Project N°: **FP7-231620**

**Project Acronym:** **HATS**

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

**Instrument:** **Integrated Project**

**Scheme:** **Information & Communication Technologies**

**Future and Emerging Technologies**

**Deliverable D5.3**

**Evaluation of Modeling**

**Due date of deliverable: (T0+36)**

**Actual submission date: 1 March 2012**

**Start date of the project: 1st March 2009**

**Duration: 48 months**

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

---

**Integrated Project supported by the 7th Framework Programme of the EC**

<table>
<thead>
<tr>
<th>Dissemination level</th>
</tr>
</thead>
<tbody>
<tr>
<td>PU</td>
</tr>
<tr>
<td>PP</td>
</tr>
<tr>
<td>RE</td>
</tr>
<tr>
<td>CO</td>
</tr>
</tbody>
</table>
Executive Summary:
Evaluation of Modeling

This document summarises deliverable D5.3 of project FP7-231620 (HATS), an Integrated Project supported by the 7th Framework Programme of the EC within the FET (Future and Emerging Technologies) scheme. Full information on this project, including the contents of this deliverable, is available online at http://www.hats-project.eu.

This deliverable evaluates developed modeling techniques using various case-studies. We evaluate the modeling capability of the ABS language by considering the application of the Delta Modeling Workflow on the Fredhopper Access Server. We then characterize the adequacy and feasibility of the HATS Variability Modeling Languages to model product lines of information systems by modeling third parties’ product lines published at the S.P.L.O.T website. This deliverable is also a followup of Deliverable D5.2.

List of Authors
Taslim Arif (FRG)
Frank de Boer (CWI)
Michiel Helvensteijn (CWI)
Karina Villela (FRG)
Peter Wong (FRH)
# Contents

## 1 Introduction

1.1 List of Papers Comprising Deliverable D5.3 ............................... 5  
1.2 The Case Studies ......................................................... 6  
  1.2.1 Fredhopper Case Study ............................................ 6  
  1.2.2 eShop Case Study ............................................... 6  
  1.2.3 Methodological and Modeling Evaluation ..................... 6  
1.3 Evaluation Criteria .................................................. 7  
  1.3.1 Fredhopper Case Study ........................................... 7  
  1.3.2 eShop Case Study ............................................... 8  
  1.3.3 HATS Methodology ............................................. 9

## 2 Requirement Analysis .................................................. 11  

2.1 Task 2.3: Testing, Debugging, and Visualization .................. 12  
2.2 Task 2.4: Types for Variability ...................................... 13  
2.3 Task 2.5: Verification of General Behavioral Properties ........ 13  
2.4 Task 3.2: Model Mining ............................................. 14  
2.5 Task 3.4: Evolvability at Bytecode Level ....................... 15  
2.6 Task 4.1: Security .................................................. 15  
2.7 Summary ........................................................................ 16

## 3 Fredhopper Case Study .................................................. 17  

3.1 Delta Modeling Workflow ............................................... 17  
3.2 Fredhopper Access Server ............................................. 19  
3.3 Variability in Replication System .................................... 21  
  3.3.1 Installation ........................................................ 21  
  3.3.2 Resources ........................................................... 22  
  3.3.3 Replication Item .................................................. 22  
  3.3.4 Load .................................................................... 23  
  3.3.5 Job Processing ..................................................... 23  
  3.3.6 Feature Constraints ............................................. 24  
3.4 Modeling the Case Study ............................................... 24  
  3.4.1 Approach ............................................................. 25  
  3.4.2 Product Line Specification ...................................... 28  
  3.4.3 Modeling Feature ReplicationSystem ......................... 29  
  3.4.4 Modeling Feature Resources .................................... 30  
  3.4.5 Modeling Feature Installation ................................ 32  
  3.4.6 Modeling Feature JobProcessing .............................. 32  
  3.4.7 Modeling Features Seq and Concur ......................... 33  
  3.4.8 Modeling Feature ReplicationItem ........................... 38  
  3.4.9 Modeling Feature Load ......................................... 40
3.5 Evaluation

3.5.1 Variability Modeling

3.5.2 Requirements

3.5.3 TS-R-1.2-1 and TS-R-1.2-2

3.5.4 TS-R-2.2-1 and TS-R-2.2-2

3.5.5 VF-R-1.2-1 and VF-R-1.2-2

3.5.6 HATS Methodology

3.6 Summary

4 eShop Case Study

4.1 Quality Model

4.1.1 Security

4.1.2 Performance

4.2 Methodology

4.3 eShop Web Systems

4.3.1 eShop Registration

4.3.2 eShop BuyPaths

4.3.3 Product Line Definition

4.3.4 Product Specifications

4.4 Extending eShop with Quality Features

4.4.1 eShop Security

4.4.2 eShop Performance

4.4.3 Extended Product Line Definition

4.4.4 Extended Product Specifications

4.5 Evaluation Based on eShop Product Line

4.5.1 Variability Modeling

4.5.2 Methodological Requirements

4.6 Summary

5 Summary

5.1 Validation of Milestone M2

5.2 Recommendations

5.2.1 Variability Modeling

5.2.2 Behavioral Modeling

5.2.3 Resource Modeling

Bibliography

Glossary

A ABS Model of SyncClient
Chapter 1

Introduction

This is HATS Deliverable D5.3. The description of Task 5.3 is the following:

In this task we apply the modeling techniques developed in WP1, WP2 and WP3 to the artifacts of the various case studies. For example, we will start modeling concurrency and security issues in FRH’s and FRG’s case studies as soon as the techniques become available. We plan to apply an iterative process of cycles that assess and adjust the modeling techniques through providing feedback to the technical WPs. This task will be the basis for verification of Milestone M2. Task 5.3 involves all partners.

In short, this deliverable evaluates developed modeling techniques using various case-studies. It is also a followup of Deliverable D5.2 [8], a link which is clarified in Chapter 2.

1.1 List of Papers Comprising Deliverable D5.3

This section lists all the papers that comprise this deliverable, indicates where they were published, and explains how each paper is related to the main text of this deliverable. As requested by the reviewers, the papers are not directly attached to Deliverable D5.3, but are made available on the HATS web site at the following url: [http://www.hats-project.eu/sites/default/files/D5.3](http://www.hats-project.eu/sites/default/files/D5.3). Direct links are also provided for each paper listed below.

**Paper 1: Delta Modeling Workflow**

This paper [16] describes a proposed workflow for developing software product lines in an abstract algebraic setting.

This paper was written by Michiel Helvensteijn and was published in the proceedings of VaMoS 2012.


**Paper 2: Delta Modeling in Practice: A Fredhopper Case-study**

This paper [15] evaluates the workflow described in [16] in the industrial setting of the Fredhopper Access Server (Chapter 3).

This paper was written by Michiel Helvensteijn, Radu Muschevici and Peter Wong and was published in the proceedings of VaMoS 2012.

1.2 The Case Studies

We test and evaluate the ABS with respect to modeling and the HATS methodology by considering two case studies: the Fredhopper case study and the eShop case study. The evaluation presented in this deliverable constitutes the validation of Milestone M2.

1.2.1 Fredhopper Case Study

The Fredhopper case study is an industrial case study. In this case study we consider the Fredhopper Access Server (FAS). 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, classified advertising etc. A more detailed description of FAS is provided in D5.1 [7, Chapter 5]. In Task 5.2, we have chosen the Replication System as one of the components of FAS for testing the core ABS language [8, Chapter 5]. The Replication System is a distributed concurrent object-oriented software system and we will consider modeling its variability using the full ABS language in this deliverable. We report on this case study in Chapter 3.

1.2.2 eShop Case Study

The eShop case study concerns a product line of Business-to-Consumer (B2C) eShop solutions with fixed-price purchasing only.

B2C sites, like amazon.com, enable retail transactions where a company sells goods or services to an individual. This is done through an eShop web site, which is sometimes referred to as a shopping cart solution.

Our case study built upon the feature model defined by Lau [23], which in its turn was defined based on literature and web research, the IBM WebSphere Commerce Documentation [http://publib.boulder.ibm.com/infocenter/wchelp/v5r6/index.jsp], an existing implementation (Shopping Cart Software: eCommerce Solutions & Hosting [http://www.ablecommerce.com/]), and field research involving several large eShop sites.

The reason for performing this specific case study were as follows:

- The eShop domain is a relatively large and well-understood domain with a large degree of variability on which serious evaluations of new approaches can be performed
- Lau [23] did serious work when identifying the set of features a provider of eShop solutions would offer
- Lau’s feature model is very well documented in [23].

Details about the eShop case study are provided in Chapter 4.

1.2.3 Methodological and Modeling Evaluation

In addition, in all three case study chapters we indicate how the specific modeling approach fits naturally into parts of the HATS methodology. We provide an evaluation of modeling using ABS and its associated tool support with respect to the requirements identified in D5.1 [7].
1.3 Evaluation Criteria

In this section we present the evaluation criteria for the validation of the modeling and the HATS methodology.

1.3.1 Fredhopper Case Study

In the Fredhopper case study, we evaluate ABS language with respect to modeling, following these criteria:

Requirement Descriptions

We evaluate the ABS language with respect to concrete requirements harvested during Task 5.2. In Task 5.1 we identified high level concerns that the HATS framework should address. These high level concerns are subsequently refined during requirement analysis sessions in Task 5.2 to generate concrete test cases and evaluation criteria. As a result, we elicited Requirements TS-R-1.2-1, TS-R-1.2-2, TS-R-2.2-1, TS-R-2.2-2, VF-R-1.2-1 and VF-R-2.2-1 for the evaluation of modeling using ABS. Their description can be found in Deliverable D5.2 [8, Chapter 2].

Note that while these requirements were harvested from specific case studies, their evaluation criteria may be applied more generally to the variability modeling of distributed object oriented systems. To this end we have chosen the Fredhopper case study as the main case study through which we evaluate ABS modeling against these requirements. We have chosen the Fredhopper case study for the following reasons:

- During Task 5.3, we have identified many aspects of the Fredhopper case study, which were not originally envisioned as high-level concerns in Task 5.1, that can be fully exploited for requirement and evaluation purposes. Specifically, in this task we have developed a complete ABS model of the Replication System that captures both variability and resource concerns (See Chapter 3 for detail).

- The Fredhopper case study is the only industrial-strength case study that has been conducted along with the development of the ABS language and tools, as well as the HATS methodology. We believe it would be natural to exploit the complexity and the coverage of this case study for evaluation of the HATS project.

- Unlike other case studies in the HATS project, the current implementation of Replication System considered in the Fredhopper case study is actually in use in a production environment by customers of Fredhopper and it would be beneficial to be able to make comparison between this implementation and the ABS model developed as part of the case study.

- By focusing on the Fredhopper case study, we can leverage maturity of the case study to evaluate the HATS methodology holistically with respect to the methodological requirements [7, Chapter 2].

Variability Modeling

We evaluate ABS with respect to the practical expressiveness of modeling design space variability. Specifically, we test ABS on the following:

Expressiveness  How easily one can express variation in the design space such as feature descriptions, and their dependencies and constraints. Variability modeling in the design space is important specifically to capture variations in an adaptable software systems such as software product lines.

Tool support  We consider the availability of the tool support for variability modeling, tool support includes editing the variability model, calculating the number of variations, type checking a given model, or checking whether a variant is valid respect to a variability model. Tool support is important as the complexity of the variability increases modeling becomes error-prone.

---

1The HATS framework consists of the HATS methodology and the ABS language with associated analysis techniques and tool support.
Behavioral Modeling

We evaluate ABS with respect to the practical expressiveness of modeling solution space variability. Specifically, we test ABS on the following:

**Expressiveness** How easily one can express behavior at the level of ABS given a feature model.

**Configuration** How easily one can connect variability at design space to behavior at the level of ABS and express dependencies between them.

**Modularity** Modularity increases model manageability, it allows a ABS model to be decomposed according to its components and features, it increases testability and encourages compositional verification.

**Reusability** Code reuse is important in a complex system as it minimizes code duplication, increases model manageability and reduces modeling time. It also encourages compositional verification, as reusing verified methods and classes reduces verification time.

**Tool support** We consider the availability of the tool support for behavioral modeling, such as editing the ABS model, generating a product of an ABS product line, simulating products in a ABS product line. Tool support is important as the complexity of the ABS model increases when the number of products the product line provides increases, and tool support must be able to scale to handle tasks like type checking, product generation and code generation. The availability of the tool support forms the basis of advanced tooling for model analysis such as verification.

Resource Modeling

We evaluate ABS with respect to the practical expressiveness of modeling resources. Specifically, we test ABS on the following:

**Scheduling** How easily one can model different scheduling polices of a distributed software system at the level of ABS. Scheduling policies requires concepts like time and priority, and it is important to be able to analyze how software system behaves specified scheduling policies at the model level such that we can be more certain how the system would operate in the real physical environment without actually running it.

**Deployment** How easily one can model the deployment of a software system at the level of ABS. Deployment captures the properties of the physical environment in which the implementation of the system will be executed. These properties include processing power, memory, network and power consumption. It is important to be able to analyze how software system behave under different deployment constraints at the model level such that we can be more certain how the system would operate in the real physical environment without actually running it.

Note that in this deliverable we do not consider analysis of ABS models with respect to scheduling policies and deployment resources, actual analysis will be conducted in Task 5.4.

1.3.2 eShop Case Study

In the eShop case study chapter 4, we analyze the Variability Modeling Languages defined in HATS for the purpose of their characterization with respect to adequacy and feasibility from the point of view of the researchers in the context of the feature models provided in the S.P.L.O.T feature model repository. The definition of this goal follows the template proposed in the Goal/Question/Metric method [2].

The adequacy of the HATS Variability Modeling languages to support the development of product lines of information systems is related to their expressiveness power. We have analyzed the S.P.L.O.T repository of feature models and verified that all models can be represented $\mu$TVL. We have used all the HATS Variability Modeling languages with focus on $\mu$TVL, CL and PSL to model a B2C eShop product line.
The feasibility of the HATS Variability Modeling languages is related to aspects that can hinder its adoption, such as learnability, usability, scalability, tool support and integration with product line engineering approaches.

Chapter 4 provides the actual findings and comments concerning the proposed analysis.

1.3.3 HATS Methodology

The HATS methodology is presented in Deliverable 1.1b (D1.1b) and has subsequently been published in the proceedings of FMSPLE. In D5.2 we presented an evaluation of parts of the HATS methodology. In this deliverable we continue evaluating the HATS methodology from empirical and requirement perspectives.

Empirical Evaluation

At this stage of the project it is not possible to consider the complete HATS methodology. Instead in our validation process, we study partially a few steps in the methodology and consider how tasks (requirement elicitation, analysis, design etc.) can fit into these steps. Since each case study is vastly different in scale, application area and contexts, the modeling approach for each case study will be different. As a result each case study considers some of the steps in the HATS methodology. Figure 1.1 shows the product line life-cycle in the HATS development methodology and highlights the phases where HATS contributes to it. Specifically, each case study considers some or all of the following steps in the methodology: Product Line Requirement Analysis, Reference Architecture, Generic Component Design, Generic Component Realization, Generic Component Validation, Application Engineering Planning and Product Construction and Integration. Details of these steps can be found in D1.1b.

![Figure 1.1: Product Line Life-cycle in the HATS Development Methodology](image-url)
Requirements

At this stage of the project it is not possible to consider all methodological requirements elicited in Task 5.1. In particular we do not consider the requirements that demand the following:

- **Iterative experiment**: At this stage of the project, it is not possible to conduct iterative experiments to test the HATS methodology. We hope to evaluate the methodology with respect to this type of requirements at a later validation task (Task 5.4). Specifically we do not consider Requirements MR9, MR14 and MR23 described in D5.1 [7].

- **Specification and Advanced Techniques**: At this stage of the project, not all techniques and tool support have been delivered. We delegate the evaluation of the methodology with respect to this type of requirements to Task 5.4. Specifically we do not consider Requirements MR5, MR15, MR16, MR17 and MR21 described in D5.1 [7].

Therefore, we consider the following requirements when testing the HATS methodology in the Fredhopper case study: Integrating Product Line Engineering (MR1), Integrating Application Engineering (MR2), Testing Reusable Artifacts (MR3), Providing Language Support for PLE (MR4), Defining Reusable Artifacts and Variation Points (MR6), Tailorability (MR7), Scalability (MR10), Learnability (MR11), Usability (MR12), Reducing Manual Effort (MR13), Integrated Environment Support (MR18), Existing Modeling Techniques Support (MR19), ABS Extensibility (MR20) and Middleware Abstraction (MR22).

Concerning the eShop case study, the applicable requirements and therefore the requirements that have been considered are: Integrating Product Line Engineering (MR1), Providing Language Support for PLE (MR4), Defining Reusable Artifacts and Variation Points (MR6), Tailorability (MR7), Incremental Adoptability (MR8), Scalability (MR10), Learnability (MR11), Usability (MR12), Reducing Manual Effort (MR13), Integrated Environment Support (MR18), and Existing Modeling Techniques Support (MR19).
Deliverable 5.1 (D5.1) presents methodological requirements that the HATS framework should meet. D5.1 also presents high level concerns derived from candidate case studies of the HATS project.

In Task 5.2, we have refined some of the high level concerns with concrete examples and evaluation criteria, such that they can be directly applicable to work tasks. This was achieved in the form of interactive requirement analysis sessions. While high level concerns are case study oriented and each concern may apply to deliverables of multiple work tasks in the project, a (concrete) requirement, on the other hand, is a concrete description of a particular aspect a work task should meet. Task 5.2 took place between project month 6 (PM6) and project month 17 (PM17), and some of the work tasks in the project did not begin until after PM17.

Task 5.3 takes place between project month 18 (PM18) and 36 (PM36), and in this task we carry out requirement analysis with tasks that have began or ended during this task period. As a result we have conducted detailed requirement analysis for Tasks 2.3, 2.4, 2.5, 3.2, 3.4 and 4.1. Table 2.1 relates each of these work tasks to its relevant section in this chapter that presents the requirement analysis result of the task.

<table>
<thead>
<tr>
<th>Task Number</th>
<th>Task Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2.3</td>
<td>Testing, Debugging, and Visualization</td>
</tr>
<tr>
<td>T2.4</td>
<td>Types for Variability</td>
</tr>
<tr>
<td>T2.5</td>
<td>Verification of General Behavioral Properties</td>
</tr>
<tr>
<td>T3.2</td>
<td>Model Mining</td>
</tr>
<tr>
<td>T3.4</td>
<td>Evolvability at Bytecode Level</td>
</tr>
<tr>
<td>T4.1</td>
<td>Security</td>
</tr>
</tbody>
</table>

Table 2.1: Requirement Analysis

Recall from D5.2 [8, Chapter 2], a requirement has the following elements.

- Unique identification of the requirement.
- Reference one or more case study scenarios where the high level concerns were identified.
- Reference to one or more high level concerns from which this requirement refines.
- A textual description of the requirement.
- Concrete test cases that are used to test whether the corresponding work task meets this requirement.
- A list of evaluation criteria that are used to interpret the results of test cases.
In the rest of this section we present each concrete requirement description in a table format following the template described in Table 2.2. We will refer to concrete test cases that will be used to evaluate outputs of the work task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirement Identifier and Name of the requirement</td>
<td></td>
</tr>
<tr>
<td>Scenarios</td>
<td>Reference to relevant scenarios in D5.1</td>
</tr>
<tr>
<td>High Level Concerns</td>
<td>Reference to relevant high level concerns in D5.1 from which this requirement refines.</td>
</tr>
<tr>
<td>Description</td>
<td>Textual description of the requirement</td>
</tr>
<tr>
<td>Test Cases</td>
<td>Textual description of the format and nature of test cases.</td>
</tr>
<tr>
<td>Accompanying Material</td>
<td>Description of any materials necessary to conduct evaluation of work task’s contribution with respect to this requirement. For example, accompanying materials may be the description of the concern and the scenario.</td>
</tr>
<tr>
<td>Evaluation Criteria</td>
<td>Criteria against which relevant work tasks would be benchmarked to measure the satisfiability of the requirement.</td>
</tr>
</tbody>
</table>

Table 2.2: Requirement Description Template

At the start of the HATS project, candidate case studies were chosen from which requirements were elicited and on which evaluation would be conducted. As the HATS project progresses, case studies have also evolved. In particular, we have identified many aspects of the Fredhopper case study, which were not originally envisioned as high-level concerns in Task 5.1, that can be fully exploited for requirement and evaluation purposes. Specifically, in the Fredhopper case study we have developed a complete ABS model of the Replication System that captures both variability and resource concerns (See Chapter 3 for detail). As the Fredhopper case study is the only industrial-strength case study that has been conducted along with the development of the ABS language and tools, as well as the HATS methodology. We would naturally wish to leverage both the coverage and the complexity of the software system being considered, and the availability of existing production implementation. Moreover, this would allow us to evaluate the HATS methodology holistically with respect to the methodological requirements [7, Chapter 2]. As a result we have chosen the Replication System as a concrete test case for the analyzed requirements presented in this chapter.

Note that we will not evaluate HATS project results against individual requirements presented in the rest of this section in isolation. This is because most of the results of these work tasks contribute to the theoretical foundation of the analysis techniques rather than the development of their tool support. The development and the integration of tool support will be conducted in Tasks 1.3, 1.4 and 1.5. As a result we will conduct evaluation as soon as the tools are available from these tasks. This activity will be the main deliverable result of Task 5.4, which begins in project month 31 (PM31), and will constitute the validation of the project milestone M3 and M4.

2.1 Task 2.3: Testing, Debugging, and Visualization

Task 2.3 aims to develop guided test case generation techniques for both glass-box and specification-based black-box testing, a unit testing framework for specifying, managing and executing test cases, and a tool-based visualization of ABS symbolic execution states and paths. In D5.1, there are the following high level concerns specific to Task 2.3: TS-C1 [7, Page 22], TS-C7 [7, Page 23], TS-C8 [7, Page 24], TS-C9 [7, Page 24], TS-C15 [7, Page 26] and VF-C9 [7, Page 33]. Table 2.3 shows Requirement FP-R-2.3-1 as the result of analyzing these high level concerns.
### 2.2 Task 2.4: Types for Variability

The results of Task 2.4 are the theoretical foundation of type checking software system with variability, such as software product lines. The aim is to be able to type check a complete software product line without type checking its individual product. In D5.1, there are the following high level concerns specific to Task 2.4: TS-C7 [7, Page 23], VF-C10 [7, Page 33] and VF-C15 [7, Page 34]. Table 2.4 shows Requirement FP-R-2.4-1 as the result of analyzing these high level concerns.

<table>
<thead>
<tr>
<th>Name</th>
<th>Detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirement</td>
<td>FP-R-2.4-1:</td>
</tr>
<tr>
<td>Scenarios</td>
<td>TS2 [7, Section 3.2.2], VF3 [7, Section 4.3.3], VF5 [7, Section 4.3.5]</td>
</tr>
<tr>
<td>High Level Concerns</td>
<td>TS-C7 [7, Page 23], VF-C10 [7, Page 33], VF-C15 [7, Page 34].</td>
</tr>
<tr>
<td>Description</td>
<td>It should be possible to type check the Replication System without type checking the product line’s products individually.</td>
</tr>
<tr>
<td>Test Cases</td>
<td>The full ABS model of the Replication System with variability modeled as deltas.</td>
</tr>
<tr>
<td>Accompanying Material</td>
<td>Textual description of the Replication System: Chapter 5 of D5.2 [8] and Chapter 3 of this deliverable.</td>
</tr>
<tr>
<td>Evaluation Criteria</td>
<td>1. It must be possible to type check parts of the Replication System without type checking the product line’s products individually.</td>
</tr>
</tbody>
</table>

Table 2.4: Requirement FP-R-2.4-1

### 2.3 Task 2.5: Verification of General Behavioral Properties

The results of Task 2.5 are the development of a program logic for core ABS based on symbolic execution and the investigation of techniques that allow incremental verification. In D5.1, there are the following high level concerns specific to Task 2.5: TS-C1 [7, Page 22], TS-C2 [7, Page 22], TS-C8 [7, Page 24], TS-C9 [7, Page 24], TS-C11 [7, Page 25], TS-C13 [7, Page 26], TS-C15 [7, Page 26], VF-C1 [7, Page 32], VF-C2 [7, Page 32].
Page 32] and VF-C16 [7, Page 34]. Table 2.5 shows Requirement FP-R-2.5-1 as the result of analyzing these high level concerns. Note that results of Task 2.5 also serve as inputs for Task 4.3. We therefore plan, as part of Task 5.4, to carry out requirement analysis and evaluation on verification techniques on Task 4.3.

<table>
<thead>
<tr>
<th>Name</th>
<th>Detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirement</td>
<td>FP-R-2.5-1:</td>
</tr>
<tr>
<td>Scenarios</td>
<td>TS1 [7, Section 3.2.1], TS3 [7, Section 3.2.3], TS4 [7, Section 3.2.4], TS5 [7, Section 3.2.5], VF1 [7, Section 4.3.1], VF5 [7, Section 4.3.5]</td>
</tr>
<tr>
<td>High Level Concerns</td>
<td>TS-C1 [7, Page 22], TS-C2 [7, Page 22], TS-C8 [7, Page 24], TS-C9 [7, Page 24], TS-C11 [7, Page 25], TS-C13 [7, Page 26], TS-C15 [7, Page 26], VF-C1 [7, Page 32], VF-C2 [7, Page 32], VF-C16 [7, Page 34]</td>
</tr>
<tr>
<td>Description</td>
<td>It should be possible to specify parts of the Replication System methods and classes based on contract-based specification.</td>
</tr>
<tr>
<td>Test Cases</td>
<td>The core ABS model of the Replication System with contract-based specification.</td>
</tr>
<tr>
<td>Accompanying Material</td>
<td>Textual description of the Replication System: Chapter 5 of D5.2 [8] and Chapter 3 of this deliverable. D2.5 [13].</td>
</tr>
<tr>
<td>Evaluation Criteria</td>
<td>1. It must be possible to specify methods and classes of the core ABS model of the Replication System based on contract-based specification.</td>
</tr>
</tbody>
</table>

Table 2.5: Requirement FP-R-2.5-1

### 2.4 Task 3.2: Model Mining

The aim of Task 3.2 is to extract ABS models from existing Java programs and to provide formal models to capture solution space variability in the form of simple hierarchical variability model (SHVM). In D5.1, there are the following high level concerns specific to Task 3.2: VF-C13 [7, Page 34], FP-C2 [7, Page 43], FP-C12 [7, Page 51], FP-C14 [7, Page 53], FP-C15 [7, Page 53] and FP-C18 [7, Page 53]. Table 2.6 shows Requirement FP-R-3.2-1 as the result of analyzing these high level concerns.

<table>
<thead>
<tr>
<th>Name</th>
<th>Detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirement</td>
<td>FP-R-3.2-1:</td>
</tr>
<tr>
<td>Scenarios</td>
<td>VF4 [7, Section 4.3.4], FP1 [7, Section 5.2.1], FP4 [7, Section 5.2.4], FP5 [7, Section 5.2.5]</td>
</tr>
<tr>
<td>High Level Concerns</td>
<td>VF-C13 [7, Page 34], FP-C2 [7, Page 43], FP-C12 [7, Page 51], FP-C14 [7, Page 53], FP-C15 [7, Page 53] and FP-C18 [7, Page 53].</td>
</tr>
<tr>
<td>Description</td>
<td>It should be possible to specify SHVM model of a given ABS model and to demonstrate correspondence between the SHVM model and the corresponding simple family</td>
</tr>
<tr>
<td>Test Cases</td>
<td>The full ABS model of the Replication System</td>
</tr>
<tr>
<td>Accompanying Material</td>
<td>Textual description of the Replication System: Chapter 5 of D5.2 [8] and Chapter 3 of this deliverable. D3.2 [14].</td>
</tr>
<tr>
<td>Evaluation Criteria</td>
<td>1. It must be possible to construct a SHVM model of the complete Replication System ABS model. 2. It must be possible to demonstrate correspondence between SHVM and a simple family.</td>
</tr>
</tbody>
</table>

Table 2.6: Requirement FP-R-3.2-1
### 2.5 Task 3.4: Evolvability at Bytecode Level

The aim of Task 3.4 is to investigate the use of bytecode level program transformations for runtime code evolution as well as rule-based compilation for ABS that preserves properties from ABS models to Java bytecode. In D5.1, there are the following high level concerns specific to Task 3.4: TS-C5 [Page 23] and FP-C19 [Page 54]. Table 2.7 shows Requirement FP-R-3.4-1 as the result of analyzing these two high level concerns. With respect to property preserving compilation from ABS to Java bytecode, the work carried out in this task focuses only on the core ABS fragment and it forms the basis for further development in Task 1.4. As a result, we do not conduct evaluation directly to this part of Task 3.4 results, but we evaluate this indirectly via Task 1.4.

<table>
<thead>
<tr>
<th>Name</th>
<th>Detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirement</td>
<td>FP-R-3.4-1:</td>
</tr>
<tr>
<td>Scenarios</td>
<td>TS1 [Section 3.2.1], TS2 [Section 3.2.2], TS4 [Section 3.2.4], TS5 [Section 3.2.5], VF1 [Section 4.3.1], VF2 [Section 4.3.2]</td>
</tr>
<tr>
<td>High Level Concerns</td>
<td>TS-C5 [Page 23], TS-C10 [Page 54], TS-C11 [Page 54], TS-C13 [Page 54], TS-C14 [Page 54], VP-C3 [Page 23], VP-C7 [Page 54]</td>
</tr>
<tr>
<td>Description</td>
<td>The ABSInliner should be able to inline a type-correct ABS model with monitor specifications given in ConSpecABS [10].</td>
</tr>
<tr>
<td>Test Cases</td>
<td>The ABS model of the Replication System with monitor specification in ConSpecABS.</td>
</tr>
<tr>
<td>Accompanying Material</td>
<td>Textual description of the Replication System: Chapter 5 of D5.2 [8] and Chapter 3 of this deliverable. D3.4 [10].</td>
</tr>
</tbody>
</table>
| Evaluation Criteria | 1. It must be possible to inline the (core ABS model of) Replication System.  
                          2. It must be possible to execute/simulate the inlined Replication System. |

Table 2.7: Requirement FP-R-3.4-1

### 2.6 Task 4.1: Security

Task 4.1 focuses on the development of mechanisms to ensure confidentiality and integrity policies for concurrent and object-oriented systems as well as distributed systems. In D5.1, there are the following high level concerns specific to Task 4.1: TS-C3 [Page 23], TS-C10 [Page 54], TS-C11 [Page 54], TS-C13 [Page 54], TS-C14 [Page 54], VP-C3 [Page 23] and VP-C7 [Page 54]. Table 2.8 shows Requirement FP-R-4.1-1 as the result of analyzing these high level concerns.

<table>
<thead>
<tr>
<th>Name</th>
<th>Detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirement</td>
<td>FP-R-4.1-1:</td>
</tr>
<tr>
<td>Scenarios</td>
<td>TS1 [Section 3.2.1], TS2 [Section 3.2.2], TS4 [Section 3.2.4], TS5 [Section 3.2.5], VF1 [Section 4.3.1], VF2 [Section 4.3.2]</td>
</tr>
<tr>
<td>High Level Concerns</td>
<td>TS-C3 [Page 23], TS-C10 [Page 54], TS-C11 [Page 54], TS-C13 [Page 54], TS-C14 [Page 54], VP-C3 [Page 23], VP-C7 [Page 54]</td>
</tr>
<tr>
<td>Description</td>
<td>It should be possible to utilize security analysis methods to enforce security requirements such as information flow policies on the Replication System.</td>
</tr>
<tr>
<td>Test Cases</td>
<td>The full ABS model of the Replication System and corresponding security requirements.</td>
</tr>
<tr>
<td>Accompanying Material</td>
<td>Textual description of the Replication System: Chapter 5 of D5.2 [8] and Chapter 3 of this deliverable. D4.1 [12].</td>
</tr>
<tr>
<td>Evaluation Criteria</td>
<td>It must be possible to specify information flow policies for the Replication System.</td>
</tr>
</tbody>
</table>

Table 2.8: Requirement FP-R-4.1-1
2.7 Summary

We conducted further requirement analyses with members of Tasks 2.3, 2.4, 2.5, 3.2, 3.4 and 4.1. For each work task, we refine corresponding high level concerns identified in D5.1. The results are requirement descriptions containing concrete test cases and evaluation criteria. Table 2.9 shows a summary of requirement descriptions and their associated work tasks, scenarios and high level concerns. We will evaluate deliverables of these tasks with respect to their requirements in Task 5.4.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Tasks</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP-R-2.3-1</td>
<td>2.3</td>
<td>Table 2.3</td>
</tr>
<tr>
<td>FP-R-2.4-1</td>
<td>2.4</td>
<td>Table 2.4</td>
</tr>
<tr>
<td>FP-R-2.5-1</td>
<td>2.5</td>
<td>Table 2.5</td>
</tr>
<tr>
<td>FP-R-3.2-1</td>
<td>3.2</td>
<td>Table 2.6</td>
</tr>
<tr>
<td>FP-R-3.4-1</td>
<td>3.4</td>
<td>Table 2.7</td>
</tr>
<tr>
<td>FP-R-4.1-1</td>
<td>4.1</td>
<td>Table 2.8</td>
</tr>
</tbody>
</table>

Table 2.9: A summary of concrete requirements
Chapter 3

Fredhopper Case Study

In this chapter we evaluate the modeling capability of the ABS language by considering the application of the Delta Modeling Workflow (DMW) \[16\] on the Fredhopper Access Server (FAS). In particular we extend the core ABS model of the Replication System presented in Deliverable 5.2 \[8\] into a full-fledged ABS model that models both the commonality and the variability of the Replication System Product Line.

The modeling of the replication system product line forms the evaluation of ABS modeling with respect to evaluation criteria identified in Section 1. These are concrete requirements TS-R-1.2-1, TS-R-1.2-2, TS-R-2.2-1, TS-R-2.2-2, VF-R-1.2-1, and VF-R-2.2-1, methodological requirements MR1, MR2, MR3, MR4, MR6, MR7, MR10, MR11, MR12, MR13, MR18, MR19, MR20 and MR22 as well as variability, behavioral and resource modeling.

The chapter is organized as follows. In Section 3.1 we introduce DMW as the development workflow and describe the necessary extensions to the ABS language in order to implement the workflow. We introduce the Replication System Product Line in Section 3.2, and report on the development of its Full ABS model in Section 3.4. In Section 3.5 we provide a practical evaluation of the ABS language and DMW. We provide a summary of this chapter in Section 3.6 and highlight some of the future work with respect to specification.

The formalization of DMW \[16\] and the development of the Fredhopper case study using DMW \[15\] have been published in conference papers at VaMoS 2012. The case study has also been presented as an invited talk at FoVeOOS 2011 \[28\]. These papers are available on the HATS web site as described in Section 1.1.

Throughout this chapter, we use a monospaced font to denote delta-names and ABS constructs, and a cursive font to denote feature-names and mathematical constructs from the DMW.

3.1 Delta Modeling Workflow

Abstract delta modeling \[6\] (ADM) describes the possible ways a product line code base can be structured by delta modeling so that it supports automated product derivation. It puts deltas, which can modify (software) products, into a partial order which restricts their application. If delta \( x \) comes earlier in this order than \( y \), then \( x \) has to be applied before \( y \). If there is no order between \( x \) and \( y \), and their changes are incompatible, a third delta can be used to mediate and resolve the conflict. The deltas carry application conditions which specify for which feature configurations they should be applied.

The Delta Modeling Workflow (DMW) \[16\] guides developers through the process of building a software product line step by step. It involves the creation of a delta for each feature in the feature model, and then the creation of a delta for each feature interaction and a delta to resolve each implementation conflict. This is done in an iterative manner as depicted in Figure 3.1.

We assume that the process starts with a product line specification \((\Psi, \oplus)\). The structural feature model \(\Psi\) is a 5-tuple \((B, \sqcup, \sqcap, \oplus, \triangleright)\), where \(B\) is a set of mandatory base-features, \(\sqcup\) is the mandatory sub-feature relation, \(\sqcap\) is the optional sub-feature relation, \(\oplus\) indicates those features that may not appear together in one product and \(\triangleright\) indicates which features require which others. These relations should all be
disjoint. $\Phi$ is used to refer to the set of feature configurations that are allowed by the feature model. $\Phi$ can be trivially deduced from $\Psi$ (which contains more information). The feature satisfaction relation $\models$ is a relation between a product $p$ and a set of features $F$. $p \models F$ indicates that $p$ satisfies the specifications of the features in $F$ as well as any desired interaction between the features in $F$.

Following the workflow should yield a product line implementation $(c, D, \preceq, \gamma)$, where $c$ is the core product that the deltas will be applied to. It is usually the empty product, although it is permissible to implement mandatory features in $c$. $D$ is the set of deltas, $\preceq$ is the partial order between these deltas and $\gamma$ is a function mapping each delta to the set of feature configurations it is applicable for with $\gamma(x) \in \Phi$ for all $x \in D$. We start with $D = \preceq = \gamma = \emptyset$. We fill in $D$, $\preceq$ and $\gamma$ while following the workflow.

The product line resulting from this workflow is guaranteed to be globally unambiguous, meaning that each feature configuration generates a unique product. If certain local guarantees are met, it is also guaranteed to be complete, meaning that each product satisfies the specifications of the features it is supposed to implement.

Figure 3.1 shows the workflow as a flow-chart. In each iteration a single feature is implemented, while also implementing necessary feature interactions and resolving conflicts that this feature might have introduced. We now explain these steps in more detail.

**Feature left to implement?**

In this stage, we choose the next feature to implement. The choice is made by following the partial order introduced by the transitive closure of $\rightarrow \cup \leftarrow \circ$. Given a feature diagram (such as the one in Figure 3.4), we work on it in a topological order from top to bottom, implementing first the base features and then their sub-features. This is because deltas implementing sub-features often need to make assumptions about – and changes to – the implementation of the base features.

This is the main stage where different developers can work on the product line concurrently and in isolation. Many features can be worked on simultaneously, so long as they are independent. While working on a feature, it is not necessary to consider possible conflicts, since there will be an opportunity to resolve them later in the workflow. (It will help, of course, if code is written in a modular way, which lends itself better to conflict resolution in the future.)

If there are no more features to implement, and for each previously implemented feature, the steps were properly followed, the product line is now finished.
Implement feature with new delta

Having chosen a feature, we need to develop a new delta in $D$ to implement it. This delta has to be applied exactly when its feature is selected, which should be reflected in $\gamma$. Its place in the partial order $\prec$ should mirror the feature’s place in the feature diagram. So, it should be greater than the delta that implements its super-feature and incomparable to all other deltas currently present. In fact, the deltas that implement features, linked by the transitive reduction of $\prec$, will form a graph that is isomorphic with the feature diagram. $D$ has to be implemented such that it introduces the new feature, but does not remove the functionality that was introduced by super-features.

Interaction to implement?

At this point, we need to know if, by introducing a new feature, there are now sets of features that require extra work to make them interact properly. We will implement any such desired interaction in the next step. Different interactions may be implemented concurrently and in isolation.

Implement interaction with new delta

Given a set of features whose interaction we need to implement, we develop a new delta in $D$ to do it. It has to be applied exactly when the interacting features are selected, which should be reflected in $\gamma$. It should be greater in the partial order $\prec$ than the deltas that implement the interacting features, as well as the deltas that implement interactions of subsets of the interacting features.

Conflict to resolve?

After implementing our feature and any desired interaction related to it, we now look for any conflicts we might have introduced in this iteration. We have to consider conflicts involving the delta that implemented the feature, all deltas we used for implementing desired interaction and all deltas we used for resolving earlier conflicts in this iteration. Formally, a conflict occurs between two deltas. However, when there is a set of deltas with many (related) conflicts, we will also want to introduce conflict-resolving deltas for larger sets that can be applied together according to $\gamma$, so all combinations are covered. Conflict-resolving deltas for different conflicts may be implemented concurrently and in isolation.

Resolve conflict with new delta

Given a set of deltas whose conflict we need to resolve, we develop a new delta in $D$ to do it. It has to be applied exactly when all conflicting deltas are applied, which should be reflected in $\gamma$. It should be greater in the partial order $\prec$ than all conflicting deltas and it should resolve all conflicts between them.

3.2 Fredhopper Access Server

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, classified advertising etc. Specifically, FAS provides to its clients structured search capabilities within the client’s data. This includes text search and structured navigation. Figure 3.2 shows a conceptual overview of FAS architecture, a more detailed informal description of each core component may be found in the requirement elicitation report [7, Section 5.1].

To minimize the possible disruption caused by updates of data and configuration in a FAS installation, each FAS installation is deployed for a customer according to the FAS deployment architecture. Figure 3.3 shows an example setup. A FAS deployment consists of a set of “environments”. In this case study we focus on two types of environments: live and staging. Briefly, the staging environment is responsible for data
and configuration updates while the live environment is responsible for serving queries from client-side web applications.

Here we provide a brief overview of these two environments. A live environment processes queries from client web applications via the Web Services technology. FAS aims to provide a constant query capacity to client-side web applications. In the deployment example in Figure 3.3 there are two live environments.

The primary role of the live environment is to serve queries from client-side application via web services. A live environment consists of only two components: the **Query Engine** and the **Synchronisation Client** (SyncClient).

**Query Engine** The query engine is responsible for processing queries from client-side web applications.

**SyncClient** The SyncClient has the responsibility to keep up-to-date the index used by the query engine.

The SyncClient on a live environment connects to the **Synchronisation Server** (SyncServer) on a staging environment and responds to incoming update changes to both data and configuration.

A staging environment is responsible for receiving client data updates in XML format, indexing the XML, and distributing the resulting indices across all live environments using the Replication System. In addition, the staging environment may run a business manager component, which allows for changing of the business configuration. A BM user may choose to publish and replicate a changed business configuration to all live instances. In the deployment example in Figure 3.3 there is a single staging environment. The example staging environment consists of a SyncServer, an indexer and a Business Manager (BM). Here we provide an overview of the responsibility of these components in a staging environment.

**Indexer** An indexer contains a XML loader, a search indexer and a navigation indexer, for processing incoming operation on items, and creating both search and navigation indices.

**SyncServer** The SyncServer in the example staging environment replicates configuration and data updates to the SyncClients running on the live environments.
Business manager  The business manager is a management console. It allows business administrators to configure, monitor and measure how FAS influences their business. Specifically it is where changes to the configuration are made.

The communication between the live and the staging are governed by the replication protocol, which is implemented by the Replication System Product Line. Specifically, the Replication System Product Line consists of a synchronization server residing in a staging environment, a set of synchronization clients residing in the live environments. The synchronization server determines the schedule of replication, as well as the content of each replication item. The synchronization client is responsible for receiving data and configuration updates. A replication item is a set of files representing a single unit of replicable data.

3.3 Variability in Replication System

The Replication System Product Line consists of several variants of the replication system. We express these variants as features. Figure 3.4 shows the feature diagram of the replication system. The feature diagram has five main sub-features: Installation, Resources, Replication Item, Load and Job Processing.

3.3.1 Installation

A Replication System that offers the feature Installation also has to offer one of the features Site and Cloud: These two features denote the two types of FAS installations.

A Replication System that offers Site is installed at the customer’s premise. This means FAS is installed onto some physical resources allocated by its customer. In this setting, we preemptively determine the necessary resources, such as processing power, physical memory, disk space and network bandwidth for the
FAS installation. This setting is static as it would be costly and time consuming to increase the amount of resources dynamically in an on-premise set up in the presence of an unexpected load to the FAS installation.

A Replication System that offers *Cloud* is installed in the Cloud. In this setting the FAS is installed onto the virtual resources provided in the cloud. A typical virtual resource provider is the Amazon EC2 web service\(^1\). This setting is dynamic as virtual resources are elastic and can be acquired and released on demand. As a result we can scale the underlying resource of the FAS installation much more cost effectively and efficiently in the presence of an unexpected load to the FAS installation.

### 3.3.2 Resources

A Replication System that offers the feature *Resources* may also offer one or both of *Server* and *Client* sub-features. The feature *Server* has attribute \(c\) that specifies the amount of CPU the SyncServer has, while *Client*’s attribute \(c\) specifies the amount of CPU each of the SyncClients has.

### 3.3.3 Replication Item

The Replication System can replicate various types of data. A FAS installation typically replicates one or more of the following types of data across its environments:

- **Item index** Item index is an embedded data base that stores details of individual customer product items.
- **Search index** Search index is the underlying data structure for providing searching capability on customer’s product items.
- **Navigation index** Navigation index is an embedded data base that stores the tree data structure of product items for spatial access. It is the underlying data structure for providing faceted navigation on customer’s product items.
- **Business configuration** After a set of product items are retrieved through either search or navigation, the business configuration specifies how this set of items are to be rendered and if there are extra information to be rendered such as promotions and facets.
- **Journal data** Journal data records what has changed during a data update; this information is important to ensure that indices are consistent, that is, live environments should not receive a search index or a navigation index that is “newer” than the item index.

\(^1\)http://aws.amazon.com/ec2
When the data or configuration is updated in the staging environment, they must be replicated to live environments. To replicate data and configuration, the Replication System Product Line offers the following sub-features of Replication Item: A Replication System may offer one or more of sub-features Dir, Journal and File: A Replication System that offers the mandatory feature Dir can replicate a complete file directory, this type of replication item is designed for replicating the search index; a Replication System that offers the optional feature File can replicate files that satisfy some rules, this type of replication item is designed for replicating the item index, navigation index and business configuration, and a Replication System that offers the optional feature Journal can replicate journal data.

3.3.4 Load

The Replication System can be subjected to various level of resource-bound settings. A Replication System that offers the feature Load has to offer Schedule and may also offer one or both of features Update and ClientNr.

A Replication System that offers feature Update allows the specification of the interval between incremental updates. Specifically, during the up-time of a FAS installation, data and configurations in the live environments have to be updated frequently to reflect that actual changes to the underlying merchandises. An incremental update defines the set of changes required to the data and configuration in the live environments. Each incremental update is broken down into a set of replication items and they are replicated from the staging to the live environments via the Replication System. The interval between updates is specified by the feature attribute \( u \).

A Replication System that offers feature ClientNr allows the specification of the number of SyncClients participating in the replication protocol. The number of SyncClients is specified by the feature attribute \( c \). For modeling purposes, the feature ClientNr also allows the specification of the number of jobs each SyncClient may schedule (attribute \( j \)) and the number of changes the SyncClient can see (attribute \( t \)) before shutting down.

A Replication System that offers feature Schedule can offer the specification of schedules. Schedules dictate when and where the Replication System should monitor for changes in the staging environment to be replicated to the live environments. Currently a Replication System may offer one or more of Search, Business and Data features.

If the Replication System offers Search, then it allows the specification of schedules on the search index directory. The specification describes the interval between which the Replication System replicate the changes from the search index directory as well as the deadline for each replication session. The interval is specified by the feature attribute \( l \), while the deadline is specified by the attribute \( d \).

If the Replication System offers Business, then it allows the specification of schedules on replicating changes about the business configuration. Again the interval between scheduled replications is specified by the feature attribute \( l \), while the deadline of each replication is specified by the attribute \( d \).

If the Replication System offers Data, then it allows the specification of schedules on replicating changes about the item and navigation indices and journal data. Again the interval between scheduled replications is specified by the feature attribute \( l \), while the deadline of each replication is specified by the attribute \( d \).

3.3.5 Job Processing

The SyncClient and SyncServer do not communicate directly. The SyncServer uses connection threads, which are Java Thread objects as the interface to the server-side of the replication protocol. The SyncClient, on the other hand, schedules jobs to handle communications to the client-side of the replication protocol. Specifically, a job is an executable task that is scheduled by a third party scheduler system. While the SyncClient is responsible to establishing jobs to be scheduled, the SyncServer is responsible to specifying the schedule for executing the client jobs.

---

2In Fredhopper, third party libraries are used to provide new features at a lower cost. For scheduling, we use Quartz API from OpenSymphony [http://www.quartz-scheduler.org/](http://www.quartz-scheduler.org/)
In a FAS installation, depending on the deployment model and the supported replication item types of the Replication System, the installation that offers Feature Job Processing also offers one of features Seq and Concur. A Replication System that offers feature Seq enables the sequential job processing mode. In this mode, jobs scheduled by a single SyncClient must be executed sequentially. A Replication System that offers feature Concur enables the concurrent job processing mode. In this mode, jobs scheduled by a single SyncClient may be executed concurrently. In particular, jobs scheduled for the same replication schedule must be executed sequentially while jobs scheduled for different replication schedules may be executed concurrently.

3.3.6 Feature Constraints

There are constraints between features defined in the Replication System feature diagram.

- The feature Business requires feature File as business configurations can be captured as a set of files in the configuration directory matching the “.xml” extension.

- The feature Data requires both features File and Journal: Both item and navigation indices are captured as a set of files in the configuration directory matching the “.jps” extension, and journal data is captured as a set of transaction log files.

- To enable concurrent job processing mode (Concur), the Replication System must be installed onto the cloud (Cloud). This is because concurrent job process mode allows multiple replications to be conducted concurrently and would therefore require more processing resource.

- For on-premise installations, that is when the Replication System offers feature Site, there is a constraint to ensure the amount of allowed CPU resources on both the staging and the live environments do not exceed 10. Moreover, in an on-premise installation, we restrict the number of live environments (ClientNr) to no more than 10 and that the unit of time between incremental updates must be greater than 10.

There are also other constraints that are not at the level of features but that of object behaviour. We informally describe them here:

- The Replication System expects replication schedules to be processed in a particular order, in particular it is necessary that the list of defined schedules for replication to must contain search index directory replication schedules followed by business configuration’s, and which is then followed by journal data’s. This means there is an order constraint between object behaviour defined on Features Search, Business, and Data.

- If feature Data is selected, replication includes journal data, and to ensure stability at the client side. we need to make sure there exist at least two SyncClient instances in the replication system as a fail-safe mechanism. This is an implicit feature interaction between Data and ClientNr.

3.4 Modeling the Case Study

In this section we present how to develop a Full ABS model of the Replication System Product Line using the DMW. We first describe some fundamental changes in Section 3.4.1 that we have made in order to capture the product line’s variability. We start executing the DMW in Section 3.4.2 in which we provide the product line specification of the Replication System Product Line. From Sections 3.4.3 to 3.4.9 we describe, step by step, the execution of DMW.
3.4.1 Approach

Our approach to develop a Full ABS model is to extend from the existing core ABS model that has been documented in Chapter 5 of Deliverable 5.2 [8]. The overall functionality of classes SyncServerAccepter, SyncServer, SyncClient, ConnectionThread, SyncServerClientCoordinator, ReplicationSnapshot and DataBase have not changed, we therefore will often refer to specific sections of that chapter throughout this section for detailed description of these parts of the ABS model. Nevertheless to express the variability of the Replication System Product Line, we have made the following fundamental changes to the ABS model; while we discuss these changes now, they are introduced into the Replication System Product Line using delta modules.

File Systems

In the core ABS model of D5.2, we express a physical file in the staging and live environments as a value of the type $\text{Pair<FileName,FileSize>}$ such that FileName is a String and FileSize is an integer value denoting the file size [8, Listing 5.10]. To model this concept, more detailed file information such as directories is included, we extend this model to express the hierarchical structure of a file system. Listing 3.1 shows the necessary data type and type synonym definitions to model a hierarchical file system. Note that our model still abstracts from actual content of files.

```
type File = Pair<FileName,FileContent>;
type Directory = Pair<FileName,FileContent>;
type FileEntry = Map<FileName,FileContent>;
data FileContent = Content(FileSize content) | Entries(FileEntry entries) | NoContent;
```

Listing 3.1: Hierarchical file system

Replication Item

In the core ABS model, we express a replication item as a pair of integer checkpoint and a set of files $\text{Pair<CheckPoint,Set<File>>}$ [8, Listing 5.10]. To express the variability on the types of replication items (See Section 3.3.3), we express each replication item as an Object of ReplicationItem. The interface definition of ReplicationItem is shown in Listing 3.2. Notice the annotation Atomic; it indicates that the annotated method contains scheduling behavior, that is, there are no asynchronous method calls, await and suspend statements, and $f$.get statement on any future $f$ in any of its execution paths. Atomicity is important as the ReplicationItem objects access the underlying file system and the ABS type system guarantees such accesses are free of data races.

```
interface ReplicationItem {
    [Atomic] Command getCommand();
    [Atomic] ReplicationItemType getType();
    [Atomic] Unit refresh();
    [Atomic] FileEntry getContents();
    [Atomic] Unit cleanup();
    [Atomic] FileName getAbsoluteDir();
}
```

Listing 3.2: ReplicationItem interface

Incremental Updates

In the core ABS model of D5.2, changes to the underlying file system are specified as static checkpoints defined in the main block of the model as a map $\text{Map<CheckPoint,Map<File,FileSize>>}$ such that each
checkpoint points to a set of files and their new size. The core ABS model would pass this map to the database of the SyncServer and update the database’s underlying files every time the method `DataBase.refresh()` is invoked [8, Section P.3]. With the introduction to a Real Time variant of ABS (RT-ABS) [3], we can capture the variability of incremental updates more faithfully. In particular we introduce an `Updater` object that randomly updates a predefined set of locations at a predefined interval. Listing 3.3 shows the interface `Updater` and its implementation. Here we see the scheduler annotation that defines the earliest deadline policy [3, Section 3.1] for `UpdaterImpl`.

```java
interface Updater {
  Unit update();
  [Atomic] Unit shutDown();
}

[Scheduler: edf(queue)]
class UpdaterImpl([Final] Int interval, [Final] SyncServer server) implements Updater {
  Bool sd = False;
  [Final] List<FileId> locations = ... 
  Unit update() {
    Fut<UpdatableDataBase> fd = server!getUpdatableDataBase();
    UpdatableDataBase db = fd.get;
    while (~sd) {
      Map<FileId,FileContent> changes = this.makeChange();
      if (changes != EmptyMap) {
        Fut<Unit> u = db!update(changes); u.get;
      }
      Int jitter = random(2);
      await duration(interval+jitter,interval+jitter);
    }
  Unit shutDown() { sd = True; }
  Map<FileId,FileContent> makeChange() { ... }
}
```

Listing 3.3: `Updater` interface and class

The `Updater` provides an update method and a shutdown method such that the update method repeatedly chooses randomly over a predefined set of file locations at the SyncServer to update at a specified interval, expressed as the `await duration(...)` statement until the shutdown method is called. Note that the scheduling policy allows one to express a higher priority to the shutdown method over the update method such that if a shutdown method is invoked, and that the update method yields a scheduling point, the shutdown method will be invoked first.

**Schedules**

In the core ABS model of D5.2, there is no concept of scheduling as the core ABS language is untimed. Using RT-ABS, we model a schedule as a tuple `(n,ls,i,d)` such that `n` is a unique name of the schedule, `ls` is a set of locations, `i` is the interval between which the SyncServer monitors the changes at `ls` and `d` specifies the default deadline under which the replication for this schedule must be completed. Listing 3.4 shows how we model schedules in ABS. Note that in ABS the parameter name of a data type constructor automatically defines an accessor function to the parameter.

```java
data Schedule = Schedule(String sname, List<Item> items, Int sched, Deadline dl) | NoSchedule;
data Item = SearchItem(FileId) | FileItem(FileId, String) | LogItem(FileId);
```

Listing 3.4: Modeling Schedules
Job Scheduling

The core ABS model of D5.2 abstracts from schedule detail and as a result SyncClient schedules client jobs with no schedule information repeatedly until the SyncServer shuts down. In the core ABS model this is detected by the null check of the return value of ServerAcceptor.getConnectionThread() method [8, Section 5.4.6]. The specification of a schedule includes monitored file locations and the intervals between which replications take place. To enact on such a specification, we extend the ClientJob implementation such that each client job is initialized for a particular schedule.

```java
public class ClientJobImpl(ToInt maxJobs, SyncClient client, JobType job, Schedule schedule) implements ClientJob {
    Maybe<Int> deadline = Nothing; Schedules schedules = EmptySet;
    ConnectionThread thread; ClientDataBase db;
    Command command = EmptyCommand;

    Unit establishSchedule() {
        Fut<Int> jcf = client!jobCount(); Int stats = jcf.get;
        if (stats >= maxJobs) {
            this.shutDownClient();
        } else {
            while (hasNext(schedules)) {
                Pair<Schedules,Schedule> nt = next(schedules);
                schedules = fst(nt); Schedule s = snd(nt);
                [Deadline: deadline(s)] client!scheduleJob(Replication, s);
            }
        }
    }

    Unit executeJob() {
        Fut<ClientId> fut = client!getId(); clientId = fut.get;
        this.clientDB(); // set data base
        thread = this.getConnectionThread(); // acquire a connection
        if (thread != null) {
            this.consumeResource(); // consume some resource...
            if (job == Boot) {
                this.becomeState(Booting);
                thread!command(ListSchedule);
                await schedules != EmptySet;
                this!establishSchedule();
            } else {
                this.becomeState(WorkOnReplicate);
                thread!command(SearchSchedule(sname(schedule)));
                await schedules != EmptySet;
                this!establishSchedule();
            }
        }
        Fut<Bool> sd = client!isShutdownRequested(); Bool shutDown = sd.get;
        if (!shutDown) this.becomeState(WaitToReplicate);
        client!nextJob(); // allow next job to proceed
    }

    Unit receiveSchedule(Schedules schedules) { this.schedules = schedules; }
}
```

Listing 3.5: Schedule-specific Client Job
Listing 3.5 shows part of the `ClientJob` class definition that shows how a client job is scheduled. Specifically, the client job main method `executeJob()` is defined as follows: If the client job is a boot job, that is, the first client job created by any `SyncClient` [Section 5.4.6], the `executeJob()` method commands the corresponding `ConnectionThread` object to send all schedules via the method `receiveSchedule(Schedules)`. Upon receiving the schedules, modeled by the statement `await schedules != EmptySet`, the client job invokes asynchronously method `scheduleJob` to create a client job for each schedule received. If the client job is a replication job the `executeJob()` method commands the corresponding `ConnectionThread` object to send the schedule specific to the job via the method `receiveSchedule(Schedules)`. Upon receiving the schedules, the client job invokes asynchronously method `scheduleJob` to create the next client job for that schedule before proceeding to the replication itself.

### 3.4.2 Product Line Specification

We now describe the development of a Full ABS model of the Replication System Product Line by applying DMW. Based on the DMW, we let \((\Psi, \models)\) be the specification of the Replication System Product Line, where \(\Psi\) is the structural feature model of the replication system and \(\Phi\) is its representative set of feature configurations. Figure 3.4 shows the diagrammatic representation of \(\Psi\). Listing 3.6 shows its corresponding \(\mu\)TVL representation.

```plaintext
root ReplicationSystem {
  group allof {
    Installation { group oneof { Site, Cloud } },
    Resources {
      group allof {
        opt Client { Int c in [1..30]; Site -> c <= 10; },
        opt Server { Int c in [1..30]; Site -> c <= 10; }
      }
    },
    JobProcessing { group oneof { Seq, Concur { require: Cloud; } } },
    ReplicationItem { group allof { Dir, opt File, opt Journal } },
    Load {
      group allof {
        opt Update { Int u in [1 .. 20]; Site -> u >= 10; },
        opt ClientNr { Int c in [1 .. 20]; Int j in [1 .. 20]; Site -> c < 10; },
        Schedule {
          group allof {
            opt Search { Int d in [1 .. 60]; Int l in [1 .. 60]; d <= l; },
            opt Business { Int d in [1 .. 60]; Int l in [1 .. 60]; d <= l; require: File; },
            opt Data { Int d in [1 .. 60]; Int l in [1 .. 60]; d <= l; require: File; require: Journal; require: File; require: Journal; }
          }
        }
      }
    }
  }
}
```

Listing 3.6: \(\mu\)TVL model of the Replication system feature diagram

The relation \(\models\) is the feature satisfaction relation. It specifies when a product satisfies the specifications of a set of features. In FAS, the feature specifications are mostly in the form of descriptions (as described in Section 3.2) and use-cases. Note that in the Replication System Product Line that the features `ClientNr`
and Data require some extra implementation effort to make them interact properly. Or, formally:

\[ \exists p : p \models \{ \text{ClientNr} \} \land p \models \{ \text{Data} \} \land p \not\models \{ \text{ClientNr}, \text{Data} \} \]

We now consider the product line specification \((\Psi, \models)\), where \(\Psi = (B, \rightarrow_\bullet, \rightarrow_o, \oplus, \rightarrow_\triangleright)\) is defined as follows,

\[ B = \{ \text{ReplicationSystem} \}, \]
\[ \rightarrow_\bullet = \{(\text{ReplicationSystem, ReplicationItem}), (\text{ReplicationSystem, Installation}), (\text{ReplicationSystem, JobProcessing}), (\text{ReplicationItem, Dir}), (\text{ReplicationSystem, Load}), (\text{ReplicationSystem, Resources}), (\text{JobProcessing, Seq}), (\text{ReplicationItem, Dir}), (\text{JobProcessing, Concur}), (\text{Installation, Site}), (\text{Installation, Cloud}), (\text{Load, Schedule})\}, \]
\[ \rightarrow_o = \{(\text{ReplicationItem, File}), (\text{ReplicationItem, Journal}), (\text{Load, ClientNr}), (\text{Load, Update}), (\text{Schedule, Search}), (\text{Schedule, Business}), (\text{Schedule, Data}), (\text{Resources, Server}), (\text{Resources, Client})\}, \]
\[ \oplus = \{(\text{Seq, Concur}), (\text{Site, Cloud})\}, \]
\[ \rightarrow_\triangleright = \{(\text{Business, File}), (\text{Data, File}), (\text{Data, Journal}), (\text{Concur, Cloud})\} \]

The feature satisfaction relation \(\models\) is then a binary relation such that its range is the subset closure of the set \(\Phi\) of 768 feature configurations. Note that at the level of the product line specification \((\Psi, \models)\) feature attributes are ignored; attributes are considered when modeling the features using deltas.

To implement this product line specification as a product \((\Phi, c, D, \prec, \gamma)\), we follow DMW: we consider the specification \((\Psi, \models)\) and start with an initial empty product line implementation \(PL = (c, \varnothing, \varnothing, \varnothing)\). To align with ABS implementation guidelines the core product \(c\) is defined as \texttt{class ReplicationSystem\_Main \{} \{ \texttt{new ReplicationSystem\_Main()}; \}}. The main block contains one instruction to instantiate a nameless, typeless object of \texttt{ReplicationSystem\_Main}. Initially class \texttt{ReplicationSystem\_Main} provides no method and field definitions.

In the rest of this section we consider the features in a linear extension of the transitive closure of \(\rightarrow_\bullet \cup \rightarrow_o\). That is, we consider first the base features, and then their sub-features.

### 3.4.3 Modeling Feature ReplicationSystem

We first work on the base feature ReplicationSystem. We model this feature by delta \texttt{ReplicationSystem\_Delta} shown in Listing 3.7.

```java
delta ReplicationSystem\_Delta {
  adds data ...;
  adds type ...;
  adds class ...
  adds class ReplicationSystem(...) {
    SyncServer getSyncServer() { ... }
    SyncClient getSyncClient(ClientId id) { ... }
    Unit run() { ... }
  }
  modifies class ReplicationSystem\_Main {
    adds Unit run() {
      List<Schedule> schedules = this.get\(\)\_Schedules();
      Set<ClientId> cids = this.get\(\)\_Cids();
      Int maxJobs = this.get\(\)\_MaxJobs();
      Int updates = this.get\(\)\_UpdateInterval();
      new ReplicationSystem(updates, schedules, maxJobs, cids);
    }
  }
}
```

Listing 3.7: Delta ReplicationSystem\_Delta
The delta ReplicationSystemDelta adds the necessary type definitions, such as data types, type synonyms as well as the core ABS classes and interfaces that model the data bases of the deployment environments as well as the definition of classes SyncServerAcceptorThread, TesterImpl, UpdaterImpl, ReplicationSystem and Network that are the commonality of this product line. Briefly SyncServerAcceptorThread provides method getConnection() that takes a client job and returns a ConnectionThread to model the connection between the SyncClient in a live environment and the SyncServer at the staging environment [8, Section 5.4.2]. The class TesterImpl models a tester that validates the consistency between the data and configurations between the SyncClient and the Sync-Server [8, Section 5.4.7]. The class UpdaterImpl models the behavior of incremental updates, and the class ReplicationSystem defines the initialisation of a Replication System. Listing 3.7 shows three methods defined in ReplicationSystem: Method getSyncServer() returns an instance of SyncServer, method getSyncClient(ClientId) returns an instance of SyncClient for the input ClientId, and method run() initializes the server, clients and the updater.

The delta ReplicationSystemDelta also modifies the definition of class ReplicationSystemMain by adding method run(). This method is executed immediately after object creation and performs the following tasks: It first records the pre-defined schedules by calling the method getSchedules: this is recorded by the List<Schedule> variable schedules: it then records the set of synchronization clients to which data is replicated; this is recorded by the Set variable cids. The variable maxJobs records the maximum number of allowable jobs a client may create for replication, and the variable updates records the interval between incremental updates. Finally, it instantiates another typeless, nameless object of class ReplicationSystem that takes variables updates, schedules, maxJobs and cids as constructor arguments. Note that the implementation of methods getSchedules, getCids, getMaxJobs and getUpdateInterval is added when applying delta LoadDelta due to the mandatory feature Load (Section 3.4.9).

After implementing base feature ReplicationSystemDelta, we have the product line implementation:

$$PL = (c, \{\text{ReplicationSystemDelta}\}, \emptyset, \{(\text{ReplicationSystemDelta}, \Phi)\}) \quad (3.1)$$

Since there is only one delta and one feature, there are neither feature interactions to be implemented nor delta conflicts to be resolved. Since ReplicationSystem is a mandatory feature, ReplicationSystemDelta is applied for all feature configurations in Φ.

### 3.4.4 Modeling Feature Resources

Next we consider the feature Resources. This is modeled by delta ResourcesDelta shown in Listing 3.8. The delta adds two methods getServerDeployment() and getClientDeployment() to the class ReplicationSystem. These two methods define the deployment component to model the CPU resources on which the server and clients execute. Each deployment component has a default 10 units of CPU capacity. The delta also modifies getSyncServer() and getSyncClient(ClientId) such that created server and client objects are deployed onto the deployment component specified by getServerDeployment() and getClientDeployment() respectively.

```plaintext
delta ResourcesDelta {
  modifies class ReplicationSystem {
    adds DeploymentComponent getClientDeployment() {
      DeploymentComponent c = new cog DeploymentComponent("c1", set[CPUCapacity(10)]); return c;
    }

    adds DeploymentComponent getServerDeployment() {
      DeploymentComponent s = new cog DeploymentComponent("s1", set[CPUCapacity(10)]); return s;
    }

    modifies SyncServer getSyncServer() {
      DeploymentComponent sd = this.getServerDeployment();
      [DC:sd] SyncServer server = new cog SyncServerImpl(listToSet(schedules),cids);
    }
  }
}
```
HATS Deliverable D5.3

Listing 3.8: Delta ResourceDelta

As a result we obtain the following product line $PL$, where we write $S^*$ for the transitive closure of binary relation $S$:

$$PL = \langle c, D, \prec, \gamma \rangle$$

Where

- $D = \{\text{ReplicationSystemDelta}, \text{ResourcesDelta}\}$,
- $\prec = \{(\text{ReplicationSystemDelta}, \text{ResourcesDelta})^*\}$,
- $\gamma = \{(\text{ReplicationSystemDelta}, \Phi), (\text{ResourcesDelta}, \Phi)\}$

(3.2)

For brevity, we omit the definition of product line $PL$ at individual steps until the final product line has been reached.

Next we consider the sub-features of Resources, Client and Server, such that Replication Systems offer these features can specify the CPU capacity on which server and clients are deployed. The CPU capacity is specified by its integer attribute. We define delta ResourcesModDelta to add a method changeDC() that takes a deployment component, a name and a cpu value and returns a new deployment component instance that has the input name and cpu value as well as all other configurations defined by the input deployment component.

Listing 3.9: Delta ResourcesModDelta

Using ResourcesModDelta, we define deltas ClientDelta and ServerDelta that implement features Client and Server respectively. Here the deltas take an integer value as the unit of CPU capacity and modify getClientDeployment() and getServerDeployment() to return a deployment component with the specified CPU capacity.

Listing 3.10: Delta ClientDelta

31
3.4.5 Modeling Feature Installation

Next we consider the feature Installation. While there are constraints related to sub-features of Installation at the level of the feature model, at our current level of abstraction we do not model actual on-premise and cloud installation. The sub-tree rooted with Installation therefore provides no behavior at the level of ABS, and hence we do not provide deltas to model features Installation, Cloud and Site.

3.4.6 Modeling Feature JobProcessing

Next we consider the feature JobProcessing. This is modeled by delta JobProcessingDelta shown in Listing 3.12.

```java
delta JobProcessingDelta {
  adds interface InternalClient { ... }
  adds interface ClientConnector { ... }
  adds interface SyncServerClientCoordinator { }
  adds interface ConnectionThread { Unit command(Command command); }
  adds interface ClientJob {
    Bool registerReplicationItems(TransactionId id);
    Maybe<FileSize> processFile(FileId id);
    Unit processContent(File file);
    Unit receiveSchedule(Schedules schedules);
    Unit executeJob();
  }
}

adds class SyncClientImpl(...) implements InternalClient, ClientConnector {
  ClientDataBase db; Network network;
  Unit setMaximumTransactionId(Int id) { ... }
  Unit makeJob(JobType jb, Schedule schedule) { ... }
  Unit finishJob(ClientJob job, Maybe<JobData> jobData) { ... }
  Bool isShutdownRequested() { ... }
  Unit requestShutDown() { ... }
  Unit scheduleJob(JobType jb, Schedule schedule) {
    this.waitFor(schedule);
    if (~shutdown) {
      this.setNext(schedule);
      [Deadline: deadline()] this.make(jb,schedule);
      hit = Cons(schedule,hit);
    } else {
      missed = Cons(schedule,missed);
    }
  }
}

adds class ClientJobImpl(InternalClient client,...) implements ClientJob {
```

Listing 3.12: Delta JobProcessingDelta
Unit executeJob() { ... }
Unit establishSchedule() { ... }
Bool registerReplicationItems(TransactionId id) { ... }
Maybe<FileSize> processFile(FileId id) { ... }
Unit overwrite(File file) { ... }
Unit continue(File file) { ... }
Unit processContent(File file) { ... }
Unit receiveSchedule(Schedules schedules) { ... }
Unit nextJob() { client!nextJob(schedule); }
}

adds class Coordinator([Far] SyncServer server, Set<ClientId> clients)
implements SyncServerClientCoordinator { }

adds class ConnectionThread([Far] ClientJob job, [Far] SyncServer server, Int id)
implements ConnectionThread {
  SyncServerClientCoordinator coord; Maybe<Command> cmd = Nothing;
  Schedules schedules = EmptySet

  Unit run() { ... }
  Schedules sendSchedule() { ... }
  [Atomic] Unit command(Command c) { cmd = Just(c); }
  Set<Set<File>> registerItems(Set<ServerReplicationItem> items) { ... }
  Unit transferItems(Set<File> fileset) { ... }
}

Listing 3.12: Delta JobProcessingDelta

The delta JobProcessingDelta adds interfaces common to both sequential and concurrent job processing. It also adds classes SyncClientImpl, ClientJobImpl, Coordinator and ConnectionThread as well as methods defined by these classes that are common to both sequential and concurrent job processing. Briefly, class SyncClientImpl is responsible to schedule the initial boot job and also defines method scheduleJob(JobType, Schedule) that takes a job type and a schedule, and creates a job of that type for that schedule [8, Section 5.4.5]. Class ClientJobImpl implements both boot and replication jobs: a boot job is responsible for acquiring the replication schedules and setting up the first set of ReplicationJob. A ReplicationJob, on the other hand, is responsible for receiving replication items [8, Section 5.4.6]. Class Coordinator is responsible to coordinate SyncClient connections. Specifically, it provides the method to initialize and finalizing the replication snapshot [8, Section 5.4.3]. Note that delta JobProcessingDelta provides an empty implementation of Coordinator since sequential and concurrent job modes require different implementations for initializing and finalizing the replication snapshot. Class ConnectionThread is created by the SyncServer and is responsible for communicating with SyncClients after the SyncClient has established a connection from Acceptor. Specifically, a ConnectionThread communicates with one SyncClient via a client job scheduled by that SyncClient [8, Section 5.4.4].

3.4.7 Modeling Features Seq and Concur

Next we consider features Seq and Concur. Feature Seq enables the sequential job processing mode and is modeled by Delta SeqDelta shown in Listing 3.13. Briefly, the delta implements method becomesState(State) for SyncClient and ClientJob, methods startReplicationUpdate() and finishReplicationUpdate() for Coordinator and ConnectionThread, and method getReplicationSnapshot() of SyncServer.

delta SeqDelta {
  adds data State = ...;
```plaintext
adds type StateMachine = Map<State, Set<State>>;
adds def StateMachine stateMachine() = ...;

modifies interface SyncServerClientCoordinator {
    adds Unit setSnapshot(ReplicationSnapshot snapshot);
    adds Unit startReplicationUpdate(ConnectionThread worker);
    adds Unit finishReplicationUpdate(ConnectionThread worker);
}

modifies interface InternalClient {
    adds Unit becomesState(State state);
}

modifies interface SyncServer {
    adds ReplicationSnapshot getReplicationSnapshot();
}

modifies class SyncClientImpl {
    adds StateMachine machine = stateMachine();
    adds State state = Start;
    adds Bool next = False;
    adds Unit setNext(Schedule schedule) { next = False; }
    adds Unit waitFor(Schedule schedule) {
        Int wait = sched(schedule);
        Int j1 = random(2); Int j2 = random(2);
        await duration(wait + j1, wait + j2) & (next || shutDown);
    }
    adds Unit becomesState(State state) { ... }
}

modifies class Coordinator {
    adds Set<[Far] ConnectionThread> threads = EmptySet;
    adds [Far] ReplicationSnapshot snapshot;
    adds Unit setSnapshot(ReplicationSnapshot snapshot) { this.snapshot = snapshot; }
    adds Unit startReplicationUpdate(ConnectionThread worker) {
        threads = Insert(worker, threads);
        if (size(threads) == 1) Fut<Unit> unit = snapshot!refreshSnapshot(); unit.get;
    }
    adds Unit finishReplicationUpdate(ConnectionThread worker) {
        if (contains(threads, worker)) {
            if (size(threads) == 1) Fut<Unit> unit = snapshot!clearSnapshot(); unit.get;
            threads = remove(threads, worker);
        }
    }
}

modifies class ClientJobImpl {
    adds Unit becomeState(State state) { ... }
}

modifies class ConnectionThread {
    adds ReplicationSnapshot startReplicationUpdate() {
        // expects only one schedule per replication job
        assert size(schedules) == 1;
        Schedule schedule = snd(next(schedules));

        // register and refresh snapshot
    }
```
Fut<Unit> rp = this.coord!startReplicationUpdate(this); await rp?

Fut<ReplicationSnapshot> sp = server!getReplicationSnapshot(); return sp.get;
}

adds Unit finishReplicationUpdate() {
    assert size(schedules) == 1;
    Fut<Unit> rp = this.coord!finishReplicationUpdate(this); await rp?
}
}

modifies class SyncServerImpl {
    adds [Near] ReplicationSnapshot snapshot;
    adds Unit setUpSnapshot() { snapshot = new ReplicationSnapshotImpl(db,schedules); }
    adds ReplicationSnapshot getReplicationSnapshot() { return snapshot; }
}

Listing 3.13: Delta SeqDelta

Figure 3.5: UML state diagram of the sequential job processing mode

Similar to the core ABS model of D5.2, throughout the execution cycle of the SyncClient and its client jobs, the SyncClient must conform to the SyncClient State Machine [8 Section 5.4.5]. A UML state diagram of the SyncClient State Machine is shown in Figure 3.5. At the level of ABS, this is provided by the function definition stateMachine(), which returns a StateMachine; StateMachine is a type synonym for Map<State,Set<State>> where State is defined as follows:

data State = Start | WaitToBoot | Booting | WaitToReplicate| WorkOnReplicate | End;

The function stateMachine() is added by Delta SeqDelta. The state diagram insists that each SyncClient can only perform one replication job at a time, as a result, Delta SeqDelta implements waitFor(Schedule) and setNext(Schedule) methods of SyncClientImpl to only allow one job to be executed at a time; Listing 3.12 shows how both waitFor(Schedule) and setNext(Schedule) are invoked by scheduleJob() of the SyncClientImpl to schedule replication jobs.

We now consider Feature Concur, which enables the concurrent job processing mode, and is modeled by Delta ConcurDelta shown in Listing 3.14.

delta ConcurDelta {
    adds data State = ...;
    adds type ManyState = Map<Schedule, State>;
    adds type StateMachine = Map<State,Set<State>>;
    adds def StateMachine stateMachine() = ...
}
modifies interface SyncServerClientCoordinator {
    adds Unit setSnapshot(Map<Schedule,ReplicationSnapshot> snapshots);
    adds Unit startReplicationUpdate(Schedule s, ConnectionThread worker);
    adds Unit finishReplicationUpdate(Schedule s, ConnectionThread worker);
}

modifies interface InternalClient {
    adds Unit becomesState(Either<Schedule,Schedules> schedule, State state);
}

modifies interface SyncServer {
    adds ReplicationSnapshot getReplicationSnapshot(Schedule schedule);
}

modifies class SyncClientImpl {
    adds StateMachine machine = stateMachine();
    adds Either<State,ManyState> state = Left(Start);
    adds Map<Schedule,Bool> nexts = EmptyMap;
    adds Unit setNext(Schedule schedule) { nexts = put(nexts,schedule,False); }
    adds Unit waitFor(Schedule schedule) {
        Int wait = sched(schedule);
        Int j1 = random(2); Int j2 = random(2);
        await duration(wait + j1,wait + j2) & (lookupDefault(nexts,schedule,True) || shutDown);
    }
    adds Unit becomesState(Either<Schedule,Schedules> schedule, State state) { ... }
}

modifies class Coordinator {
    adds Map<Schedule,Set<[Far] ConnectionThread>> threadMaps = EmptyMap;
    adds Map<Schedule,[Far] ReplicationSnapshot> snapshots = EmptyMap;
    adds Unit setSnapshot(Map<Schedule,ReplicationSnapshot> ss) {
        snapshots = ss; threadMaps = setToMap(keys(ss),EmptySet);
    }
    adds Unit startReplicationUpdate(Schedule s, ConnectionThread worker) {
        Set<ConnectionThread> threads = Insert(worker,lookup(threadMaps,s));
        threadMaps = put(threadMaps,s,threads);
        if (size(threads) == 1) {
            ReplicationSnapshot snapshot = lookup(snapshots,s);
            Fut<Unit> unit = snapshot!refreshSnapshot(); unit.get;
        }
    }
    adds Unit finishReplicationUpdate(Schedule s, ConnectionThread worker) {
        Set<ConnectionThread> threads = remove(lookup(threadMaps,s),worker);
        threadMaps = put(threadMaps,s,threads);
        if (size(threads) == 0) {
            ReplicationSnapshot snapshot = lookup(snapshots,s);
            Fut<Unit> unit = snapshot!clearSnapshot(); unit.get;
        }
    }
}

modifies class ClientJobImpl {
    adds Unit becomeState(State state) { ... }
}

modifies class ConnectionThread {
    adds ReplicationSnapshot startReplicationUpdate() {

Unlike sequential job processing mode, concurrency lies in the fact replication jobs may be executed in parallel. However, to ensure we maintain the consistency of live environment’s underlying configuration and data, two replication jobs may be executed in parallel if and only if they do not interfere. Two jobs interfere if they write to the same index or configuration. Interference can cause data inconsistency since multiple jobs may write data of different versions to the same directory at the same time. We aim to enforce this constraint in the following way: Assuming that no two schedules are defined on the same index structure and configuration, the Replication System ensures that replication jobs for the same schedule are executed sequentially.

Delta ConcurDelta makes this assumption and implements waitFor(Schedule) and setNext(Schedule) methods of SyncClientImpl to allow one job per schedule to be executed at a time, while jobs of different schedules can be executed in parallel. This is modeled in ABS by a map of schedule-Boolean pair field Map<Schedule,Bool> nexts of SyncClientImpl. The method waitFor(Schedule) takes a schedule and waits until the Boolean value associated to this schedule is True, while method setNext(Schedule) takes a schedule and sets the Boolean value associated to that schedule to False.

While independent scheduling ensures concurrent replication jobs do not interfere with respect to the underlying file system, the Replication System must also ensure the replication jobs do not interfere with respect to the in-memory internal data structure. A replication job at the staging environment accesses

```java
//expects only one schedule per replication job
assert size(schedules) == 1;
Schedule schedule = snd(next(schedules));

// register and refresh snapshot
Fut<Unit> rp = coord!startReplicationUpdate(schedule,this); await rp?

// Get replication items
Fut<ReplicationSnapshot> sp = server!getReplicationSnapshot(schedule);
return sp.get;
}

adds Unit finishReplicationUpdate() {
assert size(schedules) == 1;
Fut<Unit> rp = coord!finishReplicationUpdateFor(snd(next(schedules)),this); await rp?
}

modifies class SyncServerImpl {
adds Map<Schedule,Near ReplicationSnapshot> snapshots = EmptyMap;
adds Unit setUpSnapshot() {
Schedules ss = schedules; //snapshot per schedule
while (hasNext(ss)) {
Pair<Schedules,Schedule> ns = next(ss); ss = fst(ns);
ReplicationSnapshot shot = new ReplicationSnapshotImpl(db,set[snd(ns)]);
snapshots = InsertAssoc(Pair(snd(ns),shot),snapshots);
}
}
adds ReplicationSnapshot getReplicationSnapshot(Schedule schedule) {
assert contains(keys(snapshots),schedule);
return lookup(snapshots,schedule);
}
}
```

Listing 3.14: Delta ConcurDelta
a persistent view of the environment’s underlying data structure by creating a snapshot of the content of the underlying file system [8, Section 5.4.1]. In sequential job processing mode, the replication snapshot is initialized every time when the number of replication jobs increases from 0 to 1, and is cleared every time when the number of jobs decreases from 1 to 0; this ensures concurrent replication jobs across different live environments can share the same snapshot. Thanks to cooperative scheduling in the concurrency model of ABS, no explicit synchronization is required to ensure job accesses to the snapshot are mutually exclusive.

In concurrent job processing mode, however, it becomes possible to leverage on the assumption of non-interference between schedules and to define a map from schedules to replication snapshots such that each ReplicationSnapshot creates a persistent view of a section of the staging environment’s underlying file system. The ConcurDelta achieves this by modifying SyncServerImpl and Coordinator to operate on a data structure of Map<Schedule,ReplicationSnapshot>. Moreover, a replication snapshot specific to a particular schedule is then initialized when the number of replication jobs for that schedule increases from 0 to 1, and cleared when the number of jobs for that schedule decreases from 1 to 0. Figure 3.6 shows the UML state diagram of the SyncClient State Machine for the concurrent job processing mode of two schedules. Delta ConcurDelta models this state machine by introducing the type ManyState, which is a map Map<Schedule,State>.

![UML state diagram of the concurrent job processing mode](image-url)

**Figure 3.6: UML state diagram of the concurrent job processing mode**

### 3.4.8 Modeling Feature ReplicationItem

Next we consider the feature ReplicationItem. This is implemented by the delta ReplicationItemDelta shown in Listing 3.15.

```java
delta ReplicationItemDelta {
  adds interface BasicReplicationItem {
    [Atomic] FileEntry getContents();
    [Atomic] Unit cleanup();
    [Atomic] FileId getAbsoluteDir();
  }

  adds interface ServerReplicationItem extends BasicReplicationItem {
    [Atomic] ReplicationItemType getType();
    [Atomic] Unit refresh();
  }

  adds interface ReplicationSnapshot { ... }

  adds class BasicReplicationItemImpl( ... ) implements BasicReplicationItem {
    [Atomic] FileEntry getContents() { ... }
  }
}
```
This delta adds relevant interfaces that model replication items and implementation of a replication snapshot that are common to all sub-features of ReplicationItem. Since ABS does not support class inheritance, to reuse behavior between various implementation of ServerReplicationItem, we define BasicReplicationItem as the base implementation of all ServerReplicationItem. Delta ReplicationItemDelta also introduces the class ReplicationSnapshotImpl to model a replication snapshot [8, Section 5.4.6]. This implementation provides a version of the method replicationItem(Item) that returns null; the specific implementation of this method depends on which of the sub-features of ReplicationItem are offered.

Next, we consider the feature Dir, this is modeled by the delta DirDelta shown in Listing 3.16. To model this feature, the delta adds the class DirItem to model a replication item of a complete directory. The delta reuses the base implementation BasicReplicationItemImpl to provide basic functionality of a replication item. The delta modifies the method ReplicationSnapshotImpl.replicationItem(Item) to handle the case in which it is to create a replication item of a complete directory.

Next, we consider the feature File, this is modeled by the delta FileDelta shown in Listing 3.17. To model this feature, the delta adds the class FileItem to model a replication item of a set of files. The
For feature `Journal`, we define the delta `JournalDelta`, shown in Listing 3.18 to model this feature. Similar to deltas `DirDelta` and `FileDelta`, this delta adds class `JournalItem` to model a set of journal log data, and modifies the method `ReplicationSnapshotImpl.replicationItem(Item)` to handle the case in which it is to create a replication item of a set of journal log data.

```java
delta JournalDelta {
    adds class JournalItem( ... ) implements ServerReplicationItem {
        BasicReplicationItem base;
        ...
    }
    modifies class ReplicationSnapshotImpl {
        modifies ServerReplicationItem replicationItem(Item i) {
            ServerReplicationItem repitem = original();
            if (repitem == null && isLogItem(i)) {
                Pair<FileId,String> it = right(item(i));
                repitem = new JournalItem(fst(it),snd(it),db);
            }
            return repitem;
        }
    }
}
```

Listing 3.18: Delta JournalDelta

### 3.4.9 Modeling Feature `Load`

Next we consider the feature `Load`, this is modeled by delta `LoadDelta` shown in Listing 3.19. This delta adds methods to `ReplicationSystemMain` to set the default value for the maximum number of replication jobs per `SyncClient`, the interval between incremental updates, the number of `SyncClient` and the default set of replication schedules.

```java
delta LoadDelta {
    modifies class ReplicationSystemMain {
        adds Map<String,Pair<Int,Deadline>> scheduleMap = ... ;
    }
}
```

Listing 3.19: Delta LoadDelta
The next feature to consider is the optional feature ClientNr. It is implemented by the delta ClientNrDelta shown in Listing 3.20.

This delta modifies methods getCids() and getMaxJobs() of class ReplicationSystemMain to set the Replication System to have a number of synchronisation clients and the maximum number of replication jobs per client for the replication system, as defined by feature ClientNr.

The next feature we consider is the optional Update. It is implemented by the delta UpdateDelta shown in Listing 3.22.

This delta modifies method getUpdateInterval() of ReplicationSystemMain such that it sets the interval between which incremental updates can happen according to the feature Update.

The next feature we consider is the mandatory feature Schedule. It is implemented by the delta ScheduleDelta shown in Listing 3.22.
This delta adds methods and fields to \texttt{ReplicationSystemMain} to model various schedule information such as the types of schedules and their possible file locations from which changes are replicated.

The next feature we consider is the optional feature \textit{Search}. It is implemented by the delta \texttt{SearchDelta} shown in Listing 3.23.

\begin{verbatim}
delta SearchDelta(Int d, Int l) {
    modifies class ReplicationSystemMain {
        modifies Map<String,Pair<Int,Deadline>> getScheduleMap() {
            Map<String,Pair<Int,Deadline>> m = original();
            return put(m,"Search",Pair(d,Duration(l)));
        }
    }
}
\end{verbatim}

Listing 3.23: Delta SearchDelta

This delta modifies method \texttt{getScheduleMap()} to set the interval between replicating the search index directory and the deadline for each such replication job as specified by feature \textit{Search}. Since replicating search index directory is the default schedule as defined by feature \textit{Schedule}, this delta only modifies the specification of the schedule.

Next we consider feature \textit{Business}. This is implemented by the delta \texttt{BusinessDelta} shown in Listing 3.24.

\begin{verbatim}
delta BusinessDelta(Int d, Int f) {
    modifies class ReplicationSystemMain {
        modifies Map<String,Pair<Int,Deadline>> getScheduleMap() {
            Map<String,Pair<Int,Deadline>> m = original();
            return put(m,"Business rules",Pair(d,Duration(l)));
        }
        modifies List<Schedule> getSchedules() {
            List<Schedule> ss = original();
            Map<String,Pair<Int,Deadline>> m = this.getScheduleMap();
            return itemMapToSchedule(ss,m,businessItems);
        }
    }
}
\end{verbatim}

Listing 3.24: Delta BusinessDelta

Similar to delta \texttt{SearchDelta}, this delta modifies method \texttt{getScheduleMap()} to set the interval between replicating a set of file locations and the deadline for each such replication job. However, in addition, this delta modifies \texttt{getSchedules()} to adds schedules for business configuration to the Replication System. We notice delta \texttt{BusinessDelta} causes a conflict with delta \texttt{SearchDelta}. We resolve this conflict by providing delta \texttt{SearchBusinessDelta} shown in Listing 3.25.

\begin{verbatim}
delta SearchBusinessDelta { 
    modifies class ReplicationSystemMain { 
        modifies List<Schedule> getSchedules() { 
            List<Schedule> ss = SearchDelta.original();
            return appendRight(ss,BusinessDelta.original());
        }
    }
}
\end{verbatim}

Listing 3.25: Delta SearchBusinessDelta
Specifically, the delta `SearchBusinessDelta` resolves the conflict between `BusinessDelta` and `SearchDelta` by insisting that the returned list of schedules must contain the list of search index directory replication schedules followed by the list of business configuration replication schedules. Note that while both deltas modify `getScheduleMap()`, the order in which the modifications are applied needs not be specified, therefore, `SearchBusinessDelta` does not provide a conflict resolver for those modifications.

The final feature that must be considered is `Data` and this feature is implemented by the delta `DataDelta` shown in Listing 3.26.

```java
delta DataDelta(Int d, Int l) {
  modifies class ReplicationSystemMain {
    modifies Map<String,Pair<Int,Deadline>> getScheduleMap() {
      Map<String,Pair<Int,Deadline>> m = original();
      return put(m,"Data",Pair(d,Duration(l)));
    }
    modifies List<Schedule> getSchedules() {
      List<Schedule> ss = original();
      Map<String,Pair<Int,Deadline>> m = this.getScheduleMap();
      return itemMapToSchedule(ss,m,dataItems);
    }
  }
}
```

Listing 3.26: Delta DataDelta

Similar to delta `SearchDelta` and `BusinessDelta`, this delta also modifies method `getScheduleMap()` to set the interval between replicating a journal data and the deadline for each such replication job, and similar to `BusinessDelta`, this delta also modifies `getSchedules()` to adds schedules for journal data to the Replication System.

When replicating journal data, it is important to maintain stability at the client side. To this end, we need to make sure there exist at least two `SyncClient` instances in the replication system as a fail-safe mechanism. This means we need to implement feature interaction between features `ClientNr` and `Data`. We implement this interaction using delta `DataClientNrDelta` shown in Listing 3.27.

```java
delta DataClientNrDelta {
  modifies class ReplicationSystemMain {
    modifies Set<CId> getCids() {
      Set<CId> cs = ClientNrDelta.original();
      if (size(cs) == 1) { cs = Insert(failSafe(),c); }
      return cs;
    }
  }
}
```

Listing 3.27: Delta DataClientNrDelta

We assume the built-in function `failSafe()` to return the id of the default fail-safe synchronization client. This delta has to be applied after the deltas implementing `ClientNr` and `Data`, and only when those features are selected.

We also note that delta `DataDelta` causes a conflict with both deltas `SearchDelta` and `BusinessDelta`. The DMW would dictate that we construct a conflict resolving delta for both pairs of deltas left in an unresolved conflict as the conflict `BusinessDelta E SearchDelta` was already resolved. Then, a final conflict resolving delta would be created to handle the case where all three features are selected together. This scenario is shown in Figure 3.7. However, since all four of these conflicts are resolved consistently in each case, we can save a lot of effort if we create one conflict resolving delta `SearchBusinessDataDelta` for all
Figure 3.7: Example of the three-way conflict resolution between the scheduling features. The dashed boxes are deltas. The partial order $\prec$ is represented by the arrows and each delta $x \in D$ is decorated with a propositional logic formula representing $\gamma(x)$.

four of these cases, parametrized with the selection status of the relevant features. This delta is greater in the partial order than all three original conflicting deltas, and it is to be applied when at least one of the three relevant features is selected. Note that this disjunction differs from the usual conjunctive application conditions. This is usual when working with parametrized deltas, in which the distinction between different feature configurations is made in the code rather than on the level of delta modeling.

```java
delta SearchBusinessDataDelta(Bool Search, Bool Business, Bool Data) {
    modifies class Main {
        modifies List<Schedule> getSchedules() {
            List<Schedule> ss = Nil;
            if (Search) { ss = appendRight(ss,SearchDelta.original()); };
            if (Business) { ss = appendRight(ss,BusinessDelta.original()); };
            if (Data) { ss = appendRight(ss,DataDelta.original()); };
            return ss;
        }
    }
}
```

This delta can now replace conflict resolver SearchBusinessDelta, as it encompasses that specific case. This scenario is shown in Figure 3.8.
With no further feature interaction or conflict resolution to implement in this iteration, and no further features to implement at all, we obtain the following final product line:

\[
PL = (c, D, \prec, \gamma) \text{ where }
\]

\[
\]

\[
\]

\[
\gamma = \{\text{ReplicationSystemDelta}, \Phi), (\text{ResourcesDelta}, \Phi), (\text{ClientDelta}, \{F : \Phi \mid \text{Client} \in F\}), (\text{ResourcesModDelta}, \{F : \Phi \mid \{\text{Client}, \text{Server}\} \cap F \neq \emptyset\}), (\text{ServerDelta}, \{F : \Phi \mid \text{Server} \in F\}), (\text{JobProcessingDelta}, \Phi), (\text{SeqDelta}, \{F : \Phi \mid \text{Seq} \in F\}), (\text{ConcurDelta}, \{F : \Phi \mid \text{Concur} \in F\}), (\text{ReplicationItemDelta}, \Phi), (\text{DirDelta}, \Phi), (\text{FileDelta}, \{F : \Phi \mid \text{File} \in F\}), (\text{JournalDelta}, \{F : \Phi \mid \text{Journal} \in F\}), (\text{LoadDelta}, \Phi), (\text{ClientNrDelta}, \{F : \Phi \mid \text{Client}Nr \in F\}), (\text{UpdateDelta}, \{F : \Phi \mid \text{Update} \in F\}), (\text{ScheduleDelta}, \Phi), (\text{SearchDelta}, \{F : \Phi \mid \text{Search} \in F\}), (\text{SearchDelta}, \{F : \Phi \mid \text{Search} \in F\}), (\text{BusinessDelta}, \{F : \Phi \mid \text{Business} \in F\}), (\text{DataDelta}, \{F : \Phi \mid \text{Data} \in F\}), (\text{DataClientNrDelta}, \{F : \Phi \mid \{\text{Data}, \text{Client}Nr\} \subseteq F\}), (\text{SearchBusinessDataDelta}, \{F : \Phi \mid \{\text{Search}, \text{Business}, \text{Data}\} \cap F \neq \emptyset\})\} \quad (3.3)
\]

The corresponding product line configuration is encoded in ABS as follows.
Using the product line $PL$ we confirm that the application of DMW terminates for this case study and that all products generated by $PL$ are unique and implement the required features.

### 3.5 Evaluation

The case study described in this chapter considered the Replication System Product Line, which is part of the Fredhopper Access Server (FAS). The current Java implementation of FAS has over 150,000 lines of code.

Table 3.1 shows some metrics about the existing implementation and the ABS model of the Replication System. In particular, the number of deltas reduces from 24 to 21 if three-way conflict resolution between scheduling features is applied, while the number of products is 768 if feature attributes are ignored. We see that in terms of lines of code, the size of the existing Java implementation does not differ a lot from the size of the ABS model, this is because the ABS model describes additional model-level information such as deployment components and simulations of increment updates, which the Java implementation lacks, and also the ABS model includes behavioral descriptions for scheduling, file systems, and data bases, while the Java implementation leverages third party libraries and the Java API. These additional model-level information accounts to over 1000 lines of code.

In Table 3.2 we select a few representative class definition of the existing Java implementation and compare them with the corresponding class definition in the ABS model. Details of these classes can be found in Deliverable 5.2 [8]. Here we see that while the ABS model provides additional model-level information that are not available in the Java implementation, it does also abstract from many details that are not relevant to the modeling of the replication protocol. For example, the ABS model abstracts from
Table 3.1: Metrics about Case Study

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Java</th>
<th>ABS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nr. of lines of code</td>
<td>6400</td>
<td>5000</td>
</tr>
<tr>
<td>Nr. of classes</td>
<td>44</td>
<td>40</td>
</tr>
<tr>
<td>Nr. of interfaces</td>
<td>2</td>
<td>43</td>
</tr>
<tr>
<td>Nr. of user-defined functions</td>
<td>N/A</td>
<td>80</td>
</tr>
<tr>
<td>Nr. of user-defined data types</td>
<td>N/A</td>
<td>17</td>
</tr>
<tr>
<td>Nr. of features</td>
<td>N/A</td>
<td>21</td>
</tr>
<tr>
<td>Nr. of deltas</td>
<td>N/A</td>
<td>23 (20)</td>
</tr>
<tr>
<td>Nr. of products</td>
<td>N/A</td>
<td>768</td>
</tr>
</tbody>
</table>

Table 3.2: Comparison between Main Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Java LoC</th>
<th>ABS LoC</th>
<th>Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>SyncServer</td>
<td>300</td>
<td>100</td>
<td>Build revision, Error injection, Commands, Monitoring, Performance, I/O, Preferences</td>
</tr>
<tr>
<td>SyncClient</td>
<td>700</td>
<td>100</td>
<td>Build revision, Scheduling, I/O (shut down), Monitoring, Logging, Preferences</td>
</tr>
<tr>
<td>ConnectionThread</td>
<td>500</td>
<td>150</td>
<td>Build revision, Error injection, Performance, I/O (Sockets, Files), Real-time Scheduling, Logging, Preferences, Failure handling</td>
</tr>
<tr>
<td>ClientJob</td>
<td>650</td>
<td>200</td>
<td>Real-time Scheduling, I/O (Sockets, File), Performance, Logging (Message, Statistics), Preferences, Failure handling</td>
</tr>
<tr>
<td>Coordinator</td>
<td>250</td>
<td>200</td>
<td>Threads (Synchronisation), Time, Failure handling</td>
</tr>
<tr>
<td>Acceptor</td>
<td>140</td>
<td>50</td>
<td>Threads (Synchronisation), Logging, System Properties, Preferences, I/O (sockets), Time, Failure handling</td>
</tr>
</tbody>
</table>

In Chapter \[\text{I}\] we presented the evaluation criteria for the evaluation of modeling. Here we consider them individually:

### 3.5.1 Variability Modeling

We evaluate ABS with respect to the practical expressiveness of modeling design space variability according to the following measurements:

**Expressiveness** ABS offers $\mu$TVL as a modeling language for design space variability. In the Replication System Product Line we used $\mu$TVL to specify its feature model. The language provides a compositional hierarchical view of the dependencies between features in terms of sub-feature relationship, optional and mandatory selections and constraints between features and attributes specification of individual features. We found $\mu$TVL very usable as it provides the necessary expressiveness to model the design space variability of the Replication System Product Line. Coupling the ABS Product Selection Language, we could express precisely the feature selection we would like a product to have.

The existing Java implementation does not have an explicit well-defined feature model that captures

\[\text{http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html} \]
\[\text{http://logging.apache.org/1} \]
the relationships and constraints between them. Features offered are informally described in textual document.

**Tool support** The $\mu$TVL language has been integrated into ABS syntax and we edited the feature model using the ABS Eclipse plugin’s ABS editor perspective. In addition, the ABS tool suite offers the $\mu$TVL tool that checks whether a feature selection is valid with respect to the feature model and can calculate all valid feature selections of a given feature model. We found the checker and ABS Eclipse integration very usable for ensuring that the feature model of the Replication System Product Line has correct syntax and that products of the product line are valid with respect to the feature model. With over 700 products, we have found this tool to be convenient as it can generate all 768 valid feature selections. The existing Java implementation provides these features through configuration files, and unlike the ABS model, features are not modelled explicitly and compositionally, and features offered are informally described. We had to manually harvest parts of the functionalities of the replication systems and translate these functionalities into features.

**Behavioral Modeling**

We evaluate ABS with respect to the practical expressiveness of modeling solution space variability according to the following measurements:

**Expressiveness** The Delta modeling language offers the expressiveness to specify variability at the level of object behavior. Together with Product Line Configuration, Product Selection and $\mu$TVL languages, the ABS language offers a holistic approach to expressing variabilities as features and relating them to object behavior. Moreover, following DMW we were able to reduce the number of deltas, in the product line, while still ensuring the final product line is complete and globally unambiguous. Specifically, we introduced the delta $\text{SearchBusinessDelta}$ originally to resolve the conflict between deltas $\text{SearchDelta}$ and $\text{BusinessDelta}$. However, after introducing the delta $\text{DataDelta}$ for feature $\text{Data}$, we were able to provide the single delta $\text{SearchBusinessDataDelta}$ to resolve all conflicts between all combinations of $\text{SearchDelta}$, $\text{BusinessDelta}$ and $\text{DataDelta}$. Since $\text{SearchBusinessDataDelta}$ is greater in the partial order than all of $\text{SearchDelta}$, $\text{BusinessDelta}$ and $\text{DataDelta}$, and can semantically resolve the conflict between $\text{SearchDelta}$ and $\text{BusinessDelta}$, it replaced the delta $\text{SearchBusinessDelta}$. Reducing the number of deltas in this case also reduces redundancy and enhances the reusability of the code base.

**Configuration** We were able to use ABS to incrementally and compositionally develop the Replication System Product Line that yields members that are well-typed and valid with respect to the product line’s variability. Coupling with DMW we were able to, in a top-down fashion, systematically implement all features in the feature model to obtain the Replication System Product Line. We were also able to systematically implement all necessary feature interaction and resolve implementation conflicts between deltas, since we were directed to consider every situation by the workflow. So we avoided accidentally forgetting to implement some functionality from a complex feature model. The existing Java implementation provides no explicit relationships between features. Configuration and selection of features are defined in terms of Java preference data. However, preference data only has explicit connection to to qualified Java package and class names and not object behavior. As a result it is difficult to ensure that all combination of features are covered every time a change has to be made to the implementation.

**Modularity** The ABS module system allowed us to model both the commonality and the variability of the Replication System Product Line separately and incrementally. Moreover, the combination of ABS’s delta modeling language and DMW allowed us to incrementally and modularly specify behaviors of features. We envisage that in Task 5.4, we can perform compositional analysis on delta definitions.

**Reusability** DML provides the mechanism to express variability at the level of behavior. Together with functional and object composition, the ABS language thus provides a wide range of mechanisms for
code reuse. In particular the combination of object composition and delta modeling allows us to achieve code reusability similar to that of class inheritance. In addition, the ABS module system also allows more generic definitions such as data types and functions to be reused across the ABS model of the product line.

**Tool support** The ABS Eclipse plugin provides product selection and generation. We have found this functionality allows us to quickly generate products from the Replication System Product Line through the Eclipse plugin run configurations. Product selection reduces manual effort. Moreover, currently the ABS front end provides information about the sequence of deltas applied during product selections as well as logging on delta application. We have found this to be useful as it allowed us to inspect the process of product generation. Nevertheless the current ABS tool suite does not provide pretty printing facility for generated products. Generated products remain as ASTs until a back end is chosen to generate the products to a particular implementation language. We believe it would be beneficial to provide a pretty printer that can take some (core) ABS AST and outputs the corresponding (core) ABS model in ASCII. We envisage using such outputs for debugging a (Java) execution of the model as well as visualization during symbolic execution.

**Resource Modeling**

We evaluate ABS with respect to the practical expressiveness of modeling resources according to the following measurements:

**Deployment** The real time variant of ABS (RT-ABS) provides the construct of deployment components to capture the runtime environment of a software system. We have found it useful to be able to specify and vary the processing power of each environment of the Replication System. However, during the case study the implementation of ABS only provided the specification of process power, and at the time of writing this deliverable we know ABS implementation is being extended to also capture resources such as memory, power and network.

**Scheduling** RT-ABS allows us to specify deadline and duration of method calls as well as the scheduling of multiple method calls to an ABS object. Firstly this helped resolving the unwanted non-determinism in some part of the ABS model (See Section 6.2 of D5.2 \[8\] for more detail), and secondly, coupling with deployment modeling, we were able to specify exactly how much processing power and time a method call takes to execute.

**3.5.2 Requirements**

We evaluate ABS with respect to the requirements harvested in Task 5.2 and also enumerated in Section 1. These are requirements TS-R-1.2-1, TS-R-1.2-2, TS-R-2.2-1, TS-R-2.2-2, VF-R-1.2-1 and VF-R-1.2-2.

**3.5.3 TS-R-1.2-1 and TS-R-1.2-2**

TS-R-1.2-1 and TS-R-1.2-2 describe the following as evaluation criteria:

1. It must be possible to apply the feature modeling capability provided by the ABS language to model the variations, constraints, dependency of the features described in the textual description requirements.

2. Consistency between these features can be formalized and verified under the theory of feature refinement developed in Task 1.2.

In this case study we used \(\mu\text{TVL}\) to describe a feature model based on the textual description requirement in Section 3.3. We were also able to use the \(\mu\text{TVL}\) tool to validate that our \(\mu\text{TVL}\) model is syntactically well-formed, to verify that our product selections are valid with respect to our \(\mu\text{TVL}\) model, to calculate all valid feature selections of our model. However, Task 1.2 did not develop the notion of “feature refinement”,...
while the refinement of deltas will be considered in Task 2.6. As a result we confirm that *TS-R-1.2-1 and TS-R-1.2-2 are satisfied.*

### 3.5.4 TS-R-2.2-1 and TS-R-2.2-2

TS-R-2.2-1 and TS-R-2.2-2 concern the behavioral modeling of feature models, and describes the following as evaluation criteria:

1. It should be possible for the HATS feature integration technique to model the behavior described in the textual description requirements.

2. It should be possible to use the HATS feature integration technique to formalize the textual description and correspondingly formally show (via testing) that the failure isolation property is satisfied.

3. It should be possible for the HATS feature integration technique to formalize the behavior described in the textual description requirements, allowing one to make a formal analysis of the behaviors of features.

In this case study we used the combination of $\mu$TVL, delta modeling, product line configuration and DMW to ensure features specified in the $\mu$TVL model are directly connected to object-level behavior in terms of deltas and application conditions. This task concerns the evaluation of modeling and such that we did not carry out formal analysis or the validation of failure isolation property; these activities will be conducted in Task 5.4. As a result we confirm that *TS-R-2.2-1 and TS-R-2.2-2 are satisfied with respect to ABS modeling.*

### 3.5.5 VF-R-1.2-1 and VF-R-1.2-2

VF-R-1.2-1 and VF-R-1.2-2 concern the modeling of platform and hardware, and describes the following as evaluation criteria:

It should be possible for the HATS framework to provide the necessary language constructs (in ABS), modeling techniques and tool supports to specify variability of the platform and hardware as described in the textual description requirements.

In this case study we used the combination of $\mu$TVL, delta modeling and RT-ABS to specify the variability of platform and hardware as described in the textual description in Section 3.3. As a result we confirm that *VF-R-1.2-1 and VF-R-1.2-2 are satisfied.*

### 3.5.6 HATS Methodology

We evaluate the HATS methodology with respect to the evaluation of modeling according to the following requirements:

#### Integrating Product Line Engineering (MR1)

Currently the ABS language and its tool support can be integrated into aspects of Product Line Engineering. We saw that in the case study the combination of ABS and DMW allowed us to build reusable artifacts using deltas and build individual products by product selections over the $\mu$TVL model. However, there are other phases in Product Line Engineering such as Generic Component Validation, which would require analysis on models developed and these activities will be conducted in Task 5.4. As a result we confirm that *MR1 is satisfied with respect to ABS modeling.*
Integrating Application Engineering (MR2)

As described in the previous section on MR1, we saw that in the case study the combination of ABS and DMW allowed us to build individual products by product selections over the $\mu$TVL model. This constitutes part of the phases Product Line Instantiation, and Product Construction and Integration in Application Engineering. However, there are other phases in Application Engineering such as System Validation, which would require analysis on models developed and these activities will be conducted in Task 5.4. As a result we confirm that MR2 is satisfied with respect to ABS modeling.

Testing Reusable Artifacts (MR3)

Deltas are designed to be reusable in the context of ABS, and in the case study we saw how the complete product line consists of a minimum core and a collection of deltas and application conditions. However, the evaluation of testing will be conducted in Task 5.4. As a result we confirm that MR3 is satisfied with respect to ABS modeling.

Providing Language Support for PLE (MR4)

As described above, ABS provides language constructs to model variability via $\mu$TVL and generic components via deltas and to construct products via product selection. However, there are other phases in PLE such as System Validation, that require additional language support such as a specification language. Specification is one of the activities to be evaluated in Task 5.4. As a result we confirm that MR4 is satisfied with respect to ABS modeling.

Defining Reusable Artifacts and Variation Points (MR6)

Deltas are designed to be reusable in the context of ABS, and in the case study we saw how the complete product line consists of a minimum core and a collection of deltas and application conditions. However, there is currently development of reusable specification in the form of specification deltas. Specification is one of the activities to be evaluated in Task 5.4. As a result we confirm that MR6 is satisfied with respect to ABS modeling.

Tailorability (MR7)

From Task 5.2 we know that the familiar syntax of the core ABS language and associated Eclipse plugin made ABS to be easily integrated into the Fredhopper software development workflow. With respect to the Full ABS model, both feature and delta definition as well as product generation and execution may be carried out directly through the ABS Eclipse plugin. As a result the Full ABS can be easily tailored for the Fredhopper context. Moreover, we saw the recent development of dependency management and unit testing support for ABS made ABS even more easily tailored for the Fredhopper context from the point of view of dependency and version management, continuous integration and quality assurance process. As a result we confirm that MR7 is satisfied.

Scalability (MR10)

With the introduction of delta modeling in ABS, it became very easy to scale our Replication System Product Line to hundreds of products. However, we still need to evaluate the scalability of specification and analysis techniques such as verification, which are to be conducted in Task 5.4. As a result we confirm that MR10 is satisfied with respect to ABS modeling.
Learnability (MR11)

We consider the learnability of the ABS language and its associated tool support to enable developers in Fredhopper to learn how to use them. These, combined with the availability of language reference manual [1], tutorial [5] and case studies [28, 15] and tool support guidelines [27] made the ABS language even more easy to learn. Again, we will evaluate the learnability of specification and analysis techniques of ABS in Task 5.4. As a result we confirm that MR11 is satisfied with respect to ABS modeling.

Usability (MR12)

Similar to learnability, due to the familiarity of developer at Fredhopper with Java and the Eclipse IDE, it is straightforward for developers in Fredhopper to understand and start modeling in the core ABS language and using its associated Eclipse plug-in. Moreover, the availability of various literature on ABS (see MR11) has made ABS very usable. Since we will evaluate the learnability of specification and analysis techniques of ABS in Task 5.4, we confirm that MR12 is satisfied with respect to ABS modeling.

Reducing Manual Effort (MR13)

As pointed out in D1.2 [11, Section 2.3] and based on our experience of the ABS tool suite [27], we have found more activities such as type checking, product selection and code generation could be carried out with software support and as such ABS tool support reduces the manual effort on the use of ABS language. In Task 5.4, we will consider the tool support on specification and analysis, and as a result we confirm that MR13 is satisfied with respect to ABS modeling.

Integrated Environment Support (MR18)

An ABS Eclipse plug-in is provided that offers syntax highlighting, basic content completion and code navigation, and (location) type checking, product selection, code generation, unit testing and dependency management for ABS models. In Task 5.4 we will consider IDE support for specification and analysis of ABS model and as a result we confirm that MR18 is satisfied with respect to ABS modeling.

Existing Modeling Techniques Support (MR19)

During the case study all of the modeling had to be carried out manually. This means there is currently no support for existing modeling techniques. We believe this can be improved if the ABS tool suite provides tool support to translate existing formal specifications such as JML contracts and various UML diagrams into ABS and vice versa. This also encourages the application of ABS with existing tool support. In Task 5.4 we will also consider existing modeling techniques support in specification of ABS model and as a result we confirm that MR19 is *not* satisfied with respect to ABS modeling.

ABS Extensibility (MR20)

In the case study we have found that deltas and application conditions provided excellent extension points for the application core. We saw how it is possible to incrementally extend the ABS model using delta modeling and DMW. Moreover ABS now supports Foreign Function Interface [27]. In Task 5.4 we will also consider extensibility in the specification and analysis of ABS model. As a result we confirm that MR20 is satisfied with respect to ABS modeling.

Middleware Abstraction (MR22)

RT-ABS provided abstraction of middleware at the level of runtime deployment, and scheduling policies, while ABS Foreign Function Interface supports abstraction from third party libraries. These provisions
made ABS much better at abstracting physical environment and middleware at a level where modeling can still take them into account. In Task 5.4 we will also consider how middle abstraction affects specification and analysis of ABS model. As a result we confirm that MR22 is satisfied with respect to ABS modeling.

3.6 Summary

This chapter presented the Fredhopper case study to evaluate ABS modeling. In particular we developed a full ABS model of the Replication System Product Line. This model will be used as an input for the evaluation of specification and analysis techniques of ABS in Task 5.4. With respect of evaluation, we considered concrete requirements TS-R-1.2-1, TS-R-1.2-2, TS-R-2.2-1, TS-R-2.2-2, VF-R-1.2-1 and VF-R-2.2-1, methodological requirements MR1, MR2, MR3, MR4, MR6, MR7, MR10, MR11, MR12, MR13, MR18, MR19, MR20 and MR22 as well as variability, behavioral and resource modeling.

3.6.1 Recommendations

In this section we provide some recommendations on improving the ABS language and the ABS tool on variability, behavioral and resource modeling.

Variability Modeling

We provide some recommendations to improve the ABS language and the ABS tool suite on variability modeling:

- Features in \( \mu \)TVL may have attributes that can be used to specify constraints between features as well as parameter values to delta modules during product selections. Currently feature attributes only support ABS data types Int and Bool. We would like to see other data types in the standard library, such as String and List, to also be supported.

- \( \mu \)TVL is a formal text based language for specifying feature models. Given \( \mu \)TVL has formal syntax and semantics, we would like to see the \( \mu \)TVL tool to provide the facility to generate feature diagrams; it would be easier to navigate around a large feature model graphically.

- The \( \mu \)TVL tool is built into the ABS Eclipse plugin to provide the syntax highlighting, and type checking facilities for \( \mu \)TVL models. However, at the time of writing, the ABS Eclipse plugin does not provide code navigation and outline for \( \mu \)TVL models. We feel these functionalities increase usability of \( \mu \)TVL, especially when models become large and complex. We would therefore like to see these functionalities to be provided.

Behavioral Modeling

We provide some recommendations to improve the ABS language and the ABS tool suite on behavioral modeling.

- The current ABS tool suite does not provide type checking facility for delta modules, this means that to type check the complete product line, one needs to type check every valid product selections in the product line. This approach does not scale as the number of products can grow exponentially on the number of features the product line provides. We therefore would like to see type checking facility to be provided for delta modules. We believe there has been work towards this direction in Task 2.4 and at the time of writing, progress has been made toward providing type checking facility to delta modules [24].

- The current ABS tool suite does not provide pretty printing facility for generated products. Generated products remain as ASTs until a back end is chosen to generate the products to a particular
implementation language. We believe it would be beneficial to provide a pretty printer that can take
some (core) ABS AST and outputs the corresponding (core) ABS model in ASCII. We envisage using
such outputs for debugging a (Java) execution of the model as well as visualization during symbolic
execution.

Resource Modeling

We provide some recommendations to improve the ABS language and the ABS tool suite on resource
modeling.

- RT-ABS and its Maude back end provide ways to specify the cost of ABS statements, thereby allowing
us to simulate the effect of executing these statements over a finite amount of resources (deployment
components). At the time of writing the RT-ABS implementation supports only the modeling of
CPU resource consumption, we believe it would be beneficial to provide other resource models such
as physical memory and bandwidth.
Chapter 4

eShop Case Study

The goal of this chapter is to characterize the adequacy and feasibility of the HATS Variability Modeling Languages to model product lines of information systems.

In order to eliminate any bias, we have decided to use third parties’ feature models published at the S.P.L.O.T. website (http://www.splot-research.org/) instead of modeling an internal product line. The S.P.L.O.T. website is kept by the University of Waterloo, in Canada, with the main purpose of putting Software Product Lines research into practice through the delivery of state-of-the-art on-line tools targeting academics and practitioners in the field. The S.P.L.O.T. website includes a feature model repository which contains 174 (excluding duplicates) feature models provided by the community up to this date (2. February 2012).

We organized our empirical study in two parts: 1) a static analysis of the feature models provided in the S.P.L.O.T. feature model repository to determine whether they can be represented using $\mu$TVL, and 2) a case study based on one of those feature models to evaluate the support provided by the HATS Variability Modeling Languages to the development of product lines of information systems.

In addition, we have especially investigated whether the HATS Variability Modeling Languages are already capable of dealing with quality features. The insight acquired through the case study will support the work in Task 4.4: Auto Configuration and Quality Variability, which has the goal of developing mechanisms for the automated selection of components and features to obtain configurations that meet the desired end-to-end functionality, and performance and security requirements in an optimal, or close to optimal fashion.

According to the template proposed in the Goal/Question/Metric method [2], the aforementioned goal can be represented as follows: Analyze the Variability Modeling Languages defined in HATS for the purpose of their characterization with respect to adequacy and feasibility from the point of view of the researchers in the context of the features models provided in the S.P.L.O.T. feature model repository.

In order to achieve its goal, this chapter is organized as follows: Section 4.1 introduces the concepts that are necessary to specify quality requirements, Section 4.2 explains how the S.P.L.O.T. feature model repository was used in the analysis of the adequacy and feasibility of the HATS Variability Modeling Languages, Section 4.3 describes a specific case study based on a S.P.L.O.T. feature model, Section 4.4 extends this case study with security and performance features, Section 4.5 presents our conclusions concerning the analysis of the adequacy and feasibility of the HATS Variability Modeling Languages with special focus on the requirements defined in D5.1: Requirements Elicitation and D5.2: Evaluation of Core Framework. Finally, Section 4.6 summarizes our conclusions and provides future work.

4.1 Quality Model

In order to be able to characterize whether $\mu$TVL and the other Variability Modeling Languages in HATS are already capable of dealing with quality features, we have defined a conceptual model for software product quality based on the ISO/IEC 25000 [22, 20] and ISO/IEC 9126 [17, 18] standards. Figure 4.1 provides the
concepts and relationships of the conceptual model.

Figure 4.1: Conceptual Model for Software Quality

The definitions of the concepts presented Figure 4.1 are as follows:

**externalMetric:** *metric* used to measure the degree to which a software product enables the behavior of a system to satisfy stated and implied needs when the system including the software is used under specified conditions. Attributes of the behavior can be verified and/or validated by executing the software product during testing and operation. External measures of software quality provides a ‘black box’ view of the software. This definition is based on [20].

**internalMetric:** *metric* used to measure the degree to which a set of static attributes of a software product satisfy stated and implied needs when the software product is used under specified conditions. Static attributes include those that relate to the software architecture, structure and its components. They can be verified by review, inspection and/or automated tools. Internal measures of software quality provides a ‘white box’ view of software. This definition is based on [20].

**metric:** Entity to be measured. It defines how the measurement should be carried out and how the obtained values should be interpreted.

**metricSpecification:** Concrete specification of a *qualityRequirement* in terms of a *metric*, which establishes what is the target result of the measurement and the acceptance interval.
qualityCharacteristic: Characteristic that is used to specify, measure, and evaluate system and software product quality.

qualityInUseCharacteristic: qualityCharacteristic that refers to the quality of the system in a real or simulated operational environment. It is determined by the quality of the software, hardware, operating environment, and the characteristics of the users, tasks and social environment. All these factors contribute to the system quality in use. This definition is provided in [20].

qualityMeasure: A particular technique intended to achieve an effect in the system and software product quality.

qualityRequirement: Non functional requirement that establishes the degree to which a specific quality characteristic should be present in the system. Quality characteristics and sub-characteristics are defined in [20]. This type of requirement is mentioned in [21]. The definition presented here is another one, but the rationale behind the concept is the same. According the ISO/IEC/IEEE 29148 standard [21], the other class of non functional requirements is human factors requirements, which state required usability characteristics for system interfaces with human users, such as shapes, colors, ergonomic position for controls, and displays and information arrangement.

systemQualityCharacteristic: qualityCharacteristic that specifies the degree to which a quality characteristic should be present in a software or hardware component. This definition is provided in [20].

usageMetric: metric used to measure the overall quality of the system in its operational environment for specific users when carrying out specific tasks. It refers to users completing realistic tasks (either by user testing or in actual use). This definition is based on [20].

Table 4.1 defines the most relevant attributes of the aforementioned concepts.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Concept</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>formula</td>
<td>metric</td>
<td>Computational description of how data elements should be combined in order to provide a value to the metric.</td>
</tr>
<tr>
<td>maxAcceptanceInterval</td>
<td>metricSpecification</td>
<td>Maximum value the metric can assume that contributes to the achievement of the qualityRequirement.</td>
</tr>
<tr>
<td>measurementUnit</td>
<td>metric</td>
<td>Unit in which the measurement should be performed.</td>
</tr>
<tr>
<td>minAcceptanceInterval</td>
<td>metricSpecification</td>
<td>Minimum value the metric can assume that contributes to the achievement of the qualityRequirement.</td>
</tr>
<tr>
<td>bestValue</td>
<td>metric</td>
<td>Value that indicates the direction in which the real value should tend towards.</td>
</tr>
</tbody>
</table>

Table 4.1: Attributes of the Conceptual Model for Software Quality

In addition, some constraints are necessary to completely establish the semantics of the conceptual model.

- If a qualityCharacteristic is a qualityInUseCharacteristic, then any metric that qualityCharacteristic has is a usageMetric.
- If a qualityCharacteristic is a systemQualityCharacteristic, then any metric that qualityCharacteristic has is a (externalMetric or internalMetric).
• If $\text{metricSpecification}$ refersTo $\text{metric}$, then there is a $\text{qualityRequirement}$ that includes $\text{metricSpecification}$ and $\text{qualityRequirement}$ refersTo a $\text{qualityCharacteristic}$ that has $\text{metric}$ among its metrics.

• If a $\text{qualityRequirement}$ includes no $\text{metricSpecification}(\text{minAcceptanceInterval}, \text{maxAcceptanceInterval})$, then $\text{qualityRequirement}$ is a soft requirement.

• If $\text{minAcceptanceInterval} = \text{null}$ and $\text{maxAcceptanceInterval} = \text{null}$ for all $\text{metricSpecification}$ a qualityRequirement includes, then $\text{qualityRequirement}$ is a soft requirement.

• If a $\text{qualityRequirement}$ includes a $\text{metricSpecification}$ where ($\text{minAcceptanceInterval} \neq \text{null}$) or $\text{maxAcceptanceInterval} \neq \text{null}$), then $\text{qualityRequirement}$ is a hard requirement.

As Task 4.4: Auto Configuration and Quality Variability explicitly mentions performance and security requirements in its description of work (see mention to the goal of this task at the beginning of this chapter), we have instantiated the software product quality conceptual model for security and performance requirements, respectively.

4.1.1 Security

Security is the degree to which information and data are protected so that unauthorized persons or systems cannot read or modify them, and authorized persons or systems are not denied access to them [20]. Table 4.2 provides the sub-characteristics of security as proposed in [20] and its external metrics as proposed in [18]. There are two reasons for the few number of metrics related to security: 1) security was a sub-characteristic of functionality in ISO/IEC 9126 [17, 18] and thus it did not have its own sub-characteristics, as it has now in the new standard ISO/IEC 25010 [20]; 2) according to ISO/IEC 9126 [18], “real security metrics may only be taken in ‘real life system environment’, that is ‘quality in use’ ”. Concerning the latter reason, ISO/IEC 9126-4 [19], which is dedicated to quality in use metrics, does not provide richer set of metrics related to security. In fact, the only quality in use metric in [19] related to security is software damage, which measures the incidence of software corruption and can also measure the number of situations where there was a risk of software damage. Nevertheless the strong relation between security and functionality is clear and that is why quality measures play an important role when fulfilling and assessing security requirements: They add the functionality required to provide the desired level of security. Table 4.3 provides the security measures we have found in the literature. This initial set of security measures will be revised and extended in Task 4.4.

<table>
<thead>
<tr>
<th>Sub-characteristic</th>
<th>External Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confidentiality: Degree to which data or information are protected from unauthorized disclosure of data or information, whether accidental or deliberate</td>
<td>• Access controllability</td>
</tr>
<tr>
<td>Integrity: Degree to which a system or component prevents unauthorized access to, or modification of, computer programs or data</td>
<td>• Data corruption prevention</td>
</tr>
<tr>
<td>Non-repudiation: Degree to which actions or events can be proved to have taken place, so that the events or actions cannot be repudiated later</td>
<td>Access auditability</td>
</tr>
<tr>
<td>Accountability: Degree to which the actions of an entity can be traced uniquely to the entity</td>
<td></td>
</tr>
<tr>
<td>Authenticity: Degree to which the identity of a subject or resource can be proved to be the one claimed</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Security: Sub-characteristics and Metrics

58
<table>
<thead>
<tr>
<th>Measure</th>
<th>Sub-measures</th>
<th>Related Sub-characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Authentication</td>
<td>• Password</td>
<td>• Authenticity</td>
</tr>
<tr>
<td></td>
<td>• One-time password</td>
<td>• Accountability</td>
</tr>
<tr>
<td></td>
<td>• Digital certificate</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Biometric identification</td>
<td></td>
</tr>
<tr>
<td>User Authorization</td>
<td>• Specific user</td>
<td>• Integrity</td>
</tr>
<tr>
<td></td>
<td>• Class of users</td>
<td>• Confidentiality</td>
</tr>
<tr>
<td></td>
<td>• User groups</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• User roles</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• List of individuals</td>
<td></td>
</tr>
<tr>
<td>Encryption</td>
<td>• Symmetric (same key)</td>
<td>Confidentiality</td>
</tr>
<tr>
<td></td>
<td>• Asymmetric (public-private key)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Key length</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Algorithm</td>
<td></td>
</tr>
<tr>
<td>Secure Network Link</td>
<td>• VPN</td>
<td>Confidentiality</td>
</tr>
<tr>
<td></td>
<td>• SSL</td>
<td></td>
</tr>
<tr>
<td>Access Control</td>
<td>• Firewall</td>
<td>• Confidentiality</td>
</tr>
<tr>
<td></td>
<td>• Demilitarized zone</td>
<td>• Integrity</td>
</tr>
<tr>
<td>Event Logging</td>
<td></td>
<td>• Non-repudiation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Accountability</td>
</tr>
<tr>
<td>Data Integrity Checking</td>
<td>• Checksum</td>
<td>Integrity</td>
</tr>
<tr>
<td></td>
<td>• Hash function</td>
<td></td>
</tr>
<tr>
<td>Shared Resources</td>
<td>• Minimum shared resources</td>
<td>• Integrity</td>
</tr>
<tr>
<td></td>
<td>• Maximum shared resources</td>
<td>• Confidentiality</td>
</tr>
<tr>
<td>Attack Detection</td>
<td>• Sensor</td>
<td>• Confidentiality</td>
</tr>
<tr>
<td></td>
<td>• Sensor Fusion</td>
<td>• Integrity</td>
</tr>
<tr>
<td></td>
<td>• Event Logging</td>
<td>• Non-repudiation</td>
</tr>
<tr>
<td></td>
<td>• Control Console</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Off-line Analysis and Report</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3: Security: Measures
The metric “Data corruption prevention” in Table 4.2 can be used to illustrate the relationship metric requires qualityMeasure in Figure 4.1. This metric provides the frequency of data corruption events. However, it is impossible to recognize the occurrence of data corruption events without quality measures for data integrity checking, e.g. via checksum or hash function (see Table 4.3).

Furthermore, an example of a security requirement is: “The system should provide data integrity checking”, which means this requirement includes qualityMeasure “Data Integrity Checking” (see Table 4.3). Another example of a security requirement is: “Access auditability should be higher than 80%”, which means this security requirement includes a metricSpecification that refers to the metric “Access Auditability” (see Table 4.2) and this metricSpecification establishes the acceptance interval for the measurements related to metric.

### 4.1.2 Performance

According to ISO/IEC 25010 [20], performance is relative to the amount of resources used by a system under stated conditions, where resources may include other software products, the software and hardware configuration of the system, human resources, and materials (e.g. print paper, storage media). Table 4.4 provides the sub-characteristics of performance as proposed in [20] and its external metrics as proposed in [18]. We have not surveyed for measures related to performance, because they are more related to the way a system is structured and implemented and do not add functionality.

### 4.2 Methodology

As already mentioned at the beginning of this chapter, we have used the S.P.L.O.T. feature model repository to analyze the HATS Variability Modeling Languages concerning their adequacy and feasibility. The number of features in the 174 feature models provided in the repository ranges from 10 to 290. There are really simple feature models, which represent only a small part of a product line, but there are also complete and rich feature models, which also include quality features.

First, we analyzed the S.P.L.O.T. feature models in order to identify how many and which feature models have included quality features. Table 4.5 provides the title of the feature models and their respective top level quality features.

Second, we analyzed the S.P.L.O.T. feature models in order to assure that they can be represented using \( \mu \text{TVL} \), which means every construct supported by the S.P.L.O.T. feature model repository has an equivalent construct in \( \mu \text{TVL} \). Table 4.6 shows the mapping between constructs.

A Conjunctive Normal Form (CNF) formula is a conjunction of clauses, where a clause is a disjunction of literals. All conjunctions of literals and all disjunctions of literals are in CNF, as they can be seen as conjunctions of one-literal clauses and conjunctions of a single clause, respectively. In other words, the only propositional connectives a formula in CNF can contain are and, or, and not. The not operator can only be used as part of a literal, which means that it can only precede a propositional variable. Nevertheless, every propositional formula can be converted into an equivalent formula that is in CNF. This transformation is based on rules about logical equivalences. The S.P.L.O.T. feature model repository indicates how to represent the constraints Feature-A requires Feature-B (¬Feature-A or Feature-B) and Feature-A excludes Feature-B (¬Feature-A or ¬Feature-B) using CNF formulas.
### Sub-characteristic: Time behavior

The response and processing times and throughput rates of a system when performing its function, under stated conditions in relation to an established benchmark.

<table>
<thead>
<tr>
<th>External Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Response time</td>
</tr>
<tr>
<td>• Response time</td>
</tr>
<tr>
<td>• Mean time to response</td>
</tr>
<tr>
<td>• Worst case response time ratio</td>
</tr>
<tr>
<td>• Throughput</td>
</tr>
<tr>
<td>• Throughput</td>
</tr>
<tr>
<td>• Mean amount of throughput</td>
</tr>
<tr>
<td>• Worst case throughput ratio</td>
</tr>
<tr>
<td>• Turnaround time</td>
</tr>
<tr>
<td>• Turnaround time</td>
</tr>
<tr>
<td>• Mean time for turnaround</td>
</tr>
<tr>
<td>• Worst case turnaround time ratio</td>
</tr>
<tr>
<td>• Waiting time</td>
</tr>
</tbody>
</table>

### Sub-characteristic: Resource utilization

The amounts and types of resources used when the software performs its function under stated conditions in relation to an established benchmark.

<table>
<thead>
<tr>
<th>External Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>• I/O devices resource utilization</td>
</tr>
<tr>
<td>• I/O devices utilization</td>
</tr>
<tr>
<td>• I/O loading limits</td>
</tr>
<tr>
<td>• I/O related errors</td>
</tr>
<tr>
<td>• Mean I/O fulfillment ratio</td>
</tr>
<tr>
<td>• User waiting time of I/O devices utilization</td>
</tr>
<tr>
<td>• Memory resource utilization</td>
</tr>
<tr>
<td>• Maximum memory utilization</td>
</tr>
<tr>
<td>• Mean occurrence of memory error</td>
</tr>
<tr>
<td>• Ratio of memory error/time</td>
</tr>
<tr>
<td>• Transmission resource utilization</td>
</tr>
<tr>
<td>• Maximum transmission utilization</td>
</tr>
<tr>
<td>• Media device utilization balancing</td>
</tr>
<tr>
<td>• Mean occurrence of transmission error</td>
</tr>
<tr>
<td>• Mean of transmission error per time</td>
</tr>
<tr>
<td>• Transmission capacity utilization</td>
</tr>
</tbody>
</table>

Table 4.4: Performance: Sub-characteristics and Metrics
<table>
<thead>
<tr>
<th>Model Title</th>
<th>Model Creator</th>
<th>Quality Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>e-Shop</td>
<td>Samuel Sepuvelda</td>
<td>Security</td>
</tr>
<tr>
<td>Intelligent Tutoring System</td>
<td>Nils Klatter</td>
<td>User authentication</td>
</tr>
<tr>
<td>Stack PL</td>
<td>Markus Voelter</td>
<td>Optimization</td>
</tr>
<tr>
<td>EC2</td>
<td>Adeel Talib</td>
<td>• HighMemory</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• HighCPU</td>
</tr>
<tr>
<td>Body Comfort System</td>
<td>Sebastian Oster</td>
<td>Security</td>
</tr>
<tr>
<td>SmartHome v2</td>
<td>Mauricio Alfarez</td>
<td>Security</td>
</tr>
<tr>
<td>Web Portal</td>
<td>Marcilio Mendonca</td>
<td>• Security</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Performance</td>
</tr>
<tr>
<td>SmartHome v2.2</td>
<td>Mauricio Alfarez</td>
<td>Security</td>
</tr>
<tr>
<td>bCMS System</td>
<td>Istoan Paul</td>
<td>• Data communication confidentiality</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Authentication</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Communication layer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Integrity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Availability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Performance</td>
</tr>
<tr>
<td>HIS</td>
<td>Kyo Kang</td>
<td>• Usability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Scalability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Reliability</td>
</tr>
<tr>
<td>J2EE Web Architecture</td>
<td>Reinout Korbee</td>
<td>Security</td>
</tr>
<tr>
<td>Printers</td>
<td>Priya</td>
<td>• Performance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Security</td>
</tr>
</tbody>
</table>

Table 4.5: S.P.L.O.T. Feature Models with Quality Features

<table>
<thead>
<tr>
<th>General Construct</th>
<th>S.P.L.O.T. Construct</th>
<th>μTVL Construct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mandatory feature</td>
<td>black circle or black square before feature identifier (FID)</td>
<td>no opt before FID</td>
</tr>
<tr>
<td>Optional feature</td>
<td>empty circle or empty square before FID</td>
<td>opt before FID</td>
</tr>
<tr>
<td>And operator</td>
<td>indented FIDs</td>
<td>group allof</td>
</tr>
<tr>
<td>Or operator</td>
<td>inverted trident followed by [1..*]</td>
<td>group [1..*]</td>
</tr>
<tr>
<td>Alternative operator</td>
<td>inverted trident followed by [1..1]</td>
<td>group oneof</td>
</tr>
<tr>
<td>Cross-tree constraints</td>
<td>CNF-formula</td>
<td>require, exclude, ifout and logical expressions using !,</td>
</tr>
</tbody>
</table>

Table 4.6: Mapping between S.P.L.O.T. Constructs and μTVL Constructs
From Table 4.6 and the explanation above one can conclude that all feature models provided in the S.P.L.O.T. feature model repository can be represented in μTVL.

Finally, we have selected a feature model from the S.P.L.O.T. repository to represent in μTVL and we have extended this feature model with some hypothetical delta models, the corresponding product line model, and some valid product configurations by using the HATS Variability Modeling Languages, namely the Delta Modeling Language (DML), the Product Line Configuration Language (CL), and the Product Selection Language (PSL). We selected the feature model with title “Electronic Shopping” provided by Sean Quan Lau from the University of Waterloo, which provides 290 features that can be selected to compose specific eShop web systems. This work is reported in the next section. For the remaining of this chapter, we use “eShop” to refer to Lau’s model.

### 4.3 eShop Web Systems

E-Commerce Systems describe software systems which support business over the Internet. According to [23], they must be able to deal with a large number of visitors and transactions, and coordinate multiple stakeholders to deliver the product or service to the customers. Three common transaction patterns are Business-to-Consumer (B2C), Business-to-Business (B2B), and Consumer-to-Consumer (C2C) [26]. B2C sites, like amazon.com, enable retail transactions where a company sells goods or services to an individual. This is done through an eShop web site, which is sometimes referred to as a shopping cart solution. B2B sites are meant for the exchange of products, services or information between multiple companies. They include company web pages, product supply and procurement exchanges, information sites, and brokerage services. C2C site allows individuals to sell to other individuals. Sites can be run by an intermediary business, such as E-Bay or the Amazon Marketplace, or by consumers themselves.

The feature model we used as basis for our case study focus on supporting B2C eShop solutions with fixed-price purchasing only [23]. The feature model was created based on literature and web research, the IBM WebSphere Commerce Documentation [http://publib.boulder.ibm.com/infocenter/wchelp/v5r6/index.jsp](http://publib.boulder.ibm.com/infocenter/wchelp/v5r6/index.jsp), an existing implementation (Shopping Cart Software: eCommerce Solutions & Hosting [http://www.ablecommerce.com/](http://www.ablecommerce.com/)), and field research involving several large eShop sites. More information about the sources used to create the feature model can be found in [23].

The eShop domain is a relatively large and well-understood domain with a large degree of variability on which serious evaluations of new approaches can be performed. In addition, Lau [23] used appropriate sources to find out a likely set of features a provider of eShop solutions would offer. These were the reasons for selecting Lau’s feature model to perform our case study, despite the fact that the model does not contain any quality features originally.

Figure 4.2 shows the top level features of Lau’s feature model [23]. B2C eShop solutions can then composed by two top features:

**StoreFront:** It defines the interface that the customer uses to access the eShop. Many features are directly visible to the customer and impact their experience at the eShop. The required store front feature consists of a set of functional features representing the home page, catalog, and buy path. The store front may also include registration, wish lists, customer service and / or user behavior tracking capabilities.

**BusinessManagement:** It deals with aspects pertaining to the eShop’s operation. Most of these aspects are back-office concerns, such as product management, order processing, and marketing, which are handled by the eShop staff. Business management features can involve different stakeholders from both inside and outside the company. Business management requires the order management and administration features, which enable order processing and general e-shop management capabilities respectively. Business management capabilities can be augmented by selecting optional features, such

---

1 This model is different from the “e-Shop” model created by Samuel Sepuvelda (Table 4.5) and it does not include quality features.
as targeting, affiliates, inventory tracking, procurement, reporting and analysis, and external systems integration.

In Subsections 4.3.1 and 4.3.2, we provide the sub-features of top features Registration and BuyPaths, while showing their corresponding \( \mu \)TVL models. This sub-features are described as in [23], where one can find the complete explanation of the feature model. Furthermore, we provide the product line definition in CL (Subsection 4.3.3) and two product specifications in PSL (Subsection 4.3.4).

### 4.3.1 eShop Registration

An eShop may enable registration, which allows a customer’s information to be solicited, persisted and reused. This is a convenient feature for customers because they do not have to reenter their information every time they make a purchase. In addition, this information may also be useful for creating targeting strategies. Figure 4.3 shows the registration feature and its sub-features. Registration requires decisions to be made about enforcement and the information that is collected. In addition, user-behavior tracking information can be optionally associated with a profile.

**RegistrationEnforcement:** If registration is enabled, there must be a policy to determine which actions in the eShop are restricted to registered users only. Figure 4.3 shows three policies:

- **RegisterToBrowse:** It restricts browsing to registered customers; it is the most restrictive policy. There are many ways to define browsing permissions. A fine-grained policy would define permissions on certain products or specific details about products, whereas a coarse-grained policy would define permissions on a page type, such as product or search pages. One implementation of this policy restricts access to all product pages to registered customers, but allows guests to see lists of products.

- **RegisterToBuy:** It requires customers to register before they can make a purchase. This can be implemented by requiring customers to log in before they can add an item to their shopping cart or start the checkout process.

- **None:** It is an unrestricted policy. Any visitor can freely browse and purchase items in the eShop without going through the registration process.

**RegistrationInformation:** Registration requires that the customer provide information about themselves, which is stored in a customer profile. Figure 4.3 shows the fields which can be included in a customer profile.
LoginCredentials: It allows customers to identify themselves when they log in. The credentials include a unique identifier, such as an e-mail address, and a password.

ShippingAddress: It specifies where to send the order. Storing multiple shipping addresses may also be supported. There are two possible ways of modeling the multiple shipping address feature: 1) making it an optional sub-feature, or 2) making the shipping address feature a cloneable feature. The former was chosen because it provides a better level of abstraction. If the shipping address is stored in the profile, it is recommended that the shipping feature be supported.

BillingAddress: It specifies where to send the invoice. Storing multiple billing addresses may also be supported; the same points presented about multiple shipping addresses apply here as well. An application of the billing address is to validate credit card information; many eShops require that the billing address matches the address registered with the credit card company.

CreditCardInformation: It consists of the information that is needed to validate the card and process the payment. This information includes the name of the cardholder, card number, expiry date and, optionally, any other additional security information on the card. If the profile supports storing credit card information, the acceptance of credit cards as a form of payment is recommended.

Demographics: It includes information, such as age, income and education, about the customer. A custom demographic field allows the eShop to specify new demographic data fields at run-time. Demographics are used primarily for business intelligence activities. One application is to use previous purchase data of a customer to recommend products to other customers who have similar information in their demographic profile. Another application is to group customers with similar information into a consumer group in order to study and predict trends in consumer behavior for future marketing efforts.

PersonalInformation: It includes any data that can be used to better understand the needs of the customer, excluding any information that is covered by the demographics feature. Examples of personal information are the customer’s hobbies or interests.
Preferences: They are options that allow a customer to customize their eShop interface. They can include site options, such as the site layout, how many items to display in a product list, and the preferred language for rendering the site and any other correspondence.

Reminders: They are customer-requested notifications for pre-defined events. Notification events include informing a customer when a product becomes available or when the price of the product changes. When the event occurs, the customer will receive a notification through a communication channel, such as an e-mail or an on-screen reminder while browsing the eShop. A customer can create, edit and delete their reminders.

QuickCheckoutProfile: It is stored in the customer profile and contains default information that is used when placing an order. The information includes the payment information and, if necessary, the shipping information. If there is support for a quick checkout profile, it is recommended that the quick checkout type also be selected; however, if the quick checkout type is selected, the quick checkout profile is required.

CustomFields: The custom fields allow the eShop to define additional information to be stored in the registration profile after the eShop is deployed at run-time. Enabling custom fields requires a mechanism for the eShop staff to define these fields in terms of their representation, such as the data type, value range, and semantics. The applicability of the custom fields in other eShop workflows, such as business intelligence, depends on the ability to modify the workflows within the eShop.

UserBehaviourTrackingInformation: It allows the eShop to associate data that it collects on user actions to a registration profile. The additional information can be used to interpret the data from a marketing perspective. For example, it can combine the information sources to determine what types of products are browsed by high income visitors or what other sites younger customers tend to visit. This feature requires the selection of the user behavior tracking feature.

Listing 4.1 shows the excerpt of the $\mu$TVL model concerning the sub-tree Registration.

```
root EShop{
  group allof{
    StoreFront{
      group [3..7]{
        ...
        opt Registration{
          group [2..3]{
            RegistrationEnforcement{
              group [1..*]{
                opt RegisterToBrowse,
                opt RegisterToBuy {require: RegisteredCheckout;},
                opt None
              },
            RegistrationInformation{
              group [1..*]{
                LoginCredentials,
                opt ShippingAddress{
                  group [0..1]{
                    opt MultipleShippingAddress
                  } require: Shipping;
                },
                opt BillingAddress{
                  group [0..1]{
                    opt MultipleBillingAddress
                  }
                }
              
```
Listing 4.1: µTVL model of the Registration sub-tree

4.3.2 eShop BuyPaths

Buy paths is a grouping of features relating to the customer purchase workflow. It starts with the checkout process and ends with order placement [25] and can include actions like displaying items in a shopping cart and entering order information. Buy paths, shown in Figure 4.4, consists of three required features: shopping cart, checkout, and order confirmation.

ShoppingCart: It allows a customer to keep track of the items that they wish to purchase during their shopping session. The cart contains a list of products and each product is associated with the quantity that the customer would like to purchase. Placing an item in the shopping cart implies the intent to purchase, but there is no obligation for the customer to complete the transaction. Selection of the shopping cart feature also requires the inventory management policy and cart content page features to be selected. The shopping cart can be further enhanced by the optional cart summary page and cart saved after session features.

InventoryManagementPolicy: It specifies how actions on the shopping cart affect the inventory systems. When an item is placed into a cart, the eShop may reserve an item from the inventory.
If a customer’s session ends and there are items into the cart, it may make sense to release the inventory. When an eShop is selling distinct items, such as tickets for events with reserved seating, items can be sold out very quickly. Therefore, the eShop must associate a timer for each item once it is presented to the customer. If the item is not ordered before the timer expires, the item will be released. Another policy is to delay item reservation until later in the checkout process; however, a customer may be irritated if their order cannot be fulfilled after they have entered all order information. Depending on the policy selected, this feature may require the inventory management feature.

**CartContentPage:** It allows a customer to view all of the items that have been placed in the cart along with the desired quantity and the subtotal. The cart total may also be included. Customers are allowed to edit the quantity of an item or to remove an item from the shopping cart. Tax and shipping costs are usually excluded since they require additional information before they can be calculated. The page also contains a link that is used to start the checkout process.

**CartSummaryPage:** It contains information that is similar to the information found in the cart content page; however, the information may be condensed. The cart summary page does not provide the ability to edit the cart contents directly, but it will contain a link to the cart content page. It can be used to confirm the addition of an item into the cart, which is what Amazon.com does, or to display a summary of the items for confirmation purposes before the order is placed.

**CartSaveAfterSession:** It allows customers to save their cart contents for their next visit. There are two factors which affect this feature. The first factor is the types of customers to which this feature applies; it can apply to registered customers, guests or both. The second factor is where the cart data is saved; it can be saved locally on the visitor’s machine or remotely on the eShop’s servers. Details about the issues that arise depending on the combination of factors can be found in [23]. Regardless of the factors, cart synchronization must be addressed when the product’s price or availability changes while the product is sitting in a saved cart.
CheckOut: It encapsulates the features related to the checkout process. In the process, the customer reviews the items they have added to their shopping cart, enters their payment and shipping information, selects any shipping and gift options, and confirms the order. Checkout requires that the checkout type, taxation options and payment options features be selected; the shipping options feature may also be selected if it is necessary.

CheckoutType: There are two checkout types: registered and guest. An eShop can support both types simultaneously, but the customer will select which checkout type to use during their session at run-time.

RegisteredCheckout: It requires customers to log in before they can start the checkout process. During the checkout, customers must enter or select their shipping and payment information for the order. The selection of the register to buy policy requires the support for a registered checkout. The quick checkout is an optional feature that allows customers to place an order for the items in their shopping cart by using a default set of information from their profile. This feature requires the quick checkout profile feature in order to store the default shipping and payment information. The optional enable profile update on checkout feature allows customers to automatically propagate any profile information or default selection changes to the quick checkout profile when they are using the regular checkout.

GuestCheckout: It allows a guest to purchase products from the eShop. Guests have to enter their personal information to place an order. The information will be stored to fulfill the order and for regulatory reasons, but it will not be available for reuse in a future purchase.

ShippingOptions: It describes the options pertaining to shipping which the customer has control over while going through the checkout. Shipping options require the selection of the shipping feature. There are many optional features in shipping options, including quality of service selection, carrier selection, gift options and multiple shipments; however, the shipping cost calculation feature is mandatory since it is needed to generate the total cost of the order.

TaxationOptions: It describes all of the options available for applying the tax laws and calculating the amount of tax to be charged on an order. Taxation can be affected by many factors, including the location of the purchaser, the location of the eShop’s company, and the items purchased. Exemptions may also be applicable to certain orders depending on the purchaser or the intended use. Taxation can be handled through custom taxation rules, a tax gateway or both. Multiple tax gateways may also be required during a tax calculation depending on the circumstances surrounding the order and the capabilities of each tax gateway.

CustomTaxation: It allows the eShop to define tax calculation strategies. There are two types: fixed-rate taxation and rule-based taxation. In addition, the amount specification feature is needed to determine how to express the tax rate. Both surcharge and percentage methods can be supported, although a tax rate is only specified with one. For more details on rule-based taxation, please refer to [23].

TaxGateways: Due to the complexity of tax laws, an eShop may want to outsource the tax calculations. Tax gateways are third parties who provide tax calculation services; most gateways operate as a web service and can be integrated into the eShop’s checkout process. Tax gateways can include CertiTAX and Cybersource, but the eShop can also support the run-time addition of custom tax gateways.

PaymentOptions: It describes details pertaining to purchase payments from the customer. The payment types feature must be selected since the eShop cannot process any payments without it. In addition, payment options can be further supported by two optional features: fraud detection and payment gateways.

PaymentTypes: It denotes the forms of payment that can be handled by the eShop and can include Cash On Delivery (COD), credit cards, debit cards, electronic cheques, fax mail
orders, purchase orders, gift certificates, phone orders, and a custom payment type. The feature group remains inclusive-or for all binding times because the eShop may be required to accept multiple forms of payment for a single order. For example, a customer may make a purchase which exceeds the value of a gift certificate; therefore, a second payment type is needed to cover the balance.

**FraudDetection:** The fraud detection feature performs checks on the payment information to verify its authenticity. This can be accomplished through card authorization and verification services. Fraud detection also makes use of purchase data, neural networks and rule-based systems to generate a risk score for the eShop. This feature requires expertise that is usually available through an external service, such as a payment gateway. The specific fraud detection capabilities are dependent on the external service chosen. The actual details are service-specific and beyond the scope of this model.

**PaymentGateways:** It allows the eShop to outsource payment services. Payment gateways are third parties that can handle the verification of the payment information, fraud detection, and payment arrangements with financial institutions. Furthermore, different payment gateways can handle different payment types, so the payment types selected can limit the payment gateways which are available and vice-versa. The feature group remains inclusive-or for all binding times. Payment gateways can include Authorize.Net, LinkPoint, Paradata, SkipJack, and VeriSignPayflowPro. If the custom payment gateway feature is selected, a payment gateway can be defined and configured after the eShop is deployed at run-time.

**OrderConfirmation:** It provides an acknowledgment to the customer that the order was received by the order processor and placed successfully. An order number is usually provided to the customer for future reference. This feature is mandatory since customers require feedback after placing an order. Order confirmation can be provided through the following communication channels: electronic page, e-mail, phone or mail. Multiple channels may be used to achieve a higher-level of service; therefore, the feature group is inclusive-or for all binding times.

Listing 4.2 shows the excerpt of the µTVL model concerning the sub-tree BuyPaths.

```xml
<root EShop=
   group allof
      StoreFront{
         group [3..7]{
            ...
            BuyPaths{
               group allof{
                  ShoppingCart{
                     group [2..4]{
                        InventorManagementPolicy,
                        CarContentPage,
                        opt CarSummaryPage,
                        opt CartSavedAfterSession
                     }},
                  Checkout{
                     group [3..4]{
                        CheckoutType{
                           group [1..*]{
                              opt RegisteredCheckout{
                                 group [0..1]{
                                    opt QuickCheckout{
                                       group [0..1]{
                                          opt EnableProfileUpdateOnCheckout
                                       require: QuickCheckoutProfile;
                                    }
                                 }
                              }
                           }
                        }
                     }
                  }
               }
            }
         }
      }
}```
{}},
  opt GuestCheckout
{}},
  opt ShippingOptions{
    group [1..*]{
      opt QualityOfServiceSelection,
      opt CarrierSelection,
      opt GiftOptions,
      opt MultipleShipments,
      ShippingCostCalculation
    } require: Shipping;
  },
  TaxationOptions{
    group [1..*]{
      opt CustomTaxation{
        groupallof{
          Type{
            group [1..*]{
              opt FixedRateTaxation,
              opt RuleBasedTaxation
            }},
          AmountSpecification{
            group [1..*]{
              opt Surcharge,
              opt Percentage
            }
          }
        }
      },
      opt TaxGateways{
        group [1..*]{
          opt CertiTAX,
          opt CyberSource,
          opt CustomTaxGateway
        }
      }
    }
  },
  PaymentOptions{
    group [1..*]{
      PaymentTypes{
        group [1..*]{
          opt COD,
          opt CreditCard,
          opt DebitCard,
          opt ElectronicCheque,
          opt FaxMailOrder,
          opt PurchaseOrder,
          opt GiftCertificate,
          opt PhoneOrder,
          opt CustomPaymentType
        },
        FraudDetection,
        PaymentGateways{
          group [1..*]{
            opt AuthorizeDotNet,
            opt LinkPoint,
            opt Paradata,
            opt SkipJack,
            opt VerisignPayflowPro,
            opt CustomPaymentGateway
          }
        }
      }
    }
  }

4.3.3 Product Line Definition

This subsection provides the configuration of the *B2C eShop with Fixed Price* product line by linking the feature modeling in μTVL and the delta modeling in DML. For the sake of space, Listing 4.3 shows the excerpt of the CL model concerning the features presented in this chapter, which means the features related to the Registration and Buy Path sub-trees. The delta clauses do not intend to be complete and do not deal with interaction and conflict between deltas, because DML was already the focus of the Fredhopper case study. The *B2C eShop with Fixed Price* product line allows the specification of over one billion valid products.

```cl
productline B2CeShopFixedPrice {
    features
        // leaf features of the HomePage sub-tree
        StaticContent, WelcomeMessage, SpecialOffers, ...

        // leaf features of the Registration sub-tree
        RegisterToBrowse, RegisterToBuy, None, ...

        // leaf features of the Catalog sub-tree
        ElectronicGoods, PhysicalGoods, Services, ...

        // leaf features of the WishList sub-tree
        WishlistSavedAfterSession, EMailWishList, MultipleWishLists, ...

        // leaf features of the BuyPaths sub-tree
        InventoryManagementPolicy, CartContentPage, CartSummaryPage, ...

        // leaf features of the CustomerService sub-tree
        QuestionAndFeedbackForms, QuestionAndFeedbackTracking, ProductReturns, ...

        // leaf features of the User Behaviour Tracking sub-tree
        LocallyVisitedPages, ExternalReferringPages, PreviousPurchases, ...

        // leaf features of the Business Management sub-tree
}
```

Listing 4.2: μTVL model of the BuyPaths sub-tree
WarehouseManagement,FlatRate,QuantityPurchased, ..., DomainNameSetup;

// ========= deltas related to the Homepage sub–tree =========
...

// ========= deltas related to the Registration sub–tree =========
delta RegisterToBrowseDelta after RegistrationEnforcementDelta when RegisterToBrowse;
// it asks login credentials when someone tries to access the catalog

delta RegisterToBuyDelta after RegistrationEnforcementDelta when RegisterToBuy;
// it asks login credentials when someone tries to include an item in the shopping cart

delta ShippingAddressDelta after RegistrationInformationDelta when ShippingAddress;
// it allows the registration of a shipping address in the registration page

delta MultipleShippingAddressDelta after ShippingAddressDelta when MultipleShippingAddress;
// it allows the registration of multiple shipping addresses in the registration page

delta BillingAddressDelta after RegistrationInformationDelta when BillingAddress;
// it allows the registration of a billing address in the registration page

delta MultipleBillingAddressDelta after BillingAddressDelta when MultipleBillingAddress;
// it allows the registration of multiple billing addresses in the registration page

delta CreditCardInformationDelta after RegistrationInformationDelta when CreditCardInformation;
// it allows the registration of credit card information in the registration page

delta SecurityInformationDelta after CreditCardInformationDelta when SecurityInformation;
// it allows the registration of the security code of the credit card in the registration page

delta DemographicsDelta after RegistrationInformationDelta when Demographics;
// it creates an area to request demographic information in the registration page

delta AgeDelta after DemographicsDelta when Age;
// it allows the registration of person’s age in the registration page

delta IncomeDelta after DemographicsDelta when Income;
// it allows the registration of person’s income in the registration page

delta EducationDelta after DemographicsDelta when Education;
// it allows the registration of person’s education in the registration page

delta CustomDemographicFieldDelta after DemographicsDelta when CustomDemographicField;
// it allows the definition of custom demographic fields and their inclusion in the registration page

delta PersonalInformationDelta after RegistrationInformationDelta when PersonalInformation;
// it allows the registration of personal information in the registration page

delta PreferencesDelta after RegistrationInformationDelta when Preferences;
// it creates an area to request configuration information in the registration page

delta SiteLayoutDelta after PreferencesDelta when SiteLayout;
// it allows the selection of a specific site layout in the registration page

delta ListSizeDelta after PreferencesDelta when ListSize;
// it allows the definition of a list size for browsing in the registration page

delta LanguageDelta after PreferencesDelta when Language;
// it allows the selection of a language for the information provided in the eShop

delta RemindersDelta after RegistrationInformationDelta when Reminders;
// it allows the definition of desired reminders in the eShop

delta QuickCheckoutProfileDelta after RegistrationInformationDelta when QuickCheckoutProfile;
// it allows the creation of a quick checkout profile for usage in the eShop

delta CustomFieldsDelta after RegistrationInformationDelta when CustomFields;
// it allows the definition of custom fields and their inclusion in the registration page

// ========= deltas related to the Catalog sub-tree =========
...

// ========= deltas related to the Wishlist sub-tree =========
...

// ========= deltas related to the BuyPaths sub-tree =========

delta RegisteredCheckoutDelta after CheckoutTypeDelta when RegisteredCheckout;
// it gets the information provided by the user during registration to support the checkout process

delta QuickCheckoutDelta after RegisteredCheckoutDelta when QuickCheckout;
// it uses the information selected in the QuickCheckout Profile as the actual information for the current checkout
// process

delta EnableProfileUpdateOnCheckoutDelta after QuickCheckoutDelta when
EnableProfileUpdateOnCheckout;
// it updates the QuickCheckout Profile with the information provided in the actual information for the current checkout
// process

delta GuestCheckoutDelta after CheckoutTypeDelta when GuestCheckout;
// it gets the necessary information for the checkout process during the checkout process and this information is only
// kept in the purchase context

delta ShippingOptionsDelta after CheckoutTypeDelta when ShippingOptions;
// it calculates the shipping cost

delta QualityOfServiceSelectionDelta after ShippingOptionsDelta when QualityOfServiceSelection;
// it allows the specification of the desired quality of service of the shipment service

delta CarrierSelectionDelta after ShippingOptionsDelta when CarrierSelection;
// it allows the user to select a specific carrier
...

delta EletronicPageDelta after OrderConfirmationDelta when EletronicPage;
// it shows an electronic page that confirms the purchase

delta EmailDelta after OrderConfirmationDelta when Email;
// it sends an e-mail that confirms the purchase

delta PhoneDelta after OrderConfirmationDelta when Phone;
// it requests a phone number to be used to confirm the purchase
delta MailDelta after OrderConfirmationDelta when Mail;
// it requests a mail for confirming the purchase

// ========= deltas related to the CustomerService sub−tree =========
...

// ========= deltas related to the UserBehaviourTracking sub−tree =========
...

// ========= deltas related to the BusinessManagement sub−tree =========
...
}

Listing 4.3: CL Model of the B2C eShop with Fixed Price Product Line

4.3.4 Product Specifications

Listing 4.4 provides the specification of two examples of B2C eShop with Fixed Price. The first product (EShopBasic) is the most elementary product possible. Its homepage only contains static contents and only shopping without registration is supported. The second product (EShopRegistration) offers a slightly richer set of features. It is a product with registration enforcement for purchase purposes and therefore can support order confirmation by e-mail.

// basic product with no deltas
product EShopBasic (

// selection of leaf features of the HomePage sub−tree
StaticContent,

// selection of leaf features of the Catalog sub−tree
PhysicalGoods, BasicInformation,

// selection of leaf features of the BuyPaths sub−tree
InventoryManagementPolicy, CartContentType, GuestCheckout, CertiTax, CreditCard, ElectronicPage,

// selection of leaf features of the OrderManagement sub−tree
WarehouseManagement, FlatRate,

// selection of leaf features of the Administration sub−tree
ProductDatabaseManagement, PresentationOptions, GeneralLayoutManagement, SiteSearch, SearchEngineRegistration, DomainNameSetup);

// Product with registration enforcement and some additional features
product EShopRegistration (

// selection of leaf features of the HomePage sub−tree
StaticContent,

// selection of leaf features of the Registration sub−tree
RegisterToBuy, LoginCredentials,

// selection of leaf features of the Catalog sub−tree
PhysicalGoods, BasicInformation,

// selection of leaf features of the BuyPaths sub−tree
InventoryManagementPolicy, CartContentType, RegisteredCheckout, QuickCheckout, CertiTax,
4.4 Extending eShop with Quality Features

The feature model for the B2C eShop solutions proposed by [23] does not include quality features as other feature models in the S.P.L.O.T. feature model repository do (see Table 4.5). Nevertheless we have selected it to perform our case study due to its size and high-quality documentation.

In this section, we investigate the extension of the original feature model with quality features concerning security and performance, so that we can better understand the issues to be addressed in Task 4.4 in order to support the configuration of product lines that meet the desired end-to-end functionality, and performance and security requirements in an optimal, or close to optimal fashion.

Figure 4.5 shows our extension of Lau’s feature model [23] with the security and performance top level features.

![Extended Top level features of B2C eShop solutions](image)

4.4.1 eShop Security

Figure 4.6 illustrates an applicable set of security-related features for eShop web systems, which can be defined as follows.

**Authentication:** It defines the mechanisms that should be available in the eShop system to authenticate whether a subject or resource is the one claimed. The options are password or digital certificates, as the other authentication measures proposed in Table 4.2 would put in risk the willingness of customers to access the eShop.

Listing 4.4: PSL Model of two examples of B2C eShop with Fixed Price

```plaintext
// selection of leaf features of the OrderManagement sub−tree
WarehouseManagement, FlatRate,

// selection of leaf features of the Administration sub−tree
ProductDatabaseManagement, PresentationOptions, GeneralLayoutManagement, SiteSearch,
SearchEngineRegistration, DomainNameSetup);
```

Figure 4.5 shows our extension of Lau’s feature model [23] with the security and performance top level features.
Authorization: It consists of the mechanisms that should be available in the eShop system to control the access of its users to functions and data. The options are specific user, user roles or list of individuals. For example, the owner of a credit card should have access to his/her credit card information (access to a specific user), but shipping carrier staff should have access to shipping addresses (access according to the user roles).

Confidentiality: It defines the mechanisms that should be available in the eShop system to protect data or information from unauthorized disclosure, whether accidental or deliberate. The options are the usage of encryption, either symmetric or asymmetric, or the usage of secure protocol. The selection of secure protocol requires the selection of a specific protocol, which is considered secure, such as HTTPS.

Integrity: It consists of mechanisms to detect unauthorized modification of the eShop system or any of its data. The options are checksum or hash function.

AttackDetection: It defines the mechanisms that should be available in the eShop system to monitor network and/or system activities for malicious activities or policy violations and to produce the respective reports. The options are sensors, event logging, offline analysis and report, and monitoring console. The selection of monitoring console requires the selection of sensors and the selection of offline analysis and report requires the selection of sensors or event logging. The optional sensor fusion allows the combination of information from different sensors.

Listing 4.5 shows the excerpt of the $\mu$TVL model concerning the Security sub-tree.

```plaintext
text{Listing 4.5}
```
Listing 4.5: μTVL model of the Security sub-tree
4.4.2 eShop Performance

Figure 4.7 provides a graphical representation of an applicable set of performance-related features for eShop web systems, which can be defined as follows.

**ResponseTime**: It defines the target response time of the eShop system after a transmission key is pressed by a customer. The options are: high response time, which means target response time equal or less than three seconds, and medium response time, which means target response time between three and five seconds. It should be possible to specify different target response time for different parts of the system, for example high response time for *StoreFront* and medium response time for *BusinessManagement*.

**Throughput**: It defines the number of concurrent tasks the eShop system should be able to handle over a set unit of time. Throughput in an eShop system can be described in terms of the number of concurrent browsings, order placements, or transactions.

**ThroughputBrowsing**: It defines the number of concurrent browsings the eShop system should be able to handle over a second. The alternative options are: high browsing (capability), which means support to at least a hundred concurrent browsings over a second, and medium browsing (capability), which means support to at least fifty concurrent browsings over a second.

**ThroughputOrderPlacement**: It defines the number of concurrent order placements the eShop system should be able to handle over a second. The alternative options are: high order placement (capability), which means support to at least ten concurrent order placements over a second, and medium order placement (capability), which means support to at least five concurrent order placements over a second.

**ThroughputTransaction**: It defines the number of concurrent transactions the eShop system should be able to handle over a second. The alternative options are: high transaction (capability), which means support to at least a hundred concurrent transactions over a second, and medium transaction (capability), which means support to at least fifty concurrent transactions over a second.

**ResourceUtilization**: It defines the maximum load of resource utilization the eShop system should have as target on the server side. Resource utilization in an eShop system can be described in terms of number of CPU, memory, bandwidth or storage utilization.
**CPUUtilization:** It defines the maximum load in terms of CPU utilization. The alternative options are: low CPUPeak, which means peak of CPU utilization equal or less than 75%, and medium CPUPeak, which means peak of CPU utilization over 75% and less than 90%.

**MemoryUtilization:** It defines the maximum load in terms of memory utilization. The alternative options are: low memory utilization, which means memory utilization equal or less than 50 MB, and medium memory utilization, which means memory utilization greater than 50 MB and less than 150 MB.

**BandwidthUtilization:** It defines the maximum load in terms of bandwidth utilization. The alternative options are: low bandwidth utilization, which means bandwidth utilization equal or less than 100 Mbps and medium bandwidth utilization, which means bandwidth utilization greater than 100 Mbps and less than 150 Mbps.

Listing 4.6 shows the excerpt of the μTVL model concerning the Performance sub-tree.

```plaintext
root EShop{
group [2..3]{
  ...  
  opt SoftwareQuality{
    group [0..*]{
      ...  
      opt Performance{
        group [1..*]{
          opt ResponseTime{
            group [1..*]{
              opt MediumResponseTime{
                ifin: (varResponseTime >= 3) && (varResponseTime < 5);
              },
              opt HighResponseTime{
                ifin: varResponseTime >= 5;
              }
            }
          } // end of ResponseTime group
          Int varResponseTime;
        }, // end of ResponseTime
        opt ThroughPut{
          group [1..*]{
            opt ThroughputBrowsing{
              group oneof{
                opt HighBrowsing{
                  ifin: varThroughputBrowsing >= 100;
                },
                opt MediumBrowsing{
                  ifin: (varThroughputBrowsing >= 50) && (varThroughputBrowsing < 100);
                }
            }
          } // end of ThroughputBrowing,
        }, // end of Throughput
      }, // end of ThroughPut
    } // end of SoftwareQuality
  } // end of EShop
}
```

80
} 
} 
Int varThroughputOrderPlacement;
} // end of ThroughputOrderPlacement,
opt ThroughputTransaction{ 
group oneof{ 
  opt HighTransaction{ 
    ifin: varThroughputTransaction >= 100; 
  }, 
  opt MediumTransaction{ 
    ifin: (varThroughputTransaction >= 50) && (varThroughputTransaction < 100); 
  } 
}
Int varThroughputTransaction;
} // end of ThroughputTransaction
}, // end of Throughput 
opt ResourceUtilization{ 
group [1..*]{ 
  CPUUtilization{ 
    group oneof{ 
      opt LowCPUPeak{ 
        ifin: varCPUUtilization <= 75; 
      }, 
      opt MediumCPUPeak{ 
        ifin: (varCPUUtilization > 75) && (varCPUUtilization <= 90); 
      } 
    }
  }, // end of CPUUtilization 
  MemoryUtilization{ 
    group oneof{ 
      opt LowMemoryUtilization{ 
        ifin: varMemoryUtilization <= 50; 
      }, 
      opt MediumMemoryUtilization{ 
        ifin: (varMemoryUtilization > 50) && (varMemoryUtilization <= 150); 
      } 
    }
  }, // end of MemoryUtilization, 
  BandwidthUtilization{ 
    group oneof{ 
      opt LowBandwidthUtilization{ 
        ifin: varBandwidthUtilization <= 100; 
      }, 
      opt MediumBandwidthUtilization{ 
        ifin: (varBandwidthUtilization > 100) && (varBandwidthUtilization <= 150); 
      } 
    }
  }, // end of BandwidthUtilization 
} // end of ResourceUtilization group
4.4.3 Extended Product Line Definition

The extension of the configuration of the B2C eShop with Fixed Price product line to take into consideration security is straightforward, as the features that represent the security measures can be integrated through the corresponding delta models. However, Task 4.4 must investigate how to support the configuration of product lines taking into consideration performance features. Task 4.2 has specifically addressed resource guarantees and the goal is to use their results to allow performance metrics to be broken down using structural information from the models.

4.4.4 Extended Product Specifications

The extension of the product specifications to take into consideration security and performance features is straightforward if the same quality features should apply to the whole system. This case is illustrated in Listing 4.7, which provides the specification of the same two examples of eShop products presented in Subsection 4.3.4 but now extended with the specification of the security and performance features. However, Task 4.4 must investigate how to support the specification of quality features that only apply in the context of specific functional features, for example in the case that feature HighResponseTime should apply to all features in the StoreFront sub-tree, while MediumResponseTime is admitted for features in the BusinessManagement sub-tree.

Listing 4.6: $\mu$TVL model of the Performance sub-tree
4.5 Evaluation Based on eShop Product Line

In this section, we draw some conclusions from the experience of using the HATS Variability Modeling Languages to model an eShop product line.

4.5.1 Variability Modeling

The adequacy of the HATS Variability Modeling languages to support the development of product lines of information systems is related to their expressiveness power. We have analyzed the S.P.L.O.T. repository of feature models and verified that all models can be represented in $\mu$TVL. We have used all HATS Variability Modeling languages with focus on $\mu$TVL, CL and PSL to model a B2C eShop product line. The limitation of this case study was not to deeply investigate the adequacy of the delta clauses in the CL model.

In the description of the features ShippingAddress, CreditCardInformation and QuickCheckoutProfile (Section 4.3.1), Lau [23] refer to the fact that the inclusion of a feature is recommended (not required) in case another feature is included. The possibility of capturing recommendations is neither supported by $\mu$TVL nor by the S.P.L.O.T. repository. After analysing each case, we have the opinion that in practical terms the three recommendations would be better captured as normal require, because it is useless to have information on shipping address, credit card and quick checkout if this information cannot not be used.

The feasibility of the HATS Variability Modeling languages is related to aspects that can hinder their adoption, such as learnability, usability, scalability, tool support and integration with product line engineering approaches. One can find specific comments on those aspects in Subsection 4.5.2, which addresses the requirements of the HATS methodology. In general, scalability is an aspect to be improved in the set of languages, in particular CL and PSL. The development of tooling to support the instantiation of products from the product line was already planned in the project proposal to occur in this year.

4.5.2 Methodological Requirements

In this subsection, we make some statements based on the B2C eShop case study about how far the HATS Variability Modeling languages fulfill the pertinent methodological requirements identified in D5.1: Requirements Elicitation.

Integrating Product Line Engineering (MR1)

The HATS Variability Modeling languages are completely integrated in the Product Line Engineering (PLE) life-cycle and aligned with existing product line approaches. $\mu$TVL captures the results of the Product Line Requirements Analysis phase, DML supports the design and realization of generic components, CL establishes the link between the output of these two phases, which is used to automatically generate the product line products.

Providing Language Support for PLE (MR4)

Consortium partners are currently working on a common concept of component that will allow the definition of a product line architecture and the specification of quality properties of components. The concept

Listing 4.7: Extended PSL Model of the two examples of B2C eShop with Fixed Price

// selection of leaf features of the Performance sub-tree
MediumResponseTime, HighBrowsing, MediumOrderPlacement,
MediumTransaction, LowCPUPeak, MediumMemoryUtilization,
LowBandwidthUtilization);
of component is expected to support modularity and consequently improve the scalability of the HATS Variability Modeling approach.

Defining Reusable Artifacts and Variation Points (MR6)

The HATS Variability Modeling languages allow the definition of reusable artifacts (core ABS model and delta models) and the variation points contained therein ($\mu$TVL) as well as the according interdependencies (CL model).

Tailorability (MR7)

The HATS Variability Modeling languages have not been designed to be tailored to a specific organizational context. However, the HATS methodology, which defines how to use the several HATS approaches in an integrated way can be tailored to a specific organizational context and, therefore, take into consideration the existing development practices in an organization, the application domain, the structure of an organization, or the experience of the developers.

Incremental Adoptability (MR8)

A company can initially adopt the Core ABS framework or parts of it and then extend its usage of the HATS framework with the HATS Variability Modelling languages in order to support the development of a family of similar products. A tool to automatically translate a $\mu$TVL model into a graphical feature tree would support the adoption of this modelling language.

Scalability (MR10)

For modeling large product lines in $\mu$TVL, it would be beneficial to support the breakdown of feature models into cohesive parts and to allow their posterior composition. The same applies to the definition of product lines using CL. The language designers are currently evaluating how to improve modularity in the variability models.

Learnability (MR11)

The HATS Variability Modeling languages are well documented and their concepts are easy to understand or are already well-known by those who develop product lines.

Usability (MR12)

The ABS development environment is easy to use. CL can be improved to make it easier to refer to the features that should be included in a specific product line. The language designers are currently evaluating how to better approach this issue. Furthermore, Task 4.4 should result in a configurator that supports the selection of features to compose a specific product. The configurator will ask its users to select only those features that the configurator cannot automatically select from the information provided in the feature model, e.g. by the require and exclude constraints. Without such a configurator the specification of products is time consuming and error prone for several reasons, starting from the difficulty of identifying the leaf features in a large feature model.

Reducing Manual Effort (MR13)

This requirement has some relation with MR12. The improvement of CL to make it easier to refer to the features that should be included in a specific product line will reduce manual effort. Furthermore, Task 4.4 has also the goal of addressing this issue, by supporting semi-automatic product specification.
Integrated Environment Support (MR18)

The HATS development environment is an Eclipse plug-in which supports the usage of all HATS Variability Modeling languages with a common visual representation.

Existing Modeling Techniques Support (MR19)

This requirement has some relation with MR8, but it is the other way around here: Existing feature models could be translated into $\mu$TVL, which would also support the adoption of the language.

4.6 Summary

This evaluation of the HATS Variability Modeling Languages using the B2C eShop case study was very helpful to provide insights on how to improve their scalability. In order to perceive scalability problems, it is necessary to model large product lines. Real product lines might have thousands of features. Some alternatives to address the scalability issues have already been identified and the best alternatives will be implemented this year or as part of the exploitation plan of the consortium partners.

The B2C eShop case study was also very useful to show that the HATS Variability Modeling Languages can already deal with security and performance features at some extent. It is clear that they still need to allow the association of quality features to functional features in order to be able to specify quality requirements for a specific part of the system. Task 4.4 will address not only this issue but also how to decompose performance features and/or compose performance attributes of the system components to obtain configurations that meet the desired performance requirements.
Chapter 5

Summary

In this deliverable, we have evaluated the different modeling techniques of the project using various case studies, most thoroughly delta modeling in the Fredhopper Access Server (Chapter 3). We have also performed requirements analysis in extension of Tasks 5.1 and 5.2 [7, 8].

In Chapter 3 we considered the Fredhopper case study and evaluated ABS modeling with respect to concrete requirements TS-R-1.2-1, TS-R-1.2-2, TS-R-2.2-1, TS-R-2.2-2, VF-R-1.2-1 and VF-R-2.2-1, methodological requirements MR1, MR2, MR3, MR4, MR6, MR7, MR10, MR11, MR12, MR13, MR18, MR19, MR20 and MR22 as well as variability, behavioral and resource modeling. We made several recommendations based on this case-study, which are summarized in Section 5.2.

In Chapter 4 we considered the eShop case study and evaluated ABS modeling with respect to the adequacy and feasibility of the HATS Variability Modeling Languages from the point of view of the researchers in the context of the features models provided in the S.P.L.O.T feature model repository, as well as methodological requirements MR1, MR4, MR6, MR7, MR8, MR10, MR11, MR12, MR13, MR18, and MR19. The B2C eShop case study was also very useful to show that the HATS Variability Modeling Languages can already deal with security and performance features at some extent. It is clear that they still need to allow the association of quality features to functional features in order to be able to specify quality requirements for a specific part of the system. Task 4.4 will address not only this issue but also how to decompose performance features and/or compose performance attributes of the system components to obtain configurations that meet the desired performance requirements.

5.1 Validation of Milestone M2

As a result of this evaluation, we have validated that Milestone M2 of the HATS project has been achieved.

5.2 Recommendations

In this section we provide some recommendations to improve the ABS language and the ABS tool on variability, behavioral and resource modeling.

5.2.1 Variability Modeling

We provide some recommendations to improve the ABS language and the ABS tool suite on variability modeling:

- Features in $\mu$TVL may have attributes that can be used to specify constraints between features as well as parameter values to delta modules during product selections. Currently feature attributes only support ABS data types Int and Bool. We would like to see other data types in the standard library, such as String and List, to also be supported.
\( \mu \text{TVL} \) is a formal text based language for specifying feature models. Given \( \mu \text{TVL} \) has formal syntax and semantics, we would like to see the \( \mu \text{TVL} \) tool to provide the facility to generate feature diagrams; it would be easier to navigate around a large feature model graphically.

- The \( \mu \text{TVL} \) tool is built into the ABS Eclipse plugin to provide the syntax highlighting, and type checking facilities for \( \mu \text{TVL} \) models. However, at the time of writing, the ABS Eclipse plugin does not provide code navigation and outline for \( \mu \text{TVL} \) models. We feel these functionalities increase usability of \( \mu \text{TVL} \), especially when models become large and complex. We would therefore like to see these functionalities to be provided.

### 5.2.2 Behavioral Modeling

We provide some recommendations to improve the ABS language and the ABS tool suite on behavioral modeling.

- The current ABS tool suite does not provide type checking facility for delta modules, this means that to type check the complete product line, one needs to type check every valid product selections in the product line. This approach does not scale as the number of products can grow exponentially on the number of features the product line provides. We therefore would like to see type checking facility to be provided for delta modules. We believe there has been work towards this direction in Task 2.4 and at the time of writing, progress has been made toward providing type checking facility to delta modules [24].

- The current ABS tool suite does not provide pretty printing facility for generated products. Generated products remain as ASTs until a back end is chosen to generate the products to a particular implementation language. We believe it would be beneficial to provide a pretty printer that can take some (core) ABS AST and outputs the corresponding (core) ABS model in ASCII. We envisage using such outputs for debugging a (Java) execution of the model as well as visualization during symbolic execution.

### 5.2.3 Resource Modeling

We provide some recommendations to improve the ABS language and the ABS tool suite on resource modeling.

- RT-ABS and its Maude back end provide ways to specify the cost of ABS statements, thereby allowing us to simulate the effect of executing these statements over a finite amount of resources (deployment components). At the time of writing the RT-ABS implementation supports only the modeling of CPU resource consumption, we believe it would be beneficial to provide other resource models such as physical memory and bandwidth.
Bibliography


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.

**Abstract Delta Modeling** An algebraic and abstract formalism describing the semantics of deltas and delta models.

**ADM** Abstract Delta Modeling

**B2C** Business-to-Consumer.

**Business-to-Consumer** Common transaction pattern of E-Commerce Systems. B2C sites enable retail transactions where a company sells goods or services to an individual.

**CL** Product Line Configuration Language.

**COG** Concurrent Object Group, the unit of parallelism in ABS.

**Conflict** The condition between two incompatible, non-ordered deltas.

**Conflict Resolving Delta** The delta that resolves a given conflict between two other deltas. It has to be greater in the partial order than the conflicting deltas and equalize the two possible orderings between them.

**Core ABS** The behavioral functional and object-oriented core of the ABS modeling language. See ABS.

**Delta** A unit of functionality and conflict resolution in delta modeling, able to modify a product using invasive composition of code or other content.

**Delta Model** Generally, a means for expressing the semantics of features within product lines. In ADM, a delta model is defined more specifically as \((D, \prec)\), a partially ordered set of deltas.

**Delta Modeling Workflow** A step-by-step guide towards the concurrent and isolated development of a software product line using delta modeling, preserving useful properties.

**Delta Modeling Language** HATS Variability Modeling Language that expresses the code-level variability required for a SPL. It concerns feature integration.

**Deployment component** A modeling abstraction for deployment choices, restricting the execution capacity of different parts of ABS models.

**DML** Delta Modeling Language.

**DMW** Delta Modeling Workflow.

**FAS** Fredhopper Access Server
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.

Live environment  A live environment in the FAS deployment architecture is responsible for processing queries from client web applications via the Web Services technology.

Product Line Configuration Language  HATS Variability Modeling Language that links a feature model and delta modules together and forms the top level specification of an entire SPL.

PSL  Product Specification Language.

Product Specification Language  HATS Variability Modeling Language that expresses individual products by providing feature and attribute selection.

Replication system  The Replication System in the FAS deployment architecture synchronizes the configurations and data from the staging environment to multiple live environments. Specifically the Replication System consists of the synchronization server (SyncServer) and one or more clients (Sync-Client).

Software Family  Software Product Line.

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

SPL  Software Product Line

Staging environment  A staging environment in the FAS deployment architecture is responsible for receiving client data updates in XML format, indexing the XML, and distributing the resulting indices across all live environments using the Replication System. See Replication System.

Unambiguous Delta Model  The property in a delta model that every conflict is properly resolved, such that a unique implementation is guaranteed.

$\mu$TVL  HATS Variability Modeling Language that expresses variability on the level of feature models.
Appendix A

ABS Model of SyncClient

This chapter presents the ABS model of the class SyncClient and ClientJob that implements the sequential job processing.

```java
[C0G] class SyncClientImpl(
    [Final] Int maxTransactionId,
    [Final] Int maxJobs,
    [Final] ClientId id) implements InternalClient, ClientConnector {

    StateMachine machine = stateMachine();
    State state = Start;

    Network network;
    ServerAcceptor acceptor;
    ClientDataBase db;

    Bool shutDown = False;
    Bool next = False;
    Set<ClientJob> jobRecords = EmptySet;

    // measurement
    List<ClientJob> jobHistories = Nil;
    List<JobData> jobDatas = Nil;
    List<Int> jobDurations = Nil;
    List<Schedule> hit = Nil;
    List<Schedule> missed = Nil;

    Int currentTransactionId = -1;

    {
        // initialize the client side data base
        db = new DataBaseImpl();
    }

    [Atomic]
    Unit setMaximumTransactionId(Int id) {
        currentTransactionId = id;
    }

    [Atomic]
    Pair<Int,Int> jobCountAndMaximumTransactionId() {
        return Pair(length(jobHistories),currentTransactionId);
    }
```

92
Unit scheduleJob(JobType jb, Schedule schedule) {
    // wait for its time to be initiated
    // and the next available slot
    this.waitFor(schedule);

    // only proceed if a shutdown
    // request has not been made.
    if (!shutDown) {
        // block subsequent onces
        this.setNext(schedule);
        this.makeJob(jb, schedule);
        hit = Cons(schedule, hit);
    } else {
        // record those schedules that are missed
        missed = Cons(schedule, missed);
    }
}

Unit setNext(Schedule schedule) {
    next = False;
}

Unit waitFor(Schedule schedule) { .. }

Unit makeJob(JobType jb, Schedule schedule) {
    ClientJob job = new cog ClientJobImpl(maxTransactionId, maxJobs, this, jb, schedule);
    job!executeJob();
    jobHistories = Cons(job, jobHistories);
    jobRecords = Insert(job, jobRecords);
}

Unit finishJob(ClientJob job) {
    jobRecords = remove(jobRecords, job);
}

Unit nextJob() {
    next = True;
}

ClientId getId() {
    return id;
}

Bool isShutdownRequested() {
    return shutDown;
}

Unit requestShutDown() {
    shutDown = True;
    await jobRecords == EmptySet;
    network!shutDown(this);
}

ServerAcceptor getAcceptor() {
    return acceptor;
}
Unit run() {
    this.becomesState(WaitToBoot);
    await acceptor != null;
    this.makeJob(Boot,NoSchedule);
}

ClientDataBase getClientDataBase() {
    return db;
}

DataBase getDataBase() {
    return db;
}

Unit becomesState(State state) {
    Set<State> tos = lookupDefault(machine,this.state,EmptySet);
    assert tos != EmptySet; // this is an end state
    assert contains(tos,state); // cannot proceed to specified state
    this.state = state;
}

Unit setAcceptor([Far] ServerAcceptor acc) {
    acceptor = acc;
}

Unit setNetwork(Network network) {
    this.network = network;
}

class ClientJobImpl(
    [Final] Int maxTransactionId,
    [Final] Int maxJobs,
    [Far] [Final] InternalClient client,
    [Final] JobType job,
    [Final] Schedule schedule) implements ClientJob {

    Command start = EmptyCommand;
    Command command = EmptyCommand;
    Schedules schedules = EmptySet;
    ClientId clientId = -1;

    ConnectionThread thread = null;
    [Far] ClientDataBase db;

    ConnectionThread getConnectionThread() {
        Fut<ServerAcceptor> fs = client!getAcceptor();
        ServerAcceptor acceptor = fs.get;

        // Acquire a connection
        Fut<ConnectionThread> t = acceptor!getConnection(this); await t?;
        return t.get;
    }

    Unit clientDB() {
        Fut<ClientDataBase> fd = client!getClientDataBase();
    }
// starts the next set of replication jobs
Unit establishSchedule() {
    Fut<Pair<Int,Int>> jcf = client!jobCountAndMaximumTransactionId();
    Pair<Int,Int> stats = jcf.get;
    if (fst(stats) >= maxJobs || snd(stats) >= maxTransactionId) {
        this.shutDownClient();
    } else {
        Schedules ss = schedules;
        while (hasNext(ss)) {
            Pair<Schedules,Schedule> nt = next(ss);
            ss = fst(nt); Schedule s = snd(nt);
            client!scheduleJob(Replication,s);
        }
    }
}

Unit executeJob() {
    Fut<ClientId> fut = client!getId();
    clientId = fut.get;

    // set data base
    this.clientDB();

    // Acquire a connection
    thread = this.getConnectionThread();

    if (thread != null) {
        if (job == Boot) {
            this.becomeState(Booting);
            thread!command(ListSchedule);
            await schedules != EmptySet;
            //establish the next schedule triggers!
            this!establishSchedule();
        } else {
            this.becomeState(WorkOnReplicate);

            thread!command(SearchSchedule(schedname(schedule)));
            await schedules != EmptySet;
            //establish the next schedule triggers!
            this!establishSchedule();

            // wait for current job to start then end
            await start == StartSnapShot;
            await command == EndSnapShot;
        }
    }

    //Switch to the proper state after finishing.
    //From all states we can go to the end state to shutdown.
    Fut<Bool> sd = client!isShutdownRequested(); Bool shutDown = sd.get;
    if (~shutDown) {
        this.becomeState(WaitToReplicate);
    }
}
// allow next job to proceed
this.nextJob();
}
client!finishJob(this);

// allow next job to proceed
Unit nextJob() {
  client!nextJob();
}

Unit becomeState(State state) {
  Fut<Unit> unit = client!becomesState(state); unit.get;
}

ClientId forClient() {
  return clientId;
}

Unit shutDownClient() {
  Fut<Bool> bf = client!isShutdownRequested(); await bf;
  Bool bool = bf.get;
  if (bool) {
    Fut<Unit> unit = client!requestShutDown(); await unit;
    this.becomeState(End);
  }
}

Bool registerReplicationItems(TransactionId id) {
  Fut<Bool> reg = db!prepareReplicationItem(id, schedule);
  Bool rg = reg.get;
  if (rg) {
    Fut<Unit> u = client!setMaximumTransactionId(id); u.get;
  }
  return rg;
}

Bool hasFile(FileId id) {
  Fut<Bool> he = db!hasFile(id); await he;
  return he;
}

Maybe<FileSize> processFile(FileId id) {
  Maybe<FileSize> result = Nothing;
  Bool hasfile = this.hasFile(id);
  if (hasfile) {
    Fut<FileContent> contentf = db!getContent(id); await contentf;
    FileContent content = contentf.get;
    if (isFile(content)) {
      FileSize size = content(size);
      result = Just(size);
    }
  }
  return result;
}
Unit overwrite(File file) {
    FileId id = fst(file);
    FileSize size = fileContent(file);
    Fut<Unit> u = db!updateFile(id,size); await u;
}

Unit continue(File file) {
    FileId id = fst(file);
    FileSize size = fileContent(file);
    Bool he = this.hasFile(id);
    FileSize fsize = 0;
    if (he) {
        Fut<FileContent> s = db!getContent(fst(file)); await s?
        FileContent c = s.get;
        fsize = content(c);
    }
    size = size + fsize;
    Fut<Unit> u = db!updateFile(id,size); await u;
}

Unit processContent(File file) {
    await isAppendCommand(command);
    if (command == SkipFile) {
        skip;
    } else if (command == OverwriteFile) {
        this.overwrite(file);
    } else if (command == ContinueFile) {
        this.continue(file);
    }
}

[Atomic] Unit command(Command c) {
    if (c == StartSnapShot) {
        start = c;
    } else {
        command = c;
    }
}

Unit receiveSchedule(Schedules schedules) {
    this.schedules = schedules;
}