Project N°: FP7-231620

Project Acronym: HATS

Project Title: Highly Adaptable and Trustworthy Software using Formal Models

Instrument: Integrated Project

Scheme: Information & Communication Technologies

Future and Emerging Technologies

Deliverable D1.4

ABS System Derivation and Code Generation

Due date of deliverable: T0+42

Actual submission date: 6. September 2012

Start date of the project: 1st March 2009

Duration: 48 months

Organisation name of lead contractor for this deliverable: IoC

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

<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>
Executive Summary:
ABS System Derivation and Code Generation

This document summarises deliverable D1.4 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](http://www.hats-project.eu).

We report on the results achieved in the Task 1.4 “ABS System Derivation and Code Generation”.

We present an update on the Java backend of the ABS compiler first described in Deliverable D1.2 [2] (Section 7.4), emphasizing the new foreign function interface.

We report on a new Scala backend that is based on the Akka actor library of Scala and the continuations facility and maps the ABS concurrency model in an elegant way to high-level concurrency mechanisms of Scala. This backend supports distributed execution of ABS code. On subsets of ABS and Scala (core ABS and a small functional language with delimited continuations) we demonstrate that the compilation scheme of this backend is correct in a mathematically precise sense and also report on results of formalizing this proof in the proof assistant Agda.

We give an update on the support of delta modelling in ABS for software product lines, describing the changes and additions that we have introduced into the syntax and semantics of ABS deltas wrt. the initial description in Deliverable D1.2 (Section 5.2).

We describe a product configurator helping the user to select a product (i.e., produce a PSL specification) that satisfies the given customer requirements while conforming to the feature model of the product line (cf. Sections 5.4, 5.5 of Deliverable D1.2).

List of Authors

Taslim Arif (FRH)
Ilham Kurnia (UKL)
Radu Muschevici (KUL)
Keiko Nakata (IoC)
José Proença (KUL)
Andri Saar (IoC)
Tarmo Uustalu (IoC)
Karina Villela (FRH)
Yannick Welsch (UKL)
Contents

1 Introduction ................................................. 5

2 Java Backend ............................................... 7
   2.1 Code Generation ........................................ 7
   2.2 Runtime ................................................. 8
   2.3 Foreign Function Interface .............................. 11

3 A Scala Backend ........................................... 13
   3.1 Concurrent object groups .................................. 14
   3.2 Distributed execution .................................... 15
     3.2.1 Sample execution ................................... 15

4 Correctness of Scala backend ...................... 18
   4.1 Correctness proof ........................................ 18
     4.1.1 Source language ..................................... 18
     4.1.2 Target language ....................................... 19
     4.1.3 Compilation function .................................. 20
   4.2 Mechanized formalization of correctness proof in Agda ... 21

5 Delta Modelling .............................................. 23
   5.1 Delta Modelling .......................................... 24
   5.2 Syntax .................................................. 24
   5.3 Object-oriented modifiers .................................. 24
     5.3.1 Classes .............................................. 26
     5.3.2 Methods ............................................. 26
     5.3.3 Class interfaces ...................................... 28
     5.3.4 Fields ............................................... 28
   5.4 Functional modifiers ....................................... 28
     5.4.1 Functions ............................................ 28
     5.4.2 Data types .......................................... 29
     5.4.3 Type synonyms ....................................... 29
   5.5 Formal Semantics .......................................... 29
     5.5.1 Implementation ....................................... 31
   5.6 Unsupported modifications .............................. 31

6 End-to-end Product Derivation ..................... 33
   6.1 Motivation .............................................. 33
   6.2 General Concept .......................................... 33
   6.3 Benefits ................................................ 34
   6.4 Research Methodology ..................................... 35
   6.5 User Interaction and Interface Design .................. 36
Chapter 1

Introduction

This report documents the results of Task 1.4 “ABS System Derivation and Code Generation”. The main goal of this task was to develop and integrate the mechanisms to simulate, realize and adapt model configurations. The task included the design of the integrated infrastructure for component composition, code generation, deployment, runtime monitoring as well as dynamic adaptation. Code generation was to be founded on a solid semantical basis, so that both the method itself and generated code would be amenable to formal analysis methods. In order to maximize impact within the resources available in the project, Java was to be used as the main target language for code generation as a widely used object-oriented language.

Task 1.4 together with Tasks 1.3 (completed), 1.5 (ongoing) was first of all an implementation task meant to turn the results of Tasks 2.x, 3.x, 4.x into working solutions that can be evaluated in Tasks 5.x.

Tasks 1.1..1.5 that constitute Work Package 1 form one single chain of design and implementation efforts on the ABS modelling language. Very usefully for the project as a whole, tool development began early on in the course of the project. As a result, a lot of the work initially planned for Task 1.4 was already completed within Task 1.2 before its official start. In particular, the basic ABS compiler framework including the initial version of the main Java backend was implemented at an early stage of the project and has since then been maintained, updated and improved. The same can be said about variability modelling and platform models and configuration. The first results were reported already in Deliverable D1.2 [2] (corresponding to Task 1.2), Sections 5, 6 (variability modelling, platform models and configuration), Section 7 (code generation).

Accordingly, in this deliverable we concentrate on the changes and additions with respect to what was reported in Deliverable D1.2. These concern additions and improvements to the main Java backend of the ABS compiler framework, a new Scala backend that supports distributed execution, and additions and changes to delta modelling on top of ABS modelling. We also report a results on correctness of code generation, complementing the work done in Work Package 4 on trustworthiness of ABS models on the level of the source language.

The document is structured as follows.

In Chapter 2 we give an update on the main Java backend of the ABS compiler framework, concentrating on the progress since Deliverable D1.2, Section 7.4. In particular the chapter describes the foreign language interface for ABS. Two more experimental versions (a weakly typed version and a version supporting distribution, ABS-NET) of this backend will be described in Deliverables D3.3 and D3.5 (forthcoming).

Chapter 3 reports on a new experimental Scala backend. This is based on the Akka actor framework of Scala and Scala’s continuations mechanism. This choice allows for a mapping of the ABS concurrency model to the target language. Differently from the main Java backend of the ABS compiler, this backend supports distributed execution of different COGs on different nodes.

In Chapter 4 we discuss the correctness of compilation from ABS to Scala. Despite of the relative simplicity of the compilation scheme, correctness (semantics-preservation) is not straightforward. We concentrate on cooperative scheduling within one COG. We look at a fragment of ABS, a compilation function into a small functional language without concurrency, but providing delimited continuations, modelled after Scala. We provide a formal semantics of both languages, a formal correctness statement and a proof.
also report on the ongoing formalization of this development in the proof assistant Agda.

Chapter 5 is concerned with the changes and additions to delta modelling in ABS. The delta modelling capabilities of ABS have undergone several changes and additions since Deliverable D1.2 (Section 5.2) based on experiences obtained in case studies and user feedback. The most important of those are: added support for adding, removing and modifying interface definitions, modifying a class’s, changed semantics of deltas as modules (wrt. what a delta can see), the added feature of targeted original calls in conflict-resolving deltas.

Finally, in Chapter 6, we describe a product configurator helping the user in the selection of suitable products from a product line. The configurator assists the user in producing PLS product selection specifications satisfying the given customer requirements (required features, desired quality features, constraints) in conformance with the $\mu$TVL feature model of the given product line. Product selection is the starting point for ABS product generation according to Sections 5.4, 5.5 of Deliverable D1.2. Hence the product configurator is an essential ingredient in end-to-end product derivation in the HATS methodology: the flattened ABS code corresponding to a particular product is generated from the ABS code of the product line (core code and deltas), $\mu$TVL feature model and CL configuration (relating features to deltas) on the basis of a product selection specification in PLS.

List of Papers

This deliverable reports mostly implementation work. But the work in Chapters 4 and 6 is described in detail in the following papers.

The papers are not directly attached to the deliverable, but can be found on the web at http://www.hats-project.eu/sites/default/files/D1.4

Paper 1: Compiling cooperative multitasking of ABS to Scala This paper [8] by Andri Saar and Keiko Nakata is currently in the status of a manuscript. We plan to submit a revised version to PEPM 2012. It describes the correctness proof of the Scala backend.

Download Paper 1

Paper 2: Towards product configuration taking into account quality concerns This paper [12], written by Karina Villela, Taslim Arif and Damiano Zanardini and appearing in FMSPLE 2012, explains the use of performance, cost and security annotations in producing product selections.

Download Paper 2
Chapter 2

Java Backend

The Java backend for the ABS tool chain was initially described in Deliverable 1.2. In this chapter, we provide a more technical and detailed overview of the Java backend which includes changes that have been made since the last deliverable was written. In particular, we provide a more detailed description of the code generation and an introduction to the foreign language interface. The Java backend, depicted in Fig. 2.1, takes ABS models as input and generates runnable Java code as output. It has a built-in Java compiler so that it can also directly generate JVM bytecode. The rationale behind having a Java backend is to have an execution platform for ABS models that:

1. has a high performance and scalability so that very large ABS models can be executed and tested;
2. has a high configurability for testing and applying different scheduling strategies;
3. allows for easy observation and for integration of additional tools written in standard Java;
4. makes it possible to directly use generated code from ABS models in systems that are written in standard Java.

The Java backend uses a similar Java translation as is used by JCoBox [10], which proved to be very efficient. However, in contrast to JCoBox, the generated code of ABS has better support for configuring the scheduling strategies, for system observation, and debugging.

The implementation of the Java backend currently consists of around 11,000 lines of code (excluding blank lines and comments). It has been heavily adapted during the project; over 50 bug fixes and requests for enhancements reported by users in the HATS bug tracking system have been integrated. The backend comes in three flavors. The first one is the fast and strongly typed standard version of the backend. A second version is a (currently under development) weakly typed version that allows more flexible modification of the runtime state. For more details about the dynamic version of the backend, we refer to Deliverable D3.3 (forthcoming). The third version of the Java backend (ABS-NET) has capabilities for distribution built-in. In particular, it allows distributed execution of ABS models and migration of COGs. For more details about the ABS-NET version of the backend, we refer to Deliverable D3.5 (forthcoming).

2.1 Code Generation

The Java backend can either be used from the ABS Eclipse-based IDE, or can be used on the command line. For more detailed and up-to-date documentation of the Java backend and other tools of the project, we refer to the HATS Tools Website [http://tools.hats-project.eu](http://tools.hats-project.eu).

**Generation template** In the following, we describe how ABS elements are mapped onto elements of the Java language. For each ABS module, a Java package with the same name is used. ABS classes are mapped onto Java classes (with _c suffix). ABS interfaces are directly mapped onto Java interfaces (with _i suffix).
For ABS data types, there is no direct corresponding concept in Java. Here, the data types are mapped onto Java classes with subclasses for each constructor (no suffix). ABS functions are mapped onto Java classes (with \_f suffix) with a static apply method that has the signature of the function. Pattern matching is implemented using anonymous classes.

For example, the following command generates Java code for the ABS program PeerToPeer.abs (provided as an example in Deliverable 1.2) into the javagen directory:

```
java -cp absfrontend.jar abs.backend.java.JavaBackend -d javagen PeerToPeer.abs
```

If the command is successful it will generate Java source and JVM class files into the javagen directory. For this example, the following files are generated:

```
Catalog.java  findServer_f$1$3$4$6.class  Network_i.class  Node_c$9.class
Client_i.class findServer_f$1$3$4$5.class  Network_i.java  Node_c.class
Client_i.java  findServer_f$1$3.class  Node_c$10.class  Node_c.java
DataBase_i.java findServer_f$1.class  Node_c$11.class  OurTopology_c.class
DataBase_i.java findServer_f.class  Node_c$12.class  OurTopology_c.java
DataBaseImpl_c.class findServer_f.java  Node_c$13.class  Packet.java
DataBaseImpl_c.java Main$1.class  Node_c$14.class  Peer_i.class
File.java Main$2.class  Node_c$15.class  Peer_i.java
Filename.java Main$3.class  Node_c$16.class  Server_i.class
Filenames.java Main$4.class  Node_c$17.class  Server_i.java
findServer_f$1$2.class Main.class  Node_c$18.class
findServer_f$1$3$4$5.class Main.java  Node_c$19.class
```

If the ABS model specifies a software product line, variability is resolved prior to generating Java code. This is achieved by generating code for one specific product of the product line. On the command line, the switch \-product=<name> is used to select a specific product.

### 2.2 Runtime

The Java backend generates, for every Main block that exists in an ABS model, a corresponding Main class that contains a standard Java main method. Thus, the generated Java code can be executed like any other standard Java code by using the java command. The generated Java code relies on a runtime library (included in the absfrontend.jar), which must be provided when executing the system.
Concurrency and Execution Model  The concurrency model, based on COGs, is implemented as follows. Each COG in the ABS model is represented by an explicit COG object in the Java runtime. Each Java object that represents an ABS object refers to its COG object. Each COG has a Scheduler object that manages the tasks for the COG.

Synchronous method calls are handled by standard Java method calls. Asynchronous method calls result in the creation of Task objects. Execution of Task objects is handled by the scheduler of the target COG. The implementation is similar to JCoBox [10]. To keep the implementation simpler than the JCoBox version, however, we are currently using plain threads instead of thread pools.

To execute the code generated from the PeerToPeer.abs example, one can use the following command on the command line:

```java
java -cp javagen:absfrontend.jar PeerToPeer.Main
```

As ABS does not natively support I/O operations, executing an ABS models on the command line does not produce any output. In this case this command will just finish without further information. In case an assert statement fails during the execution, a corresponding error message will be shown. For example:

```
Error in Task (1) [COG [Main] (1), Method: Main.main block]:
PeerToPeer.abs:181:3: Assertion failed
```

Debugging and Monitoring Support  In order to be able to observe an ABS system, the Java backend supports a flexible observation mechanism. This mechanism allows the user to write system observers in Java. Observer interfaces, which are provided by the ABS Java runtime, have to be implemented by concrete observers for the handling of observable events. View interfaces are provided by the runtime to query the state of the running system and to register observers. To allow a flexible definition of different observers, the runtime provides various observation interfaces: SystemObserver, ObjectCreationObserver, ObjectObserver, TaskObserver. In the following we give a more detailed account of the TaskObserver:

```java
public interface TaskObserver {
    void taskStarted(TaskView task);
    void taskFinished(TaskView task);
    void taskBlockedOnFuture(TaskView task, FutView fut);
    void taskRunningAfterWaiting(TaskView view, FutView fut);
    void taskStep(TaskView task, String fileName, int line);
    void taskDeadlocked(TaskView task);
    void stackFrameCreated(TaskView task, TaskStackFrameView stackFrame);
    void localVariableChanged(TaskStackFrameView stackFrame, String name, ABSValue v);
}
```

The corresponding view that allows to query the state of a task looks as follows:

```java
public interface TaskView {
    TaskView getSender();
    ObjectView getSource();
    ObjectView getTarget();
    COGView getCOG();
    String getMethodName();
    List<ABSValue> getArgs();
    FutView getFuture();
    void registerTaskListener(TaskObserver listener);
    int getID();
}
```
To implement an observer, the following steps are necessary. First, the `SystemObserver` interface must be implemented, which provides the hook to integrate the observer into the system. Observers can be specified on the command line by using the `-Dabs.systemobserver=<observerlist>` parameter. It is possible to specify several observers, which allows the user to use multiple observers at the same time, e.g., visualizing the state by using the graphical debugger and see the high-level communication by using the UML sequence chart generator. The following command executes the PeerToPeer example and visualizes the system by using the graphical observer. In this case, the `-Dabs.debug=true` option must be given to be able to observe the stepping of tasks.

```
java -cp javagen:absfrontend.jar -Dabs.debug=true \
   -Dabs.systemobserver=abs.backend.java.debugging.GraphicalDebugger PeerToPeer.Main
```

Currently, a number of observer implementations are available:

1. A **Console Observer**, which prints information on the standard output.
2. A **Graphical Debugging View**, which shows the system state using a Java-based graphical user interface. It shows information about the object-state of the system as well as the state of all executing tasks including their source positions.
3. An **UML Sequence Chart Generator**, which generates UML sequence charts. It provides a high-level view of the communication between COGs. The sequence charts are visualized by using a modified version of the existing tool SDEdit\(^1\).
4. Various internal observers are used within the implementation of the Eclipse-based IDE. For example, an observer is used to record histories of system runs.

### Debugging ABS Systems

The Java backend provides a flexible configuration mechanism to define the scheduling strategies that are used during the execution of an ABS system. This is important as ABS is a non-deterministic language and the scheduling is crucial for the execution of the system. Currently, the following scheduling strategies are implemented:

1. The **default scheduler**, which executes the system in a non-deterministic way.
2. A **random scheduler**, which also executes the system in a non-deterministic way, but uses a configurable random seed. With a given seed the system is actually executed deterministically, so that a system execution can be reproduced by providing the seed.
3. An **interactive scheduler**, which provides a graphical user interface that allows the user to make scheduling decisions manually.
4. A **replay scheduler**, which replays a given scheduling history. Such a history can be created by a user by using a scheduler and then saving the scheduling history to a file.

Schedulers are specified by using the command line option `-Dabs.totalscheduler=<scheduler>`. Schedulers and observers can be combined in an arbitrary way. A typical use scenario is to use the interactive scheduler in combination with the graphical debugger view:

\(^1\)[http://sdedit.sourceforge.net/](http://sdedit.sourceforge.net/)
2.3 Foreign Function Interface

The foreign language interface (FLI) of the Java backend makes it possible for ABS code to interact with legacy Java code, e.g., by using existing library code written in the Java language. The connection is done by annotating ABS classes with the [Foreign] annotation. These classes must then be implemented in the Java language.

To declare a class as a foreign class one just has to annotate it with the [Foreign] annotation, which is defined in the ABS.FLI module. It is also necessary to define an interface which is implemented by the foreign class. Finally, the class must not have any class parameters.

```java
import * from ABS.FLI;

interface Hello {
    String hello(String msg);
}

[Foreign]
class HelloImpl implements Hello {
    String hello(String msg) {
        return "default implementation, no foreign language was found at runtime";
    }
}

Hello h = new HelloImpl();
h.hello("Hi there");
```

The foreign class must provide a default implementation for the implemented interface. This default implementation is used at runtime when no foreign implementation is available (e.g., on other backends). After the Java code for the ABS model has been generated, a class `Test.HelloImpl_c` is available. This class now has to be extended by a standard public Java class. The Java class must have a public parameterless constructor (or no explicit constructor). To implement the behavior of the class, the programmer has to override the methods of the superclass with names that are prefixed by `fli_`.

The print methods defined in the FLIHelper class can be used to print messages to the console that will appear on the command line as well as the Eclipse console.

```java
package myjavaflitest;

// included in the absfrontend.jar
import abs.backend.java.lib.types.ABSString;

// generated by the ABS Java backend
import Test.HelloImpl_c;

public class FLITest extends HelloImpl_c {
    @Override
```
public ABSString fli_hello(ABSString msg) {
    FLIHelper.println("I got "+msg.getString()+" from ABS");
    return ABSString.fromString("Hello ABS, this is Java");
}

After implementing the foreign class in Java, the Java implementation must be connected to the corresponding ABS class. This is done at runtime when executing the generated Java code by using Java properties. The following three possibilities to establish this connection are supported:

Using naming conventions The easiest way to establish the connection is by naming convention. In that case the overriding Java class must be defined in a package that corresponds to the ABS module of the ABS class and must be named like the ABS class with a _fli suffix. For example, if an ABS class SomeClass is defined in a module My.Module then the Java class must be in a package My.Module and must be called SomeClass_fli. The methods that need to be overridden are those that start with fli_ followed by the name of the ABS method.

Using system properties Instead of using the convention mechanism, one can use system properties. The properties have to be of the form

abs.fli.class.ABSCLASSNAME=JAVACLASSNAME

where ABSCLASSNAME is the fully qualified name of the ABS class, and JAVACLASSNAME is the fully qualified name of the corresponding Java class. For the example above, the following needs to be specified:

java -cp ... -Dabs.fli.class.Test.HelloImpl=myjavaflitest.FLITest Test.Main

Using a separate properties file Another possibility is to specify the mapping of ABS classes to Java classes in a separate properties file. In that case the keys are just the fully qualified ABS class names and the values are the fully qualified Java class names. For example, a file abstojava.properties could look as follows:

Test.HelloImpl=myjavaflitest.FLITest

The system property abs.fli.properties is used to specify the name of the properties file:

java -cp ... -Dabs.fli.properties=abstojava.properties Test.Main

Note that the abstojava.properties file must available on the Java classpath. If there is a file with the exact name absfli.properties available on the Java classpath, this file is taken automatically.
Chapter 3

A Scala Backend

The Scala backend is motivated to serve as the first backend to support distributed execution of ABS models. The implementation is rather light weight. Scala is a hybrid functional and object-oriented programming language, featuring, among others, case classes, pattern matching and higher-order functions. This makes Scala a good match for ABS: large part of the code generation is simple syntactic translation. For distributed execution, we employ an industrial strength Akka actor library.

The ABS Scala backend generates Scala code from ABS models which can then be compiled with the standard Scala compiler and executed on the Java virtual machine, see Fig. 3.1. The concurrent objects in ABS are translated to Akka actors that exchange asynchronous messages between each other and various system components (which are also implemented as Akka actors), thus enabling distributed execution of ABS models. The code generator is integrated into the standard ABS Tool Suite and includes support for the Maven dependency management system. For modelers using Maven, executing a model requires only slight modifications of the project’s POM (adding the Scala compiler plugin to the build process and changing the target platform for the ABS plugin).

Every entity that needs to communicate with any other entity in the system (concurrent objects, object groups, tasks, etc.) are implemented as Akka actors that communicate via asynchronous messages. On a high level, the architecture of the system is as follows:

- Every concurrent object belongs to a concurrent object group (COG).
- Invoking a method of a concurrent object creates a new task in the system, tied to the COG of the object.
- COGs take care of scheduling of tasks.
- Tasks double as futures: other entities can register themselves as asynchronous listeners and will receive a notification when the task completes.

The internal state of tasks and concurrent object groups can be modeled as finite state machines.

![Figure 3.1: Compilation to Scala and Java bytecode](image-url)
3.1 Concurrent object groups

Concurrent object groups keep track of the tasks in the system and handle the scheduling of tasks. One COG may have at most one task running at any time. The task may willingly give up its right of execution (by finishing or yielding) or block on some condition; if the task blocks, the COG is unable to schedule some other task for execution and must wait until the condition is fulfilled and the original task can finish its execution.

An overview of the internal states and messages that can cause transitions between the states is shown in Fig 3.2. Initially, the COG starts in the Idle state, with no tasks. The most important message is the Work message: if the COG is in the idle state and it receives a Work message, it iterates through its list of tasks and if there is some task that can be executed, starts the execution of that task and goes to the Busy state. If the COG receives any Work messages while in the Busy state, they are ignored.

A task that has been executed can send two messages back to the COG: either Done, which means that the task has finished (for now—this also covers yielding) or Blocked, which means that the task is waiting on some future to become ready. If the task has finished, the COG transitions back to the Idle state and sends a Work message to itself. If the task has become blocked, the COG transitions to the Blocked state and awaits for Work messages. When it receives a Work messages, it checks the condition of the task to see whether the task is no longer blocked: if not, it transitions back to the Busy state and resumes the execution of the task. Otherwise, the COG will stay in the Blocked state awaiting further Work messages.

Whenever a method of a concurrent object is invoked, the object will send a Run message to the COG. When the COG receives a Run message, it creates a new task, appends it to the internal list of tasks and sends a Work message to itself. A Run message will never cause a state transition directly (although the Work message may trigger a transition). A reference to the newly created task is returned to the sender, as this can be used as a future.

Tasks

Tasks handle the execution of actual code and double as futures, keeping track of interested parties and notifying them when the task has finished. In the Idle state, a task has a guard: a condition that must be
true if this task is to be executed. Initially, the guard is simply \texttt{true} and the task may be scheduled by the COG at any time.

Similar to COGs, the internal state of a task can be modeled as a state machine, shown in Figure \ref{fig:task-state-machine}. When the task receives a Work message, it fires off a separate runner thread that executes the code of the method. This is done so that the task object itself can respond to incoming messages and respond to the senders; if the task executed the code directly, we would run into the risk of getting deadlocks. The runner is also an actor, with only one possible message, Run, sent by the task when transitioning to the Running state. Once the task has told the runner to start, it will not send any other messages to the actor until it receives a message from the runner stating that the execution has stopped.

When the runner reaches a statement that depends on the outer system—awaiting on some future to become ready or resolving a future (possibly blocking until it becomes ready)—the runner does not save the rest of the computation into a continuation, storing it until the condition is satisfied. This has the positive effect that blocked tasks do not hold threads and resuming a suspended computation is easy.

There are three possible reasons for a runner to stop the execution:

- The code has finished. In this case, the task will transition to the Finished state, notifies any listeners that the value is ready, and sends a message to its COG stating that the work is done.
- The code is (asynchronously) awaiting on some condition. The task saves a function to check the condition in the guard and notifies the COG that it is finished for now.
- The code is blocked on some condition. Similar to the previous case, but the COG will receive a message noting that the task is blocked.

As the task doubles as a future, there are two other messages that the task can receive: Get and Listen. The first message, Get, will send back a message containing the result of the computation (if the task is in the Finished state) or notify the sender that the task is not finished yet. Listen is used for asynchronous notification of COGs: interested parties will be saved to a internal list of listeners and when the task transitions to the Finished state, it will send a Work message to all of the listeners. If the task receives a Listen message while in the Finished state, it will immediately respond with a Work message.

### 3.2 Distributed execution

The Scala backend supports executing COGs on different nodes. This is possible thanks to the remoting features provided by Akka: all actor references are location independent and thus actors do not care whether the other actor they are communicating with resides on the same node or on a different node.

In ABS source code, the distribution is exposed via the CogLocation annotation, where the developer can specify the node when creating a new COG. Every node participating in concurrent execution has a node manager actor, with a well-defined path, that manages the creation of COGs in that particular node. If the CogLocation annotation is not present at the new cog statement, the new COG is created by the node manager residing on the current node; otherwise, a path to the remote node manager is built from the information contained in the annotation and the remote node manager creates the COG. Once the new COG has been created, the system is completely oblivious whether the COG resides on the same node or some remote node.

Right now the backend requires that the COG actor, all concurrent object actors and tasks related to those objects reside on the same node. Migration of COGs or objects between nodes is not supported.

#### 3.2.1 Sample execution

As an example of how to use the Scala backend, we examine the distributed PingPong example:

```scala
module PingPong;
import CogLocation from ABS.DC;
```
data PingMsg = Fine | HelloPing | ByePing;

data PongMsg = NoMsg | Hello(Ping) | HowAreYou | ByePong;

interface Ping {
    Unit ping(PingMsg m);
}

interface Pong {
    Unit hello(Ping ping);
    Unit pong(PongMsg m);
}

class PingImpl(Pong pong) implements Ping {
    Unit run() {
        pong!hello(this);
    }

    Unit ping(PingMsg msg) {
        PongMsg reply = case msg {
            HelloPing => HowAreYou;
            Fine => ByePong;
            ByePing => NoMsg;
        };

        if (reply != NoMsg) {
            Fut<Unit> fu = pong!pong(reply);
            fu.get;
        }
    }
}

class PongImpl implements Pong {
    Ping ping;

    Unit hello(Ping ping) {
        this.ping = ping;
        ping!ping(HelloPing);
    }

    Unit pong(PongMsg msg) {
        if (msg == HowAreYou)
            ping!ping(Fine);
        else
            ping!ping(ByePing);
    }
}

Pong pong = new cog PongImpl();
This example program creates two concurrent objects, Ping and Pong, both running concurrently in different concurrent object groups. With the CogLocation annotation we denote that we want the PingImpl object to be created on the remote node identified by the host name host.example.com.

In order to execute the program, we need to start up the NodeManager on the remote host. Assuming that the Maven build system is used, this can be done by issuing the following command:

```shell
mvn scala:run -DmainClass=abs.backend.scala.runtime.NodeManager
```

The NodeManager will start listening for incoming connections. Assuming that the default log level is used, you may see a message similar to the following in the output:

```
[INFO] [08/17/2012 10:41:41.696] [main] [ActorSystem(ABS-Scala)]
   REMOTE: RemoteServerStarted@akka://ABS-Scala@host.example.com:2552
```

On the main node, you can start the program by executing

```shell
mvn scala:run -DmainClass=PingPong.Main
```

As ABS programs can’t output anything on their own, it is advisable to execute the program with the logging level raised to DEBUG.
Chapter 4

Correctness of Scala backend

This chapter continues on the Scala backend from the previous chapter. In the previous chapter, we described how the Scala backend supports the concurrency between COGs including distributed execution. In this chapter, we look at cooperative multitasking within one COG and explain how it is implemented in the Scala backend.

Inside one COG we have cooperative multitasking with the control being released only at certain scheduling points: the await e and the suspend statements. The statement await e suspends the execution of the current task until the guard expression e evaluates to true. The suspend statement yields the control. These statements are a scheduling point: the scheduler may decide to switch tasks and proceed with some other task. In order to implement this efficiently we use the experimental continuations plugin for Scala. Every time we reach a scheduling point, we capture the rest of the task in a continuation, store it together with its guard in the task set and give control back to the scheduler of the COG. The scheduler will then go through its list of tasks, evaluate the guards and pick a new task whose guard evaluates to true. If there are no such tasks, then none of the tasks in the COG are executable at that moment. In this case, the COG blocks until some external event causes some task to be ready for execution.

In order to provide a provably correct code generator for ABS to Scala, as the first step, we look at what happens inside one concurrent object group and define a simplified language based on ABS that lacks objects but retains asynchronous calls and cooperative scheduling between tasks. For this language we define a compilation function, modelled from our implementation, into a target language inspired by Scala that lacks concurrency but provides delimited continuations and closures. We then proceed to prove, in a paper and pencil fashion, that compiling programs in the source language to a program in the target language preserves the operational behaviour. This gives us a proof-of-concept for the correctness of our compilation methodology. We are currently formalizing the correctness proof for a larger subset of ABS, using a dependently typed programming language Agda. In the following subsections, we elaborate on these.

4.1 Correctness proof

We have designed two languages, both endowed with small-step operational semantics, that are simpler than both ABS and Scala, yet exhibit the behaviour we are interested in. The source language models one COG in ABS by having asynchronous calls, await statement and cooperative multitasking. The target language includes primitive statements to manage the task set and supports continuations as the method to switch between tasks, which are used to implement a scheduler.

4.1.1 Source language

The source language is designed to be as minimalistic as possible while still retaining the interesting properties: thus, we have removed objects, COGs and futures from the language, resulting in a simple imperative

\[1\]This is a simplification: if the guard immediately holds, we are not allowed to switch tasks in ABS.
programming language with asynchronous procedure calls and a way to wait for some condition to become true.

The statements $S$ of the language are defined by

$$S ::= x := e \mid \text{skip} \mid S_1 ; S_2 \mid \text{await } e \mid \text{if } e \text{ then } S_1 \text{ else } S_2 \mid \text{while } e \text{ do } S \mid f(\tau)$$

Besides statements, we have pure expressions $e$ which consist of usual arithmetic and boolean expressions. The notation $\tau$ (resp. $\pi$ and $\tau$) denotes a possibly empty sequence of expressions (resp. variables and values).

A program $P$ consists of a function environment $env_F$ which maps function names to pairs of a formal argument list and a statement, and global variable declarations with their initial values. The entry point of the program will be the procedure named $\text{main}$.

The scheduler may only switch between tasks when evaluating an $\text{await } e$ statement or when the currently running task terminates. Procedure calls $f(\tau)$ return immediately with no result and the original task will proceed. Communication between different tasks is achieved via global variables (which may be seen as analogues to object fields in ABS).

During runtime, the program is represented as a configuration $c$ that consists of an active task identifier $n$, a global variable mapping $\sigma$ and a set of tasks $Ts$. A task has an identifier and may be in one of the three forms: (a) a triple $\langle c, S, \rho \rangle$, representing a task that is awaiting to be scheduled, where $c$ is the guard expression and $\rho$ is the local variable mapping; (b) a pair $\langle S, \rho \rangle$, representing the currently active task and (c) a singleton $\langle \rho \rangle$, representing a terminated task.

$$\text{Configuration} \quad c ::= n, \sigma \triangleright Ts$$

$$\text{Task sets} \quad Ts ::= n\langle e, S, \rho \rangle \mid n\langle S, \rho \rangle \mid n\langle \rho \rangle \mid Ts || Ts$$

The order of tasks in the task set is irrelevant: the parallel operator $||$ is commutative and associative. Transition rules in the semantics are in the form $env_F \vdash c \rightarrow c'$, A subset of the rules is shown in Figure 4.1. We assume an evaluation function $\llbracket e \rrbracket_{\rho, \sigma}$, which evaluates $e$ with respect to the local and global variable mappings $\rho$ and $\sigma$.

The first rule in Figure 4.1 allows us to focus on a subset, of the whole task set, that advances the transition. If the currently active task is in the form $\langle S, \rho \rangle$, then we proceed to evaluate the statement $S$. Otherwise, it has terminated, i.e., it is in the form $\langle \rho \rangle$, or it has reached a scheduling point, i.e., in the form $\langle e, S, \rho \rangle$. Then we may pick another task from the task set whose guard evaluates to $\text{true}$. When a procedure $f$ is invoked, we evaluate the argument expressions $\tau$ and put a new task $n'(\text{true}, S, [\tau \mapsto \tau])$ to the task set, where $S$ is the body of the procedure, $[\tau \mapsto \tau]$ maps formal argument names $\tau$ to their values $\tau$ and $n'$ is a fresh identifier. The execution will proceed in the active task and the new task may only get scheduled once we reach a scheduling point.

### 4.1.2 Target language

The target language is modeled on Scala. Instead of containing explicit support for scheduling points, the target language supports delimited continuations via the $\text{shift}$ and $\text{reset}$ statements and the scheduler is explicitly specified when compiling programs from the source language to the target language. In order to not clutter the language with arrays, we also include four primitives that handle adding and picking tasks from a
\[
\begin{align*}
\text{env}_F &\vdash (S, \rho, \sigma) \Rightarrow (S', \rho', T', \sigma') \\
\text{env}_F &\vdash (E[S], \rho, T, \sigma) \Rightarrow (E[S'], \rho', T', \sigma')
\end{align*}
\]

\text{E does not cross reset}

\[
\begin{align*}
\text{env}_F &\vdash \{\text{reset} \{E[\text{shift } k \{S\}]\}, \rho, T, \sigma\} \Rightarrow \{\text{reset} \{S\}, \rho\}
\end{align*}
\]

\[
\begin{align*}
\text{env}_F &\vdash \{\text{reset} \{E[\text{skip}]\}, \rho, T, \sigma\} \Rightarrow \{\text{reset} \{E[\text{skip}]\}, \rho, T, \sigma\}
\end{align*}
\]

\[
\begin{align*}
\text{env}_F &\vdash \{\text{reset} \{E[\text{getTask}()]\}, \rho, T, \sigma\} \Rightarrow \{\text{return } \rho', T, \sigma\}
\end{align*}
\]

\[
\begin{align*}
\text{env}_F &\vdash \{\text{addFunc } f \{\bar{\tau}\}, \rho, T, \sigma\} \Rightarrow \{\text{skip}, \rho, T \cup (e, [S, \rho'])\}
\end{align*}
\]

Figure 4.2: Reduction semantics for the target language (excerpt)

task set. During runtime, we make use of closures and the return statement, which may only appear during evaluation but may not be used in programs. Closures are represented as pairs \([S, \rho]\) of a statement and a variable mapping. The special return statement allows us to restore local variable mappings. The language also contains the usual arithmetic and boolean expressions, and simple loop and branching statements, just as our source language.

\[
\begin{align*}
\text{Expressions} \quad e &::= \ldots | [S, \rho] \\
\text{Statements} \quad S &::= \ldots | \text{reset } \{S\} | \text{shift } x \{S\} | \text{invoke } x | \text{return } \rho
\end{align*}
\]

We introduce three primitive statements and one primitive expression for operating on the task set. A task is represented as a pair of a guard (a delaying expression) and a closure. The statement addFunc \(f \{\bar{\tau}\}\) will add to the task set a pair of a guard \(\text{true}\) and a closure representing a call to the procedure \(f\) with \(\bar{\tau}\) immediately evaluated and used as the arguments to the procedure. The statement addClosure \(e x\) will add a pair of a guard \(e\) and the closure stored at \(x\) to the task set. The expression TaskAvail evaluates to true if there is any task in the task set whose guard holds. The statement getTask() invokes a closure from the task set whose guard evaluates to true. When there are several valid tasks, which of the tasks will get invoked is unspecified. In our Scala backend, these primitives are implemented using a global array and first-class functions.

We specify the operational semantics of the target language using evaluation contexts

\[
\begin{align*}
\text{Evaluation contexts} \quad E &::= [] | E; S | \text{reset } \{E\}
\end{align*}
\]

The configurations in the target semantics are in the form \((S, \rho, T, \sigma)\) where \(S\) is the statement, \(\rho\) is the local variable mapping of the current task, \(T\) is the task set, and \(\sigma\) is the global variable mapping. The function environment \(\text{env}_F\) is a mapping from function names to pairs of parameter lists and statements, similar to the source language. The reduction rules are in the form \(\text{env}_F \vdash c \mapsto c'\), and a subset of the rules is shown in Figure 4.2.

Due to the evaluation context, informally, we will always reduce the leftmost statement, ignoring the enclosing resets. The shift statement is used to capture the continuation up to the nearest enclosing reset.

4.1.3 Compilation function

The compilation function generates a program in the target language for every program in the source language. The interesting part of the definition is given by

\[
\begin{align*}
\llbracket \text{fun } f(\bar{x}) \text{ is } S \rrbracket &\Rightarrow \text{fun } f(\bar{x}) \text{ is } \{S\}; \text{invoke } \text{sched} \\
\llbracket \text{await } c \rrbracket &\Rightarrow \text{shift } k \{\text{addClosure } e k; \text{invoke } \text{sched}\} \\
\llbracket f(\bar{\tau}) \rrbracket &\Rightarrow \text{addFunc } f \{\bar{\tau}\}
\end{align*}
\]

20
Every compiled program contains a prelude, which declares a global variable `sched` that will contain the scheduler continuation and a statement that adds the `main` function to the task set. The control is then passed on to the scheduler:

```
var sched : Closure = [skip, []];
addFunc main ();
reset {while TaskAvail do shift k {sched = k; getTask()}}
```

The scheduler checks if there is any task ready to be executed (TaskAvail); if there is, the rest of the computation (which means invoking the while loop again) is stored in the global variable `sched`. When the task terminates, the closure is invoked and the control passes back to the scheduler, which will proceed by trying to find another suitable task for execution. A similar pattern is followed with the `await e` statement: we store the rest of the execution in a continuation, add the continuation with its guard to the task set and pass the control back to the scheduler.

The following theorem states that the compilation function is correct: a one-step reduction in the source language is simulated by possibly multiple-step reduction in the target language after the compilation. To state the theorem formally, we extend the compilation function to configurations and define a similarity relation `≤`, which disregards the number of adjoining resets, between configurations in the target language.

**Theorem 1 (Correctness of compilation)** For all configurations `c_S`, `c'_S` in the source language and `c_T` in the target language such that `c_S → c'_S` and `∥c_S∥ ≤ c_T`, there exists a configuration `c'_T` in the target language such that `c_T ↦→ + c'_T` and `∥c'_S∥ ≤ c'_T`.

### 4.2 Mechanized formalization of correctness proof in Agda

We are currently formalizing the correctness proof for a larger subset of ABS, using the dependently typed programming language Agda [11]. Agda is an interactive proof assistant, namely a computer program that can verify mathematical proofs. It is based on the Curry-Howard (aka. proofs-as-programs) correspondence. It comes with a very expressive programming language in which one can write and prove logical assertions. Notably, one can write ordinary programs in Agda, which are correct by construction, where the correctness is machine-verified. This is in contrast to paper-and-pencil proofs which are more prone to mistakes, e.g., omission of limit cases. Mechanizing the correctness proof is particularly interesting for us since the compiler is a intricate piece of software, which involves syntax of a programming language, in our case ABS, that has many possible instructions.

To scale up the proof-of-concept to the actual compiler, we take a larger subset of ABS in the Agda formalization. We include COGs, a yield statement for explicitly introducing scheduling points, an avail statement for polling the message queue, (synchronous) method invocation, asynchronous message passing between COGs and spawning new tasks within the same COG and in a new COG. We choose slightly different primitives from ABS to simplify our Agda formalization, but most of ABS language constructs (except for the guard construct) are expressible as macros. We adopt the strongly typed term representation to define the syntax of the language. This can be seen that we define the syntax and the type system simultaneously, so that only well-typed terms can be constructed. For instance, the syntax of expressions is parameterized by contexts for functions (FCtx), variables (Ctx), constants (CCtx) and tasks (TCtx) and defined by two mutually recursive datatypes ExpList and Exp:

```agda
data ExpList (Δ : FCtx) (Γ : Ctx) (Ξ : CCtx) (T : TCtx) : Ctx → Set where
  [] : ExpList Δ Γ Ξ T []
  _::_ : ∀ {σ Σ} → Exp Δ Γ Ξ T σ → ExpList Δ Γ Ξ T Σ → ExpList Δ Γ Ξ T (σ :: Σ)

data Exp (Δ : FCtx) (Γ : Ctx) (Ξ : CCtx) (T : TCtx) where
  var : ∀ {τ} → Var Γ τ → Exp Δ Γ Ξ T τ
```

21
The datatype `ExpList` denotes lists of expressions (typed under the same contexts) and the constructors of the datatype `Exp` define syntactic constructs forming expressions. For instance, the `val` and `const` constructors look up the respective contexts `T` and `Ξ`. The `spawn` constructor takes a function name (from the context `Δ`) and a list of expressions as arguments to the function. To form an if-then-else expression, one has to supply to the `If_then_else` constructor an expression of type `bool` (`Bo`) and two expressions of the same type `τ`.

Constant and variable contexts are simply lists of types, whereas a function context is a list of pairs of parameter types and the return type. A task context is a pair of a number denoting the number of COGs in the system and a vector of natural numbers of that length, containing the number of tasks in each COG:

\[
\text{TCtx} : \text{Set} \\
\text{TCtx} = \exists \lambda \text{mgid} \to \text{Vec} \mathbb{N} \text{zero mgid}
\]

We have defined and formalized in Agda an operational semantics for this source language, and proved type preservation. In fact, since we can only construct well-typed terms, type preservation is obtained for free (as soon as we are able to define the operational semantics.) We are currently proving the progress property (in a suitable formulation since the type system does not guarantee the absence of synchronization errors such as deadlock.)
Chapter 5

Delta Modelling

The ABS language directly supports the modelling of software product lines and automatic derivation of products. This is achieved through the functional interaction of several language extensions. The $\mu$TVL extension is designed for encoding feature models, and the delta modelling extension allows the encapsulation of variable behavioural (code) artifacts. Features and code artifacts are connected via a configuration language. These language features are detailed in Chapter 5 of Deliverable 1.2 [2]. With this infrastructure in place, a user only needs to select a product of the product line; the executable model for that product is generated automatically and without any further user intervention.

Delta modelling is a new programming paradigm, and its potential and areas of application have not been fully explored. The ABS delta modelling framework is constantly evolving based on experiences gained in case studies and from user feedback. These changes, which improve the usability of ABS when modelling variable systems such as software product lines, serve two goals:

1. Complement and complete ABS delta modelling capabilities; and
2. Offer a simplified and more streamlined modelling experience.

This chapter documents the current variability modelling capabilities of ABS and presents their syntax and formal semantics with an emphasis on the additions and changes that occurred since Deliverable 1.2 [2]. All items described here have been implemented. The main changes and additions are:

- Support for adding, removing and modifying interface declarations
- Support for modifying a class’s interface
- Support for adding functions, data types and type synonyms
- Deltas as modules. Initially, delta modules were designed as elements within modules in the same way as, for example, classes reside inside modules. This raised practical issues related to deltas being subject to the namespace restrictions imposed by modules. As practice has shown, deltas are often used to implement the behaviour of (parts of) a feature. Hence they modify an ABS model in a cross-cutting manner, often affecting several classes and modules. Accordingly, the semantics of deltas has been changed to provide them with an all-seeing view of the model.
- Targeted original calls. Deltas are typically used to implement individual features of a software product line. Different features sometime modify a core program in incompatible ways. To use such features together in a product, the conflicts have to be resolved. This can be done by defining conflict resolving deltas, which modify the specific code of the features in conflict. Targeted original calls help getting to the specific code. They specify which version of an “original” method implementation to invoke by prefixing the call with the name of a delta module.
5.1 Delta Modelling

ABS implements the delta-oriented programming model [9], an approach that aids the development of a set of programs simultaneously from a single code base, following the software product line engineering approach [7]. In delta-oriented programming, features defined by a feature model are associated with code modules that describe modifications to a core program. In ABS, these modules are called delta modules. Hence the implementation of a software product line in ABS is divided into a core and a set of delta modules.

The core consists of a set of ABS modules containing the classes that implement a complete software product of the corresponding software product line. Delta modules (or deltas in short) describe how to change the core program to obtain new products. This includes adding new classes and interfaces, modifying existing ones, or even removing some classes from the core. Delta modules can also modify the functional entities of an ABS program, that is, they can add functions, data types and type synonyms.

Deltas are applied to the core program by the ABS compiler front end. The choice of which delta modules to apply depends on the selection of a set of features, that is, a particular product of the SPL. The role of the ABS compiler front end is to translate textual ABS models into an internal representation and check the models for syntax and semantic errors. The role of the compiler back end is to generate code for the models targeting some suitable execution or simulation environment.

5.2 Syntax

Figure 5.1 specifies the ABS syntax related to delta modeling. Non-terminals written in purple (gray) refer to core ABS symbols, whose intended meaning should be immediate. We focus on the evolution of the grammar since it was presented in Deliverable 1.2 [2]. The DeltaDecl clause specifies the syntax of delta modules, consisting of an unique identifier, a module access directive, a list of parameters and a sequence of module modifiers. The newly introduced module access directive gives the delta access to the namespace of a particular module. In other words it specifies the ABS module to which the modifications specified by the delta apply by default. A delta can still apply changes to several modules by fully qualifying the TypeName of module modifiers.

The ModuleModifier clause describes the syntax of modifications at the level of modules. Such a modification can add a class or interface declaration, modify an existing class or interface, remove a class or interface, and also add functions, data types and type synonyms. Class modifications now include the ability to change the interface of a class by adding or removing items from the class’s list of implemented interfaces. The InterfaceModifiers clause describes how to modify existing interface declarations, either by adding new or removing existing method signatures.

The Modifier clause specifies the modifications that can occur within a class or interface body. These include adding and removing fields and methods, and modifying methods, which amounts to replacing a method implementation with a new one, while enabling the original method to be called using the original keyword. The aim of original is to enable the method being replaced to be called from the delta module that replaces it. This is implemented by renaming the original method, and replacing the call via keyword original with a call to the renamed method. As replacement of a method can occur multiple times by applying a succession of deltas, a targeted version of the original call was introduced. The target here is a specific delta or the core program. This gives the user a tighter control over the program’s behaviour. It also makes the code less flexible because calling original on a particular target delta introduces the assumption that the target delta has been already applied. Such a dependency could be invalidated for instance by changes to the SPL configuration, which dictates which deltas should be applied when certain features are selected.

5.3 Object-oriented modifiers

To modify an object-oriented ABS program, delta modules support adding new classes and removing existing ones. Existing classes can be also modified by adding new methods and also removing or modifying existing
\[ \text{DeltaDecl} ::= \text{delta} \text{TypeId} [\text{DeltaParams}] ; [\text{ModuleAccess}] \text{ModuleModifier}^* \]

\[ \text{ModuleModifier} ::= \text{adds} \text{ClassDecl} \]
\[ \text{removes class} \text{TypeName} ; \]
\[ \text{modifies class} \text{TypeName} \]
\[ \text{adds} \text{TypeId} (, \text{TypeId}^*) [\text{removes} \text{TypeId} (, \text{TypeId}^*)] \]
\[ \{ \text{Modifier}^* \} \]
\[ \text{adds} \text{InterfaceDecl} \]
\[ \text{modifies interface} \text{TypeName} \{ \text{InterfaceModifier}^* \} \]
\[ \text{adds} \text{FunctionDecl} \]
\[ \text{adds} \text{DataTypeDecl} \]
\[ \text{adds} \text{TypeSynDecl} \]

\[ \text{InterfaceModifier} ::= \text{adds} \text{MethSig} ; \]
\[ \text{removes} \text{MethSig} ; \]

\[ \text{Modifier} ::= \text{adds} \text{FieldDecl} \]
\[ \text{removes} \text{FieldDecl} \]
\[ \text{adds} \text{MethDecl} \]
\[ \text{removes} \text{MethSig} \]
\[ \text{modifies} \text{MethDecl} \]

\[ \text{DeltaParams} ::= (\text{DeltaParam} (, \text{DeltaParams})^*) \]

\[ \text{DeltaParam} ::= \text{Identifier HasCondition}^* \]
\[ \text{Type Identifier} \]

\[ \text{ModuleAccess} ::= \text{uses} \text{TypeId} ; \]

\[ \text{HasCondition} ::= \text{hasField} \text{FieldDecl} \]
\[ \text{hasMethod} \text{MethSig} \]
\[ \text{hasInterface} \text{TypeId} \]

Figure 5.1: ABS Grammar: Delta Modules.
methods. Deltas can also introduce new interface declarations and modify existing ones by adding or removing operations. Furthermore, deltas can change the interface of a class by adding or removing interfaces from the class’s list of implemented interfaces. Lastly, delta modules can introduce new fields to classes or remove existing fields.

5.3.1 Classes

Deltas can introduce new classes and remove existing classes. The syntax is illustrated by the following examples.

```
delta D1;
adds class MyModule.DataBase(Map<Filename,File> db) implements DB {...}
```

```
delta D2;
uses MyModule;
removes class Node();
```

The first delta D1 above declares a new class `DataBase` inside the module `MyModule`. Delta D2 removes the class `Node` from the same module. Specifying to which module such code modifications apply can be done in two ways. First, as exemplified by delta D1, the class name can be qualified with a module name. An alternative way is to include a `uses` `<Module Name>` clause at the beginning of the delta module, which opens a module so that names don’t need to be qualified. When a delta specifies modifications to a single module, this method is more concise. When a delta specifies modifications across multiple modules, it is more convenient to qualify each class modifier with a module name. Using both methods together is also possible, in which case unqualified class names will refer to classes defined inside the used module.

Deltas can also modify existing classes by adding new methods and removing or modifying methods; by adding or removing fields; and by manipulating the list of interfaces that the class implements. These operations are illustrated in the following sections.

5.3.2 Methods

Methods can be added, removed or modified from within deltas. The following example shows a delta module designed to modify the behaviour of the class `Greeter` by modifying its `sayHello` method. The class is assumed to have been declared in the core program inside the `Hello` module.

```
delta N1;
uses Hello;
modifies class Greeter {
  modifies String sayHello() {
    return "Hallo wereld";
  }
}
```

The above N1 delta module applies its changes to the core ABS module `Hello`, as specified by the `uses` clause. It provides a new implementation for the method `sayHello()` in class `Greeter` by declaring a so-called method `modifier`. The method modifier is introduced by the `modifies` keyword and followed by the method signature and a block of code providing the method’s new implementation.

Adding entirely new methods is also supported using the `adds` keyword followed by the method signature and its implementation. Similarly, it is possible to remove methods from classes using `removes` followed by the method signature, as shown in the following.

```
delta D;
modifies class M.Foo {
  adds Int bar() { return 17; }
}
Calling original

Calling original from within a method modifier body makes it possible to access the method’s previous behaviour, that is, the behaviour implemented in the previously applied delta or in the core. This is similar to calling super to access the superclass behaviour of a method in a language with class inheritance such as Java. An original call has to supply a list of arguments that conforms with the original method’s list of formal parameters.

Targeted original calls

Original calls can be targeted towards a given delta by prefixing the call with the name of the delta, or towards the core ABS code by using the keyword core:

```plaintext
core.original(params);
Delta.original(params);
```

Regular (untargeted) original calls invoke the method behaviour defined by the previously applied delta. For example, if a method m is defined in the core, and then a set of deltas D1..D3, which each modify m, are applied in sequence, then calling original from within m’s modifier in D3 will run the version of m defined in D2. With a targeted call, one can access any version of m, that is the versions defined in D2, D1 and in the core.

This allows a tighter control of which code is actually executed when calling original. As the order of delta application is often not uniquely defined, it is not always determinable which behaviour will be invoked upon calling original. With a targeted original call, the user can specify exactly which code to execute and even invoke multiple versions of a method. This, of course, implies that the target delta has been applied already; otherwise the compiler will indicate an error.

Consider the above example. D1 and D2 both modify method m in different, non-compatible ways. We say that these two deltas are in conflict. Assume that D1 and D2 can be applied in any order, and that delta Resolve has to be applied after D1 and D2. By calling original from within Resolve, we cannot be sure which version of m will actually be invoked: this depends on whether D1 or D2 has been applied last. By targeting the original call towards a specific delta, we can control the behaviour precisely, and resolve the conflict in a meaningful way.
Targeted original calls were required for the implementation of the delta modelling workflow (DMW) [5, 4], which is described in more detail in Deliverable 5.3 [3]. The DMW describes a process of applying delta modelling to obtain a model of a software product line that is globally unambiguous and complete. A focus of DMW is the systematic reconciliation of conflicting feature functionality.

### 5.3.3 Class interfaces

ABS offers a maximum amount of flexibility when changing the behaviour of classes. To this end, a delta module can also change the list of interfaces that a class implements. Adding or dropping interfaces from that list is achieved using the familiar **removes** and **adds** keywords.

The following example shows a core ABS program defining a `Logger` class that implements the `Output` interface. It further declares a delta module that modifies the `Logger` class to the extent that it implements a different interface. This new IO interface is also introduced by the delta.

```plaintext
module M;
interface Input { String read(); }
interface Output { Unit write(String s); }
class Logger implements Output {
    Unit write(String s) {...}
}

delta IO;
adds interface IO extends Input, Output {}
modifies class Logger adds IO removes Output {
    adds String read() {...}
}
```

### 5.3.4 Fields

In addition to modifying object behaviour, ABS allows adding or removing fields. New fields are introduced by the **adds** keyword followed by the field’s type, name, and an optional value assignment. Similarly, fields can be removed using the **removes** keyword. The following example demonstrates this.

```plaintext
delta D;
modifies class M.Foo {
    adds List<Item> items;
    adds Int itemsCount = items.size();
    removes String name;
}
```

### 5.4 Functional modifiers

Functional program elements can also be modified from within deltas. ABS supports the addition of functions, data types and type synonyms. Qualifying functional elements with the module name is currently unsupported, therefore when adding functional elements, a **uses** clause has to be specified.

#### 5.4.1 Functions

Example of adding a function.

```plaintext
delta MyDelta;
uses MyModule;
```
5.4.2 Data types

Example of adding a data type.

```plaintext
adds data Schedule = Schedule(
    String schedname,
    List<Item> items,
    Int sched,
    Deadline dline) | NoSchedule;
```

5.4.3 Type synonyms

Example of adding a type synonym.

```plaintext
adds type ClientId = Int;
```

5.5 Formal Semantics

Applying a delta module $\Delta$ to a core ABS program $P$ yields a new core ABS program. Thus a product is constructed by successively applying delta modules, one at a time, to a core module. This section presents a formal semantics of delta modules similar to the semantics presented in Deliverable D1.2 [2], but considering the existence of targeted original calls. The key difference to the previously presented semantics is about how calls to previous versions of a method are handled. The code within the body of a modification clause of a method can contain not only references to the previous method definition, but it can contain references to any version of this method, indexed by the delta identifier that originated the change. Consequently, extra references to delta identifiers are kept around. Other modifications of the semantics of delta modules include formalising modified methods as new methods with fresh names, as opposed to inlining their method bodies; and simplifying the treatment of errors by postponing these to compile time when possible.

This formalisation is based on the more abstract presentation of Clarke et al. [1]. That work also describes the composition of delta modules with each other, which is essential for reasoning about conflicting delta modules, but this feature is elided from the current presentation. ABS programs, classes and delta modules will be represented in terms of finite maps from identifiers to the corresponding contents of the program, class, or delta module, in order to more cleanly present the semantics. The semantics only describes the modifications of methods; dealing with fields, functions, and so forth is a straightforward extension. Parameters are omitted. These have been covered in D1.2.

Let $CIdentifier$, $MIdentifier$, and $DIdentifier$ be the set of identifiers for classes, methods, and deltas, respectively, and let $MethBody$ be the set of method bodies, including the parameter and return types, with possible references to (targeted and untargeted) original methods. In the following domains, $Modify$ and $Remove$ are used to tag the various branches of sum data types.

- $Program = CIdentifier \rightarrow ClassBody$
- $ClassBody = MIdentifier \rightarrow (MethBody \times DIdentifier)$
- $Delta = CIdentifier \rightarrow DeltaBody$
- $DeltaBody = Modify (MIdentifier \rightarrow ((MethBody \uplus Remove) \times DIdentifier))$

A program is a map from class names to classes. Class bodies are collections of pairs of a method body and the identifier of the delta used in its creation or last update. Initially all class bodies have their methods
associated to the special delta identifier \texttt{core}. A delta is a map from class names being modified to delta bodies. Note that, for technical convenience, the \texttt{DIdentifier} is included in the delta bodies and not in \texttt{Delta}. Delta bodies consist of two different types of modification: \texttt{Modify} modifies a class in place or creates a new class if it does not exist, where the two elements within a \texttt{Modify} clause correspond to either (1) replacing or adding a method with a new body from \texttt{MethBody}, or (2) removing the method. Finally, \texttt{Remove} denotes the removal of the class.

\textbf{Notation 5.5.1} Let $f : X \rightarrow Y$ denote a partial function from $X$ to $Y$. If $f(x)$ is undefined for $x \in X$, write $f(x) = \bot$, where $\bot \notin Y$. For set $A$, let $A \bot = A \cup \{ \bot \}$, where $\bot \notin A$. We freely shift between partial functions $X \rightarrow Y$ and functions $X \rightarrow Y \bot$. If $\circ : A \bot \times B \bot \rightarrow C \bot$, define the lifting of $\circ$ to partial functions over index set $I$ as

$$
\overline{\circ} : (I \rightarrow A) \times (I \rightarrow B) \rightarrow (I \rightarrow C)
$$

$$(f \overline{\circ} g)(i) = f(i) \circ g(i), \quad \text{where } i \in I.
$$

\text{Notation 5.5.2} Given class update $u : \texttt{Mldentifier} \rightarrow ((\texttt{MethBody} \sqcup \texttt{Remove}) \times \texttt{DIdentifier})$, define function $u^* : \texttt{Mldentifier} \rightarrow (\texttt{MethBody} \times \texttt{DIdentifier})$ as follows. For $i \in \texttt{Mldentifier}$:

$$
u^*(i) = \begin{cases} 
\bot & \text{if } u(i) = (m,d) \text{ and } m \in \texttt{Remove}, \\
u(i) & \text{otherwise.}
\end{cases}
$$

\text{Notation 5.5.3} Given $d \in \texttt{DIdentifier}$ and $i \in \texttt{Mldentifier}$, let $\epsilon(i,d) \in \texttt{Mldentifier}$ be a method identifier uniquely defined by $d$ and $i$. Given class update $u : \texttt{Mldentifier} \rightarrow ((\texttt{MethBody} \sqcup \texttt{Remove}) \times \texttt{DIdentifier})$ and class body $c : \texttt{Mldentifier} \rightarrow (\texttt{MethBody} \times \texttt{DIdentifier})$, define function $\xi(u,c) : \texttt{Mldentifier} \rightarrow (\texttt{MethBody} \times \texttt{DIdentifier})$ as follows.

$$
\xi(u,c) = \{ \epsilon(i,d) \mapsto (m,d) \mid (i \mapsto (m,d)) \in c, i \in \text{dom}(u) \}
$$

\text{Definition 5.5.4 (Delta module application)} The application of a delta module to a program is specified by the following functions:

\[
\begin{align*}
\text{apply} & : \text{Delta} \times \text{Program} \rightarrow \text{Program} \\
\text{apply}(d,p) & = d \overline{\circ_c} p
\end{align*}
\]

where $\overline{\circ_c} : \text{DeltaBody} \bot \times \text{ClassBody} \bot \rightarrow \text{ClassBody} \bot$

$$
\overline{\circ_c} \bot x = x \quad \text{(Modify } u) \overline{\circ_c} \bot = u^* \\
\text{Remove} \overline{\circ_c} \bot = \bot \quad (\text{Modify } u) \overline{\circ_c} c = u \overline{\circ_m} c \cup \xi(u,c)
$$

and $\overline{\circ_m} : ((\text{MethBody} \sqcup \texttt{Remove}) \times \texttt{DIdentifier}) \bot \times (\text{MethBody} \times \texttt{DIdentifier}) \bot \rightarrow (\text{MethBody} \times \texttt{DIdentifier}) \bot$

$$
\overline{\circ_m} \bot x = x \quad (m,d) \overline{\circ_m} \bot = (m,d) \\
(\text{Remove},d) \overline{\circ_m} \bot = \bot \quad (m,d) \overline{\circ_m} (m',d') = (m|m_{id},d'),d)
$$

where $m \in \text{MethBody}$, $m_{id} \in \text{Mldentifier}$ is the identifier of method $m$, and $d,d' \in \text{DIdentifier}$. 
5.5.1 Implementation

In practice, for every delta body Modify a for class C and for each \( m_{id} \rightarrow (m, d) \) with \( m \in \text{MethBody} \), the following steps are performed:

1. if exists \( m' \in \text{MethBody} \) and \( d' \in \text{DIdentifier} \) such that \( m_{id} \rightarrow (m', d') \) is in the class body of C, then:
   - replace it with \( m_{id} \mapsto (m[m_{id}, d'], d) \), and
   - add \( \epsilon(m_{id}, d') \mapsto (m', d') \);
2. otherwise add \( m_{id} \mapsto (m, d) \).

The example at the end of Section 5.3.2 illustrates this process with concrete code. The modified method in that example is \( m \), belonging to class \( C \). Originally \( C = \{m_{id} \mapsto (m, \text{core})\} \), where \( m \) is the method body of \( m \) and \( m_{id} \) is its identifier. A possible sequence of applying the three deltas is \((D1, D2, \text{Resolve})\).

Application of D1. Let \( m_1 \) be the new method body of \( m \) in \( D1 \). First we calculate \( m_1[m_{id}, \text{core}] \) by replacing \( \text{original} \) by \( \epsilon(m_{id}, \text{core}) \) in \( m_1 \), using \( \epsilon(m_{id}, \text{core}) = \text{m$\text{ORIGIN\_core}$} \). Second we add \( m_{id} \mapsto (m_1[m_{id}, \text{core}], D1) \) and \( \epsilon(m_{id}, \text{core}) \mapsto (m, \text{core}) \) to class \( C \).

Application of D1. Let \( m \) be the method body for \( m \) after applying \( D1 \) and \( m_2 \) be the new method body of \( m \) in \( D2 \). First we calculate \( m_2[m_{id}, D1] \) by replacing \( \text{original} \) by \( \epsilon(m_{id}, D1) \) in \( m_2 \), using \( \epsilon(m_{id}, D1) = \text{m$\text{ORIGIN\_D1}$} \). Second we add \( m_{id} \mapsto (m_2[m_{id}, D1], D2) \) and \( \epsilon(m_{id}, D1) \mapsto (m, D1) \) to class \( C \).

Application of Resolve. Let \( m \) be the method body for \( m \) after applying \( D1 \) and \( m_r \) be the new method body of \( m \) in Resolve. First we calculate \( m_r[m_{id}, D2] \) by replacing \( D1, \text{original} \) by \( \epsilon(m_{id}, D1) \) in \( m_r \). Second we add \( m_{id} \mapsto (m_r[m_{id}, D2], \text{Resolve}) \) and \( \epsilon(m_{id}, D2) \mapsto (m, D2) \) to class \( C \).

The application of the three deltas results in the following code.

```java
module M;

class C {
    String m(String s) { return prefix + m$\text{ORIGIN\_core}$s + suffix; };

    String m$\text{ORIGIN\_core}$s(String s) { return(s); }

    String m$\text{ORIGIN\_D1}$s(String s) { return prefix + m$\text{ORIGIN\_core}$s; }

    String m$\text{ORIGIN\_D2}$s(String s) { return m$\text{ORIGIN\_D1}$s(s) + suffix; }
}
```

After the application of all three deltas, the class \( C \) has four methods: \( m$\text{ORIGIN\_core}$, m$\text{ORIGIN\_D1}$, m$\text{ORIGIN\_D2}$ and \( m \). Both \( m \) and \( m$\text{ORIGIN\_D2}$ call \( m$\text{ORIGIN\_D1}$, while \( m$\text{ORIGIN\_D1}$ calls \( m$\text{ORIGIN\_core}$). Observe that the method \( m$\text{ORIGIN\_D2}$ is added to \( C \) but never called. A simple optimisation is to postpone its addition to \( C \) until it is called from within any method body. Furthermore, unreachable versions of methods can be safely removed.

5.6 Unsupported modifications

While delta modelling supports a broad range of ways to modify an ABS model, not all ABS program entities are modifiable. These unsupported modifications are listed here for completeness. While these modifications could be easily specified and implemented, we opted not to overload the language with features that have not been regarded as necessary in practice.
Class parameters and init block Deltas currently do not support the modification of class parameter lists or class init blocks.

Functional program elements Deltas currently only support adding functional program elements such as functions or data types. Removal and modification is not supported; for example one cannot add cases to the data type declared in Section 5.4.2.

Modules Deltas currently do not support adding new modules or removing modules.

Imports and Exports Deltas currently do not support the addition, removal, or modification of import or export statements to/from modules.

Main block Deltas currently do not support the modification of the program’s main block.
Chapter 6

End-to-end Product Derivation

6.1 Motivation

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. As feature models [6] have been the main approach for capturing variability in product lines, 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.

As of Deliverable D1.2 [2], the HATS framework was already capable of deriving the code for a concrete product by using the product specification represented in the Product Selection Language (PSL) to resolve the variability in the product line assets. However, there was no support for the selection of the appropriate features to be delivered in a concrete product. In this previous scenario, the person in charge of configuring and therefore deriving a product from a product line only received the feedback whether the product specification was valid or not. Moreover, this feedback was received probably after spending considerable time in the selection of features and, in case of failure, without indication on how to correct the product specification in order to successfully generate the product.

In order to support end-to-end product derivation, it is necessary to address the initial phases of the application engineering process, which in the HATS context means how to derive a valid and appropriate product specification.

6.2 General Concept

This section presents the general concept of a product configurator that takes into consideration:

1. the dependencies and constraints among features
2. the different stakeholders who deal with external and internal features
3. the desired degree of quality
4. and cost constraints.

The dependencies and constraints (item 1) are provided in the feature model. In this task, we have investigated how to ask the person in charge of the configuration of a product only the information he/she has to provide, which means external features he/she is very much sure that should be included in the
product (related to item 2). We have then defined the concept and design of the configurator that is in charge of proposing a set of valid configurations that obey the cost constraints (item 4).

In Task 4.4 (Auto-Configuration and Quality Variability) we have been extending this concept to take into consideration variability in quality (item 3) and to refine which information has to be requested from the person in charge of the configuration process in case of supporting quality variability (related to item 2).

Figure 6.1 shows the application engineering process and the respective artefacts when assuming semi-automated product configuration and automated product derivation in the HATS framework. The process starts with the elicitation of the customer’s requirements, which should cover business goals, key functional requirements, quality concerns, and established constraints. The customer’s requirements are then mapped into an appropriate set of features. In the subsequent phase of the process, the product configurator checks the correctness of the initial set of features based on the dependencies and constraints of the feature model, indicates how to correct an invalid configuration with a minimal number of changes, completes the set of selected features while trying to optimize quality concerns such performance and security. The output of this phase is a product specification in PSL. The Generation of ABS Code phase translates the product specification into a flattened ABS code, by applying the appropriate deltas to the core ABS according to specifications captured in the Product Line Configuration Language (CL). These specifications define when and in which order to apply the deltas. The final phase is the Generation of Backend Code, when code for different backends can be generated from ABS code.

Figure 6.1: HATS End-to-end Product Derivation (Steps and Artefacts)

While the result of the application engineering process is a concrete product, the result of the domain engineering process is the product line artifact base with several assets to be reused when deriving the different products. In the HATS framework, the product line artifact base is composed of μTVL (text-based feature modeling language, pronounced either micro textual variability language or simply mu tee vee ell, an extended subset of TVL), delta modules, CL (the product line configuration language which links feature models specified in μTVL with delta modules to provide a specification of the variability in a product line), and core ABS files. The product configurator briefly described above requires the extension of the feature model of the product line with cost, performance, and security annotations. Moreover, if variability in security should be supported, security features and the respective delta modules should be available.

6.3 Benefits

The product configurator can help the person in charge of the configuration process to find out all possible configurations, complete any partial configuration, check the correctness of a configuration, identify errors, correct a configuration with the minimum number of changes, find out maximal or minimal configurations in terms of number of features, among others. In Figure 6.2 a scenario is presented where the features indicated by the user as required lead to an invalid configuration. The configurator can indicate a valid configuration that requires the minimum number of changes in the user’s input. The green boxes represent the features
selected by the user. The violet boxes (filled and unfilled) are the corrections suggested by the configurator to get a valid configuration. Filled boxes indicate the features that have been included by the configurator in the proposed valid configuration, where as the unfilled boxes indicate features that the configurator has excluded. The configurator also calculates the distance between the initial and proposed configurations. Each change of a feature from selected to de-selected and from de-selected to selected counts as 1. In this example the distance between the two configurations is 3.

![Figure 6.2: Minimal Distant Valid Configuration](image)

### 6.4 Research Methodology

As already mentioned, the work reported in this deliverable is planned to be continued in the scope of Task 4.4 (Auto-Configuration and Quality Variability). The overall methodology is composed of steps as follows.

1. Literature review on Quality Variability Modeling
2. Literature review on Product Configuration
3. Definition of the solution concept
4. Definition of the solution design and implementation
   
   (a) Cost extension
   
   (b) Performance extension based on static analyses provided by COSTABS
   
   (c) Security extension based on reusable expert knowledge

The results of step 1, 2 and 3 have been published in [12]. In the next sections we provide details on item 4 without focusing on the quality concerns, as they will be addressed in Task 4.4 (Auto-configuration and Quality Variability).
6.5 User Interaction and Interface Design

Our product configurator will support the following interaction with its users, people responsible for configuring products from product line infrastructures.

1. The configurator uses information from the feature model about mandatory features to automatically select those for the current product configuration.

2. The configurator asks the user to select the required features. Required features are those which the user is confident about their inclusion in the product. If they do not infringe on any constraint in the feature model, the configurator will not propose deselecting them in any of the provided solutions.

3. The user sees the automatically selected mandatory features in a graphical representation and indicates the required features by selecting them.

4. The configurator checks the correctness (not the completeness) of the set of features that has been selected so far. If the selected set of features triggers requires and excludes relationships, the configurator will either select features or make them unavailable. If there is an error, the configurator will at least indicate the constraints that have been infringed.

5. If necessary, the user corrects his/her selection of required features.

6. The configurator helps the user to provide cost constraints and/or to inform qualifiers for the quality concerns of interest.

7. The user provides information about the cost constraints and/or quality concerns of interest. For the quality concerns, the user also provides their respective weight.

8. The configurator provides a set of configurations that include the required features and obey the cost constraints, while at the same time trying to achieve the desired degree of quality.

9. The user chooses the most suitable configuration out of the recommended configurations and decides to derive the concrete product.

Figure 6.3 shows the prototype of the configurator’s screen. On the left side, the user follows the steps of the configuration process that require information or action from him/her. The information/action is required on the right side. In this case, the user visualizes the automatically selected mandatory features (in green), the required features (in blue), and the selection of features suggested by the configurator (in yellow). On the top, the user is informed that this is configuration number 5 and he/she can navigate through the complete set of valid configurations that adhere to the customer’s requirements. The bars on the right side illustrate the estimated degree of security and performance of the current configuration and also illustrate its cost position regarding the cost constraints.

6.6 Architectural Design

The architecture of the product configurator (Figure 6.4) has 3 main parts - the user interface part, the application logic part and the model management part. In the following paragraphs, these three parts are described briefly.

- View part - This part of the architecture is responsible for interacting with the person in charge of the configuration and allowing the visualization of the feature model and the product specification. It includes the UserInteraction and Visualization components.
Application logic part - It consists of several core components. The central component is the ConfigurationManager. It takes all the configuration input from the UserInteraction component and takes services from other components (muTVLAnnotation, Solver, ABSCodeBuilder, BackendCodeGenerator) for generating the code of the product. muTVLAnnotation is responsible for annotating the feature model in µTVL with different quality and cost annotations. To do this, it takes services from the concrete annotator components (PerformanceAnnotation, SecurityAnnotation, and CostAnnotation). The Solver component can solve the Constraint Satisfaction Problem generated by the ConfigurationManager and stores product specifications in PSL through the ProductManager component. The ABSCodeBuilder component is responsible for generating the flattened ABS code from a product specification in PSL. It stores the flattened ABS code through CodeManager component. The BackendCodeGenerator component generates the source code of the appropriate type (Java/Maude/Scala) that corresponds to the flattened ABS code.

Model part - It is composed of the PLManager, ProductManager, and CodeManager components, which are responsible for inserting and updating different entities and managing the persistence of those entities. Each of these components stores, updates and provides its respective artefacts (µTVL, DL, CL and Core ABS files; product specifications in PSL; and flattened ABS and source code).

Several of the aforementioned components (e.g. ABSCodeBuilder and BackendCodeGenerator) have been integrated into the proposed architecture to achieve end-to-end product derivation, but they are not the result of the work reported in this section.

6.7 Detailed Design

In this section quality concerns (performance and security) are considered only as far as they have impact in the overall design of the product configurator. As already mentioned, quality concerns are going to be specifically addressed in Task 4.4 (Auto-configuration and Quality Variability).

In order to help the selection of features for a concrete product that obey certain cost constraints, it is necessary to indicate how each feature in the feature model contributes to the total cost of the product. The same applies to the quality concerns and their specific metrics. In order to support the selection of features for a concrete product so that performance is optimized, it is necessary to indicate how each feature in the feature model contributes to the total performance of the product. After literature review and the analysis of HATS variability languages (µTVL, DL, CL), we have decided to annotate µTVL feature models with information related to cost, performance, and security. The annotated µTVL belongs to a specific product line, as there is no way to annotate a feature with security or performance information without considering
how the feature is implemented. Depending on the elements of the cost of features, the information on cost also depends on how the feature is implemented.

Another considered aspect was the fact that the performance and the security of a feature cannot be precisely analyzed in isolation. On-the-fly analyses of product configurations would provide more precise results, however this would make the configurator too slow as all product configurations would have to be analysed for performance and security. Thus we have decided to annotate features with cost, performance and security information in a pre-processing stage, so that the Solver can already take those pieces of information into consideration, which narrows the solution space. Due to resulting possible loss of information, there is no guarantee that the recommended configurations which will be the best in terms of performance and/or security. However, the annotations are still useful for guiding the configuration process since, in most cases, the recommended configurations will be close to the optimal result.

Figure 6.5 illustrates how feature models and more specifically feature models in $\mu$TVL are annotated. In this figure, the product line is a Weather Station Product Line and the graphical feature model and $\mu$TVL feature model are annotated using internal performance metrics: memoryConsumption and responseTime. They are called internal metrics because they are measured by using static analyses carried out by COSTABS and not by running the system. The right and bottom part of the figure shows how annotations are represented in $\mu$TVL. The StormAlert feature is annotated with Memory Consumption = 3 and Response Time = 1. The remaining information in the extension is common for all features and can be derived automatically.

## 6.8 Implementation

Most of the HATS tools and components are implemented in the Eclipse framework as Eclipse plugins. In order to promote an integrated ABS software development environment, the product configurator has been also developed based on Eclipse. We have used Java as programming language and Eclipse Version 3.6.2 as development environment. For solving the Constraint Satisfaction Problem, ChocoJava Version 2.1.1 has been used. ChocoJava is a Java based CSP solver and the reasons for adopting it were as follows: 1) the mapping of the product configuration problem into CSP is intuitive, 2) ChocoJava is already used for...
other purposes in the HATS framework and therefore its adoption promotes an integrated ABS software development environment, and 3) there are translators of CSP into Satisfiability Modulo Theories (SMT) that can be adopted to deal with potential performance problems of CSP solvers if required.

The source code of the current version of the product configurator is available in the HATS consortium’s SVN repository. The current version does not implement the user interface components.

6.9 Conclusion

In this task, we have developed an infrastructure for product configuration that receives as input the customer’s requirements and suggests configurations that meets those requirements. This allows for deriving ABS code and the respective source code for the concrete product as soon as the desired configuration is chosen. In Task 4.4 (Auto-configuration and Quality Variability) we are going to extend this infrastructure to support the configuration of a product based on quality (performance and security) concerns.

Two research issues were identified that concerns this first stage of the work:

- User empowerment vs. Auto-configuration: According to our work, the configuration process is automatized as much as possible. However, decisions are taken by the user who also provides essential information for the efficiency of the configuration process. We believe this to be an interesting balance between automatic and interactive configuration.
- Performance of the CSP Solver: White et al. [13] reported a response time of 3 seconds for a 3-step configuration process based on a feature model with 500 features. We want to find out the response time of our configurator for some thousand features. In any case, we have already investigated that is possible to translate a problem modeled in CSP into a problem modeled in SMT, which should offer a much better response time.

The presented configurator has been developed to achieve end-to-end system derivation and it will certainly improve efficiency and shorten the time to generate concrete products from an ABS product line.

Figure 6.5: An Example of Annotated Feature Model
Chapter 7

Conclusions

In this task we brought together those efforts of the project that pertain to system derivation and code generation from ABS models.

Since tool development started out at an earlier stage of the project than first planned and very successfully, much of the work initially planned for Task 1.4 was already done within Task 1.2 before its official start. In particular, the basic ABS compiler framework with its Java backend was ready by an early stage of the project and has since then been maintained, updated and improved continously to support the work of all other tasks of the project. Since the early work on code generation was already reported in Deliverable D1.2\cite{2}, Section 7, in the current deliverable we concentrated on updates to these tools and the new element of the tool kit: a Scala back-end of the compiler that supports distributed execution of ABS models—running different COGs at different nodes.

Trustworthiness of software is one of the key objectives of the whole project. This concerns first of all trustworthiness of ABS models produced with the HATS methodology, targeted in Tasks 1.3, 2.5 and 4.x. But equally important is that code generation from ABS models is correct. We have therefore worked hard on organizing compilation from ABS to Scala in a way which is at once both practical and enables a sensibly structured proof of correctness of compilation (preservation of semantics), requiring subtle reasoning about delimited continuations.

Regarding the compiler frontend (resolving variability) and system derivation at large, we made various adjustments to the delta modelling capabilities of ABS and developed a product configurator tool that helps the user to put together PSL product selection specifications that serve as input to product derivation from user requirements and ABS+$\mu$TVL+CL product lines.
Bibliography


Glossary

Terms and Abbreviations

ABS Abstract Behavioral Specification language. An executable class-based, concurrent, object-oriented modeling language based on Creol, created for the HATS project.

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

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

CSP Constraint Satisfaction Problem

CL The (Product Line) Configuration Language (of the HATS framework)

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

Delta Synonymous with Delta Module

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

DMW The Delta Modelling Workflow describes a process of applying delta modelling to obtain a model of a software product line that is globally unambiguous and complete.

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

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

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

PSL The Product Selection Language (of the HATS framework)

μTVL Micro Textual Variability Language—the feature modelling language of the HATS framework