Project Nº: **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 D4.4**

**Auto Configuration and Quality Variability**

Due date of deliverable: (T0+47)

Actual submission date: 1st February 2013

---

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

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

---

<table>
<thead>
<tr>
<th>Dissemination level</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PU  Public</td>
<td>✓</td>
</tr>
<tr>
<td>PP  Restricted to other programme participants (including Commission Services)</td>
<td></td>
</tr>
<tr>
<td>RE  Restricted to a group specified by the consortium (including Commission Services)</td>
<td></td>
</tr>
<tr>
<td>CO  Confidential, only for members of the consortium (including Commission Services)</td>
<td></td>
</tr>
</tbody>
</table>

Final version
Executive Summary:
Auto Configuration and Quality Variability

This document summarizes deliverable D4.4 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.

We report on the results achieved in the Task 4.4 "Auto Configuration and Quality Variability".

ABS Product Configurator: We present a visualizer for the text-based $\mu$TVL language developed for variability modeling in HATS Framework in the Task 1.2 "Full ABS Modeling Framework". We specify a quality variability modeling approach in HATS. We also present a configurator that provides functional auto configuration. In addition to that we present how in the presence of desired quality analyzers for different quality metrics, quality aware configuration can be achieved. We describe the intended tool chain dealing with the configuration, and the current state of the implementation.

Performance-Aware Configuration: We present different scenarios for Performance-aware configuration, and outline a prototype implementation of one of them. Performance-aware configuration aims at providing awareness of the performance throughout the configuration process. The static resource analyzer COSTABS (developed in Task 4.2) is extended to analyze Deltas (corresponding to features) of ABS. The result obtained by the extended COSTABS can be used by the ABS product configurator.

Feature-Level Security Analysis: We describe ways of modeling and analyzing the security variability of computer systems using security features. We present how to break down security requirements into detailed properties, and thereafter specific security mechanisms that realize particular features.

List of Authors
Karina Villela (FRG)
Taslim Arif (FRG)
Elvira Albert (UPM)
Damiano Zanardini (UPM)
Anders Moen Hagalisletto (NR)
Chapter 1

Introduction

Configuring concrete products from a product line infrastructure is the process of resolving the variability captured in the product line, based on a company’s market strategy or requirements from specific customers. Feature models have been the main approach for capturing variability in product lines, so the configuration process usually consists of selecting those features that are applicable to the product and assembling the (partial) product from the product line assets. Several aspects influence the selection of features for a concrete product, such as dependencies and constraints among features, the different stakeholders involved dealing with external and internal features, the desired degree of quality, and cost constraints. As real-world feature models normally have hundreds or even thousands of features, the selection of a correct and appropriate set of features can be a very cumbersome task.

In the HATS project, Task 4.4 (Auto Configuration and Quality Variability) had the aim of developing mechanisms for the automated selection of features and components to obtain configurations that meet the desired end-to-end functionality as well as performance and security requirements in an optimal, or close to optimal fashion. This challenging goal was divided into two sub-tasks: 1) auto-configuration based on end-to-end functionality and 2) the more exploratory sub-task of determining procedures for automatically determining optimal or near-optimal configurations that take into account performance and security.

In terms of auto-configuration based on end-to-end functionality, the HATS framework was already capable of deriving the code for a concrete product from a selection of features specified in the Product Selection Language (PSL), but there was no support for the selection of the appropriate features to be delivered in a concrete product. Our vision was to develop an interactive tool where the person in charge of configuring a certain product would determine only its key features and the tool would automatically select the rest of the features based on the dependencies and constraints defined in the feature model. This configurator was partially developed in Task 1.4 (System Derivation and Code Generation) and it builds upon ChocoJava, a Constraint Satisfaction Problem solver, to check the correctness of the key features selected by the user and find out all possible configurations that include those key features. In Task 4.4 we have extended our configurator to make it possible to use a graphic representation of the feature model as a user interface for the selection of the key features and for the correction of an invalid set of key features interactively, in case the configurator detects the need for it.

However, auto-configuration only based on functionality is not effective in the real-world scenario of feature models with hundreds or thousands of features, because it still results in too many possible configurations. It is essential to use quality concerns and cost constraints to narrow the set of possibilities and make auto-configuration feasible.

In terms of using performance and security to this end, we have envisioned a quality-aware configurator in which the user would also specify quality concerns and cost constraints, and the configurator would find the optimal or close to optimal configurations based on the user’s quality concerns and cost constraints. This vision is based on the assumption that the selection of a feature has an impact in the quality attributes of the final product, as well as the interaction among the selected features. The first step towards this vision was then the characterization of performance and security in terms of sub-characteristics and metrics,
which was reported in Task 5.3 (Evaluation of Modeling), and the study of quality variability modeling
techniques capable of capturing the impact of the selection of a feature or a set of features in the quality of
the final product. Our approach is based on the extended feature model technique due to its learn-ability and
usability. Our configurator was then extended to request and process quality concerns and cost constraints.

The exploratory character of the task is due to the difficulty of annotating a feature model with sound
quality indicators, in our case sound indicators of performance and security. For performance, the vision was
to use the COSTABS analyzer to provide estimates of the resource consumption, where resource consumption
refers to a range of performance metrics in a generic approach w.r.t. the metric used. For security, the vision
was to have reusable features and delta models to implement them, which could be attached to the domain-
specific feature model and deltas models and be used to configure the final product.

Soltani et al. [44] and Siegmund et al. [40] have already identified performance as a business concern
in their configuration approach. However, as far as we know, there is no approach to resource-aware
configuration that use automated resource analysis to assist the configuration process. We present three
scenarios that use resource-consumption estimates as a heuristic for guiding the selection of features. The
COSTABS analyzer, an automated resource analyzer, provides estimates of the resource consumption and
help to select features in a (partially-built) product. In our prototype implementation and case study, we
use the amount of allocated memory and the number of executed instructions as quality metrics to estimate
the degree of performance of a product.

Security has also been considered by Soltani et al. [44] and Siegmund et al. [40]. However, to best of
our knowledge, there is no thorough study on how security can be modeled in a formalism compatible with
feature models. As a first step towards security aware configuration, we looked into ways of modeling and
analyzing the security of systems using security features. Through the identity management case study, we
showed how we can introduce variabilities i.e. features and their corresponding deltas i.e. source code. This
can be used for configuring secure products once we are able to measure the implications of those features
to the overall security.

The evaluation of the proposed approach, in which we are currently work on, is a prerequisite for its
publication in conferences and journals of relevance. Therefore, we have just started to publish it.
Chapter 2

The AS-IS ABS Language

This task "Auto configuration and quality variabilities" is mostly tied with the HATS framework, i.e., the different languages developed during the course of the project span. Those languages are already reported in other deliverables (D1.1, D1.2). For the sake of completeness of this document, we briefly describe the different languages here so that the next chapters can be followed properly. We also present the Fredhopper case study in brief that we will use as an example throughout this document. The Fredhopper case study is reported in D5.3.

2.1 Language Stack

HATS provides a stack of languages for the whole product line development life cycle. The list is shown in Table 2.1.

Table 2.1: Full ABS Language

<table>
<thead>
<tr>
<th>Language</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Micro Textual Variability Language (µTVL)</td>
<td>Feature models, attributes and constraints on them</td>
</tr>
<tr>
<td>Delta Modeling Language (DML)</td>
<td>Modifications to core behavioral modules</td>
</tr>
<tr>
<td>Product Line Configuration Language (CL)</td>
<td>Mappings between features and delta modules, configurations of delta modules with attributes</td>
</tr>
<tr>
<td>Product Selection Language (PSL)</td>
<td>Feature and attributes selections plus product initialization block</td>
</tr>
<tr>
<td>Core ABS</td>
<td>Core behavioral modules (independent of extensions)</td>
</tr>
</tbody>
</table>

Software Product Line Engineering (SPLE) is a software-development paradigm characterized by two main processes: (1) Family Engineering, where product-line assets that are part of the product-line infrastructure are created; and (2) Application Engineering, where these assets are reused to create specific products according to customer requirements[56]. The process of Application Engineering becomes a Product-Derivation process when it is mainly concerned with the configuration of a product and its automatic derivation from the product-line assets. These two phases of the product line can again be split into two dimensions, namely the problem space and the solution space. In the problem space, we extract and describe the requirements and in the solution space we address those requirements with concrete implementation. The languages listed above can be organized using the schema in Figure 2.1.
2.2 SPLE in ABS on a Case Study

For the sake of concreteness, this section describes the artifacts produced in the two processes (family and application engineering) using languages of the HATS\(^1\) language stack and illustrates them with real examples from the Fredhopper case study.

2.2.1 Case Study

The Fredhopper Access Server (FAS) is a distributed and concurrent system that provides search and merchandising services to e-Commerce companies. FAS provides to its clients structured search capabilities within the client data. FAS is structured as a set of live and staging environments. A live environment processes queries from client web applications via web services, with the aim of providing a constant query capacity to client-side web applications. A staging environment receives data updates in XML format, indexes the XML, and distributes the resulting indices across all live environments according to the replication protocol implemented by the Replication System. The replication system consists of a SyncServer at the staging environment, and one SyncClient for each live environment. The SyncServer determines the schedule of replication, as well as its contents, while every SyncClient receives data and configuration updates.

There are several variants of the replication system that were developed as a software product line, which is used as a running example in this paper.

2.2.2 Feature Models

A feature model \[29, 11\] represents a hierarchy of features, which are properties of domain concepts relevant to some domain stakeholder and used to discriminate between concept instances. The hierarchy is organized as a tree: it starts from a root feature, which has a group of sub-features. An “AND”, “OR”, or “XOR” (alternative) relation can hold between features in the same group \[37\]. In an “OR” group, it is also possible to set a minimum and maximum number \((n_1, n_2)\) of features that have to be present in any product. Moreover, a feature can be mandatory, if it is common to all possible instances of the concept, or non-mandatory, if it is marked as optional (opt in Table \[2.2\]) or belongs to an alternative (“XOR”) group. In addition to the hierarchical relations, cross-tree relations control the selection of non-mandatory features:

\(^1\)HATS project website: [http://www.hats-project.eu/](http://www.hats-project.eu/)
If a feature $f_1$ is selected and there is a relation “$f_1$ requires $f_2$”, then $f_2$ has to be selected too. In contrast, if $f_1$ is selected and there is a relation “$f_1$ excludes $f_2$”, then $f_2$ has to be deselected.

<table>
<thead>
<tr>
<th>General Construct</th>
<th>$\mu$TVL Construct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mandatory feature</td>
<td>no opt before feature identifier</td>
</tr>
<tr>
<td>Optional feature</td>
<td>opt before feature identifier</td>
</tr>
<tr>
<td>AND relation</td>
<td>group allof</td>
</tr>
<tr>
<td>OR relation</td>
<td>group $[n_1..*]$, group $[n_1..n_2]$</td>
</tr>
<tr>
<td>XOR relation</td>
<td>group oneof</td>
</tr>
<tr>
<td>Cross-tree relations</td>
<td>require, exclude, ifout and logical operators !,</td>
</tr>
</tbody>
</table>

Figure 2.2: Feature Diagram of the Replication System

The Micro Textual Variability Language ($\mu$TVL) [9] is a text-based feature modeling language that extends a subset of TVL [10]. Table 2.2 shows its main constructs. A feature model is represented textually as a tree of nested features, each with a collection of boolean or integer attributes. Additional cross-tree relations can also be expressed.

Listing 2.1 shows an excerpt of the $\mu$TVL feature model for our case study. The replication system has eight mandatory features (e.g., ReplicationSystem, Installation, and Dir; one is not shown here), plus a number of optional features; finally, Site and Cloud are alternative, as well as Seq and Concur. The selection of Concur requires that Cloud is also selected. Some features like Client have an integer parameter $c$ whose value must be between 1 and 30; moreover, $c$ cannot be greater than 10 whenever Site is selected.

2.2.3 Feature Implementations

Feature implementations specify at the code level how each feature contributes to the behavior of the final product. Several approaches have been used to this end, such as aspect-oriented [30], feature-oriented [7], and delta-oriented programming [39].

The features of the replication system have been implemented using delta-oriented programming. The implementation of a software product line in delta-oriented programming is divided into a core model and a set of delta modules (or deltas). The (possibly empty) core model consists of the classes that implement a complete product of the product line, while deltas describe how to change the core model to obtain new
products. The choice of which deltas to apply is based on the selection of desired features for the final product. The \textit{Delta Modeling Language} (DML) \cite{9} is used to define deltas, and provides constructs for modifying code, such as \texttt{adds}, \texttt{removes} or \texttt{modifies}, which can refer either to classes, interfaces, or methods. Listing \ref{delta_1} shows an excerpt from a delta module of the replication system in which, among other things, a new class \texttt{ReplicationSystem} is added and the class \texttt{ReplicationSystemMain} is modified.

\begin{verbatim}
delta ReplicationSystemDelta;
  adds data JobType = Replication | Boot;
  adds type ClientId = Int;
  adds class ReplicationSystem(
      [Final] Int maxUpdates, ... ,
      SyncServer getSyncServer() { ... }
      SyncClient getSyncClient(ClientId id) { ... }
      Unit run() { ... }
  }
  modifies class ReplicationSystemMain{
    adds Unit run() {
      List<Schedule> schedules = this.getSchedules();
      Set<ClientId> cids = this.getCids();
      Int maxJobs = this.getMaxJobs();
      Int maxUpdates = this.getMaxUpdates();
      new cog ReplicationSystem(
          maxUpdates, schedules, maxJobs, cids); }
  }
\end{verbatim}

Listing 2.2: A delta module of the replication system

### 2.2.4 Linking Feature Models to Feature Implementations

A feature-oriented product-line infrastructure is composed, at least, of a feature model and the code that implements the features in it. The \textit{Product Line Configuration Language} (CL) \cite{9} links feature models specified in \(\mu\)TVL with deltas in order to provide a specification of the variability in a product line. A product-line configuration consists of a set of features assumed to exist, and a set of \textit{delta clauses}. Each delta clause specifies a delta and the conditions for its application, called \textit{application conditions}. These conditions contain (1) propositional formulas over the set of known features and attributes (\texttt{when} clauses), and (2) a partial ordering on deltas (\texttt{after} clauses). When the condition holds for a given product, the delta is said to be \textit{active}. The partial ordering indicates which deltas, when active, should be applied before the considered delta. Listing \ref{cl_1} provides an excerpt of the CL specification of our product line.
### 2.2.5 Product Specification

*Product specifications* are used to define the products of a product line by stating which features should be included in each of them and setting the attributes of features which need it. This provides trace-ability and supports automatic derivation of products from the product-line infrastructure. Listing 2.4 shows two products from the replication system product line using the *Product Selection Language* (PSL) [9].

```plaintext
product DefaultProduct(
    ReplicationSystem, Installation, Resources,
    JobProcessing, ReplicationItem, Dir, Load, Schedule,
    // non-mandatory features
    Site, Seq);

product TwoClients(
    ReplicationSystem, Installation, Resources,
    JobProcessing, ReplicationItem, Dir, Load, Schedule,
    // non-mandatory features
    Site, Seq, File, Journal, ClientNr{c=2,j=5},
    Update{u=3}, Search{d=10,l=20}, Business{d=10,l=20});
```

Listing 2.4: Two products in the product line

### 2.2.6 Product Generation

Given a feature model $FM$, a core model $CM$, a set of deltas $D$, a product-line configuration $LC$, and a product specification $S$, the following steps are systematically performed to build the final software product:

1. Check the product specification $S$ against $FM$ for validity in order to assure that the set of features in $S$ obey the relations provided in $FM$; (2) use $LC$ to activate the deltas from $D$ with valid application conditions according to $S$; (3) apply the active deltas to the core model $CM$ in the prescribed order. Applying all active deltas yields the final product $P$. 

---

```
productline PL;
  features ReplicationSystem, Resources, Client, ClientNr, ... , Data;

delta ReplicationSystemDelta when ReplicationSystem;
delta ResourcesDelta
  after ReplicationSystemDelta when Resources;
delta ClientDelta(Client.c)
  after ResourcesDelta when Client;
delta DataClientNrDelta after ClientNrDelta,DataDelta
  when Data && ClientNr;
```

Listing 2.3: CL specification of the replication system
Chapter 3

Quality Variability Modeling

3.1 Literature Survey

Concrete products of a software product line may vary with regard to quality characteristics, particularly the ones that are relevant to end-users. This chapter is especially concerned with security and performance. According to Exteberria and Sagardui [19], quality variability can be of different types:

- Variability in the relevance of quality characteristics: for example, performance might be considered as important for one product, while security might be regarded as the main issue for another product.
- Variability in the degree of presence of a quality characteristic: for example, one product might require high security, whereas others might require a medium or even low degree of security.
- Indirect variation: functional features may impact differently on different quality characteristics.

Exteberria and Sagardui [19] also identified several requirements for quality variability modeling techniques: (1) allow automatic analysis and reasoning, (2) provide concrete quality characterization, (3) capture optionality, (4) capture the impact of one functional feature on quality, (5) capture the impact of a set of functional features on quality, and (6) capture the inter-dependencies among quality characteristics. A functional feature can impact the quality and this impact can be different in the presence of other functional features. The impact of the two features can be more or even can be less than their individual independent impacts. Feature interaction plays a role here. Requirements number 4 is about individual impact of any functional feature and number 5 is about joint impact of a set of functional features on any quality attribute.

Several quality variability modeling techniques have been presented in the literature. The major quality modeling techniques are [19]:

**Goal-based Model:** It proposes two sub-models: one model represents the functional variability and the other model (soft-goal model) represents the quality characteristics the system should achieve. The correlation between the functional and the soft-goal models captures the impact of functional goals on soft-goals and is usually represented as ++, +, ?, -, --. These qualitative labels are converted into quantitative values, and priorities are also set among the different soft-goals to do some automatic reasoning on the model [20].

**Bayesian Belief Network:** This captures functional variability in a feature model, a Bayesian Belief Network is used to capture the impact of functional variants on quality characteristics. This means directed edges are annotated with a number that represents the domain expert’s belief regarding the impact of functional variants on the quality characteristics [48].

**Definition Hierarchy:** A hierarchical logical AND tree is used to model variability. The topmost nodes are the quality characteristics and the other nodes are design decisions. Any edge in the tree represents that this quality characteristic is partially fulfilled by this design decision. Each node also gets a priority to represent the importance of that node in supporting its parent [31].
Variability Modeling Framework (COVAMOF): This captures functional variability through variation points and quality variability through dependencies. Three types of associations are used to relate dependencies to variation points: (1) abstract, which indicates there is an influence, but there is no information on how, (2) directional, where the influence is not fully known, but there is some information on how the value depends on its binding, (3) logical, where the influence is fully known [43].

Feature-Soft-goal Inter-dependency Graph (F-SIG): It uses a feature model to represent functional variability and soft-goal inter-dependency to represent quality characteristics. Two types of impacts (explicit and implicit) of features on quality characteristics can be modeled and the degree of influence (for example, ++, +, ?, - , --) can be specified [28].

Extended Feature Model: It extends feature models with attributes of features and relationships among those attributes, which are used to model quality concerns [8].

3.2 $\mu$TVL Annotation in ABS

In the previous section we have seen several approaches for modeling quality variabilities. Now, to achieve configuration based on quality variabilities we need to select some mechanism. The mechanism should fulfill all the requirements provided by Exteberria and Sagardui [19]. On the other hand the approach should not cause many changes to the different ABS languages and should be built on top of the current development of HATS framework. Having the requirements from Exteberria and Sagardui [19] we choose to use the Extended Feature Model approach. We found that it is easily integratable to the HATS framework while achieving most of the quality variability modeling requirements. Below we will show how the requirements can be fulfilled by the Extended Feature Model approach, i.e., annotating the $\mu$TVL with quality concerns.

```plaintext
extension Business {
  Int im_memoryConsumption in {0,4};
  ifout: im_memoryConsumption == 0;
  im_memoryConsumption == 4;
  Int im_responseTime in {0,3};
  ifout: im_responseTime == 0;
  im_responseTime == 3;
}
```

Listing 3.1: Quality Annotation

The $\mu$TVL annotation approach can fulfill most of the requirements. It can directly be used for automatic analysis and reasoning. It can nicely represent the impact of an individual functional feature on quality. It can also be used to represent the impact of a set of features. Optionality is already supported in $\mu$TVL as built-in.

In quality characterization one high level quality is broken down into several low level concrete quality attributes and it goes on down the hierarchy (For example, performance can be characterized by breaking it into several lower level quality attributes, response time, memory consumption, throughput and so on. These lower level attributes can again be characterized). A corresponding effect can be achieved in a $\mu$TVL annotation by composing higher level attributes out of the lower level attributes and it can also go up to any depth of the hierarchy. Inter-dependencies of quality attributes are also representable by $\mu$TVL annotation. Moreover they can always be represented as standalone rules outside $\mu$TVL if the inter-dependencies do not depend on the functional features. Group Impact, quality characterization and inter-dependencies of quality attributes - these can have scalability problems while modeling for very big product line using annotation but this problem cannot be solved by using other modeling techniques also.

Most of the requirements can be fulfilled by the approach nicely. The $\mu$TVL annotation approach is easy to understand, easy to apply which is very important for its adoption in the industry. An example quality annotation is shown in Listing 3.1. It shows the performance annotation of the feature Business by
two attributes \textit{im\_memoryConsumption} and \textit{im\_responseTime} with their respective values if the feature is finally selected in the configuration.
Chapter 4

Quality-Aware Configuration

4.1 Literature Survey

For configuring a product, developers must find a selection of features from the feature model, that satisfies the requirements of a concrete product and adhere to the rules of the feature model [47]. This configuration process involves reasoning over a complex set of constraints to meet a final goal. Interactive configuration support uses constraints to propagate configuration choices made by the user [14], whereas automatic configuration support provides a set of configurations that satisfy the rules of the feature model and the user’s requirements and constraints.

Hubaux [26] classifies the techniques for feature-based configuration into five groups.

Unguided configuration The feature model is used without following a well-defined approach. Usually the features are merely selected in a top-down fashion.

Staged configuration The product configuration is performed in stages, each stage eliminating configuration choices in the feature model. Different dimensions can be associated with the configuration stages, for instance the times of the product life or the roles of the parties performing the configuration [12].

Multi-level staged configuration This extends the staged configuration by adding configuration levels, where each level is represented by a feature model linked to the other level’s feature model through inter-level links [13]. This provides modularization and therefore enhances scalability.

Probabilistic configuration This augments traditional feature models with conditional probabilities regarding the selection of features and legal joint probability distributions (JPD) [14]. JPD assign a probability to each possible configuration of a software product line.

Dynamic configuration This considers the evolution of the product configuration at run-time [32]. In contrast to static configuration, both context evolution and stakeholder actions entail automated re-configurations of the running product.

Furthermore, White et al. [47] present a formal model of multi-step configuration that allows making continuous progress on the development of a product according to a change constraint, such as the maximum development budget per year. The output is a path of configurations (from the starting configuration to the desired final configuration) that meet the multi-step constraints. Their technique, which translates the feature selection problem into constraint satisfaction problem (CSP), can also be used for single-step or staged configuration.

The same research group [46] provides a polynomial-time approximation algorithm, called Filtered Cartesian Flattening, for selecting a highly optimal set of features that adheres to a set of system resource constraints. In Filtered Cartesian Flattening, the problem of feature selection with resource constraints is represented as an approximate Multi-dimensional Multi-choice Knapsack Problem (MMKP), which can be
Table 4.1: Support for Feature Selection

<table>
<thead>
<tr>
<th>Main Characteristic</th>
<th>Support Type</th>
<th>NF Concerns</th>
<th>Underlying Technology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-level staged</td>
<td>Interactive</td>
<td>Security</td>
<td>Specialized FMs</td>
</tr>
<tr>
<td>Probabilistic</td>
<td>Interactive</td>
<td>No</td>
<td>Conditional probabilities and legal Joint Probability Distributions</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Automatic</td>
<td>No</td>
<td>Binding analysis and reconfiguration strategy</td>
</tr>
<tr>
<td>Multi-step</td>
<td>Automatic</td>
<td>Cost</td>
<td>Constraint Satisfaction Problem</td>
</tr>
<tr>
<td>Polynomial-time</td>
<td>Automatic</td>
<td>Yes</td>
<td>Multi-dimensional Multi-choice Knapsack Problem</td>
</tr>
<tr>
<td>Business concern</td>
<td>Automatic</td>
<td>Yes</td>
<td>Hierarchical Task Network</td>
</tr>
<tr>
<td>annotation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multi-view</td>
<td>Interactive</td>
<td>No</td>
<td>Workflow management tool</td>
</tr>
<tr>
<td>Performance measurement</td>
<td>Automatic</td>
<td>Resource</td>
<td>SAT Solvers</td>
</tr>
</tbody>
</table>

Soltani et al. [44] propose the extension of feature models with annotations reflecting different business concerns of the stakeholders. Examples of business concerns are “high security”, “high customer satisfaction”, “high performance” and “potential for high international sales”. They make use of Hierarchical Task Network (HTN) planning to select the maximal sets of features which satisfy the stakeholders’ requirements and business concerns, and have a total cost lower than or equal to the expected cost.

While these three related works address automatic configuration, Czarnecki et al. [14] support interactive configuration based on probabilistic feature models. In their technique, probabilistic propositional formulas are used to express both hard and soft constraints. Such formulas have well-defined semantics denoting a set of joint probability distribution over features. The authors also show how probabilistic feature models can be mined from a given set of product configurations. On the one hand, the main advantage of this technique is the capability of expressing preference among the legal configurations. On the other hand, either a representative sample set of configurations to apply feature model mining or a domain expert is required.

Abbasi et al. [1] also support interactive configuration. Taking into consideration that the configuration process involves several stakeholders and might not be organized in a linear manner, they extend an existing feature-based configurator with multi-view support and integrate it with a work-flow management tool. Views let stakeholders concentrate on those parts of the feature model that are relevant to them and the work-flow management tool drives the configuration of these views.

Table 4.1 summarizes a literature review on product configuration support. Concerning Support Type, the selection of features in our resource-aware configurator is mainly automatized. However, the user has a central role providing not only information on concerns (e.g., memory consumption) and constraints (e.g., that the cost has to be lower than x), but also on the key features of the product. If those features do not infringe upon any rule, the configurator will not propose deselecting them in any of the provided solutions. On the one hand, this information is essential for the efficiency and effectiveness of a product configurator. On the other hand, it provides an interesting balance between automatic and interactive configurations.

Several efforts have been made around research on product configuration. Quality-aware configurations
require modeling variability in quality attributes. Sinnema et al. [43] described COVAMOF, a modeling technique for variability. Etxeberria et al. [18] presented a survey on existing approaches including COVAMOF for specifying variability in quality attributes, the requirements for a quality variability modeling technique and comparison of the techniques. Hubaux et al. [27] presented the multidimensional separation of concerns in feature-based configuration, i.e., generating concern-specific configuration views. They also presented in [25] the feature configuration workflow, with scheduling parts of the feature diagram as part of the configuration process. Tun et al. [45] showed a systematic approach to relate requirements to features.

As regards Non-Functional Concerns, several approaches take into consideration cost constraints, but only few of them consider quality concerns such as performance and security. Some notable exceptions are [42, 40, 41]. The most related work to ours is [40] which is also concerned with deriving optimal products with respect to non-functional requirements by showing customers which features must be selected. Similar to our feature-level analysis, they propose an approach to predict product’s non-functional properties by aggregating the influence of each selected feature on a non-functional property to predict a product’s properties. They also try to generate and measure a small set of products and, by comparing measurements, they approximate each feature’s influence on the non-functional property in question. A fundamental difference is that our approach to estimate the performance of features is based on a formal method (i.e., static resource analysis), while [40] performs measurements. It is well-known that the use of formal methods has some advantages. In our case, we rely on a static analysis which infers approximations which are safe for any possible input data value. In addition to the advantage of having a result which is valid for any input data, an important consequence of this different choice is that we can analyze partial products and focus on the performance behavior of fragments of the product (e.g., the footprint) while, because they perform measurements, they need to analyze it globally. This gives us further flexibility. The interaction of features is not taken account in our analysis explicitly, though there is some implicit interaction which is taken into account with the mandatory features (since they are in the minimal product). The techniques proposed in [40] to take into account such interaction can be also used in our approach without requiring any conceptual change to our scenarios.

### 4.2 HATS Approach for Quality-Aware Configuration

After considering the current configuration mechanisms in the literature, we finalized our approach. Configuration of a product is done at different stages of the product life cycle. Our approach in this task is to support development-time quality-aware configuration and it is part of the overall end-to-end product derivation. Therefore we first present the work-flow of end-to-end ABS product derivation. Then we describe the configuration activities, actors and artifacts in detail. As a part of the configuration task, we integrated FeatureIDE for visualizing \( \mu \)TVL. The integration steps are briefly described at the end.

#### 4.2.1 Work-flow of End-to-end Product Derivation

In this section we describe the end-to-end ABS product derivation process. We show different steps towards product derivation and the artifacts generated in each step in Figure 4.1. Below we briefly present all the steps. This helps us pointing out to which step our auto-configuration is contributing.

**Elicitation of Customer Requirements** The derivation of the product starts with the requirements of the customer or stakeholder. There can be more than one stakeholder involved for any product. A systematic elicitation process is required to handle conflicting, ambiguous and incomplete requirements. Business goals, functional and quality requirements are elicited, so that they can be mapped to features in the product line feature model.

**Complete Selection of Product Features** In this step, the features obtained from the requirements elicited in the previous step are made complete. This is necessary because features have interdependencies among themselves that can result in inclusion of new features, or exclusion of previously
selected features. Our functional auto-configuration helps in this stage, to find out products with different characteristics. For example, it can find the minimal product with respect to the number of features respecting the user requirements. There can be several minimal products. It is at the end the user’s decision to choose the suitable product out of the recommended ones. The configurator generates the PSL (product specification language in ABS) of this product. Similarly it can generate products with other functional characteristics.

In case of quality aware requirements the auto-configuration helps to find a configuration that respects the quality constraints (e.g., memory consumption should not cross some boundary and so on) and optimizes (maximizes or minimizes) some objective function. The objective function is formulated with desired quality attributes and with their relative priorities. For example, we consider the simplest case - a user wants to have the thinnest product (i.e., minimal product in terms of resource consumption) of the product line as she has memory constraints in her device. The configurator minimizes the memory consumption metrics and finds suitable products from the product line. For our case study, one of most memory efficient configurations suggested by the configurator is shown in Listing 4.1. The user will see the visualizations of the suitable products and can select the best one for her. Once selected, PSL will be generated for the next step.

**Generation of ABS Code** Once the PSL is generated by the auto-configurator, the ABSFrontend (plugin for ABS language) can generate ABS code out of it. Core ABS is modified with the deltas corresponding to the features specified in the PSL, in a specific order described in CL to generate the ABS code of the configured product.

**Generation of Back-end Code** Several back-end code generators have been developed in the HATS project. From the flattened ABS code, it is possible to generate Java, Scala or Maude code by designated back-end code generators.

```plaintext
product MemoryEfficientProduct(
    ReplicationSystem, Installation, Cloud,
    Resources, JobProcessing, Concur, ReplicationItem,
    Dir, Load, Schedule);
```

Listing 4.1: PSL for Memory Efficient Product
The configuration we presented above is development-time configuration. The product can be further configured during deployment-time. Features cannot be deployed independently in the system. At the end it is the deploy-able units (i.e., components mostly) which will run. HATS already developed a component model in Task 3.1 and it can be used for run-time or deployment-time configuration.

The product can be further configured or adapted at run-time. The product quality metrics (performance, security, etc.) can be monitored during run-time and based on the requirements it can be adapted. In Task 3.5 HATS is doing the monitoring of the ABS program and finding ways to dynamically select new configurations which better approximate the given end-to-end objectives.

Figure 4.2: The ABS Product Configuration Work-flow

4.2.2 ABS Product Configurator

In this section we describe the product configuration activities, the organizational roles and their contribution for supporting product configuration in ABS. Any system is built up by several stakeholders. To understand clearly the auto-configuration process, we describe the organizational setting of the ABS product configuration work-flow in Figure 4.2. The main activities are described below in sequence.

1. Creating $\mu$TVL, Core ABS, Deltas and CL

Before the configuration starts, the product line needs to be developed. The domain engineers and developers generate the product line using the various languages ($\mu$TVL, Core ABS, Deltas and CL) provided by HATS. One such product line is shown in the case study above in Section 2.2. In Figure 4.2 the artifact generated is named ABS Model which is the product line with common base source code and functional variabilities. It does not contain any quality variability yet unless some quality variability is mentioned as a functional feature in the $\mu$TVL.
2. Annotating $\mu\text{TVL}$ with quality and cost

In the second step, the product line feature model is annotated with cost and quality attributes. Our current approach is to annotate features with their impact in cost, performance and security based on static analyses of features in isolation. The domain expert annotates the $\mu\text{TVL}$ with cost, adding attributes to features. She sets the cost based on her expertise and also following company strategy. In parallel other automatic analyzers (Performance Analyzer) or different quality experts (Security Expert) annotate the features with designated quality attributes. All these annotations generate extensions of $\mu\text{TVL}$ as artifacts shown in Figure 4.3. Analysis done only statically and in isolation cannot generate a realistic estimation of quality rating. Several mechanisms can be used to improve the accuracy of annotations based on static analyses of features in isolation, such as (partial) product analysis OR run-time monitoring. We intend to adopt product analysis to come up with a more accurate quality estimation of features, but run-time monitoring goes beyond the scope of this task.

Figure 4.3: Quality Annotation

3. Integrating the annotations and ABS model

In the third step we need to integrate the two artifacts generated above i.e. the ABS Model and different $\mu\text{TVL}$ extensions for quality and cost. Integration is necessary because the annotations done by different parties can be heterogeneous in nature. For example, some analyzers may annotate a parent feature considering the child feature and some may not. If the parent feature is not annotated at all or annotated without considering the child features, then we need some mechanism to propagate the values of the children to the parent. The integration process identifies appropriate ways of propagating the values and calculating homogeneous values of parent features for different quality attributes.

Figure 4.3 shows different steps of annotation. The configuration preprocessor combines the annotations $a_f$ provided in the previous step. Listing 4.2 shows an excerpt with the definition of preprocessing options for memory consumption. It states that the annotation of a parent-level feature is done using the compositionOperator, which is “+” in this case.

The resulting annotation will be a formula, where $im\_memoryConsumption$ is the attribute for internal metrics of memory consumption—we distinguish between external and internal metrics as we are just
considering the static measures here i.e., the internal metrics:

\[ \sum_{i=0}^{n} \text{childFeature}_i.im\textunderscore memoryConsumption } \]

which represents the sum of the memory consumption of all children present in the configuration; if \( \text{childFeature}_i \) is not selected in the configuration, \( \text{childFeature}_i.im\textunderscore memoryConsumption = 0 \). The appropriate composition function can be easily defined for each specific metric and application. Empirical study and domain knowledge are required to find an appropriate composition function.

4. Specifying quality concerns and functional features

Steps 1 to 3 are part of the domain engineering process. Once the product line is developed with the functional and quality variabilities, it is time for deriving the specific products. After eliciting functional and non-functional requirements for the product, the user of the ABS Product Configurator translates those requirements into functional features, quality concerns and/or constraints, and/or cost constraints. Quality concerns are the quality attributes of interest and the priority among those. There are three modalities of product configuration depending on the information provided by the user of the ABS Product Configurator.

**Basic Configuration** The user specifies those features of which she is sure about inclusion. The set of required features might not be valid. The configurator suggests a minimally distant valid product in case of invalid input from the user. The elicitation screen of the ABS Product Configurator is shown in Figure 4.4.

**Cost-aware Configuration** The user specifies the required features and the cost limit of the desired product. The configurator finds out the maximum product within the cost limit respecting the user recommended features.

**Quality-aware Configuration** In this case, again the user specifies the required features and cost. In addition to that the user specifies the percentage of importance of the quality concern (for example performance 60% and security 40%). The user can even prioritize among the different metrics of any higher level quality attribute. For example, in our case study we are using memory consumption and response time as performance metrics. The user can set memory consumption 25% and response time 75%. The configurator forms an objective function out of those and finds those products which are within the cost limit and optimizes the function. In this example case, the objective function could look like this

\[ 0.6(0.25 \times \text{memoryConsumption} + 0.75 \times \text{responseTime}) + 0.4 \times \text{security} \]

Figure 4.5 shows one example of specifying the functional features by the user. The filled square boxes are the features selected by the tool because either the feature is mandatory or the feature is required by some other features due to inter-dependencies. The user selected features are shown as ticked in the diagram. The configurator shows the validity (real-time configuration status) and number of possible open products at real-time to guide on specifying the functional features.
Figure 4.4: User concern elicitation for configuration

Figure 4.5: Real-time guidance in ABS Product Configurator
5. Configuring ABS product

Once the objective function is determined, the product can be configured automatically by using any constraint solver. If there are only functional requirements, the configurator builds the minimal products with respect to changes of user recommendations in terms of features. In case of quality-aware configuration, the configurator generates the products that optimize the objective function the most.

In order to find valid solutions for the configuration problem, our configurator uses the Java-based CSP solver called Choco Java, which converts the feature model and the objective function into a Constraint Satisfaction Problem (CSP) and asks the CSP solver to solve it. For visualizing the results and eliciting the user’s quality and functional requirements, the open source tool FeatureIDE is used.

4.2.3 Feature Model Visualization

To support the configuration process, some work in this task was dedicated to provide a graphic visualization of the feature model, originally captured in $\mu$TVL, so that one can easily select the key features of a to-be product.

Rather than developing the visualizer from scratch we looked for a suitable open source feature model visualizer. Among all the possible options we selected the FeatureIDE tool because of its modularity in development and extend-ability. The ABSFrontend is the tool, developed in the HATS project and we use it for the purpose of parsing and compiling $\mu$TVL. As both ABSFrontend and FeatureIDE are Java based Eclipse plugins, the integration becomes much easier. This is also one fundamental reason for selecting FeatureIDE.

Despite the integratability of FeatureIDE, some integration work was required. As both were Java based Eclipse plugins, we just had to modify the manifest file (MANIFEST.MF) appropriately so that the two plugins can communicate. The second issue was the different format of feature model in the two different programs. For FeatureIDE the format is XML based which is shown in Listing 4.3 and for the ABSFrontend the format is $\mu$TVL which is shown in Listing 2.1. After comparing the two formats, we identified the conversion algorithm for converting $\mu$TVL to FeatureIDE-Format. In Table 4.2 we have shown how some of the conversions are done.

Table 4.2: $\mu$TVL to FeatureIDE conversion

<table>
<thead>
<tr>
<th>$\mu$TVL</th>
<th>FeatureIDE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cardinality</td>
<td></td>
</tr>
<tr>
<td>oneof</td>
<td>&lt;alt&gt; tag</td>
</tr>
<tr>
<td>allof</td>
<td>&lt;and&gt; tag</td>
</tr>
<tr>
<td>group</td>
<td>&lt;or&gt; tag</td>
</tr>
<tr>
<td>Optionality of</td>
<td>attribute mandatory=true/false</td>
</tr>
<tr>
<td>feature</td>
<td></td>
</tr>
<tr>
<td>Constraints (requires)</td>
<td>&lt;imp&gt; tag</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

The group cardinality cannot be represented in the FeatureIDE data model. Nevertheless the solver in the back-end uses the $\mu$TVL to reason about feature model. So even if it is not represented in the visualization, the end configuration is always valid.

The transformed $\mu$TVL of our case study to the XML format of FeatureIDE is shown in Listing 4.3. Once we have the transformation FeatureIDE can visualize it in different styles. One of them is shown in Figure 4.6.

4.2.4 Benefit and Threat Analysis

The usage of the ABS Product Configurator offers clear benefits:

**Visualization** The ABS Product Configurator greatly simplifies the specification of the features desired in a product through its integration with FeatureIDE. The user can better understand the overall

\[1\text{FeatureIDE website: } \text{http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/}\]
Listing 4.3: Feature model of the case study in the FeatureIDE data format
commonality and variability of the product line graphically and it can interact with the graphic representation to indicate the product requirements.

**Real-time Guidance** The tool also shows in real-time the validity of the current selections of features and the number of possible configurations. Therefore, it can guide in selecting appropriate features before completing the selection of features towards the semi-automatized product configuration.

**Basic Configuration** The tool finds the product configuration according to the functional requirements. All the constraints, dependencies and other relationships are resolved automatically by the configurator.

**Heuristics based optimal quality product at development-time** Heuristics are used to find out a product with the desired degree of quality. There is no guarantee that the recommended configurations will be the best in terms of quality, but it is better to have some information about quality than having no information at all.

**Integrated in the (ABS) development environment** The tool is developed in eclipse and integrated with ABSFrontend (the ABS language compiler) and FeatureIDE (feature model visualizer). All those integrated tools are readily available within the Eclipse development environment.

Quality-aware configuration is still a research problem. We have addressed some of the problems but still a number of threats are there. Some of them are below:

**Accuracy of Static Analysis** The quality annotation of features are done statically without considering the run-time behavior which leads to imprecise estimation of quality.

**Accuracy of Feature Interaction** Feature interactions are not considered during estimation, even though $\mu$TVL annotations can be used to model interactions. Feature interaction can play a crucial role for

![Figure 4.6: $\mu$TVL visualized in FeatureIDE](image)
quality deviation from independent isolated analysis. The domain expertise about the feature interaction is not incorporated into the approach.

**Accuracy of Composition Operators** The approach requires specific composition operators for every specific metric. Empirical studies must be performed to find out a suitable operator for a specific metric.

**Scalability of Feature Annotation** Providing $\mu$TVL annotations might be cumbersome for big feature models. However, a tool is in charge of providing performance annotations. For quality annotations to be provided by a domain expert, the ABS Product Configurator can be extended or another tool can be provided to ease the quality annotation of features. In addition, the feasibility of developing further static and/or run-time analyzers should be investigated.

The ideas developed and implemented in the ABS Product Configurator can be easily used with feature models represented in other languages. As it is developed in a modular way, we need to replace parser with the new parser, analyzers with the new ones and then the configurator will work the same way it is working for ABS.
This chapter is concerned with the configuration problem from the perspective of product performance, as this is a frequently desired quality for software products. Resource consumption metrics such as the amount of allocated memory and the number of executed statements are the product quality metrics we use herein to estimate the performance of a product. Note that these metrics correspond to the resource consumption metrics that can be inferred by the resource analyzer for ABS, described in Deliverable 4.2 [16]. We present three scenarios for resource-aware configuration of software product lines. The common idea in all scenarios is the use of resource-consumption estimates as a heuristic for guiding the selection of features. The crux is the use of an automated resource analyzer (e.g., any of [21, 24, 3] can be used) that provides estimates of the resource consumption of a product and helps to select features in a partially-built product. The estimates are used to guide the configuration process towards more efficient products, while keeping intact all the key features (i.e., the essential features from the users point of view) of the product, and adhering to economic cost constraints and to the dependencies and constraints specified in the feature model.

The scenarios for resource-aware configuration differ in the way the interaction between the configurator and the resource analyzer is designed: (1) In Product-Level Analysis, resource analysis is run a posteriori for selected product configurations; (2) in Partial-Product-Level Analysis, the resource analyzer is invoked to analyze partial products obtained after the selection of features along the configuration process; (3) in Feature-Level Analysis, the resource analyzer is run a priori to estimate the impact that each feature in isolation may have on the resource consumption of products. Soltani et al. [44] have already identified performance as a business concern in their configuration approach. However, as far as we know, a full-fledged product configuration approach that shows how performance efficiency can be estimated for product variants, like ours, and how to use this estimate in the product configuration process is missing.

5.1 Preliminaries: Product Configuration

It is not our purpose to formalize the product configuration process, but just to fix the basic notions needed to accurately describe the scenarios for resource-aware configuration in SPLE.

5.1.1 Definition

Given a product line $PL$ with a feature model $FM$, product configuration is the process of selecting those features that comply with $FM$ and fulfill the stakeholders’ requirements, which results in the product specification $S$.

A set of key features $K = k_1, \ldots, k_s$ might be optionally provided. Key features are those features whose selection the user of the configurator is confident about; therefore, they should be part of all solutions provided by the configurator, as long as they respect the feature model. The solution provided by the
configurator is a set of candidate configurations $Conf \equiv \{C_1 \ldots C_n\}$, where each $C_i$ is defined as a set of features $\{f_1, \ldots, f_m\}$ (providing initial values for attributes). All configurations include the set of mandatory features and the set of key features $k_1, \ldots, k_s$ selected by the user, and must be valid w.r.t. the feature model.

**Example 1** Two candidate configurations for our case study are those in Listing 2.4 which, in the above notation, are: $C_1 = \{\text{ReplicationSystem}, \text{Installation}, \text{Resources}, \text{JobProcessing}, \text{ReplicationItem}, \text{Dir}, \text{Load}, \text{Schedule}, \text{Site}, \text{Seq}\}$ and $C_2 = \{\text{ReplicationSystem}, \text{Installation}, \text{Resources}, \text{JobProcessing}, \text{ReplicationItem}, \text{Dir}, \text{Load}, \text{Schedule}, \text{Site}, \text{Seq}, \text{File}, \text{Journal}, \text{ClientNr}(c=2,j=5), \text{Update}(u=3), \text{Search}(d=10,l=20), \text{Business}(d=10,l=20)\}$. Observe that in $C_2$ we provide initial values for the attributes of certain features.

For each candidate configuration $C_i$, the unique associated product $P_i$ denoted as $P(C_i)$ in the following can be automatically derived from the product-line infrastructure (Sec. 2.2.6) by taking $S$ to be equal to $C_i$.

**Example 2** Consider for instance the configuration $C_1$ above, we generate a product from it by following the CL specification in Listing 2.3 and applying the deltas in Listing 2.2 to the core module. The result is a program written in the ABS language.

5.1.2 Configuration Trees

In order to define the resource-aware configuration scenarios, it is useful to view configuration as the construction of a decision tree, referred to as the configuration tree, whose nodes are partial configurations $C \equiv \{f_1, \ldots, f_n\}$, i.e., the set of features selected so far while traversing the tree. A branch $C \rightsquigarrow C'$ in the tree adds one or more features to $C$ such that the child $C'$ is $C \cup \{f_{n+1}, \ldots, f_m\}$. The reason why several features might be added in one step are the constraints in the feature model; e.g., it might be the case that one cannot select $f_{n+1}$ without selecting $f_{n+2}, \ldots, f_m$. Nodes can have several children; e.g., both $C'$ and $C''$ may be children of $C$ when we choose among optional or alternative features; this may happen if $C'$ has an optional feature that $C''$ does not add, or $C'$ adds the feature $f'$ to $C$ while $C''$ adds the feature $f''$, and $f'$ and $f''$ are alternative.

**Example 3** For instance, the configuration tree for the case study has a branch $\text{ReplicationSystem} \rightsquigarrow \text{Installation} \rightsquigarrow \text{Resources} \rightsquigarrow \text{JobProcessing} \rightsquigarrow \text{ReplicationItem} \rightsquigarrow \text{Dir} \rightsquigarrow \text{Load} \rightsquigarrow \text{Schedule} \rightsquigarrow \text{Site}$ (we assume that each node contains the new feature plus all the above features in the branch). From the last node, we will have as children configurations $C_1$ and $C_2$ of Example 1. For $C_1$, we add one further step $\rightsquigarrow \text{Seq}$ and the resulting configuration becomes a leaf of the tree. For $C_2$, we have to add all remaining features.

Given the product line infrastructure $PL$ and the set of key features $K$, we rely on a generic configurator invoked as $\text{Configurator}(PL, K)$ that computes a decision tree $\tau$ as described above. In the following, $Conf$ denotes the leaves of the tree (i.e., the set of valid configurations), and $PartConf$ denotes the intermediate nodes (i.e., the set of partial configurations).

**Example 4** The configuration tree for our case study has 768 leaves, so one could generate 768 candidate products. Two of those leaves are $C_1$ and $C_2$.

5.2 Static Resource Analysis

In our resource-aware configuration approach, resource-consumption estimates are computed by an off-the-shelf resource analyzer and used to select the most promising configuration candidate(s). We rely on the generic resource-analysis tool $\text{Analyzer}$, which, given a fragment of code $P$, an entry method $m_0$, and a resource metric of interest $R$, is invoked as $\text{Analyzer}(P,m_0,R)$ and analyzes the resource consumption of $m_0$, as well as of those $n$ methods transitively invoked from it, w.r.t. $R$. As a result, it returns a set of $n+1$ pairs $(m_i, u_i)$, where $m_i$ is a method name and $u_i$ is an upper bound to its resource consumption.
Example 5 Consider a fragment of method `transferItems(Set<File> files)` shown in Listing 5.2 which is part of our case study. This method has been pointed out in [15] as a hot spot in the execution of the case study. This method traverses the set of files received as input parameter and on each element of the set performs a number of processing operations. It is not relevant for our purposes to understand the behaviour of the method which includes also primitives for concurrency (like future variables, await operations and asynchronous calls) which are completely outside the focus of this work. The important point to notice is the external `while` loop on the parameter `files` and the use of auxiliary operations inside the loop.

Let us analyze its resource consumption using the COSTABS tool [3], an implementation of a resource analyzer for ABS programs. We select the cost model that counts number of steps, since this is the metric which is most related to execution time. COSTABS returns the following (asymptotic) upper bound: 
\[ \text{nat}(\text{files})^2 \cdot \text{nat}(\text{max(rid)}) + \text{nat}(\text{files})^3 \]

which is a polynomial of degree 3 on the size of the argument `files` and the class field `rid`. Function `nat` is defined as: 
\[ \text{def Int nat(Int a) = if a > 0 then a else 0; } \]

and used by the analyzer in order to avoid negative evaluations of the upper bound expression.

```
Unit transferItems(Set<File> files) {
    while (hasNext(files)) {
        Pair<Set<File>,File> nf = next(files);
        files = fst(nf);
        File file = snd(nf);
        FileSize tsize = fileContent(file);
        Fut<Unit> rp = job!command(AppendSearchFile); await rp?;
        Fut<Maybe<FileSize>> fs = job!processFile(fst(file));
        await fs?;
        Maybe<FileSize> content = fs.get;
        FileSize size = 0;
        if (isJust(content)) {
            size = fromJust(content);
        }
        if (size > tsize) {
            rp = job!command(OverwriteFile);
            await rp?;
            rp = job!processContent(file);
            await rp?;
        } else {
            // omitted a fragment of the method
        }
    }
```

Listing 5.1: Excerpt of method `transferItems` of case study

The most relevant points of static resource analysis relevant to our study are: (1) Upper bounds are cost expressions that might include polynomial, logarithmic, exponential subexpressions (and any combination thereof). (2) For simplicity, in the example we have shown the result of COSTABS in asymptotic form (or big O notation), i.e., we have removed all the constants as the expression was rather large. The result provided by the analyzer is a precise upper bound that includes also constants. (3) The upper bound is given in terms of the input parameters (e.g., `files`) and of the class fields (e.g., `rid`). This is the case for the upper bound in most methods unless they have constant cost. (4) In order to compare the cost of two fragments of code, we need to be able to compare upper bound expressions of the above form, this problem has been studied in [4]. We thus assume the existence of an operator “<” which allows us to compare two upper bounds. (5) The upper bound is ensured to be correct, i.e., it is a safe approximation of the worst-case resource consumption of running the program for any possible input data.

5.3 Scenarios for Resource-Aware Configuration

In our resource-aware configuration approach, resource-consumption estimates are computed by the off-the-shelf resource analyzer COSTABS and used to select the most promising configuration candidate(s).

29
Without loss of generality, we assume the existence of an entry method that corresponds to the main method from which the analysis starts. If there is no entry method, then the same approach can be applied to any other method or set of methods of interest.

In the remainder of the section, we discuss three ways to carry out the interaction between Analyzer and Configurator, and point out advantages and drawbacks of each of them.

5.3.1 Product-Level Analysis

In the first scenario, Analyzer obtains the resource estimates directly from the final products. The process consists of three steps:

1. Given the product-line infrastructure $PL$ and the set of key features $K$, we first obtain the set of final (valid) configurations $Conf$ (Sec. 5.1);
2. for each $C_i \in Conf$, we generate a product $P_i \equiv P(C_i)$, and analyze it by running $Analyzer(P_i, entry, R)$ where $R$ is the resource metric of interest;
3. the best candidate is the product $P$ whose resource consumption $u$, corresponding to the pair $(entry, u)$, is the minimum among all products.

**Advantages** The main advantage of this approach is that it can be potentially implemented using existing technology since (1) there are tools that behave like Configurator; (2) there exist product generators for valid configurations; (3) a static analyzer Analyzer for final products can be used; (4) we have techniques for comparing upper bounds and choosing the minimum [4].

This scenario produces the most accurate results, though soundness of the selected candidate is not ensured. Note that the resource analysis guarantees that the obtained upper bounds $u_i$ are sound, i.e., $u_i$ correctly over-approximates the resource consumption of the product $P_i$ for any possible input data. However, it is not guaranteed that $P_m$ is the best candidate, since the static analyzer performs several approximations in order to obtain a sound result, and the loss of information in the analysis of one product can be larger than the loss in the analysis of another. One can easily provide examples for which this leads to selecting a “best” candidate that is actually not the best. Thus, the analysis is used as a heuristic for guiding the selection rather than as a guarantee. Still, this scenario should produce very accurate results.

**Disadvantages** The main drawback of this approach is its inefficiency. For a configuration tree with $k$ leaves, we need to invoke the product generator and the analyzer $k$ times. Each analysis is performed on a full product, which can be a large and complex piece of software. The results from analyzing one product cannot be reused when analyzing the next one, as there is no knowledge on which parts of the product are the same as those of previous products. Unfortunately, static analysis tools for a property as complex as resource usage are not yet developed at an industrial level. While they can handle medium-sized programs, their application to commercial products is still a research challenge. In conclusion, we argue that, although this scenario is feasible in theory, it is beyond the current state of the practice.

**Example 6** In the case study, this approach involves generating 768 different products, analyzing each of them, and choosing the one that shows the best performance behavior. Most products are, in terms of lines of code, even bigger than the code implementing the whole product line, thus making the analysis of each single product very expensive. In general, the number of products to be generated, together with their size, can make this task prohibitive.

5.3.2 Partial-Product-Level Analysis

It is quite natural to think of an interleaved cooperation between the resource analyzer and the configurator in such a way that Configurator invokes Analyzer along the configuration process to be aware of the
resource consumption associated with partial (i.e., in the process of being computed) configurations. This approach requires being able to estimate the resource consumption of partial products associated with the partial configurations built throughout the configuration process. The resource consumption will allow the configurator to decide if it is worth continuing the construction of such a configuration, or if it is better to reject that path of the configuration tree. Given the product line infrastructure PL and the set of key features $K$, we proceed to construct the configuration tree $\tau$ (Sec. 5.1). Partial-product-level analysis consists of the following steps:

1. For each partial configuration $C \in \text{PartConf}$ of $\tau$, we generate a partial product $P(C)$;

2. we incrementally analyze the partial product executing $\text{Analyzer}(P(C), entry, R)$, reusing the results inferred for the partial products of ancestor nodes of the configuration tree;

3. we decide whether the estimated resource $(entry, u)$ for $P(C)$ is “acceptable”; otherwise, we prune this branch of $\tau$ (i.e., the children of $C$ will not be considered).

In order to decide whether the resource consumption is acceptable, the user can set up a threshold (or maximal amount of resources) $Limit$ before starting the configuration process. Thus, in step 3, we simply check if $u > Limit$ to decide if the branch must be pruned. Another possibility is to compare the resource consumption of all candidates by keeping the results for the best product constructed so far (namely, $u_{\text{min}}$), and prune a branch if a partial product already exceeds $u_{\text{min}}$. This scenario corresponds to a branch-and-bound optimization in search.

Example 7 Let us assume that the user imposes as threshold $Limit = \text{nat}(\text{fileset})^2$, i.e., the set of files that are to be transferred can be traversed at most a quadratic number of times. During the construction of the configuration tree, as soon as we choose a feature that triggers a delta that includes method $\text{transferItems}$ (see Example 5), the threshold provided by the user is exceeded since the resource consumption of previously computed $\text{nat(fileset)}^2 * \text{nat(max(rdir))} + \text{nat(fileset)}^3 > \text{nat(fileset)}^2$. Thus, the current branch of the configuration tree is pruned and this feature is not selected.

Advantages The main advantage of this approach comes from pruning the configuration tree and avoiding building products whose resource consumption exceeds the provided threshold. Furthermore, as (partial) products are built incrementally, incremental resource analysis [5] can be used, so that information gathered in the analysis of previous partial products is reused whenever it is valid. In a different context and for a different language, it is proven in [5] that incremental resource analysis can save up to 50% of the time when compared with non-incremental program analysis.

Disadvantages A problem with this approach is that it is not feasible using current technology. First, generating partial products is not always feasible: a partial product is, in general, incorrect code since relevant parts of the code are missing. The product generator aims at building a final product; as soon as it finds a method that is not defined, the whole process fails. As a consequence, most nodes in PartConf cannot be evaluated since there is no tool for generating the product. Even if there were a product generator for partial products, this approach has some efficiency and soundness issues.

Concerning efficiency, Analyzer has to be invoked, in the worst case, on all intermediate nodes PartConf and leaves Conf of $\tau$. Consider a complete binary tree with $n$ leaves: the analyzer will have to be invoked up to $m = 1 + 2 + 4 + \ldots + n$ times, where each summand corresponds to the number of nodes in the corresponding level of the tree. Thus, for our case study $m > 768$. In contrast, in product-level analysis (Sec. 5.3.1) it is invoked only $n$ times, which is smaller than (although comparable to) $m$. The analyses can be performed incrementally and, according to Albert et al. [5], each partial analysis can achieve a gain of around 50%.

\footnote{We do not have the concrete value of $m$ for our case study because we use the tool that computes the configuration tree as a black box in our implementation.}
w.r.t. analyzing the full product every time. Therefore, according to the current state of the practice, the efficiency approach is comparable to product-level analysis.

On the other hand, soundness is potentially lost whenever a branch in the selection tree is pruned. This is because choosing the local best solution does not necessarily lead to the global best solution, since a feature added in a later selection might affect the resource consumption significantly. Let us illustrate this issue with a very simple example.

**Example 8** Consider the partial product resulting from applying the delta $\text{foo1}$ to `entry` in Listing 5.2. If we now measure the number of statements executed by `entry`, the resource analysis infers a linear cost, namely, $n_0 \cdot x_0$ statements, where $n_0$ and $x_0$ refer to the initial values of $n$ and $x$ respectively. Observe that the while loop in the `entry` method is executed $n - x$ times, explaining the linear cost obtained. On the other hand, consider the partial product that results from applying `foo2` to the previously constructed (and analyzed) product: The resource consumption of `entry` is $\text{nat}(n_0 - x_0) \cdot \text{nat}(x_0)$, which is quadratic. This is because we invoke method `incr` inside the while loop of `entry` and each of the invocations executes the while loop of the new implementation of `entry`. The latter while loop performs $x$ iterations, which multiplied by the number of iterations of the while loop of `entry` leads to the quadratic cost.

```c
int entry(int x, int n) { 
  while (x < n) x = incr(x); return x + n; } 

delta foo1{ modifies int incr(int x) { return x++; } }

delta foo2{ modifies int incr(int x) { 
  int x0 = x; while (x > 0) { f++; x--; }; return x0++; } }
```

**Listing 5.2: Unsoundness of Partial-Product Analysis**

The above example reveals that it is not sound to prune $\tau$, as a future modification might affect the resource consumption of a previously analyzed (partial) product. The consumption can be increased (as in the example) but also reduced (e.g., if the deltas are applied in the inverse order). Therefore, sound results can only be obtained by building full configuration trees and analyzing the resulting complete products. We conclude that this scenario can be of interest only if: (1) we have tools to build partial products; (2) incremental analysis performs much more efficiently than whole-program analysis; and (3) we have a language in which the new increments (deltas) applied to a partial product cannot interfere with the existing code in the partial product. In the absence of such conditions, product-level analysis seems to be a better choice.

### 5.3.3 Feature-Level Analysis

With the aim of devising a more practical scenario, we consider a third possibility: assessing the resource consumption due to each feature $f$ in the product line by generating and analyzing a product containing only $f$ plus the minimal number of features needed to get a valid configuration. We denote by $P_0$ a product with the minimal number of features needed to get a valid configuration. Then, the minimal product for $f$ (denoted $P_{\text{min}}(f)$) is obtained by applying feature $f$ to $P_0$. We note that, due to requirements among features, adding $f$ may result in adding other features, as the following definition states.

**Definition 5.3.1 (minimal product)** A minimal product for an optional feature $f$ consists of:

(a) all mandatory features;

(b) the feature $f$ under consideration;

(c) for every group of features to be included, $n_1$ features chosen randomly (preserving the validity of the configuration), where $n_1$ is the minimum number of features to be selected;

---

2 It should be noted that $P_0$ can be non-unique because we may want to randomly re-select every time alternative features.
(d) any feature that is required by this selection based on cross-tree relations.

Minimal products for features in alternative groups are obtained in a similar manner.

As discussed in Section 2.2.2, non-mandatory features are (1) those marked with an opt modifier, or (2) those appearing in an alternative group (group oneof construct). Concerning disjunctive groups (group [n1...n2] or group [n1...*]), every product must include n1 to n2 features belonging to the same group, where the * symbol stands for n2 when the group has n2 features; the construct group oneof is a special case of disjunctive group where n1 = n2 = 1.

The intuitive idea of the feature-level scenario is that the resource consumption due to feature f, denoted \( U_f \), can be approximated as \( U_f = R_f - R_0 \), where \( R_f \) is the resource consumption obtained by analyzing \( P_{min}(f) \) and \( R_0 \) is the resource consumption obtained by analyzing \( P_0 \) for the considered entry. The generation and analysis of \( P_{min}(f) \) needs to be performed only for non-mandatory features, since the analysis of \( P_{min}(f) \) aims at deciding whether selecting \( f \) is good from the point of view of performance. This makes no sense for mandatory features because they will be selected in any case. Given a product line with non-mandatory features \( f_1...f_n \), the feature-level analysis is performed as follows:

1. generate the minimal product \( P_0 \), invoke once \( Analyzer(P_0, entry, R) \), and get the resource estimate (entry, \( R_0 \));
2. generate the minimal products \( P_{min}(f_i) \) for \( 1 \leq i \leq n \);
3. analyze all products by running \( Analyzer(P_{min}(f_i), entry, R) \), get resource estimates (entry, \( u_i \)) and obtain the resource consumption due to feature \( i \) as \( U_i = u_i - R_0 \);
4. given a candidate configuration \( C\{f_{i1},...,f_{is}\} \in Conf \), its resource consumption is \( U \equiv U_{i1} \oplus ... \oplus U_{is} \) (using some specific operator \( \oplus \)), where \( U_{ij} \) is the resource consumption obtained for \( f_{ij} \) in step 3.

Note that if the minimal product \( P_0 \) does not contain an entry method, then \( R_0 = 0 \). Similarly, if the selection of one feature does not modify the cost, then \( u_i \) will be the same as \( R_0 \) and the resource consumption associated to the feature will be 0. In the above steps, we use a generic operator \( \oplus \) to combine the resource consumption of a set of features. One could think of using + and simply accumulate the resource consumption contributed by all features (as we do in the implementation), or using max to be able to discard products including features with very high resource consumption. Given \( k \) candidate configurations \( C_1...C_k \), the best candidate is the one whose resource consumption is minimal.

**Advantages** This methodology has a number of practical advantages with respect to the others: (1) the number of minimal products to be analyzed is much smaller than the number of products (in the case study, 13 instead of 768), thus making this approach much more feasible than the first two ones; (2) the whole analysis process can take place before the configuration begins and thus there is no need to design a complex interaction between Analyzer and Configurator, this has an important advantage over the partial-product analysis.

**Disadvantages** Clearly, this approach is not sound because it does not consider the interaction of different non-mandatory features in a product (i.e., products that are not minimal in the sense described above). However, as we have noted before, the other two scenarios are not sound either, though the loss of soundness in the first scenario should be rather negligible. In order to consider the interaction among features, one could adopt intermediate solutions (e.g., those proposed in [10] for a different measurement-based approach) and generate minimal products also for combinations of interacting features, instead of considering them in isolation.

**Example 9** In the ReplicationSystem example, there are nine optional features and two alternative (group oneof) groups, each containing two alternative features; thus, 13 minimal products will be generated and
analyzed. This is a great improvement over the 768 products to be analyzed in the first approach. However, the feature-level approach does not allow appreciating how different features behave when coexisting in a product: for instance, Search and Business can both be selected in a given product, but no minimal product contains both.

All in all, what the feature-level methodology can provide is a heuristic that describes the performance behavior of each feature and helps the configuration process in the challenging task of choosing one configuration which in addition to be valid is efficient w.r.t. some performance metric.

### 5.4 A Feature-Level Resource-Aware Configurator

This section describes our implementation of the feature-level scenario and applies it to the case study. We have developed a prototype resource-aware product configurator that implements the feature-level scenario described in Section 5.3.3. Figure 5.1 provides an overview of the workflow in our tool.

The first phase consists in generating performance annotations by carrying out the following steps:

1. Given a product-line infrastructure $PL$ and a selection of key features, the component $MinimalProductGenerator$ generates the minimal products for each non-mandatory feature;

2. the off-the-shelf resource analyzer $COSTABS$ is used to analyze the minimal products; and

3. the component $PerformanceAnnotator$ takes the cost expression returned by $COSTABS$ and uses it to annotate the feature model with performance annotations.

The final output of this phase is an $PartiallyAnnotatedFeatureModel$.

The second phase is the $product$ configuration itself, in which:
(2a) the PartiallyAnnotatedFeatureModel is preprocessed to derive annotations for upper-level features from
the annotations provided by the PerformanceAnnotator;

(2b) the component Visualization&UserInteraction asks the user to provide his/her quality constraints and
concerns;

(2c) the configurator suggests a small set of valid product configurations that best fit the objective function
representing the user’s input; and

(2d) the user selects one of those configurations.

The final output is a PSL specification from which the product can be generated.

We describe below the main decisions made during the implementation of the above components.

5.4.1 Performance Annotation

The implementation of the performance annotator comprises steps (1a), (1b) and (1c) described in Sec-
tion 5.3.3. The main decisions that have been taken in the implementation are: (1) considering the resource
consumption of those methods potentially modified by the selection of a feature (named footprint of the
feature), instead of looking at the entry method only, as described below; and (2) providing the configura-
tor with performance annotations that are natural numbers instead of the cost expressions inferred by the
resource analysis.

Generation of minimal products First, a minimal product \( P_{\text{min}}(f) \) is computed and generated for
every non-mandatory feature \( f \). Computing a minimal product involves reading the \( \mu \)TVL definition of
the feature tree and identifying the set of non-mandatory features. Afterwards, the features that will be
included in a “minimal product” \( P_{\text{min}}(f) \) are selected following the definition of minimal product given in
Section 5.3.3. Given the feature specification for \( P_{\text{min}}(f) \), the actual product can be generated by existing
tools available in the project.

Example 10 In the case study, the minimal product for \( P_{\text{min}}(\text{Cloud}) \) for Cloud includes all mandatory
features, Cloud itself, and either Seq or Concur (chosen randomly). This configuration is a minimal valid
configuration containing Cloud. The minimal product for Concur involves selecting all mandatory features,
Concur itself, and Cloud, which is required by Concur. Finally, \( P_{\text{min}}(\text{Client}) \) consists of all mandatory
features, Client, one between Site and Cloud, and one between Seq or Concur (when Concur is selected,
Cloud has also to be chosen). In all these cases, no optional features are selected unless they are strictly
required.

In this case study, the total size of the code (including deltas, the product-line declaration, etc.) is around
4,000 lines. All the products (i.e., the code produced by the product generator tool) generated in the tests,
either minimal or not, have roughly the same size. Only considering the core-ABS code (i.e., excluding
delta code, product declarations, etc.) gives a size of around 2,000 lines for all generated products (this
will be discussed later in Table 5.1), including the final best product selected by the configurator (2,025 lines
of code). This means that generating minimal products does not need to give any significant advantage in
terms of code size with respect to generating “complete” products. As pointed out above, the advantage is
that minimal products are only a small part of the set of valid products.

Static Analysis Once minimal products have been generated, COSTABS analyzes each of them. The
resource of interest is a parameter of the analyzer, which, in the current implementation of our solution, is
either response time\(^3\) or memory consumption.

\(^3\)It must be clear that response time does not refer to actual execution time in milliseconds; rather, it represents the number
of executed statements.
When analyzing a piece of code, a typical choice in static analysis, is to analyze the entry method (e.g., the main method in a Java program). This usually involves analyzing all or most of the code, since such a method will probably invoke many other methods during its execution.

On the other hand, for a given minimal product $P_{\text{min}}(f)$, we are interested in reasoning only on the resource consumption of methods that are specific to the feature $f$ under consideration. In general, the selection of $f$ only affects a limited portion of the code, and the analysis benefits from identifying this portion. The definition of the product-line (Section 2.2.4) specifies which deltas must or could be active when $f$ is selected. Conservatively, we take all deltas whose associated delta clause has an application condition where $f$ occurs; this is an approximation since, it can be the case that a delta is only active when both $f_1$ and $f_2$ are selected, so that the selection of $f_1$ does not imply the activation of the delta. For instance, the delta clause $\text{delta DataClientNrDelta}$ in Listing 2.3 states that this delta is active and must be applied after the deltas $\text{ClientNrDelta}$ and $\text{DataDelta}$ (if they are also active) whenever the expression $\text{Data && ClientNr}$ holds. Conservatively, this delta is considered when analyzing the minimal product for $\text{Data}$, even though the effective activation of this delta also requires the selection of $\text{ClientNr}$.

Once the set of deltas that can be active when $f$ is selected has been obtained, an inspection of the delta declarations collects all the methods added or modified by such deltas. This set is called the footprint of $f$, and can be computed statically. For every method in $\text{Footprint}(f)$, COSTABS is run and a cost expression is obtained. Thus, the output of this step is a set of upper bounds $(m_i, u_i)$ for each $m_i \in \text{Footprint}(f)$.

Example 11 The number of methods defined in the minimal product $P_{\text{min}}(\text{File})$ is 138. However, the footprint of the feature $\text{File}$ only contains nine methods, which are those modified or created by delta $\text{FileDelta}$ which

- adds class $\text{ReplicationFilePattern}$, containing six methods;
- modifies one method in class $\text{ReplicationSnapshotImpl}$;
- modifies two methods in class $\text{TesterImpl}$.

Performance annotations While the cost expressions $u_i$ that COSTABS outputs in the previous step provide a precise upper bound of the resource consumption, manipulating them in the subsequent configuration phase is rather complex. In a product-line with a large number of products and core assets, managing such expressions grows increasingly difficult, and the results become hard to interpret, especially from the point of view of the user. For instance, deciding if a cost expression $e$ is smaller than another one $e'$ requires the use of specific techniques [4]. The result of the comparison is often not simply a Boolean answer, but rather constraints on the variables of $e$ and $e'$ under which the comparison can be proved.

As a practical solution, we consider transforming cost expressions into performance annotations, which indicate the resource consumption of executing a method by means of an integer number: the higher the number, the higher the consumption. In other words, $(m_i, u_i)$ is transformed into $(m_i, a_i)$, where $a_i$ is the annotation. In order to carry out this mapping, we first transform $u_i$ into asymptotic form (big O notation). This transformation can always be applied, and can be done locally and efficiently [2]. The next step is to map the asymptotic bound to a performance annotation. This mapping can be done using different heuristics, and the effectiveness of the transformation should be assessed experimentally. In practice, the heuristic used should be derived by observation using typical (case study-specific) input sizes. For example, the current implementation makes the following choice:

0 : constant cost
100 : logarithmic, sublinear or linear cost
200 : polynomial cost (up to degree 3)
300 : high-degree polynomial or exponential cost
400 : unknown (the analyzer could not get an upper bound)

4Methods that are removed by a delta are not included because there is no code to be analyzed anymore.
Example 12 Consider the cost expression that has been obtained from the analysis of method `transferItems` in Example 5 which is already in asymptotic form. According to the above choice, the expression (polynomial of degree four) will be mapped into the annotation 300.

The next step is to combine (by $\oplus$, see Section 5.3.3) per-method annotations $a_i$ into a per-feature annotation $a_f$ describing the performance of $f$. A reasonable heuristic is required here, whose discussion is beyond the scope of this deliverable. Simple heuristic functions such as addition, average or max can be easily implemented and evaluated experimentally. In the implementation, we are currently using the average of the performance levels of all methods in the footprint of the feature.

Example 13 We have analyzed all methods in the footprint of `File` (Example 11) w.r.t. the cost metrics “response time”. Once COSTABS generates performance annotations for all of them, the overall annotation $a_{File}$ has been obtained as the average and the result is 133. This number is obtained by analyzing all the nine methods in the footprint: 5 of them have constant cost (performance level 0), 2 of them give a low-degree polynomial upper bound (performance level 200) while the last two cannot be analyzed (performance level 400). The final performance level is given to Configurator as an extension using the following syntax:

```cpp
extension File {
    Int im_responseTime in {0,133};
    ifout: im_responseTime == 0;
    im_responseTime == 133;
}
```

Listing 5.3: Performance Annotation by COSTABS

The first instruction declares `im_responseTime` as an integer that can take values 0 or 133. The second line states that, if the feature is not selected (ifout), then it must take value 0. Finally, according to the last line the performance annotation is 133 whenever the feature is selected. Three lines are output for every each cost model considered in the analysis.

5.4.2 Product Configurator

The configuration preprocessor combines the annotations $a_f$ provided in the previous phase, as described in Section 4.2.2. Listing 4.2 showed an excerpt with the definition of preprocessing options for memory consumption that states that the annotation of a higher-level feature is done using the `compositionOperator`, which is “+” in this case.

After obtaining the fully-annotated feature model, any objective function can be defined on the root feature’s attributes. The quality concerns provided by the user are translated into an appropriate objective function. In the objective function, the priorities of different metrics can be reflected. For example, `im_memoryConsumption` can be more important than `im_responseTime` to the user. The user can directly quantify how important they are using several standard approaches for eliciting prioritization. The Analytical Hierarchical Process (AHP) [38] is one popular approach for finding priorities from relative importance of different criteria.

In addition to the objective function, the user can also set quality constraints by providing a threshold that cannot be exceeded. Quality constraints can be related to one quality metric or it can be related to multiple metrics. Once the objective function and the quality constraints are elicited, the configurator finds suitable product configurations for the user.

Example 14 In our example feature model, we have the feature `Client` and the user wants his product to be launched in a very thin client with respect to memory. She can set a constraint on that feature specifying how much memory consumption she can tolerate. The constraint is specified as follows:

```
Client.im_memoryConsumption <= 2
```
5.4.3 Preliminary Experiments on the Case Study

We have implemented the generator of performance annotations as an extension of the COSTABS analyzer: it takes as input the ABS files containing all the code, and outputs a $\mu$TVL file with the performance annotations described in Example 13. In the case study, the code is contained in two files: ReplicationSystem.abs contains the module declaration, the core code, the delta definitions, the product-line configuration, and some product specifications; Features.abs contains the feature model. The total size of both files is 3548 lines of code.

The resource under consideration is response time. As mentioned before, in the feature model there are 8 mandatory features and 13 optional or alternative features; consequently, 13 minimal products have to be generated. For each of them, the footprint is computed, and the core part of COSTABS (i.e., the static analyzer properly said) is called once for each method in the footprint. Experiments on performance annotation have been carried out on a MacBook Pro with a 2.4 GHz Intel Core i5 processor and 4Gb of memory, running Mac OS 10.7.5. The execution has been repeated 5 times, and reported times (expressed in milliseconds) are computed as the average of all the executions. Table 5.1 summarizes the experiments:

<table>
<thead>
<tr>
<th>Feature</th>
<th>$t(P_{\text{min}}(f))$</th>
<th>$F(P_{\text{min}}(f))$</th>
<th>$M(P_{\text{min}}(f))$</th>
<th>$L(P_{\text{min}}(f))$</th>
<th>$FP(f)$</th>
<th>$t(a_f)$</th>
<th>$a_f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>1643</td>
<td>11</td>
<td>132</td>
<td>2030</td>
<td>3</td>
<td>1783</td>
<td>0</td>
</tr>
<tr>
<td>Server</td>
<td>1655</td>
<td>11</td>
<td>132</td>
<td>1977</td>
<td>1</td>
<td>1694</td>
<td>0</td>
</tr>
<tr>
<td>File</td>
<td>1669</td>
<td>11</td>
<td>140</td>
<td>2053</td>
<td>9</td>
<td>15675</td>
<td>133</td>
</tr>
<tr>
<td>Journal</td>
<td>1647</td>
<td>11</td>
<td>139</td>
<td>2069</td>
<td>7</td>
<td>4309</td>
<td>86</td>
</tr>
<tr>
<td>Update</td>
<td>1659</td>
<td>11</td>
<td>131</td>
<td>1972</td>
<td>1</td>
<td>1686</td>
<td>0</td>
</tr>
<tr>
<td>ClientNr</td>
<td>1645</td>
<td>11</td>
<td>131</td>
<td>2024</td>
<td>2</td>
<td>1746</td>
<td>0</td>
</tr>
<tr>
<td>Search</td>
<td>1652</td>
<td>11</td>
<td>132</td>
<td>2030</td>
<td>2</td>
<td>2042</td>
<td>200</td>
</tr>
<tr>
<td>Business</td>
<td>1641</td>
<td>12</td>
<td>142</td>
<td>2063</td>
<td>2</td>
<td>2035</td>
<td>200</td>
</tr>
<tr>
<td>Data</td>
<td>1641</td>
<td>13</td>
<td>150</td>
<td>2160</td>
<td>3</td>
<td>2112</td>
<td>133</td>
</tr>
<tr>
<td>Seq</td>
<td>1633</td>
<td>10</td>
<td>130</td>
<td>1972</td>
<td>21</td>
<td>9952</td>
<td>76</td>
</tr>
<tr>
<td>Concur</td>
<td>1658</td>
<td>10</td>
<td>133</td>
<td>2025</td>
<td>24</td>
<td>14874</td>
<td>163</td>
</tr>
<tr>
<td>Site</td>
<td>1657</td>
<td>10</td>
<td>132</td>
<td>1972</td>
<td>0</td>
<td>1658</td>
<td>0</td>
</tr>
<tr>
<td>Cloud</td>
<td>1645</td>
<td>10</td>
<td>132</td>
<td>1972</td>
<td>0</td>
<td>1645</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.1: Experimental evaluation on the case study

Let us draw some conclusions from the table. We can observe that all minimal products are very similar in size, since most code is shared by all of them, and that the time needed to generate them (most of which is taken by the execution of the HATS tools for product generation) is also similar. The most significant difference lies in the size of their footprint: this is consistent with the intuition that the difference between two features is related to the portion of code they directly affect. Note also that some features have no methods in their footprint; this means that, actually, they are “dummy” features which do not modify the code, so that they are given a default performance annotation 0. As regards the efficiency of the analysis process, there is a common pre-processing task which is the same for every feature, and takes around 1350 milliseconds. Column “$t(a_f)$” shows the total time taken by COSTABS; this time is obtained by putting a 10-seconds timeout on each call to the analyzer (such timeout is only reached once when analyzing a method in Footprint(File)). It must be pointed out that all the minimal products have been analyzed separately, while
the implementation could have been optimized by reusing several parts of the computation; for example, most of the work done by COSTABS on a method can be reused for other methods in the same footprint, and part of the work on a product can be reused for other products. To improve the efficiency following these and other directions is part of the future work.

By using the performance annotations shown in the table, the resource-aware configurator suggested the following configuration:

\{
ReplicationSystem, Installation, Resources, JobProcessing, ReplicationItem, Dir, Load, Schedule, Site, Seq
\}

In absence of performance annotations, the configurator had suggested another configuration:

\{
ReplicationSystem, Installation, Resources, JobProcessing, ReplicationItem, Dir, Load, Schedule, Site, Concur
\}

whose overall performance annotation (according to the preprocessing for configuration described in Listing 4.2 which simply sums the performance annotations of all features) is worse than the one chosen by considering performance issues. As expected, using the annotations, we obtained a configuration that has a better overall performance level.

We argue that our experiments, even if still at a very preliminary stage, constitute a proof of concept that resource-aware configuration is feasible. However, it still remains to see how close our performance annotations are to the actual resource consumption of the products. This requires defining and evaluating heuristics for the different operators we have used in the feature-level scenario. In the experiments, we used the average performance annotation for all the methods in the footprint as the final annotation of a feature, and the addition of the performance annotations of all features as the performance annotation of a product. Obviously, other choices could have been taken. Future work includes proposing new heuristics that allow us to have annotations which are closer to the actual resource consumption, and undertaking a thorough experimental evaluation.
In this chapter we look into ways of modeling and analysing the security of computer systems using security features. A security feature is a specific implementable function in a system that supports some part of the system’s security policy. The security policy includes constraints on functions and flow among the components in the system: Hence it covers security properties like confidentiality, integrity, authenticity, access control, etc. Thus the goal of the current chapter is to break down security requirements into detailed properties, and thereafter specific security mechanisms that realise particular features.

6.1 Variability in Security

Automated techniques have shown to be successful in verifying small-sized protocols, this research was initiated by Gavin Lowe’s groundbreaking work from 1996 [33]. Yet although research has progressed, current tools cannot efficiently handle protocols larger than TLS (10-12 messages). Security critical enterprise systems typically contain more than 100 messages. The number of attacks on real systems have increased dramatically the last years [17] and the cost of developing secure systems increases. The methods for verifying systems can only handle isolated components or small sub-systems. Unfortunately research is lagging behind current industry needs for security design and assurance.

There are two main challenges in assuring that enterprise systems are secure: configuration and system complexity. In most real-world systems there are many options to configure in submodules - the consequences of choosing one particular set of options might have undesirable effects on the overall security and the system design is typically very complex.

Security indicators and metrics have been used to understand enterprise systems and get a high level view on the security. Yet there are some pitfalls - in general too little is known about the security of the existing configurations of real systems:

- Only small fragments of distributed systems are well-understood with respect to the security: typically only security barriers and isolated modules and sub-protocols have been tested separately - and little is known about the security of the entire system, including the complete flow of messages. In contrast, performance-tests can be performed on entire enterprise systems - not only to submodules.

- Security is highly non-compositional: if we have two secure modules $A$ and $B$, then their composition $A \oplus B$ will not necessarily be secure. For instance, if $A$ and $B$ both provide confidentiality for a group of agents $G$ with respect to some content $F$, this does not necessarily imply confidentiality in case of $A \oplus B$. Composition $\oplus$ can be interpreted either as sequential composition or by interleaving messages or even layering messages within channels. Independent of how we interpret compositionality the core security properties may be compromised. If keys or key material are used or reused in careless manners over different sub-modules, then confidentiality of the overall system can be broken, although each sub-module in isolation can be proven to be secure.
• Security systems contain “spaghetti of dependencies”: Security protocols are constructed by making dependencies between data units, keys, protocols and data-structures.

• No consensus (either industry or academics) on the meaning of main security properties: For most practical purposes the main security properties like confidentiality, authenticity and integrity is specified as general requirements, and thus not suitable in order to generate concrete code.

The initial vision using the feature-based process, was the following: First a feature tree of independent security features is constructed. The features are then used as a guideline for a moderator. The moderator of the feature-based approach picks out desirable security features with associated source code, in order to construct the final product. But dependency of security concepts, specialized (context dependent) interpretation of main security properties, and the non-compositionality of security make this hard to convey for real systems. Automated configuration requires that the main goals and mechanisms are independent. Particular features are chosen from a given predefined general library of features. The challenge with a feature-based approach to security is two-fold: First, for some security features it is hard to generate the resulting code, since the feature should be implemented in several distinct places in the code-base. A general security property like confidentiality of a certain entity can be protected by several distinct mechanisms throughout the message flow. Second, if a security feature is chosen and a particular mechanisms realize this feature, careless use elsewhere in the code of parameters from the particular feature can make undermine the intended security effect of the feature.

Hence it does not seem to be practically feasible to cover security by choosing among a set of security features from a feature tree. Instead we make a pragmatic turn, and take one step back and ask: Is it possible to build a secure system where several of the building blocks can be varied? We try to give a somewhat realistic scenario that could be a design of a system used currently. Therefore we used the latest version of TLS (TLS version 1.2), and deploy three main processes (protocols), Registration, Authentication and Transaction. The Registration protocol involves submission of credentials for authentication. The authentication protocol includes two basic elements, the submission of credentials from the client to the identity management server and the hand-over of credentials to the service provider.

6.2 Method

In Task T4.1 a method for automated construction of protocol implementations was developed. The process is described in Figure 6.1. Initially a formal specification of the protocol is written in PROSA \cite{23}. Then this condensed formal specification is automatically refined into an executable specification. This specification contains local pre- and post-conditions for each message transmission in addition to the messages themselves. The entire refined specification is then split into separate views - one for each agent. The view for the agent $A$ contains all assumptions and message-events that concern $A$. Then each assumption- and transmission-event gives rise to ABS code fragments. For each snippet of refined PROSA specifications, a piece of executable ABS code is generated. The output of this process is a set of COG’s, where each COG corresponds to an agent role in the protocol.

Although we have fixed the underlying connection protocol (TLS), the main purpose of the application and some specific communication design, there are several options that can be varied. One such example is the notion of authentication factors. Authentication factors are elements that connect the user’s identity to a particular service. There are three authentication factors, personal - something the user knows, technical - something the user has and finally human something the user is (typically called biometry). Personal authentication factors include password and username. Technical factors might include deviceID, nonces, timestamps, IP addresses, location, and smartcard(s). Biometrical authentication factors may include recognition of voice, fingerprints, retina scan, walking pattern and similar. In Figure 6.1, we show how the process of automated construction of ABS implementations from specifications can be expanded to include authentication factors. The authentication factors are injected into the specification, and are propagated into the refined specification and then finally to the generated ABS code.
6.3 Identity Management Case Study

Identity management systems are used to manage identities, authenticate clients, authorise them to services, set roles and privileges. In Europe a great majority of the citizens use more than one electronic ID (eID) for performing transactions towards banks, private e-commerce or public sector. Hence application of eID has become a necessary part of everyday life in modern society - including banking and tax payment.

Figure 6.2 shows the three main protocols of the identity management service, Client Registration, Authentication and Service Transaction. In Client Registration a particular client A registers main identity attributes and authentication credentials as for instance dynamic or static passwords. The authentication factors to be used later are submitted to the Identity Provider (IDP). The authentication protocol has two main goals: The IDP authenticates one client A, already registered with the IDP, in order to hand over a session request for a particular service to a Service Provider (SP).

In the initial system there are four components, a client (A), an identity provider (IDP), a certification authority (CA), and a service provider (SP), as depicted in Figure 6.3.

The certification authority distributes certificates to the client and the identity provider. Registration is performed by the client initiating a TLS session, and then transmitting credentials for further use in the authentication process. Authentication starts with a TLS session succeeded by a client hand-over to the service provider.

6.3.1 Security channels

In any security application the transport of messages is wrapped inside security channels. An example of a typical way to organize layers of security properties is given in Figure 6.4. There are two main communicating parties Alice and Bob, and a component PROXY that forwards messages. The channels are organized as follows: At the innermost layer there are signatures of important data or credentials (marked purple). The application logic (marked yellow), includes these signatures and extra messages that describe the main security functionality of the application. The application channel is typically wrapped inside a channel providing confidentiality (marked blue). At the outermost level (marked green) there is a communication-enabling channel for secure transport of each message. This is the most common layering of security structures. But there are other ways to organize the layers. The most frequently discussed alternative
consists in encrypting the content and then signing the result.[1]

There are several potential variations on the basic architecture: the authentication factors - whether to use one, two or three factor authentication, the authentication strength of TLS in registration or authentication (zero, one, two-way authentication). The final Service Transaction protocol might be encapsulated inside a secure channel that gives integrity, confidentiality between the client and the service provider.

6.3.2 System specification

In this section we present the main architecture, the details of the message interaction and the information transmitted, using a standard Alice-Bob notation. The complete specification in PROSA is described in Section A.1, while the automatically generated ABS code of the Identity Management Casestudy is described in Section A.4.

The system proposed contains three main stages, registration, authentication and transaction, and one initial stage where certificates are distributed. The different processes in the application are described in Figure 6.5. TLS is used to set up communication between the client and the identity provider (IDP). After running TLS, the TLS-session-keys are used to protect the payload of the application.

The system is specified formally using the PROSA language [23]: Each sub-process is defined as a

---

protocol with a protocol header containing the protocol name, session instance, roles and then a sequence of message transmissions. For convenience we use a neutral Alice-Bob notation in the presentation of the system. Each message transmission is written

\[ A \rightarrow B : F \]

meaning that agent (or component) \( A \) sends to agent (or component) \( B \) the payload \( F \). The payload might contain textstrings written \( W^{\text{TEXT}} \), nonces \( N_A \), agent names \( A, B \), concatenation of elements ‘,’ (comma) or composite constructors for hashing \( \text{Hash}[F] \) and encryption \( E[\text{key} : F] \).

**TLS**

“The primary goal of the TLS protocol is to provide privacy and data integrity between two communicating applications. (…)\(^2\) TLS is designed as a “lazy sheriff”, nonces, credentials and certificates are built up during the protocol run. Although an attacker can intervene at early stages in the protocol, it has been shown to be hard to succeed at the end (first work on the topic was [34], using inductive reasoning - theorem proving, later, reachability methods were used in the AVISPA project [6]). TLS accumulates credentials transmitted during the run and at the end, the entire payload history is wrapped inside the final two messages. The most stripped-down version of TLS is without certificates, shown below. It might be called “zero-way” TLS, because there is no distribution nor checking of certificates involved, thus both parties do not have any guarantee of the authenticity of the other party.

\[
\begin{align*}
(tls_1^0) \ A & \rightarrow S : A, N_A, N_{A}^{\text{SID}}, W_{A}^{\text{CryptoOffer}} \\
(tls_2^0) \ S & \rightarrow A : N_S, N_{A}^{\text{SID}}, W_{S}^{\text{CryptoOffer}} \\
\text{authentication (signed certificate)} & \text{ optional messages}
\end{align*}
\]

The core specification of TLS is given in Figure 6.6. In the protocol the client \( A \) is connecting to a server \( S \). In the first message (\( tls_1^0 \)), the client \( A \) sends her name, a client nonce, a session ID and a crypto-offer to the server \( S \). The crypto-offer includes the preferred cryptographical algorithms used later in the session. The server then creates a nonce and returns it together with a crypto counter-offer (what algorithms the server can use in the interaction) and the session ID in message (\( tls_2^0 \)). Next, in message (\( tls_3^0 \)), the client encrypts the PMS (Pre Master Secret) with the public key of the server. Then at the end the structure Finished is sent from \( A \) to \( S \) encrypted with ClientKey and then decrypted and sent back using ServerKey. The session keys are defined by the following:

\[
\text{clientKey} = \text{key}(A, N_A, N_S, \text{MasterSecret})
\]

\[
\text{serverKey} = \text{key}(S, N_A, N_S, \text{MasterSecret})
\]

where the Master-secret is constructed by concatenating the preMasterSecret value (PMS), together with the nonces for the client \( C \) and the server \( S \), and then hashing the result:

\[
\text{MasterSecret} = \text{Hash}[N_A^{\text{PMS}}, W^{\text{MasterSecret}}, N_A, N_S]
\]

The finished structure is defined by collecting all data elements from the initial message interactions and concatenating it with \( \text{MasterSecret} \), and then hashing the result:

\[
\text{Finished} = \text{Hash}[\text{MasterSecret}, \text{MessageData}]
\]

Message data contains all the data snippets from the previous message interactions, collecting the payload from (\( tls_1^0 - tls_5^0 \)).

\[
\text{MessageData} = A, S, N_A, N_S, N_S^{\text{ID}}, W_A^{\text{CryptoOffer}}, W_S^{\text{CryptoOffer}}, E[K_S : N_A^{\text{PMS}}], E[K_A^{-1} : \text{Hash}[N_S, S, N_A^{\text{PMS}}]]
\]

In case TLS is configured with one- or two-way authentication, certificates are exchanged in advance in a separate process. The certificate exchange can be modelled as follows, where a Certification Authority (CA) signs the public key of an agent \( X \):

\[
\text{SignedCertificate}(X) = \text{Sign}(K_{CA}^{-1}, X, K_X)
\]

where the signing is defined in the standard way: \( \text{Sign}(\text{Key}_{\text{private}}, F) = F, E[\text{Key}_{\text{private}} : \text{Hash}[F]] \). The private key in our case is given by \( \text{Key}_{\text{private}} = K_{CA}^{-1} \).

The process of distributing certificates should be modelled as a separate protocol. Distribution of certificates can be done in several ways. For the sake of simplicity we model the process as a request response pattern:

\[
\text{distributeCertificate}(X) = \begin{align*}
X & \rightarrow CA : W^{\text{RequestCertificate}} \\
CA & \rightarrow X : W^{\text{DeliverCertificate}} X, K_X, E[K_{CA}^{-1} : \text{Hash}[X, K_X]]
\end{align*}
\]

The core specification of TLS - defined in the PROSA syntax is given in Figure 6.6. The relation between a standard Alice-Bob notation and PROSA is straightforward, for most of the primitives there is a direct translation. In some cases PROSA contains more type-information, like for instance the notion of key.
protocol[TLSversion1.1Zerowayauth, 0,
role(A) \& role(S), role(A) \& role(S), role(A),

(tls\textsubscript{1}) \begin{align*}
A \rightarrow S : & \text{Agent}(A) \land \text{isNonce}(n(ANONCE, A)) \land \text{isNonce}(n(SID, A)) \land \text{Text}(CryptoOffer, A)
\end{align*}

(tls\textsubscript{2}) \begin{align*}
S \rightarrow A : & \text{isNonce}(n(SNONCE, S)) \land \text{isNonce}(n(SID, A)) \land \text{Text}(CryptoOffer, S)
\end{align*}

authentication (signed certificate) optional messages

(tls\textsubscript{3}) \begin{align*}
A \rightarrow S : & E[\text{key(asym, public, S)} : \text{isNonce}(n(PMS, A))]
\end{align*}

verify certificate optional

(tls\textsubscript{4}) \begin{align*}
A \rightarrow S : & E[\text{ClientKey} : \text{Finished}]
\end{align*}

(tls\textsubscript{5}) \begin{align*}
S \rightarrow A : & E[\text{ServerKey} : \text{Finished}]
\end{align*}

\[
\text{MasterSecret} = \text{Hash}[\text{isNonce}(n(PMS, A)) \land \text{Text}(\text{MasterSecret}) \land \\
\text{isNonce}(n(ANONCE, A)) \land \text{isNonce}(n(SNONCE, S))]
\]

\[
\text{Finished} = \text{Hash}[\text{MasterSecret} \land \text{MessageData}]
\]

\[
\text{MessageData} = A, S, N_A, N_S, N^{\text{SID}}, \\
W^{\text{CryptoOffer}}_A, W^{\text{CryptoOffer}}_S, E[K_S : N^{\text{PMS}}_A], E[K_A^{-1} : \text{Hash}[N_S, S, N^{\text{PMS}}_A]]
\]

\[
\text{ClientKey} = \text{key}(\text{Agent}(A) \land \text{isNonce}(n(ANONCE, A)) \land \text{isNonce}(n(SNONCE, S)) \land \text{MasterSecret})
\]

\[
\text{ServerKey} = \text{key}(\text{Agent}(S) \land \text{isNonce}(n(ANONCE, A)) \land \text{isNonce}(n(SNONCE, S)) \land \text{MasterSecret})
\]

\[
\text{SignedCertificate}(X) = \text{Sign}(\text{key(asym, priv, CA)}, \text{Agent}(X) \land \text{isKey}(\text{key(asym, public, X)}))
\]

where \[
\text{Sign}(\text{Key}_{\text{private}}, F) = F \land E[\text{Key}_{\text{private}} : \text{Hash}[F]]
\]

\[
\text{distributeCertificate}(X) = \\
\text{protocol}[\text{distributionOfCertificates}, 0, \text{role}(X) \land \text{role(CA)}, \text{role}(X) \land \text{role(CA)}, \text{role}(X),
\]
\[
X \rightarrow CA : \text{Text}(\text{Request Certificate})
\]
\[
CA \rightarrow X : \text{Text}(\text{Deliver Certificate}) \land \text{Agent}(X) \land \text{isKey}(\text{key(asym, public, X)}) \land \\
E[\text{key(asym, priv, CA)} : \text{Hash[Agent(X) \land isKey(key(asym, public, X))]}]
\]

Figure 6.6: Specification of TLS using PROSA syntax.
Protocol - Exchange Credentials

The initial protocol uses TLS (typically one-way authentication), in order to submit the initial credentials from the client $A$ to identity provider $IDP$. Protocol Exchange Credentials establishes one registration of a client. Observe that there is one occurrence of a monadic second-order variable $X^{2O}_{AUTHFACTORS}$ - a placeholder for the authentication factors to be used in the particular configuration. Initially one TLS session is used, then two messages are exchanged as described below in the exchangeCred protocol:

\[
(reg_1) \quad A \rightarrow IDP : E[\text{clientKey} : W_{\text{RegistertoAS/IDP}}, X^{2O}_{AUTHFACTORS}, A, N^{\text{REGISTER}}_A]
\]

\[
(reg_2) \quad IDP \rightarrow A : E[\text{serverKey} : W_{\text{RegistertoAS/IDP}}, A, IDP, N^{\text{REGISTER}}_A, N^{\text{REPLY}}_{IDP}]
\]

In the first message, the client $A$ encrypts the name of the agent, the authentication credentials for the client $A$, and a session ID nonce $N^{\text{REGISTER}}_A$, using the TLS session key for encryption clientKey. The request is notified and an acknowledgement is sent back using the serverKey. The response message includes the client and identity provider’s name, the session nonce and a reply nonce.

Authentication protocol

The purpose of the authentication protocol is to authenticate the user client towards a particular service. The identity provider (IDP) is the entity that creates, maintains, and manages identity information for principals and provides principal authentication to other service providers within a federation, such as web browser.

In the system we propose, the IDP is the only module that sees the authentication factors. IDP authenticates the user on behalf of the Service Provider, and establishes the secure communication channel between the client $A$ and $SP$. Hence the authCred-protocol is given as follows:

\[
(auth_1) \quad A \rightarrow IDP : E[\text{clientKey}^* : W_{\text{authenticatetoservice}}, A, SP, N^{\text{AUTH}}_A]
\]

\[
(auth_2) \quad IDP \rightarrow A : E[\text{serverKey}^* : W_{\text{RegistertoAS/IDP}}, A, IDP, N^{\text{AUTH}}_A, N^{\text{AUTHREPLY}}_{IDP}]
\]

\[
(auth_3) \quad A \rightarrow IDP : E[\text{clientKey}^* : W_{\text{authenticatetoservice}}, A, X^{2O}_{AUTHFACTORS}, N^{\text{AUTH}}_A, N^{\text{AUTHREPLY}}_{IDP}]
\]

\[
(auth_4) \quad IDP \rightarrow SP : E[K_{SP} : A, N^{\text{AUTH}}_A, N^{\text{AUTHREPLY}}_{IDP}, \text{Hash}[X^{2O}_{AUTHFACTORS}]]
\]

\[
(auth_5) \quad SP \rightarrow IDP : E[K_{IDP} : A, N^{\text{AUTH}}_A, N^{\text{AUTHREPLY}}_{IDP}, N^{\text{AUTHREADY}}_{SP}]
\]

\[
(auth_6) \quad IDP \rightarrow A : E[\text{serverKey}^* : N^{\text{AUTH}}_A, N^{\text{AUTHREPLY}}_{IDP}, N^{\text{AUTHREADY}}_{SP}]
\]

The authentication protocol is initiated by a TLS session, resulting in a pair of client-server keys, denoted clientKey* and serverKey*. Observe that a registration session, including a TLS session, is supposed to occur prior to the authentication process - that includes a new pair of session keys (marked with a •). In the first message the client sends a request to authenticate, including its name $A$, the service provider $SP$, and a session nonce $N^{\text{AUTH}}_A$, encrypted with the client session-key. The identity provider returns the nonce including a response nonce, encrypted with the TLS generated session key. Then the client $A$ sends to $IDP$ the authentication credentials: its name $A$, the authentication factors and the nonces encrypted with clientKey*. Then in the fourth message, $IDP$ is handing the credentials over to the service provider $SP$, along with the session nonces. The content is encrypted with the public key of the service provider. The service provider returns the nonces used so far, in addition to a freshly generated nonce, encrypted with the public key of the identity provider. Finally the nonces are forwarded to the client encrypted with the TLS session key.

\(^3\text{From the online Glossary Term Definition of Identity Provider (OASIS)}\)
Transaction protocol

The purpose of the transaction protocol is to enable the communication between the user client and the service provider. The transaction is a simple request-response pattern.

\[(tr_1) \quad A \rightarrow SP : W^{RequestService}, N_A^{SESSIONID}\]

\[(tr_2) \quad SP \rightarrow A : W^{DeliverService}, N_A^{SESSIONID}\]

There are several ways to add security to the user-transaction - either keeping the content secret or assuring the integrity of the message payload or the integrity of the originator of the transaction. In this section we present one way of obtaining confidentiality to the channel between the client and the service provider, while Section 6.3.3 contains several other variations of user-transaction. We define confidentiality as a relation between a data-structure and a set of agents. We way that a piece of data \( D \) is confidential with respect to two agents \( A \) and \( B \) if and only if no other agent (different from \( A \) and \( B \)) can obtain \( D \). In the context of message-oriented systems we might define more specific notions of confidentiality: A one-way channel from agent \( A \) to agent \( B \) is one-way confidential iff the data \( D \) sent from \( A \) to \( B \) cannot be obtained by a third agent intercepting the message. A two-way channel between two agents \( A \) and \( B \) is two-way confidential, iff the one-way channels from \( A \) to \( B \) and from \( B \) to \( A \) both are one-way confidential. The channel could provide confidentiality using either symmetric keys or using public keys. The former corresponds to two-way confidentiality: if one participant \( C \) knows the key used for encrypting communication between agent \( A \) and \( B \), then \( C \) can see and manipulate every message on the \( AB \) channel, encrypt and decrypt messages going from both \( A \) and \( B \).

In the first version a symmetric key is used that consists of elements from the authentication session, in addition to the authentication credentials. Thus the session key, containing the agent variable \( X \)

\[KEY^X_{SESSION} = \text{key}(\text{Hash}[X, N_A^{AUTH}, N_{SP}^{AUTHREPLY}, N_{SP}^{AUTHREADY}, \text{Hash}[X^{20^{AUTHFACTORS}}]])\]

is used to make the content (payload) confidential. The client \( A \) encrypts using \( KEY^A_{SESSION} \) while the service provider encrypts using \( KEY^SP_{SESSION} \). The transaction protocol for confidentiality is a simple request-response protocol. The client \( A \) requests a service through the confidentiality channel, then a particular service is delivered from the service provider \( SP \) to the client.

\[(tr_1)^{SYM} \quad A \rightarrow SP : E[KEY^A_{SESSION} : W^{RequestService}, N_A^{SESSIONID}]\]

\[(tr_1)^{SYM} \quad SP \rightarrow A : E[KEY^SP_{SESSION} : W^{DeliverService}, N_A^{SESSIONID}]\]

When receiving the first message the Service Provider decrypts the payload ciphertext by constructing the client version of the session key \( KEY^A_{SESSION} \):

\[D[KEY^A_{SESSION} : E[KEY^A_{SESSION} : W^{RequestService}, N_A^{SESSIONID}]] = W^{RequestService}, N_A^{SESSIONID}\]

The service is delivered to the client \( A \) protected by the client session key \( KEY^SP_{SESSION} \). The message is decrypted by the client that receives the service in a similar fashion since the client possesses all the required constituents of the key. Hence by performing

\[D[KEY^SP_{SESSION} : E[KEY^SP_{SESSION} : W^{DeliverService}, N_A^{SESSIONID}]] = W^{DeliverService}, N_A^{SESSIONID}\]

the client obtains the service from \( SP \).

6.3.3 Variation points

In the Identity Management casestudy many security features can be varied. The most insecure variant of the entire system does not include any application of TLS and succeeding session keys for encryption. Hence
the Registration Protocol is given by
\[ (\text{reg}_{1}^{\text{plain}}) \quad A \rightarrow IDP : W^{\text{RegistertoAS/IDP}}, X^{20}_{\text{AUTHFACTORS}}, A, N_{A}^{\text{REGISTER}} \]
\[ (\text{reg}_{2}^{\text{plain}}) \quad IDP \rightarrow A : W^{\text{RegistertoAS/IDP}}, A, IDP, N_{A}^{\text{REGISTER}}, N_{IDP}^{\text{REPLY}} \]

Attacks on this protocol are trivial to construct given the Dolev Yao model. An eavesdropper can obtain all the required credentials by interrupting the first message and then constructing a reply by injecting a bogus nonce in the reply message.
\[ (\text{reg}_{11}^{\text{plainattack}}) \quad A \rightarrow I(IDP) : W^{\text{RegistertoAS/IDP}}, A, IDP, N_{A}^{\text{REGISTER}} \]
\[ (\text{reg}_{12}^{\text{plainattack}}) \quad I(IDP) \rightarrow A : W^{\text{RegistertoAS/IDP}}, A, IDP, N_{A}^{\text{REGISTER}}, N_{IDP}^{\text{REPLY}} \]

The attacker I can now impersonate as the client A and use A’s credentials in order to perform malicious transactions later on. If the Authentication Protocol is designed without the confidentiality mechanisms described in Section 6.3.2 severe attacks on the authentication process can be performed, since every transaction is performed in plaintext:
\[ (\text{auth}_{1}^{\text{plainAttack}}) \quad I(A) \rightarrow IDP : W^{\text{authenticateoservice}}, A, SP, N_{I}^{\text{AUTH}} \]
\[ (\text{auth}_{2}^{\text{plainAttack}}) \quad IDP \rightarrow I(A) : W^{\text{RegistertoAS/IDP}}, A, IDP, N_{A}^{\text{AUTH}}, N_{IDP}^{\text{IDP}} \]
\[ (\text{auth}_{3}^{\text{plainAttack}}) \quad I(A) \rightarrow IDP : W^{\text{authenticateoservice}}, A, X^{20}_{\text{AUTHFACTORS}}, N_{A}^{\text{AUTH}}, N_{IDP}^{\text{AUTHREPLY}} \]
\[ (\text{auth}_{41}^{\text{plainAttack}}) \quad IDP \rightarrow I(SP) : A, N_{I}^{\text{AUTH}}, N_{IDP}^{\text{AUTHREPLY}}, \text{Hash}[X^{20}_{\text{AUTHFACTORS}}] \]
\[ (\text{auth}_{42}^{\text{plainAttack}}) \quad I(IDP) \rightarrow SP : A, N_{I}^{\text{AUTH}}, N_{IDP}^{\text{AUTHREPLY}}, \text{Hash}[X^{20}_{\text{AUTHFACTORS}}] \]
\[ (\text{auth}_{51}^{\text{plainAttack}}) \quad SP \rightarrow I(IDP) : A, N_{I}^{\text{AUTH}}, N_{IDP}^{\text{AUTHREPLY}}, N_{SP}^{\text{AUTHREADY}} \]
\[ (\text{auth}_{52}^{\text{plainAttack}}) \quad I(SP) \rightarrow IDP : A, N_{I}^{\text{AUTH}}, N_{IDP}^{\text{AUTHREPLY}}, N_{SP}^{\text{AUTHREADY}} \]
\[ (\text{auth}_{6}^{\text{plainAttack}}) \quad IDP \rightarrow I(A) : N_{I}^{\text{AUTH}}, N_{IDP}^{\text{AUTHREPLY}}, N_{SP}^{\text{AUTHREADY}} \]

The attacker has authenticated to the system - impersonating the client A - fooling both the Service provider and the Identity Provider.

Even if TLS is used in the authentication process some attacks still apply given the weakest security configuration of registration. The security strengths of TLS are well understood (see for instance [35], and several tools verifying TLS in the AVISPA project [6]). TLS is particularly robust towards Man-in-the-Middle attacks (MIM): Even the weakest authentication modus (no certificates) does not permit straightforward MIM. Without certificates, any attacker can impersonate either as the client or server. If the client is fooled to accept a fake certificate for the server then the following attack is applicable:
\[ (\text{tls}_{1}^{\text{Attack}}) \quad A \rightarrow I(S) : A, N_{A}, N_{A}^{\text{SID}}, W_{A}^{\text{CryptoOffer}} \]
\[ (\text{tls}_{2}^{\text{Attack}}) \quad I(S) \rightarrow A : N_{I}, N_{I}^{\text{SID}}, W_{I}^{\text{CryptoOffer}} \]
\[ (\text{tls}_{3}^{\text{Attack}}) \quad A \rightarrow I(S) : E[K_{I} : N_{A}^{\text{PMS}}] \]
\[ (\text{tls}_{4}^{\text{Attack}}) \quad A \rightarrow I(S) : E[\text{ClientKey}_{I} : \text{Finished}_{I}] \]
\[ (\text{tls}_{5}^{\text{Attack}}) \quad I(S) \rightarrow A : E[\text{ServerKey}_{I} : \text{Finished}_{I}] \]
The attacker intercept the first message, storing the payload. In the next message $tls^{\text{Attack}}_2$, a bogus nonce $N_I$ and a suitable bogus cryptooffer is created by the attacker. The client $A$ encrypts the PreMasterSecret she created using the public key of the attacker $K_I$ in $tls^{\text{Attack}}_3$. Finally in the two last messages, a compromized pair of session keys $ServerKey_A$ and $ServerKey_I$ is used together with a bogus data-structure $\text{Finished}_I$.

Consequently, the client exposes all credentials to the server and the generated session keys are compromised, the attacker can communicate with the client on the final channel where the authentication credentials are transmitted - hence the attacker captures all the required factors used in authentication.

For more typical cases, TLS will be used in order to establish a communication channel between the user and the system with some security properties. In general, authentication is considered to be a one way process: some client $A$ authenticates to a particular server $S$. This means that $A$ presents some evidence to the server $S$ that the client $A$ is who she claims to be. Two parties $A$ and $S$ can mutually authenticate each other by running two authentication processes, whereby both parties presents some evidence to each other - evidence which they verify on each side. TLS can be used with three distinct authentication options - what we call authentication directions. These are based on the use of optional certificates in TLS. The first option does not include any certificates - we call it zero-way, one-way where the client has the certificate of the server, and finally two-way authentication - where each participant has a certificate. Hence depending on the strength of the authentication, suitable distribution protocols should be chosen. The authentication factor might be only one, one-factor (something the client knows), two factors (something the client knows and has), and three-factor (something the client knows, has and is). Depending on the option decided in the registration protocol, the authentication protocol should deploy the corresponding authentication factors.

In our example we have used the following instances of the factors, static password, dynamic password and finger print.

<table>
<thead>
<tr>
<th>strength</th>
<th>example</th>
<th>relation to user</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-factor</td>
<td>$W^X_{\text{StaticPassword}}$</td>
<td>has</td>
</tr>
<tr>
<td>2-factor</td>
<td>$W^X_{\text{StaticPassword}}$, $N^X_{\text{DynamicPassword}}$</td>
<td>has and knows</td>
</tr>
<tr>
<td>3-factor</td>
<td>$W^X_{\text{StaticPassword}}$, $N^X_{\text{DynamicPassword}}$, $W^X_{\text{FingerPrint}}$</td>
<td>has and knows and is</td>
</tr>
</tbody>
</table>

In Figure 6.7, we show how the initial example can be varied. Several of the options do not make sense: for instance if the registration process includes two way TLS and two-factor authentication then authentication cannot be three-factor, and the distribution of certificates cannot be empty. There are also selections of features that give insecure products.

There are several ways to order the security strength of the two authentication features authentication factors and authentication directions. One can give two partial orders of the features: either give the highest weight to the factors or the directions, as described in Figure 6.8. The weakest authentication is 0-way (no application of certificates) with only one authentication factor, while the strongest is 2-way authentication with three factors.

What are the most important - the factors or the directions? Unfortunately it is not easy decide - it depends on the strength of each factor and whether distribution of certificates is performed in a a secure way. Three weak authentication factors can give weaker security than one strong factor. One long and hard-to-guess static password can be more secure than short static password, easy-to-guess dynamic passord and weak implemented biometrical factors. Mutual checking of certificates do not give any security advantage compared to no-certificate validation, if the system permits compromized certificates or the implementation of certificate validation is poor or erroneous.

In Section 6.3.2 we presented a version of the transaction protocol providing confidentiality using symmetric keys. There are several other ways to configure the transaction protocol: Another typical way of providing confidentiality (directed confidentiality), is using public keys. In that case each communicating party encrypts the payload using the receiver’s public key:

$$\begin{align*}
(t^{\text{publ}}_1) & \quad A \rightarrow SP : E[K_{SP} : W^{\text{RequestService}}, N^A_{\text{SESSIONID}}] \\
(t^{\text{publ}}_2) & \quad SP \rightarrow A : E[K_A : W^{\text{DeliverService}}, N^A_{\text{SESSIONID}}]
\end{align*}$$

50
The result is a two-way confidentiality channel built up of two one-way channels. Integrity is typically explained as the property that data cannot be modified without being detected. This is called *data-integrity*. Hence integrity includes three particular notions: *sender-integrity*, *receiver-integrity* and *data-integrity*. By *sender-integrity* we mean that the receiver can verify that the message was created by the sender and not any other agent. By *receiver-integrity* we mean that the sender has guarantees that only the receiver can access the content.

In some cases strong confidentiality might imply integrity of the payload protected by encryption. In the latter variant of the transaction protocol, integrity does not come for free. The following attack shows why integrity can be broken.

\[(tr_{publAttack}^{1.1}) \quad A \rightarrow I(SP) : E[K_{SP} : W^{RequestService}, N^{SESSIONID}_A]\]

\[(tr_{publAttack}^{1.2}) \quad I(A) \rightarrow SP : E[K_{SP} : W^{RequestAnotherService}, N^{SESSIONID}_I]\]

\[(tr_{publAttack}^{2.1}) \quad SP \rightarrow I(A) : E[K_A : W^{DeliverAnotherService}, N^{SESSIONID}_I]\]

The attacker has successfully compromised the initial request message, and fooled the Service Provider to accept a bogus request. But the session nonce limits the attackers possibility of compromising the response. The attacker can only forward the message received from *SP*.

\[(tr_{publAttack}^{2.2}) \quad I(SP) \rightarrow A : E[K_A : W^{DeliverAnotherService}, N^{SESSIONID}_I]\]

A way to avoid attacks on integrity is to introduce one particular integrity-protecting mechanism, the *digital signature*. One common option is to sign the content before encryption, in order to bind the content...
to the sender of the message, either the request or the response or both. In our next example we do both:

\[(tr_1^{sign}) \quad A \rightarrow SP : E[\text{KEY}_A^{\text{SESSION}} : \text{Sign}(K_{A^{-1}}^{\text{RequestService}}, N_A^{\text{SESSIONID}})]\]

\[(tr_2^{sign}) \quad SP \rightarrow A : E[\text{KEY}_{SP}^{\text{SESSION}} : \text{Sign}(K_{SP}^{-1}, W^{\text{DeliverService}}, N_A^{\text{SESSIONID}})]\]

The final signature requires that the Service Provider certificate is distributed in advance or as part of the authentication process (Authentication Protocol), as one variant.

### 6.4 How to Introduce Variabilities and Deltas

There are several ways one could introduce variability into security architectures. One way could be to define a core or primary architecture and then add deltas to it. In practice this would mean that a given ABS implementation could be augmented with extra code-lines. The possibility for error, and therefore potential vulnerability occurs in such a situation.

A better solution is to make a high level precise design, including several variants, and then automatically generate the implementations of the variants. This means that we first construct protocol specifications for each variant and then afterwards generate code for each variant. If the specification is secure and the transformation from specification to implementation is secure (does not introduce security flaws), then the resulting ABS-implementation is secure. By starting with specifications and afterwards generating code, deltas are generated implicitly. The delta is given as the difference between two automatically generated implementation variants. Variations of security products can typically occur in three different ways:

**V1** Making variants of data: which means the substitution of a given fixed data-structure (containing the variation) inside a protocol for a second-order variable (see Figure 6.9).

**V2** Addition of one or more messages including payload (see Figure 6.10).

**V3** Removal of entire message(s) from the core specification (see Figure 6.11).

We consider the former case in depth, the latter two are solved by making extra versions of the entire protocol. It is possible to solve **V2** and **V3** using protocol algebra and functions but that is left to future work.

**Making variants of data**

First we consider the former case, how to make variants of data: Since the basic payload-structures are represented as sentences, we introduce second-order monadic variables in the core protocol template and substitution over second-order variables. We write the second-order monadic variables as \(X\). Substitution \(\text{sub}(G, X^{20}, F(X^{20}))\) reads “replace the sentence \(G\) for the second-order variable \(X^{20}\) in the sentence \(F\)”. Substitution over protocol specifications is defined by obvious recursion on the sentences in PROSA. Since the substitution is over sentences and not terms, type casting over keys involves potential extra substitutions when a key is composite - meaning that the key is built up of another key material. Thus we have the following special clauses:

\[(e_1) \quad \text{sub}(G, X^{20}, \text{key}(KEY)) = \text{key}(\text{sub}(G, X^{20}, KEY)) \text{ if Sentence}(KEY)\]
\[(e_2) \quad \text{sub}(G, X^{20}, \text{key}(KEY)) = \text{key}(KEY) \text{ if not Sentence}(KEY)\]
\[(e_3) \quad \text{sub}(G, X^{20}, \text{isKey}(\text{key}(KEY))) = \text{isKey}(\text{sub}(G, X^{20}, KEY)) \text{ if Sentence}(KEY)\]
\[(e_4) \quad \text{sub}(G, X^{20}, \text{isKey}(\text{key}(KEY))) = \text{isKey}(\text{key}(KEY)) \text{ if not Sentence}(KEY)\]
\[(e_5) \quad \text{sub}(G, X^{20}, E[KEY : F]) = E[\text{sub}(G, X^{20}, KEY) : \text{sub}(G, X^{20}, F)]\]
\[(e_6) \quad \text{sub}(G, X^{20}, \text{HMAC}[KEY : F]) = \text{HMAC}[\text{sub}(G, X^{20}, KEY) : \text{sub}(G, X^{20}, F)]\]

The predicate Sentence(\(KEY\)), checks whether \(KEY\) is of type Sentence. If \(KEY\) is not of that type then substitution is skipped, else it is a composite session key - and substitution can be applied. In case of the
cryptographic operations \( (e_5) \) and \( (e_6) \) the type-checking is performed in the second round (using clause \( (e_1) \) and \( (e_2) \)). In the appendix (Section A.2) we present the remaining rules covering non-cryptographic operations.

### 6.4.1 Formal system specification using protocol algebra

The system includes two applications of the TLS protocol. Since we use open sentences (no explicit first order quantification), the second application of the TLS protocol should have set of variables distinct from the former application. We introduce a function for generating fresh variable inside a protocol specification \( P \), written \( \mathcal{F}(P, n) \). The function \( \mathcal{F}(P, n) \) takes a protocol specification \( P \) and a natural number \( n \), and constructs the \( n \)th fresh version of the protocol. The function takes a protocol as input and slightly changes the variables for nonces in the original protocol such that corresponding variables differ, yet the original variables can be retrieved if required. Thus \( \mathcal{F}(\text{TLS}, 1) \) means the first reuse of TLS. Protocol composition \( P_1 \oplus P_2 \), is defined by concatenating the former protocol \( P_1 \) with the latter \( P_2 \) \[22\]. Hence, the three main protocols in the identity management can be defined precisely by:

- Registration: \( \text{TLS} \oplus \text{exchangeCred} \)
- Authentication: \( \mathcal{F}(\text{TLS}, 1) \oplus \text{authCred} \)
- Transaction: \( \text{transaction} \)

Note that \( \mathcal{F}(\text{TLS}, 1) \), means that fresh variables are generated for the second round of TLS.

A complete run of the entire system where one client, first registers and submits credentials, then authenticates one time, and thereafter performs one transaction is defined formally via protocol composition by:

\[
(\text{TLS} \oplus \text{exchangeCred}) \oplus (\mathcal{F}(\text{TLS}, 1) \oplus \text{authCred}) \oplus \text{transaction}
\]

When we adapt the specification with a second-order variable \( X^{20} \) for authentication factors we write:

\[
\text{CompleteIDM} = (\text{TLS} \oplus \text{exchangeCred}(X^{20})) \oplus (\mathcal{F}(\text{TLS}, 1) \oplus \text{authCred}(X^{20})) \oplus \text{transaction}(X^{20})
\]

Three classes of variants of this application can be designed as indicated earlier. By performing substitutions on the second-order variable \( X^{20} \) for the three different kinds of factors used we obtain three versions of authentication:
In order to vary the protocol options, we define specifications for each option. In case of the service exchange credentials, three distinct versions of TLS can be deployed, without certificates, one-directional authentication - including server certificate, and bidirectional authentication - including both client and server certificates.

6.4.2 Generating ABS code for the variants

When we generate code from a selected configuration we perform the choice of protocol variant and substitution in advance. Observe that there is no obvious way to separate particular security features from the rest of the source code. One particular variant of a security feature does not correspond to a bounded code snippet - instead the code associated with a particular feature is spread out in several locations of the code. In contrast to the library of protocols used as case study in Task T4.1 - where each protocol was rather limited, the current identity management system gives rise to 158 Kbyte ABS code, and 2060 lines of code. Although the code-footprint is considerably larger than the single protocols explored in Task T4.1, it is relatively small compared to real IDM systems.

6.4.3 Giving security features values

We have designed a system which has several similarities with real systems. We have indicated that it is in general not possible to assign scores on the particular security features described in this chapter. Yet, we do not have exact data on real systems, but we suspect that a real system implemented in Java would include several ten-thousands lines of code.
under appropriate assumptions and given a particular application a scoring can be justified. If we assume that static entities with a given low probability can be stolen (static passwords or public or private keys), then we can assign values to the variants proposed for confidentiality and integrity as follows:

<table>
<thead>
<tr>
<th>Conf. mechanism</th>
<th>Confidentiality Score</th>
<th>Integrity mechanism</th>
<th>Integrity Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plaintext (none)</td>
<td>0</td>
<td>Plaintext (none)</td>
<td>0</td>
</tr>
<tr>
<td>Public keys</td>
<td>1</td>
<td>Symmetric keys</td>
<td>1</td>
</tr>
<tr>
<td>Symmetric keys</td>
<td>2</td>
<td>Digital Signature</td>
<td>2</td>
</tr>
</tbody>
</table>

Two orderings of authentication strength was presented in figure 6.8. Whether it is possible to construct a feature-tree based on these initial orderings of the main security mechanisms is left as an open issue.
Chapter 7

Conclusion

In this task we brought together efforts in the project related to product auto configuration, quality variability modeling and static analysis of performance and security with the aim of supporting quality-aware auto configuration of products from a product line infrastructure.

In terms of product auto configuration, the ABS Product Configurator supports the developer in finding a selection of the features from the feature model that satisfies the requirements of a concrete product and adhere to the feature model rules. Our configurator asks for only the key features of the product under development and optionally its cost constraint, checks whether they satisfy the feature model rules, and completes the selection of features according to the feature model and the user’s indication of minimizing cost or maximizing it within a certain constraint.

In terms of quality variability, we investigated a number of modeling techniques. Having in mind the requirements for variability modeling techniques but focusing on learn-ability and usability, we decided for the adoption of the extended feature model technique: features in a $\mu$TVL model are annotated with quality attribute indicators. A quality attribute indicator provides an estimate on how a certain feature affects a certain quality attribute of the product. Our configurator uses this information plus the quality attributes of relevance for the user (i.e., the percentage of relevance of those quality attributes) to provide a set of product configurations that have the best chance of maximizing the quality attributes of interest (i.e., quality concerns). In practice, we use the information about the percentage of relevance of the quality attributes for the product in an objective function that is given to a Constraint Satisfaction Problem solver.

As learn-ability, usability and thereby adoption by practitioners of the HATS technologies are crucial points in the HATS project, our configurator provides a graphic interface for visualizing features models, allowing the selection of the key features of a certain product and supporting the correction of invalid set of features interactively. This was achieved by integrating FeatureIDE.

Two quality attributes have been especially investigated in this task, as already proposed in its description of work, namely performance and security. Concerning performance, we have introduced the notion of resource-aware configuration which strives for finding a selection of features which leads to a product that has an efficient performance and that complies with the quality constraints provided by the user. We have envisaged several scenarios for resource-aware configuration and described a prototype implementation of the most practical scenario. Our implementation shows that it is feasible to use an off-the-self analyzer to obtain performance indicators that can be used to annotate feature models. Using the annotated feature models, the configurator is able to suggest a small set of valid product configurations that best fit the objective function representing the user’s input.

Our implementation and its application to the case study constitute a proof of concept for resource-aware configuration. However, a thorough experimental evaluation is required to assess the accuracy of the envisaged scenarios and, in particular, to define appropriate heuristics that lead to efficient products. In future work, and outside the context of the HATS project, we plan to define and evaluate different heuristics to combine the contribution of each method to the resource consumption of the feature, and also more refined heuristics to map cost expressions into performance annotations.
With regard to security, this task had an even stronger investigative character. As a first step towards modeling variability in security we tried to identify the sources of variability in security through a case study. The vision is to have a feature model for security that can be integrated into the application domain feature model and used by the ABS Product configurator to automatically select security features taking into consideration the relevance of security for the product.
Bibliography


Glossary

Terms and Abbreviations

\(\mu\text{TVL}\) Micro Textual Variability Language

DML Delta Modeling Language

**Delta** Synonymous with Delta Module

**Delta Module** A specification of modifications to core ABS classes and interfaces

**DoW** Description of Work

ABS Abstract Behavioral Specification (Language)

ASM Abstract State Machine

**AST** Abstract Syntax Tree

**CL** (Product Line) Configuration Language

**Core ABS** The behavioural functional and object-oriented core of the ABS modeling language

**COG** Concurrent Object Group

**CSP** Communicating Sequential Processes

**CCS** Calculus of Communicating Systems

**Deployment Component** An abstract resource monitor for platform modeling.

**Full ABS** Core ABS language plus all extensions (\(\mu\text{TVL}, \text{DML}, \text{CL}, \text{PSL}, \text{etc}\))

**Future** A place holder for the result of an asynchronous method call.

**IDE** Integrated Development Environment

**OOL** Object-Oriented (Programming) Language

**KeY** A formal software development tool that aims to integrate design, implementation, formal specification, and formal verification of object-oriented software as seamlessly as possible.

**PLE** Product Line Engineering, i.e., SPLE

**PSL** Product Selection Language

**RMI** Remote Method Invocation

**SPLE** Software Product Line Engineering

**UML** Unified Modeling Language
Appendix A

Variability in Security Specification

A.1 PROSA specification of the main protocols

In this section we present the detailed system specifications written in PROSA, the three main protocols for exchanging credentials, authentication and transaction. Each protocol specification begins with a header containing protocol name, session number, the total roles involved in the protocol, the agent-specific roles and finally the role that initiates the protocol session. The main part consists of protocol clauses: message transmissions including complete payload structure.

A.1.1 Protocol exchange Credentials

```
protocol[exchangeCred, 0,
  role(A), role(IDP), role(A), role(IDP), role(A),

A → IDP : E[clientKey : W^RegistertoAS/IDP, AUTHFACTORS^A, N_A^REGISTER]
IDP → A : E[serverKey :
  W^RegistertoAS/IDP, A, IDP, χ_A^REGISTER, N_IDP ]
```

A.1.2 Authentication protocol

```
protocol[AuthenticationCred, 0,
  role(A) ∧ role(IDP) ∧ role(SP),
  role(A) ∧ role(IDP) ∧ role(SP),
  role(A),

A → IDP : E[clientKey^* :
  Text(authenticate to service) ∧ Agent(A) ∧ Agent(SP) ∧ isNonce(n(AUTH, A))]
IDP → A : E[serverKey^* :
  Text(Register to AS/IDP) ∧ Agent(A) ∧ Agent(IDP) ∧
  isNonce(n(AUTH, A)) ∧ isNonce(n(AUTHREPLY, IDP))]
A → IDP : E[clientKey^* :
  Text(authenticate to service) ∧ Agent(A) ∧
  AUTHFACTORS^A isNonce n(AUTH, A) ∧ isNonce n(AUTHREPLY, IDP)]
IDP → SP : E[K_SP :
  Agent(A) ∧ isNonce n(AUTH, A) ∧ isNonce n(AUTHREPLY, IDP) ∧
  Hash[AUTHFACTORS^A]]
SP → IDP : E[K_IDP :
  Agent(A) ∧ isNonce n(AUTH, A) ∧
  isNonce n(AUTHREPLY, IDP) ∧ isNonce n(AUTHREADY, SP)]
IDP → A : E[serverKey :
  isNonce n(AUTH, A) ∧ isNonce n(AUTHREPLY, IDP) ∧
  isNonce n(AUTHREADY, SP)]
```

63
A.1.3 Transaction protocol

protocol[Transaction, 0,
role(A) \land role(IDP) \land role(SP),
role(A) \land role(IDP) \land role(SP),
role(A),
A \rightarrow SP : E[key(Hash(\text{Agent}(A) \land isNonce(n(\text{AUTH}, \text{A}))) \land isNonce(n(\text{AUTHFACTORS}, \text{IDP}))) \\
\land isNonce(n(\text{AUTHREADY, IDP})) \land Hash[\text{AUTHFACTORS}^2]) : \\
\text{Text(Request Service)} \land isNonce(n(\text{SESSIONID}, \text{A})))
SP \rightarrow A : E[key(Hash(\text{Agent}(SP) \land isNonce(n(\text{AUTH}, A))) \land isNonce(n(\text{AUTHFACTORS, IDP}))) \\
\land isNonce(n(\text{AUTHREADY, IDP})) \land Hash[\text{AUTHFACTORS}^2]) : \\
\text{Text(Deliver Service)} \land isNonce(n(\text{SESSIONID}, \text{A})))
]

A.2 Definition of second order substitution

For the sake of completeness we present the standard clauses for second order substitution below:

Definition A.2.1

(i) \[ \text{sub}(G, X^{20}, X^{20}) = G \]
(ii) \[ \text{sub}(G, X^{20}, Y^{20}) = Y^{20} \text{ if } X^{20} \neq Y^{20} \]
(iii) \[ \text{sub}(G, X^{20}, \text{ATOM}) = \text{ATOM} \]
(iv) \[ \text{sub}(G, X^{20}, F_1 \land F_2) = \text{sub}(G, X^{20}, F_1) \land \text{sub}(G, X^{20}, F_2) \]
(v) \[ \text{sub}(G, X^{20}, \text{protocol}[\mu, N, \xi_{Tot}, \xi_A, \xi_{Start}, \Phi]) = \text{protocol}[\mu, N, \xi_{Tot}, \xi_A, \xi_{Start}, \text{sub}(G, X^{20}, \Phi)] \]
(vi) \[ \text{sub}(G, X^{20}, \text{Bel}_A(F)) = \text{Bel}_A(\text{sub}(G, X^{20}, F)) \]
(vii) \[ \text{sub}(G, X^{20}, \text{Enforce}_A(F)) = \text{Enforce}_A(\text{sub}(G, X^{20}, F)) \]
(viii) \[ \text{sub}(G, X^{20}, F \otimes \Phi) = \text{sub}(G, X^{20}, F) \otimes \text{sub}(G, X^{20}, \Phi) \]
(ix) \[ \text{sub}(G, X^{20}, A \rightarrow B : F) = A \rightarrow B : \text{sub}(G, X^{20}, F) \]
(x) \[ \text{sub}(G, X^{20}, \varepsilon) = \varepsilon \]

A.3 Expanded protocol specifications

Below we show all the three protocols including the macros expanded.

A.3.1 Protocol exchange Credentials (expanding macros)

\[
\text{protocol}[\text{exchangeCred}, 0, \\
role(A) \land role(IDP), \\
role(A) \land role(IDP), \\
role(A), \\
A \rightarrow IDP : E[\text{key}(\text{Agent}(A) \land isNonce(n(\text{ANONCE}, \text{A}))) \land isNonce(n(\text{IDPNONCE}, \text{IDP}))) \\
\land Hash(isNonce(n(\text{PMS}, \text{A}))) \land \text{Text}(\text{MasterSecret}) \land \\
isNonce(n(\text{ANONCE}, \text{A})) \land isNonce(n(\text{IDPNONCE}, \text{IDP}))) : \\
\text{Text}(\text{Register to AS/IDP}) \land \text{AUTHFACTORS}) = \text{Agent}(A) \land isNonce(n(\text{REGISTER}, \text{A}))) \\
IDP \rightarrow A : E[\text{key}(\text{Agent}(\text{IDP}) \land \\
isNonce(n(\text{ANONCE}, \text{A}))) \land isNonce(n(\text{IDPNONCE}, \text{IDP}))) \\
\land Hash(isNonce(n(\text{PMS}, \text{A}))) \land \text{Text}(\text{MasterSecret}) \land \\
isNonce(n(\text{ANONCE}, \text{A})) \land isNonce(n(\text{IDPNONCE}, \text{IDP}))) : \\
\text{Text}(\text{Register to AS/IDP}) \land \text{Agent}(A) \land \text{Agent}(\text{IDP}) \land isNonce(n(\text{REGISTER}, \text{A})) \land isNonce(n(\text{REPLY, IDP}))) \\
]
A.3.2 Authentication protocol (expanding macros)

\[
\text{protocol} [\text{AuthenticationCred}, 0, \\
\text{role}(A) \land \text{role}(IDP) \land \text{role}(SP), \\
\text{role}(A) \land \text{role}(IDP) \land \text{role}(SP), \\
\text{role}(A) \land \text{role}(IDP) \land \text{role}(SP), \\
\text{role}(A), \\
A \rightarrow \text{IDP} : \\
E[\text{key} (\text{Agent}(A) \land \text{isNonce}(n(\text{ANONCE}^*, A)) \land \text{isNonce}(n(\text{IDPNONCE}^*, IDP))) \\
\land \text{Hash}(\text{isNonce}(n(PMS^*, A)) \land \text{Text} (\text{MasterSecret}) \land \text{isNonce}(n(\text{ANONCE}^*, A)) \land \text{isNonce}(n(\text{IDPNONCE}^*, IDP))) : \\
\text{Text} (\text{authenticate to service}) \land \text{Agent}(A) \land \text{isNonce}(n(\text{AUTH}, A))] \\
\text{IDP} \rightarrow A : \\
E[\text{key} (\text{Agent}(IDP) \land \text{isNonce}(n(\text{ANONCE}^*, A)) \land \text{isNonce}(n(\text{IDPNONCE}^*, IDP))) \\
\land \text{Hash}(\text{isNonce}(n(PMS^*, A)) \land \text{Text} (\text{MasterSecret}) \land \text{isNonce}(n(\text{ANONCE}^*, A)) \land \text{isNonce}(n(\text{IDPNONCE}^*, IDP))) : \\
\text{Text} (\text{authenticate to service}) \land \text{Agent}(IDP) \land \text{isNonce}(n(\text{AUTH}, A)) \land \text{isNonce}(n(\text{AUTHREPLY}, IDP))] \\
A \rightarrow \text{IDP} : \\
E[\text{key} (\text{Agent}(A) \land \text{isNonce}(n(\text{ANONCE}^*, A)) \land \text{isNonce}(n(\text{IDPNONCE}^*, IDP))) \\
\land \text{Hash}(\text{isNonce}(n(PMS^*, A)) \land \text{Text} (\text{MasterSecret}) \land \text{isNonce}(n(\text{ANONCE}^*, A)) \land \text{isNonce}(n(\text{IDPNONCE}^*, IDP))) : \\
\text{Text} (\text{authenticate to service}) \land \text{Agent}(A) \land \text{isNonce}(n(\text{AUTH}, A)) \land \text{isNonce}(n(\text{AUTHREPLY}, IDP))] \\
\text{IDP} \rightarrow SP : \\
E[K_{SP} : \text{Agent}(A) \land \text{isNonce}(n(\text{AUTH}, A)) \land \text{isNonce}(n(\text{AUTHREPLY}, IDP)) \land \\
\text{Hash}[\text{AUTHFACTORS}] \\
SP \rightarrow \text{IDP} : \\
E[K_{IDP} : \text{Agent}(A) \land \text{isNonce}(n(\text{AUTH}, A)) \land \\
\text{isNonce}(n(\text{AUTHREPLY}, IDP)) \land \text{isNonce}(n(\text{AUTHREADY}, SP))] \\
\text{IDP} \rightarrow A : \\
E[\text{key} (\text{Agent}(IDP) \land \text{isNonce}(n(\text{ANONCE}^*, A)) \land \text{isNonce}(n(\text{IDPNONCE}^*, IDP))) \\
\land \text{Hash}(\text{isNonce}(n(PMS^*, A)) \land \text{Text} (\text{MasterSecret}) \land \text{isNonce}(n(\text{ANONCE}^*, A)) \land \text{isNonce}(n(\text{IDPNONCE}^*, IDP))) : \\
\text{isNonce}(n(\text{AUTH}, A)) \land \text{isNonce}(n(\text{AUTHREPLY}, IDP)) \land \\
\text{isNonce}(n(\text{AUTHREPLY}, SP))] \\
\]

A.3.3 Transaction protocol (expanding macros)

Transaction protocol (expanding macros).

\[
\text{protocol} [\text{Transaction}, 0, \\
\text{role}(A) \land \text{role}(IDP) \land \text{role}(SP), \\
\text{role}(A) \land \text{role}(IDP) \land \text{role}(SP), \\
\text{role}(A), \\
A \rightarrow \text{IDP} : E[\text{key} \text{Hash}(\text{Agent}(A) \land \text{isNonce}(n(\text{AUTH}, A)) \land \\
\text{isNonce}(n(\text{AUTHREPLY}, IDP)) \land \text{isNonce}(n(\text{AUTHREPLY}, SP)) \land \\
\text{Hash}[\text{AUTHFACTORS}^2_0]) : \\
\text{Text} (\text{Request Service}) \land \text{isNonce}(n(\text{SESSIONID}, A))] \\
\text{IDP} \rightarrow A : \\
E[\text{key} \text{Hash}(\text{Agent}(SP) \land \text{isNonce}(n(\text{AUTH}, A)) \land \\
\text{isNonce}(n(\text{AUTHREPLY}, IDP)) \land \text{isNonce}(n(\text{AUTHREPLY}, SP)) \land \\
\text{Hash}[\text{AUTHFACTORS}^2_0]) : \\
\text{Text} (\text{Deliver Service}) \land \text{isNonce}(n(\text{SESSIONID}, A))] \\
\]

A.4 Automatically generated ABS code for the IDM system

The automatically generated version of the Identity Management system with the lowest security level can be found in the HATS website [http://www.hats-project.eu/]. The code is generated using the module for automated construction of security protocol defined in Task 4.1.