavioral contract, the HATS framework should provide the methodology and technique for reasoning about these properties as well as generating the implementation code from *compatible model*. (T1.1, T1.3, T1.4, T1.5 and T4.3)

Concern (FP-C5)**. Incremental validation:** Additional desired properties could be provided about the interaction behavior by further studying the notion of behavioral compatibility. For example, given three components P, Q and R, it would be useful to show that the interaction behavior between P and Q could not invalidate the interaction behavior of Q and R. This is a notion of incremental development of software

[1] The rule engine stores business rules as a Rete network [14].

systems and has already been studied at a more abstract level [28]. We envisage that one of the results of from HATS project is the analysis of similar formal notion in the object oriented setting. This is so that not only these properties could be ascertained at an abstract (ABS) level, but corresponding implementation code could be generated with the same guarantee. (T1.1, T1.3, T1.4, T1.5 and T4.3)

Maintainability As suggested by our scenario and concrete example, glue code of interactive components is often scattered around the structure of individual components. This results in low maintainability of the glue code. Specifically if there are changes in individual components, it would become very difficult to implement corresponding changes to their glue code. To this end we envisage that the HATS framework would provide the supports for maintaining the structure of glue code and managing the changes to the glue code.

Concern (FP-C6)**. Evolvability and code generation:** By modeling the changes to component's behavior, it should be possible to either deduce the changes to the glue code from both the model of change and the behavioral contract. This deduction should lead to some form of (guided) code generation. We envisage the HATS framework would provide the methods and tools for managing such changes. (T1.4, T1.5, T3.1, T3.3, T3.5 and T4.3)

Concern (FP-C7)**. Tool support for navigability:** The HATS framework should provide the tool support for managing the structure of glue code of any two interacting components. Unlike correctness concerns, here we are concerned with improving navigability of the glue code. For example, the HATS framework could provide the tool support to visually relate a piece of glue code to its corresponding behavioral contract. As the glue code and its behavioral contract become complex, the provided tool support should ease the validation and the verification of the glue code. (T1.4, T1.5 and T4.3)

5.2.3 Scenario FP3: Test system provisioning

In this section we consider the scenario on the generation of test systems. In FAS, a test system consists of the following three testing procedures:

- Automatic unit testing – during development of a component, unit tests are written to validate the correctness of the component and to detect regressions. A unit test exercises a unit of functionality and makes assertions about the state after the execution of that unit. Unit tests are written as a program. They are executed automatically when the component containing the units of code re-compiles. Unit tests make assumption about the state of the system before the unit's execution. Therefore these tests may be run independently from the rest of the system.

- Automatic integration testing – after a component is unit tested, it would be integrated to the rest of the system. To ensure that this integration does not introduce errors to the system, the integration must be tested. An integration test exercises several units of functionality across many components in the system to ensure that components still perform correctly.

- Manual testing – both unit and integration testings are executed automatically each time the system is re-compiled. After automatic testings, the system must be manually tested. These tests are performed by testers, who would follow detail test cases to exercise different functionalities of the system.

In this section we focus on the provision of unit tests. Unit tests are somewhat *independent* from the "specification" of the component. This means changes to a component would require manual and idependent changes to its unit tests. When the component does not have a precise specification, it is difficult to implement "correct" change for its unit test. In the following section, we follow the development and support process of the targeting diagnostic tool.

Concrete example

This section looks how we generated and maintained the test systems of the targeting diagnostic tool in the following steps:

1. When a component is added to FAS, unit tests are implemented for every unit of functionality of the component. These unit tests are written manually based on informal documentation of the component. Each unit is then tested using its unit test before being approved and integrated into FAS.

   Step example: *Currently we provide a certain level of guarantee about the correctness of the diagnostic tool by running unit tests on several examples. Each unit test initializes the diagnostic tool by generating particular types of rules (rules with/without conjunction, with/without negation, with/without variables, with/without expression). The unit test asserts expected properties on the inference results for each example.*

2. The new component is integrated to FAS and will be made available to clients.

   Step example: *A business client from a sport webshop has setup a business rule for a product promotion. The rule specifies that a special offer about running shorts should be displayed if the users access a page about running shoes. The business client then uses the targeting diagnostic tool to see if the rule about this promotion is correct. The business client carries out this test by first adding this business rule to the rule engine; he then submits a URL of a page displaying running shoes to the tool. The diagnostic tool then sends this URL to the rule engine, which carries out the inferencing on the URL. The tool collects information about this inferencing session. It let the client knows if the business rule was evaluated and whether the page displaying the running shoes would also display the promotion.*

3. When adding a new feature to a component, new unit tests should be written to specify and validate the correctness of the feature. Similar to the existing unit tests, the new tests are implemented based on informal documentation of the component. Existing unit tests may also be changed depending on whether or not the new feature would change the existing behavior of the component. After the unit tests has been changed, the component with the new feature must pass the updated unit tests before being integrated to FAS.

   Step example: *An update is included in the forthcoming release of the diagnostic tool. This update adds a graph visualization feature to the tool so that the relationship between different rules submitted to the rule engine may be viewed as a graph. This feature should not change the existing behaviors of the diagnostic tool. Therefore it is necessary for the diagnostic tool passes all existing unit tests before this update is approved.*

4. Currently when a new feature is added to a component in FAS, the complete software system must be shut down, that is, the query engine must be stopped. The old version of the component is then replaced by the updated one before FAS is restarted again.

   Step example: *To update his copy of the diagnostic tool, the business manager stops the query engine that is currently running, replace old version of the diagnostic tool with the updated version and restarts the query engine.*

5. When an update to FAS introduces a regression, the error, together with instructions to reproduce the error in FAS, will be reported back to developers. The developers would then attempt to identify the cause of the error by debugging the system using the given test case. This usually involves setting breakpoints in the code and stepping through them until the cause of error can be identified. Once the error has been corrected, new unit tests would be implemented to detect future regression.

   Step example: *After FAS has been updated, the business client continues to use the targeting diagnostic tool to see if the business rule about the promotion on running shorts is correct. However after*

testing the business rule with the URL once, the diagnostic tool no longer renders any information on any of the subsequent tests. This error is then reported back to the developers. The developers have consequently spent two person days to identify the cause of the error. The error was eventually fixed and new unit tests were added.

Concerns and envisioned support by HATS

Through the concrete example in Section 5.2.3, we have identified two major issues about the current approach in test system provision:

Generation of test systems Lack of precise understanding of the component's semantics and of the relationship between the components units of functionality. This means one could easily have incorrect assumptions about the state of the system before invoking a function of the component. At the same time this also leads to an incorrect assumption about the state of the system after the performance of that function. The argument for a formal specification has already been put forward through studying previous scenarios. Once this specification has been obtained, it should be used to assist the generation of the component's test system. We believe that such test generation method would provide better coverage over the state space of the component's execution. Therefore we envisage that the HATS framework would provide the supports for generation of test systems:

Concern (FP-C8)**. Test case generation:** Provision for assisting the generation of unit tests and manual test cases for units of code. While Scenarios FP1 and FP2 have already identified the need for a formal and complete specification of component's behavior, the concrete example above identified the need for generation of test systems from the formal specification of units of code. With a better understanding of the component behavior, it is more likely that these tests would have better coverage and would produce less false positive results. Note that we envisage there are times where automatic test generations might not be possible. In this case the HATS framework should incorporate usable techniques and tools to *guide* such tests generation. (T1.1, T1.4, T1.5 and T4.3)

Evolution of test systems Inability to change an existing test system according to the changes of the component's implementation. The concrete example shows that there is a lack of understanding to the side effects produced by changing of component's behavior. This understanding requires a high-level specification that allows better change management and consequently provides guidance to generation of corresponding test systems. To this end we envisage that the HATS framework would provide the supports for modification of component's behavior and its test systems:

Concern (FP-C9)**. Modeling behavioral changes:** Modeling of changes to component's behavior, and identification of those that could invalidate *consistency* properties. 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. Consider the concrete example described in Section 5.2.3. To maintain the precondition of the tool's operation, addition of the graphical visualization feature must be consistent. However, if the change is inconsistent, a developer should be notified about it. This also requires a formal treatment of the notion of *evolution consistency*. (T3.1, T3.3 and T4.3)

Concern (FP-C10)**. Test system evolution:** Assisting the generation of test systems for testing consistency properties. While the ideal situation is to be able to prove consistency of changes to components, this might not be tractable in practice. Consider the concrete example described in Section 5.2.3. One way to identify whether the addition of the graphical visualization feature would invalidate the precondition of the tool, is to update the test system of the diagnostic tool systematically using *a formal model of software evolution*. (T1.1, T1.5, T3.1, T3.3 and T4.3)

5.2.4 Scenario FP4: Concurrency

In Scenario FP1 of Section 5.2.1, we have identified issues regarding the lack of guarantees about correctness of sequential code. In this section we consider issues regarding concurrency in FAS. Unlike sequential programs, for which it is possible could provide test systems, for concurrent programs[2], where multiple threads are accessing shared resources, it is notoriously difficult to generate useful test systems. This is specifically detrimental when threads access mission critical sections of a component.

As a concrete example, we consider a small fragment of code from the query engine component responsible for updating the search indexes. Listing 5.5 shows an excerpt of the class `QueryEngine`. It defines the static method `receiveIndexUpdate()`, which takes an update object, containing some update instruction and data, and performs the corresponding updates to the search index by calling the method `updateSearchIndex()`. Partial definition of the method `updateSearchIndex()` is shown in Listing 5.6. For efficiency reason, we ensure that the objects calling `receiveIndexUpdate()` do not require to wait for the update to complete, that is, the call to `receiveIndexUpdate()` does not block the execution of the caller object itself. Therefore every time the method `receiveIndexUpdate()` is invoked, a thread is created to carry out the update to the search index. Note that in definition `updateSearchIndex()` shown in Listing 5.6, two level locking is implemented to ensure that *atomicity* of the update is maintained.

```
1  public class QueryEngine {
2    public static void receiveIndexUpdate(Update update) {
3      ...
4      Thread uproc = new Thread(new Runnable() {
5        public void run() {
6          QueryEngine.getSearchIndex().updateSearchIndex(update);
7        }
8      });
9      uproc.start();
10   }
11 }
```

Listing 5.5: An excerpt of an implementation of the class `QueryEngine`

```
1  public class SearchIndex {
2    private final Lock flock, slock;
3    public void updateSearchIndex(Update update) {
4      flock.lock();
5      ...
6      slock.lock();
7      ...
8      slock.unlock();
9      ...
10     flock.unlock();
11   }
12 }
```

Listing 5.6: An excerpt of an implementation of the class `SearchIndex`

[2]FAS is implemented in Java [16] primarily

Concrete example

Unlike sequential code, it is very difficult to create test case to exercise concurrent behavior, and if an error is introduced, it would be equally difficult to identify such error by conventional debugging facilities. For example, it is difficult to set up breakpoints for multithreaded code as threads interleave by definition.

Step example: *Listing 5.7 shows an excerpt of an attempt to define unit test for* `receiveIndexUpdate()`. *Specifically the unit test* `testreceiveIndexUpdate()` *invokes a list of updates predefined as data points. The method then tests if all updates are successful via the method* `hasAllUpdates()`. *Unfortunately even if we "randomize" the rate and the order in which these updates are invoked, it still provides very weak guarantee to the correctness of the updating procedure, and as a consequence there has been many errors associated with this procedure.*

```
1 public class QueryEngineTest {
2   @Test
3   public void testreceiveIndexUpdate() {
4     List<Update> updates = ... // predefined updates
5     for (Update update : updates)
6        QueryEngine.receiveIndexUpdate(update);
7     assertTrue(hasAllUpdates());
8   }
9 }
```

Listing 5.7: An excerpt of the proposed unit test for the class `QueryEngine`

Concerns and envisioned support by HATS

The cause of the issue identified in the example is that there is no precise specification of the concurrent behavior of the multithreaded program. This means that it is very difficult to diagnose errors such as deadlocks or livelocks as well as to derive test(s) for ensuring validity of the concurrent behavior and detecting regressions. Similar to sequential programming, we would like to be able to assert precise behavioral correctness about units of code that involve consistency. These properties range from general state safety, *deadlock* and *livelock* freedom to more specific ones about fairness and other liveness properties. Currently each unit of concurrent code in FAS is tested via unit and integration testings, and these tests would have to be setup and implemented manually. However testings can only at best show the existence of deadlock and livelock but they cannot guarantee their absence. Therefore we would advocate the application of formal verification. In particular concurrent program should be specified formally and be verified against this specification for correctness. While complete formal verification is a software engineering ideal, existing methods may be time consuming and therefore cost ineffective. Therefore current verification method must be coupled with tools and techniques to assist generation of test systems. To this end we envisage that the HATS framework would provide the following support for concurrent programming:

Concern (FP-C11)**. Specification of concurrent programs:** Similar to the sequential programming case discussed in Section 5.2.1, the HATS framework should provide the mechanism for formally specifying units of code that exhibit concurrent behavior. One possible way of achieving this is by providing the language and calculus for modelling concurrent behavior. In the area of formal engineering methods, extensive research has been carried out towards modeling and reasoning about concurrent behavior. Notable area includes *process algebras* [1] in which software systems are modeled as algebraic processes describing their possible behavior. Behavioral property specifications are then provided as either logical expressions or abstract processes. The use of a process algebraic approach requires knowledge in abstraction, and this creates a

conceptual gap in between programming and modeling. A complementary approach is the development of implementation level language with rigorous formal foundation. This approach not only facilitates formal reasoning and associated tool support, but it is also much closer to general purpose programming languages like Java [16]. Notable research result includes the development of Creol [21]. Creol is an experimental high-level object-oriented language for distributed objects, and specification in Creol may be translated in Maude, for which various tools such as a theorem prover have been developed. However, either approach requires mathematical knowledge, which cannot be assumed to be accessible to software developers and testers. HATS should therefore incorporate usable techniques and tools to assist the generation of these behavioral properties such that formal specification becomes more amenable. (T1.1, T1.3, T1.5 and T4.3)

Concern (FP-C12)**. Verification of concurrent programs:** Similar to sequential programming discussed in Section 5.2.1, the HATS framework should provide the mechanism to breach the gap between specifying and verifying program correctness. The need for verification of behavioral properties has been identified in the concrete example above and the last paragraph has identified how concurrent behavioral properties may be specified either algebraically or logically. For HATS framework to be a usable technology, it should incorporate techniques and automated tools (e.g. a model checker) to assist the verification process. We would also expect HATS to provide the mechanisms to (a) move both verification and respective specification processes up to an abstract behavioral model level, and (b) be able to transfer verified properties back to the code level. (T1.1, T1.4, T1.5, T3.2 and T4.3)

5.2.5 Scenario FP5: Using third party library

This scenario covers the case of FAS utilizing a third party library. During the development of FAS, third party libraries are used to provide new features at a lower cost. For example, FAS uses many open source libraries of the Apache project group.

In the next section we look at the utilization of a third party library for graph layout. This library provides the algorithm to layout the graph representing the internal data structure for capturing business rules in the rule engine; a brief overview of this graph layout feature is provided in Section 5.2.2.

Concrete example

Here we identify the typical steps for utilizing third party library. For each step we provide an example relating to the concrete example.

1. We identify specific information about the library, such as the version of the library that would be used; the public and interface methods exposed by the library as well as the corresponding (informal) documentation about the potential values and types of the input arguments and return values of methods.

   Step example: *We identify the version of the graph layout library, the method for creating a graph object for laying out using the library, and the methods for adding nodes and edges to the graph object. We also identify the interface, which the library provides for implementing view navigation listeners. This allows inclusion of additional behavior when a node is clicked during graph visualization.*

2. We implement the logic to *transform* the internal FAS rules data structure to one which could be used by the graph layout library.

   Step example: *The diagnostic tool contains information about business rules stored in the rule engine. We implement the transformation procedure to convert this information into a graph object for visualiation using the graph layout library.*

3. We implement the code to invoke the graph layout library's functionalities and integrate this code into our existing component.

Step example: *To implement the code for setting up the layout, we studied demo programs provided by the graph layout library. This included identifying the correct algorithm provided by the library for laying out the nodes of the graph, choosing the color scheme of the layout, and implementing the listener for allowing the diagnostic tool to provide specific information based on the particular node pressed. We also implemented the actual selection listener, that contains the programming logic to relate the diagnostic information with the node selected by a user in the graph visualisation.*

4. We write unit tests and manual test cases for both the third party library's as well as the programming logic (glue code) that we have implemented for transforming data structure and invoking the library logic.

   Step example: *We implement unit tests to validate the following programming logic*

   - *The transformation procedure for converting the business rules stored in the rule engine to the graph object, which the library accepts. We must ensure that the graph repects how business rules are related in the rule engine.*
   - *The glue code for setting up the display and implementing the functionalities provided by the user interface. Specifically when business rules store in the rule engine are visualized as a graph, each rule is represented as a node in the graph. We must ensure that when a node is selected, diagnostic information about that business rule is displayed correctly.*

5. For every change and version update to the third party library, we review the changes manually by studying release note and updated documentation on the library's API. We then implement the necessary changes to the data transformation procedure and the glue code, as well as the changes to the test systems. We execute the unit tests on the updated code before approving it to enter the product.

   Step example: *A change to the interface for adding nodes and edges to the graph object requires corresponding changes to the transformation procedure. This updated procedure is then unit tested.*

Concerns and envisioned support by HATS

The graph layout library documentation provides informal textual information about the functionality of its interfaces as well as information about the potential side-effects that may arise from invoking these interfaces. It is necessary to be a able to provide a more precise description of the third party library's functionality. Furthermore, we want to provide means to handle unanticipated changes to library's internal and/or external behaviors. In the rest of this section we highlight specific issues about *functional correctness*, *integration* and *testing* from using third party libraries.

Functional correctness Our components must instantiate the libary classes and invoke its methods through public interfaces according to the library's "intended behavior". That is, we must understand the functionality of each public interface, and the side-effects that may be caused by invoking these interfaces. Unlike Scenarios FP1 and FP2, where one might be able to provide precise behavioral description of methods down to *class level*, third party libraries are black boxes and as such we have no control with internal behavior or any behavioral changes. Therefore it is necessary to have the ability to model and specify behavior at the interface level. To this end we envisage the HATS framework would provide the supports for addressing these issues at the *interface level*:

Concern (FP-C13)**. Partial specification:** Modeling of black box components and specification of behavioral properties such as pre/postcondition and invariants about them. Note that unlike the case presented in Scenario FP1, it is often impossible to provide a complete behavioral specification of a black box component. Therefore, the HATS framework should allow models to have *formal but incomplete specification.* (T1.1, T1.5 and T4.3)

Concern (FP-C14)**. Model mining:** Derivation of models (model mining) from black box components such that black box components may be described in the same way as white box components but with higher level of abstraction. (T3.2)

Concern (FP-C15)**. Evolution of black box components:** Provision of guidance to model evolution due to unanticipated changes of black box components. Unlike the case introduced in Scenario FP1, it often is impossible to enforce boundary of changes to black box components. HATS framework should therefore identify changes to the black box components via model mining and provide useful information based on some measure of model difference. (T3.1, T3.2, T3.3 and T3.5)

Integration To integrate a third party library into FAS, we manually provide the programming logic for two tasks: a) *transforming* the data models between the third party library and FAS, and b) *coordinating* the interaction between the library and FAS. This means the orders in which the library's interface methods are invoked.

We aim to factor out the glue code from the components interacting with the third party library. This would allow better management of the glue code as well as adaptation to changes the third party library's APIs. While integration of new components has been addressed in Section 5.2.2, it is more difficult to anticipate the changes of the third party library. These unanticipated changes would make the integration of the library to FAS much harder. To this end we envisage the HATS framework would provide the following supports:

Concern (FP-C16)**. Modeling data transformation:** Modeling of the transformation between the data models of the third party library and that of FAS. For doing so, the HATS framework should provide (guided) code generation for implementing the transformation procedure. In our concrete example, given that we have a specification of the abstract syntax of the internal data structure of the rule engine and the abstract syntax of the graph object for layout, it should be possible to specify declaratively the *rules* for transforming between these data models. This declarative specification could then be used to generate the implementation code of the transformation. (T1.1, T1,4, T1.5 and T4.3)

Concern (FP-C17)**. Modeling interactions with black box components:** Modeling of interacting behavior between black box components and the rest of the systems. The HATS framework should be able to provide (guided) generation for the program code for handling the interaction between the black box components and the rest of the systems. In our concrete example in Section 5.2.5, we can see how it may be difficult to validate the correctness of program code for a) *transforming* the data models between the third party library and FAS, and b) *coordinating* the interaction between the library and FAS. This becomes extremely difficult without a precise specification of the program code. Therefore we must first provide a formal specification of the interacting behavior between the third party library and FAS. The HATS framework should then provide a tool support to assist the generation of the program code. (T1.1, T1.4, T1.5 and T4.3)

Concern (FP-C18)**. Propagation of changes:** Modeling evolution due to unanticipated changes to black box components. Specifically evolution must be formally specified and properly propagated to the models of the program code for a) transforming the data models between the third party library and FAS, and b) interacting behavior between the third party library and FAS. Similar to the evolution of test systems, how evolution are propagated to the program code must be formally studied. This is so that evolutions could maintain consistency property. (T3.1, T3.2, T3.3 and T3.5)

Testing In Section 5.2.3, we have identified the need for the HATS framework to provide methodology and technique to assist the generation of test systems for individual components. This requirement should be extended to third party libraries. Specifically the HATS framework should provide support to assist the generation of the test system for the program code for transforming data models between the library and FAS and for coordinating their interaction. The problem with testing behavior involving third party

libraries is that tests are often written *independently* from the third party library's "specification". This means it becomes notoriously difficult to ensure that we do not get false positive results. Furthermore, it is very difficult to make *changes* to the test system while preserving consistency properties. To this end we envisage the HATS framework would provide the following:

Concern (FP-C19)**. Test case generation:** Provision of (guided) code generation of test systems based on properties like compatibility specified using the HATS framework. Unlike maintaining consistent evolution of white box components, it is more difficult to prove total correctness of black box components. In our concrete example, instead of manually implementing unit tests for the graph layout library and corresponding transformation procedures and glue code, once a partial specification of the library interfaces has been obtained, the HATS framework should be able to provide guidance for generating unit tests and manual integrated test cases for rule engine interface as well as the interaction between the rule engine and the diagnostic tool. (T1.1,T1.4,T3.1,T3.4 and T3.5)

5.2.6 Scenario FP6: Performance

We now consider the scenario, which covers the performance issues that arise in the development process of FAS. In the current development of FAS, there are two main metrics for performance benchmark: *response time* and *queries per second*. Those two elements form the FAS's *service level agreement*.

Response time For an individual interaction with FAS, the main unit of measurement is response time. In particular it is the time it takes for a request to enter FAS and FAS to prepare a response to return to the user. Note that this is not the total time an interaction would take. FAS is a server-based software system. Therefore clients would normally implement specialised applications to support such interaction. Consequently the computation of such applications would add overhead to the *total response time*. Other overheads such as network latency would also contribute to the total response time.

Queries per second FAS is a server-based software system, which provides the capability to allow the client to serve multiple requests concurrently. Queries per second is the number of requests completed by FAS per second. Note that this measurement is orthogonal to response time of individual request and as such does not imply bounds to response time. For example, if FAS receives ≥ 20 requests per second, it guarantees completion of ≥ 20 requests per second, that is, 20 queries per second, while the response time of individual request might take up to 200 milliseconds to complete.

As a concrete example, we study the development process of FAS, in particular we look at how performance analysis is carried out in Fredhopper.

Concrete example

In this section we describe how the performance of a particular functionality of FAS is analysed. To maintain the service level agreement, rigorous performance analysis has to be carried out at different levels of granularity. During performance analysis a library of *stress test cases* are executed. Each test case defines the workload, number of requests as well as where and when measurements are taken. Measurements are taken for certain units of activity; here an activity is a sequence of instructions executed in FAS. For each activity, the following measurements are taken:

- Activity duration – This is the amount of time an activity takes to complete. This is measured by taking a timestamp via `System.currentTimeMillis()` before and after the execution of the activity.
- Triggered activity duration – If an activity takes longer than expected, that is, longer than its *threshold* value.

- Activity occurrences – This measures the number of times and/or the number of locations where the completion of a particular activity may be recorded.

Each piece of measurement is identified with an activity. Identification is in terms of source code location.

- Stack trace location – Stack trace location provides accurate location in the source code, Since the procedure for obtaining a stack trace of the current thread by calling `getStackTrace()` of the current thread object would cost several milliseconds, this would be taken into consideration when carrying out analysis.
- User-defined location – A unique identifier, specified by the developer/tester, could be assigned to a particular location
- Request identifier – This is a unique identifier composed of identifier of the current thread and an integer specifying the number unique of requests processed by this thread.

For every measurement we produce an individual report containing the measurment's unique identifier, its value, and the location where it is taken. Certain measurements may be aggregated, for example, if it measures the number of events at the same location. For every test case, some standard statistical calculations (average, standard deviation etc.) are made on the set of measurements obtained, and also corresponding (average) response time and (average) queries per second are obtained.

Step example: *FAS defines a `PageServlet` class, which is a subclass of `HttpServlet`[3], for creating a frontend web page for response for FAS. It overrides the method `doGet()`, which is called by the server (via the service method) to allow a servlet to handle a GET request. We would like to measure the time a GET request [18] would take for FAS to serve. For this measurement instructmentations are inserted at the beginning and the end of the `doGet()` method body.*

Concerns and envisioned support by HATS

The method for performance analysis described in the example above has a number of shortcomings. It is time consuming, a stress test case might be long-running, ranging from 30 minutes to several hours, and this is not scalable under constant changes to the software system behavior such as adding new feature or modifying existing ones. It is not exhaustive, test cases cannot provide complete guarantee on performance requirements. It is not extensible, performance requirements are currently hard wired rather modeled declaratively and the model for specifying requirements is not compositional. To this end we envisage the HATS framework would provide the theory and tool support for the following:

Concern (FP-C20)**. Specification and analysis of Performance requirements:** The HATS framework should provide the mechanism and analysis technique for formally specifying performance requirement and resource guarantee of units of code. Proposed method should both be declarative and extensible by means of compositionality. In the area of formal *cost analysis* [26, 12, 2], a program's cost function are a set of cost relations, each defined by a set of recurrence equations, in which cost of a program is defined as a function of its input data size. However the formulation of this model requires knowledge, which cannot be assumed to be accessible to software developers and testers. HATS should therefore also incorporate usable techniques and tools to assist the generation of cost models such that formal specification of performance requirement might become more amenable. (T4.2)

Concern (FP-C21)**. Verification of performance requirement:** The HATS framework should provide the mechanism to breach the gap between specifying and verifying performance requirement and resource guarantees. The need for formal and *approximate* verification of performance requirement has been identified to overcome the shortcoming of the inexaustive method of stress testing. This is where performance and

[3] An abstract Jave class to be subclassed to create an HTTP servlet suitable for a Web site.

resource guarantee of programs are specified in program logic [3], and then formally proved for correctness. Similar to specification of functional requirements, HATS should incorporate usable techniques and tools to assist verification process. (T1.5 and T4.2)

5.3 Summary

In this chapter we have presented the industrial software system FAS and used general scenarios with concrete examples to identify concerns and issues with the development and support process of FAS. We then suggest where in the HATS project these concerns and issues may be addressed. As mentioned at the beginning of this chapter, corresponding detail requirement analyses for these scenarios and examples would be carried out in Task 5.2, using which concrete requirements would be harvested for the HATS framework.

Specifically Scenarios FP1 and FP4 in Sections 5.2.1 and 5.2.4 identified concerns for verifications of sequential and concurrent behavior; Scenario FP2 in Section 5.2.2 identified concerns for components integration; Scenario FP3 in Section 5.2.3 identified concerns for with generation and evolution of test systems; Scenario FP5 in Section 5.2.5 identified concerns for the utilization of third party libraries, and Scenario FP6 in Section 5.2.6 concerned performance analysis. Table 5.1 gives a summary of the concerns harvested from this case study and Table 5.2 shows the relationship between concerns identified in this case study and tasks to be carried out in the HATS project.

Req. Identifiers	Req. Labels	Tasks	Reference
FP1: Correctness of sequential programs			
FP-C1	Specification of sequential programs	1.1, 1.3, 1.5, 4.3	Page 43
FP-C2	Verification of sequential programs	1.1, 1.4, 1.5, 3.2, 4.3	Page 43
FP2: Integration of a new feature component in FAS			
FP-C3	Correctness of interactions	1.1, 1.3, 1.4, 1.5, 4.3	Page 45
FP-C4	Behavioral compatibility	1.1, 1.3, 1.4, 1.5, 4.3	Page 45
FP-C5	Incremental validation	1.1, 1.3, 1.4, 1.5, 4.3	Page 45
FP-C6	Evolvability and code generation	1.4, 1.5, 3.1, 3.3, 3.5, 4.3	Page 46
FP-C7	Tool support for navigability	1.4, 1.5, 4.3	Page 46
FP3: Test system provisioning			
FP-C8	Test case generation	1.1, 1.4, 1.5, 4.3	Page 48
FP-C9	Modeling behavioral changes	3.1, 3.3, 4.3	Page 48
FP-C10	Test system evolution	1.1, 1.5, 3.1, 3.3, 4.3	Page 50
FP4: Concurrency			
FP-C11	Specification of concurrent programs	1.1, 1.3, 1.5, 4.3	Page 50
FP-C12	Verification of concurrent programs	1.1, 1.4, 1.5, 3.2, 4.3	Page 51
FP5: Using third party library			
FP-C13	Partial specification	1.1, 1.5, 4.3	Page 52
FP-C14	Model mining	3.2	Page 53
FP-C15	Evolution of black box components	3.1, 3.2, 3.3, 3.5	Page 53
FP-C16	Modeling data transformation	1.1, 1,4, 1.5, 4.3	Page 53
FP-C17	Modeling interactions with black box components	1.1, 1.4, 1.5, 4.3	Page 53
FP-C18	Propagation of changes	3.1, 3.2, 3.3, 3.5	Page 53
FP-C19	Test case generation	1.1, 1.4, 3.1, 3.4, 3.5	Page 54
FP6: Performance			
FP-C20	Specification and analysis of performance requirements	4.2	Page 55
FP-C21	Verification of performance requirement	1.5, 4.2	Page 55

Table 5.1: High level requirements harvested from the Fredhopper case study

Work-packages and Tasks	*Concerns harvested from Fredhopper case study*																				
	FP1		FP2					FP3			FP4		FP5							FP6	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1 Framework																					
Task 1.1	•	•	•	•	•			•		•	•	•	•			•	•		•		
Task 1.2																					
Task 1.3	•		•	•	•						•										
Task 1.4		•	•	•	•	•	•	•				•				•	•		•		
Task 1.5	•	•	•	•	•	•	•	•		•	•	•	•			•	•				•
2 Variability																					
Task 2.1																					
Task 2.2																					
Task 2.3																					
Task 2.4																					
Task 2.5																					
Task 2.6																					
3 Evolvability																					
Task 3.1						•			•	•					•			•	•		
Task 3.2		•										•		•	•			•			
Task 3.3						•			•	•					•			•			
Task 3.4																			•		
Task 3.5						•									•			•	•		
4 Trustworthiness																					
Task 4.1																					
Task 4.2																				•	•
Task 4.3	•	•	•	•	•	•	•	•	•	•	•	•	•			•	•				
Task 4.4																					

Table 5.2: Overview of the scenarios and concerns of the Fredhopper case study and related tasks

Chapter 6

Summary

This document provides the description of abstract requirements for the HATS method. The requirements are divided into methodological requirements, which are described from an industry and a research perspective, and concrete requirements, coming from different scenarios of the three case studies.

The methodological requirements can be seen as general high-level requirements, which should be supported by the HATS method in order to be accepted by the research community and by industry.

The requirements of the case studies are elicited by means of different scenarios, which cover different challenges and problems for each case study. As the three case studies greatly differ in profile, size, and application area, they cover very different aspects of the software development life cycle. Tables 3.2, 4.2 and 5.2 together give the complete overview of the scenarios and concerns identified via the case studies and their corresponding work packages and tasks. As can be seen, the scenarios and concerns provide a complete task coverage, where the different case studies vary in focus.

The Trading System case study is an academic system. It is small enough to be completely modeled by ABS. It will be used to apply nearly all techniques developed in HATS. The different scenarios of the Trading System case study are thus designed to cover most problem domains, which are addressed by HATS. In particular, it will be used to evaluate tasks that are not addressed by the other two case studies. However, being an academic system it cannot serve as a realistic example from industry, as the elicited requirements are artificial and not driven by real customer needs.

The Virtual Office of the Future case study is a research system, coming from a semi-industrial domain. The system is in particular challenging for ABS because of its highly distributed system structure. The scenarios of this case study have no main focus and cover aspects of all work packages. As the VOF case study is developed by standard industry methods, it will be an important case study to validate the integration of HATS framework into existing development methods.

Finally, the Fredhopper case study is a system used and developed in industry. This case study differs in two main points from the other ones. The first is that it is in active use by many customers and that it is constantly evolving and adapted. The second is, being an industry system, it relies on many third party software components. Thus it cannot be assumed that the whole program source is available. This introduces several interesting challenges to the HATS framework, which might, however, not all be solvable in the scope of HATS. In particular, it requires that the ABS language is able to modularly describe and verify certain parts of a software system and to abstract from a concrete environment and the implementation of other software parts. The main focus of the case study, however, is evolvability. In particular, the Fredhopper case study provides evolution steps, which are driven by customer needs and are thus not artificial. Evolution in the Fredhopper system is critical in several aspects. First, evolution is required for Fredhopper in order to keep their product competitive. Second, evolution has to be applied to systems, which are already deployed and are running. And third, evolution must not introduce bugs, resulting in down times of systems deployed at customers.

Deliverable D5.1 is the starting point for the validation of the HATS framework. The requirements collected in this document are described on a rather abstract level. These requirements will be refined and

scoped in the later validation phases, starting with D5.2, which will validate the Core ABS language.

Bibliography

[1] L. Aceto and A. D. Gordon, editors. *Proceedings of the Workshop Algebraic Process Calculi: The First Twenty Five Years and Beyond*, volume 162 of *ENTCS*, 2006.

[2] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Cost Analysis of Java Bytecode. In Rocco De Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer-Verlag, March 2007.

[3] D. Aspinall, L. Beringer, M. Hofmann, H-W. Loidl, and A. Momigliano. A program logic for resource verification. In *17th International Conference on Theorem Proving in Higher Order Logics*, volume 3223 of *LNCS*, 2004.

[4] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel. *Component-based product line engineering with UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[5] J. Bayer, C. Gacek, D. Muthig, and T. Widen. Pulse-i: Deriving instances from a product line infrastructure. *Engineering of Computer-Based Systems, IEEE International Conference on the*, 0:237, 2000.

[6] S. Becker, W. Hasselbring, A. Paul, M. Boskovic, H. Koziolek, J. Ploski, A. Dhama, H. Lipskoch, M. Rohr, D. Winteler, S. Giesecke, R. Meyer, M. Swaminathan, J. Happe, M. Muhle, and T. Warns. Trustworthy software systems: a discussion of basic concepts and terminology. *SIGSOFT Software Engineering Notes0*, 31(6), 2006.

[7] C. Canal, E. Pimentel, and J. M. Troya. Compatibility and inheritance in software architectures. *Science of Computer Programming*, 41(2):105–138, 2001.

[8] S. Carpineti, G. Castagna, C. Laneve, and L. Padovani. A formal account of contracts for web services. In *WS-FM'06*, volume 4184 of *LNCS*, 2006.

[9] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley Longman, 2001.

[10] The CoCoME Website, July 2009. `http://www.cocome.org`.

[11] K. Czarnecki, U. Eisenecker, and K. Czarnecki. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, 2000.

[12] S. K. Debray and N. W. Lin. Cost analysis of logic programs. *TOPLAS*, 15, 1993.

[13] Sophia 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.

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

[15] Fredhopper Product. http://www.fredhopper.com.

[16] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[17] Highly Adaptable and Trustworthy Software using Formal Methods, March 2009. http://www.hats-project.eu.

[18] Hypertext Transfer Protocol – HTTP/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616.html.

[19] D. Jackson. A direct path to dependable software. *Comm. ACM*, 52(4):78–88, 2009.

[20] M. Jazayeri, A. Ran, and F. van der Linden. *Software architecture for product families: principles and practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

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

[22] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, 1997.

[23] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall, 3rd edition, 2004.

[24] D. Muthig. *A Lightweight Approach Facilitating an Evolutionary Transition Towards Software Product Lines*. PhD thesis, University of Kaiserslautern, 2002.

[25] K. Pohl, G. Böckle, and F. Van Der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, Heidelberg, 2005.

[26] F. A. Rabhi and G. A. Manson. Using complexity functions to control parallelism in functional programs. Technical Report TR. CS-90-1, Department of Computer Science, University of Sheffield, UK, 1990.

[27] A. Rausch, R. Reussner, R. Mirandola, and F. Plasil, editors. *The Common Component Modeling Example: Comparing Software Component Models [result from the Dagstuhl research seminar for CoCoME, August 1-3, 2007]*, volume 5153 of *LNCS*. Springer, 2008. Preliminary version of the chapter describing the Trading System is available at: http://agrausch.informatik.uni-kl.de/CoCoME/downloads/documentation/cocome.pdf.

[28] J. N. Reed, J. E. Sinclair, and A. W. Roscoe. Responsiveness of interoperating components. *Form. Asp. Comput.*, 16(4):394–411, 2004.

[29] Steffen Thiel and Klaus Pohl, editors. *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings. Second Volume (Workshops)*. Lero Int. Science Centre, University of Limerick, Ireland, 2008.

[30] Research Highlight: Virtual Office of the Future. http://www.iese.fraunhofer.de/research/vof/vof.jsp.

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.

AE See Application engineering

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

Application engineering Application engineering is a process that builds a single product by reusing artifacts in the product line artifact base.

Artifact An artifact in a product line is the output of the product line engineering process. Artifacts encompass requirements, architecture, components, tests etc.

ELT Extract, Transform and Load Toolkit.

Family engineering Family engineering is a process that builds reusable artifacts that are stored in a product line artifact base. See also Product line artifact base.

FAS See Fredhopper access server

FE See Family engineering

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

Feature model An expression of the variability within product lines. Abstractly it may be seen as a system of constraints on the set of possible feature configurations.

Fredhopper access server Fredhopper access server 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.

PLE See Product line engineering.

Product line artifact base A repository in a software product line containing all reusable artifacts.

Product line engineering A development methodology for software product family. It splits development into Family engineering and Application engineering processes. See also Family engineering and Application engineering.

Queries per second Queries per second is the number of requests completed by FAS per second. See also FAS.

Response time Response time is the time that is required for a request to enter FAS and FAS to prepare a response to return to the user.

Service level agreement Service level agreement is a part of a service contract where the level of service is formally defined.

Software product family A family of software systems with well-defined commonalities and variabilities.

SWPF See Software product family.

UML Unified Modeling Language

Virtual office of the future A P2P framework that enables office workers to perform their office tasks seamlessly independent of their current location.

VOF See Virtual office of the future