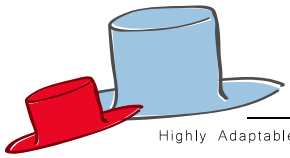

Project N°: **FP7-231620**

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 D1.5
ABS Tool Platform and Methodology

Due date of deliverable: (T0+48)

Actual submission date: March 1, 2013

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

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

Final version

Integrated Project supported by the 7th Framework Programme of the EC		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Executive Summary:
ABS Tool Platform and Methodology

This document summarises deliverable D1.5 of project FP7-231620 (HATS), an Integrated Project supported by the 7th Framework Programme of the EC within the FET (Future and Emerging Technologies) scheme. Full information on this project, including the contents of this deliverable, is available online at `http://www.hats-project.eu`.

This deliverable has two key contributions: First, an integrated tool architecture, that is, an architecture for the set of tools developed in the HATS project, and second, the HATS methodology, which describes how to use these tools to build product line software. The HATS methodology presented here is an update of the previous version from deliverable D1.1b.

List of Authors

Taslim Arif (FRG)
Bjarte M. Østvold (NR)
Karina Villela (FRG)
Balthasar Weitzel (FRG)
Peter Wong (FRH)

Numerous tool authors in the project provided text on tool architecture or on tool usage in the methodology.

Contents

Chapter 1

Introduction

In this deliverable we are concerned with two main subjects: the ABS tool platform and the HATS methodology. Regarding the ABS tool platform, we present in Chapter 2 how several tools developed in the project have been integrated into an ABS tool platform and how this prototypical tool platform can evolve towards an industrial tool platform. For that, we have documented the current structure of the individual tools, designed an *industrial tool platform architecture*, and defined the steps for migrating each tool to this improved architecture. The *tool platform* itself comprises all the tools developed in the HATS project: modelling and specification languages, compilers, simulators, debuggers, testers and test generators, specification and modelling frameworks, verifiers and analysers, model generators, and integrated development environments.

Regarding the *HATS methodology*, we present in Chapter 3 a process that describes the activities to be carried out to build product line software, and we identify input and output artifacts of those activities. This process description is enriched with procedures that explain how to carry out the process activities using HATS framework. The HATS methodology supports tool integration since it describes how an artifact produced by one tool may be required by another tool.

In addition, this deliverable points to tutorial material on the ABS language and tools in Chapter 4. A glossary of the most central terms appears at the end.

Relationship to deliverable D1.1b and the HATS Description of Work. The methodology presented in Chapter 3 is based on the version presented in deliverable D1.1b [8]. The present version retains the basic structure of the previous version—processes and phases—but it includes more details: stakeholders, activities, artifacts, and procedures. We refer to the earlier deliverable for an overview of product line engineering (PLE) [8, §2].

The methodology provided in this deliverable goes beyond what was promised in the HATS Description of Work, because it is quite detailed in explaining how the tools can be used, and how they share results and participate in a structured process of software product line development.[1]

1.1 Approach

Regarding the integrated tool platform, the overall task was divided into the following steps:

a) Collect high level architectural information about the tools, done with the questionnaire in Appendix A (*All tool developers*)

b) Sketch the integrated tool architecture (*Task 1.5 contributors*)

c) Sketch possible migration steps (*Task 1.5 contributors*)

[1] The emphasis on methodology motivated the change of name of this deliverable with respect to the Description of Work.

The questionnaire aimed at gathering the necessary information to document the current architecture of each tool on a high abstraction level. The result of this questionnaire served as input for the sketch of a common tool architecture. In order to be sure that the platform architecture would be capable of integrating all the currently available tools it was crucial to understand their architectures and the constraints that result from architectural decisions made during their realisation.

Regarding the HATS methodology, we started with the previous version provided in D1.1b, which was already based on the classical organization of the software product line engineering activities in the two main processes of family engineering and application engineering. We then refined the previous set of phases into more detailed activities and identified for each activity the stakeholders involved in it, the artifacts required as input, as well as the artifacts produced as output. After describing activities, which indicate what has to be done without providing hints on exactly how to perform them, we have described procedures to specify in detail how to use HATS methods, techniques and tools to carry out all activities supported by the HATS framework. All HATS project members were requested to indicate in the respective procedures how their contribution to the project fits into the HATS methodology. We believe the integration of all research contributions into a systematic methodology is crucial for the transfer of the HATS results to the industry.

Chapter 2

ABS Tool Platform and Architecture

2.1 Introduction

Task 1.5 had the goal of providing a prototypical tool platform integrating the different tools developed in the HATS context and, in particular, defining the architecture for such a platform.

In general, a software architecture documents the fundamental design decisions made before or during the implementation of the software. It describes which elements the software comprises and how they are related to each other.

The investigative characteristic of a research project does not allow many design decisions to be made in advance (prescriptive architecture). Therefore, this task adopted a stepwise approach.

A *prototypical tool platform* has been delivered, where integration was pursued through the use of:

- an abstract syntax tree infrastructure as the main integration element. Several tools consume and provide services or data related to this infrastructure.
- the Eclipse IDE as the underlying development and maintenance environment. Several tools have been delivered as Eclipse plugins. This decision ensured a seamless GUI experience for the developers using the HATS approach.
- Maven for integrated build management. It supports a uniform build system, with a clear definition of what a project consists of and the dependencies on varying types. It also provides a repository for sharing standard utilities among different builds.

The second step was the *architectural documentation of each individual HATS tool*. The architecture of a fully integrated tool platform should reflect all existing HATS tools and therefore take into consideration their main characteristics. Furthermore, tools that play a substantial role in the overall HATS methodology need to be considered, too. The overall goal is to offer an integrated and fluent tool chain to support the HATS methodology.

As it goes beyond the scope of a research project to deliver a robust and modular tool platform integrating all project results, the third step consisted of elaborating *coarse migration steps* towards the integrated platform architecture. This plan explains on an abstract level how to evolve the tools to achieve better integration into the platform. In this case, integration means that the tool is built on the platform and does not appear as a separate tool for the user. Every tool developer can then decide when and how to realise the coarse migration steps. To do so, each migration step has to be transformed into a concrete change request on the implementation level.

2.2 Prototypical Tool Architecture

A number of tools have been developed in the context of the project. In order to provide an integrated platform for ABS developers, we developed a prototype of such integration based on three integration

approaches.

A significant number of tools have been integrated based on the common parsing and abstract syntax tree (AST) infrastructure provided by the *ABS Frontend* tool. This infrastructure includes Java classes representing all language features of Core and Full ABS, which many tools make use of. For example, the *Type Checker* and the *Feature and Delta Modelling Extension*, which checks products for consistency against a product line definition, were two previously separate tools that have been merged into the *ABS Frontend* tool. Closely tied to the *ABS Frontend* are the backends, since they explore every part of the abstract syntax tree object structure. There is no practical reason for decoupling them, since AST changes arise from language changes, and language changes mandate backend changes anyway.

Integration in terms of GUI functionalities is provided via Eclipse plugins. An Eclipse plugin, which also relies on the common parsing and AST infrastructure, provides the usual services of a code development environment (navigation, syntax highlighting, error markup, invocation of executable code generation). COSTABS, several backends and a Sequence Diagram Visualiser are also available as individual plugins, which are part of an integrated set of plugins. Tool developers spent quite some time separating core functionality and GUI in order to make integration in terms of GUI functionalities based on Eclipse possible.

The third integration approach is through Maven, itself an Eclipse plugin for ABS-based build management. Integration is achieved in this case through thin wrapper scripts that allow Maven to invoke different HATS tools in a sequence that is appropriate for the dependencies between those tools, thereby supporting systematic building. The developed Maven plugin supports generating Java and Maude code from ABS. It also supports executing and simulating the generated Java and Maude code.

Therefore, data, presentation and control integration have been addressed in the prototypical tool platform. Table 2.1 shows the tools of the prototypical tool platform according to the integration approach.

Table 2.1: Integrated tools according to the integration approach.

AST	Eclipse GUI	Maven
ABS Frontend Eclipse IDE Java Backend Maude Backend Scala Backend COSTABS Sequence Diagram Visualiser KeY-ABS ABS Product Configurator PEKeY SDA Type Checker Feature and Delta Modelling Extension aPET	ABS Frontend Eclipse IDE Java Backend Maude Backend COSTABS Sequence Diagram Visualiser Type Checker	ABS Frontend Java Backend Maude Backend Scala Backend

2.3 How We Understand Architecture

We consider software architecture to be the set of main design decisions made about the software system. These decisions impact the way the software system fulfils its requirements. In general, there are multiple ways to design a software system that offer the same functionality, but each kind of realisation exposes differences, especially regarding non-functional requirements.

2.3.1 Documentation of Architecture based on Views

This section briefly explains the architecture documentation approach based on views [2] that was used to document the current architecture of the HATS tools.

There are three *dimensions* for structuring architectural information, each addressing a different subset of tool characteristics.

***Dimension* Context:** How does the tool interact with its environment? The tool is seen as a black box and the focus is on its relationships to its environment.

***Dimension* Runtime:** How does the tool work at runtime? The runtime entities and their relationships with each other are the key aspects of this dimension. A decomposition into elements that are present at runtime is done.

***Dimension* Development Time:** How is the tool implemented? What development technologies are used? During development, the tool is in most cases structured differently than at runtime. The focus is on the distribution of runtime entities to elements used during development. If you take a typical Eclipse extension as an example, there might be one functional runtime entity (an editor, for example) that comprises different plugins, with some of them being written manually and some of them being generated. Information about these situations is expected to be represented in this dimension, which should help to discuss how the development of the tool is organized.

As these dimensions are too coarse grained for a clear architectural documentation, there are *viewpoints* in every dimension that can be separated from each other:

***Viewpoint* Data:** This viewpoint describes the data that is handled by the system. Depending on the dimension, this data is used internally or in exchange with other systems.

***Viewpoint* Functions:** This viewpoint describes how the functionality (defined by the functional requirements) is realised. Depending on the dimension, this functionality is used internally or provided to other systems.

***Viewpoint* Allocation:** This viewpoint describes aspects regarding allocation and deployment of the elements of the software tool.

***Viewpoint* Processes:** This viewpoint describes how workflows and business processes are related to the elements of the software tool.

***Viewpoint* Technologies:** This viewpoint describes when and how technologies (like a framework) are used in the tool and what their impact is.

These *viewpoints* are applied to each *dimension*, resulting in different *views*, each explaining a specific aspect of the software architecture. In a semi-formal way, the views are the result of:

$$Dimensions = \{Context, Runtime, Devtime\}$$

$$Viewpoints = \{Data, Functions, Allocation, Processes, Technologies\}$$

$$Views = Dimensions \times Viewpoints$$

Some views can be merged to reduce the effort for creating them and to decrease the need of a view to refer to another one. Not every view is needed in all cases. For the purpose of architecture documentation in the context of this task, we used the views indicated in Table 2.2 as mandatory (•) or optional (○).

2.3.2 Required Views

As illustrated in Table 2.2, three views were considered mandatory or required.

		Viewpoints			
		Data	Functions	Allocation	Processes	Technologies
Dimensions	Context	●			○	
	Runtime	●			○	●
	Devtime					○

Table 2.2: Mandatory and optional views.

***View* Data and Functions in Context:**

- What data is exchanged with other tools?
- Which format does this data have?
- Which functionalities are used from connected systems or tools?
- Which functionalities are provided to other tools or systems?

This view should be documented with the tool as a black box component with connections to the outside world. Each connection should be annotated with the type of data that is transported over it. An example is shown in Figure 2.1.

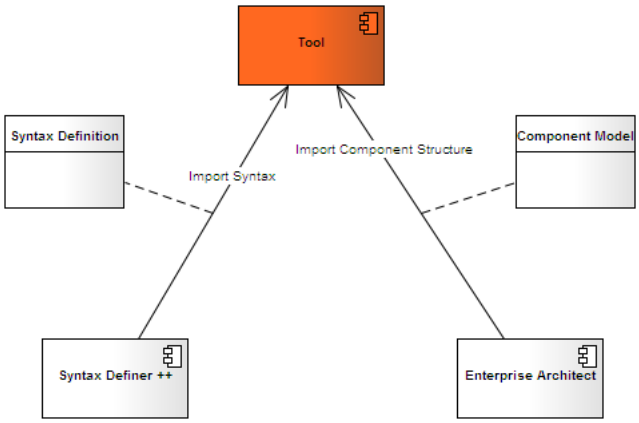

Figure 2.1: Example diagram: data and functions in context.

***View* Data and Functions at Runtime:**

- What data is used in the tool?
- When is it exchanged for what purpose?
- Which functions are provided by the elements of the tool?
- How is the data structured?

This view should be documented with the tool decomposed into its components and the connections among them. Each connection should be annotated with the type of data that is transported over it. An example is shown in Figure 2.2.

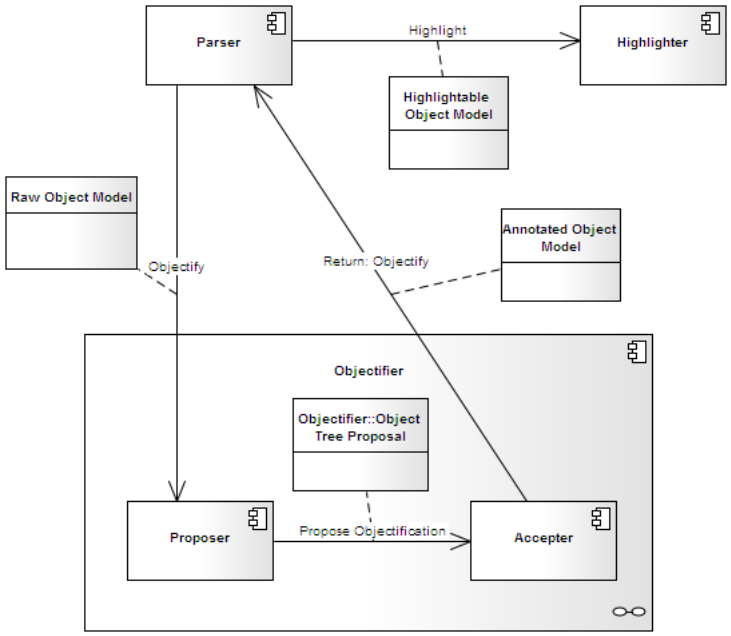

Figure 2.2: Example diagram: data and functions at runtime.

***View* Technologies at Runtime:**

- What technologies are used while the tool is running?

This view should be documented mainly in a textual manner and enhanced with notes in the diagrams used for Data and Functions in Context and Runtime. An example is shown in Figure 2.3.

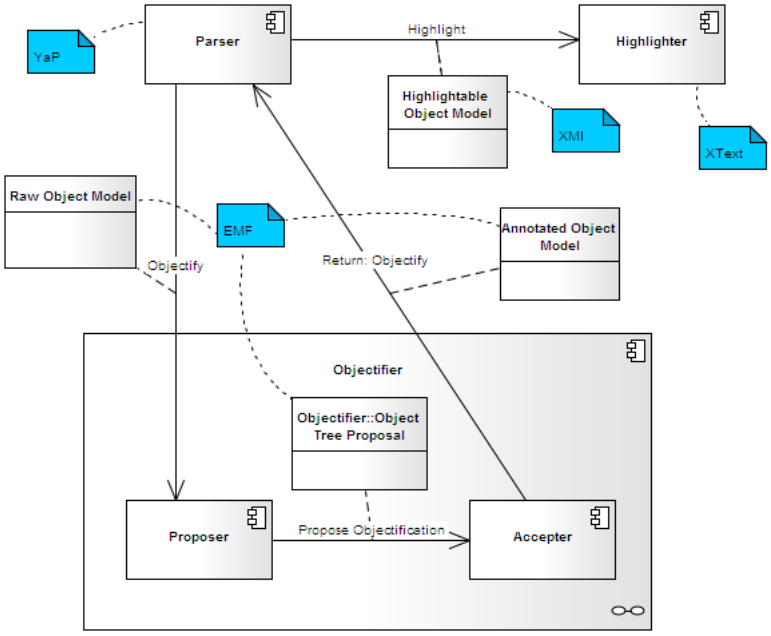

Figure 2.3: Example diagram: technologies at runtime.

2.3.3 Optional Views

There are additional views that are helpful for the understanding of the tool architecture. If they can be easily created, they should be included in the response. They can be documented in a textual manner.

***View* Processes in Context** How is the tool integrated into the processes of its environment, especially with respect to the HATS methodology?

***View* Processes at Runtime** How is the workflow that should be supported by the tool realised by runtime entities?

***View* Technologies at Development Time** What technologies are used at development time for realising the artifacts of the tool? Are there technologies used to generate code? This view should be documented mainly in a textual manner and enhanced with notes in the diagrams used for Data and Functions in Context and Runtime.

2.4 Documentation of Individual Tool Architectures

In this section, the architectures of the individual tools are documented using the view-based approach. Each tool is described by its goals, use cases, and various architectural views.

2.4.1 ABS Frontend

Goals

- Well-formedness of ABS programmes: The tool is used to check the well-formedness of ABS programmes.
- AST-like representation of ABS programmes: It provides a uniform programmatic way to an AST-like representation of the ABS programme.

Use Cases

- Check that an ABS programme is well-formed
- Development of ABS language extensions
- Development of ABS type system extensions (e.g., Location Type System)
- Development of ABS compiler backends (Java, Scala, Maude)
- Development of static analyses for ABS programmes

Important Architectural Views

1. *View Data and Functions at Runtime* Upon invocation, the tool takes a set of ABS source files as input. Error messages regarding the well-formedness of the given ABS source files are put out as a response to the invocation.
   - Data: Abstract Syntax Tree - AST representing ABS code
2. *View Data and Functions in Context*
   - Input: ABS Files - Takes as input files containing ABS programmes
   - Provided Functionality: Command Line Interfaces - Can be invoked from command line or programmatically

- Provided Functionality: Java Interface - Provides API to uniform Java representation of ABS programmes

3. *View Technologies at Runtime*

   - Java Runtime Environment: The tool is executed on the JRE.

4. *View Technologies at Development Time* The tool is implemented in Java using the Beaver parser and the JastAdd framework for the AST representation.

   - Beaver is used as parser generator.
   - JastAdd is used to generate the AST representation (see `http://jastadd.org`).

2.4.2 Feature and Delta Modelling Extension

The feature modelling extension of ABS provides means for encoding and analysing feature models. The delta modelling extension provides *delta modules*, which are used to encapsulate behavioural artifacts as ABS code, and control their inclusion in the final software product. The two facilities are connected through the *product line configuration*, a flexible mechanism based on *application conditions*: the selection of features affects the inclusion/exclusion of code provided in delta modules. The feature and delta modelling capabilities of ABS have been described in detail in project deliverable D1.2 [9].

Goals

- Support SPLE - These ABS language extensions are provided to support the development of Software Product Lines (SPL) in ABS.

Use Cases

- Feature Model Analysis - Find the valid products of a feature model
- Feature Model Analysis - Check the validity of a given product with respect to the feature model
- SPL Specification - Fully specify software product lines
- Product Specification - Specify the software products of an SPL that are of particular interest
- Product Generation - Generate the code for given, valid software products

Important Architectural Views

1. *View Data and Functions in Context* The tool takes the AST of an ABS programme (which represents an SPL) and generates a modified AST representing a particular, valid product of that SPL.

   - Provided Functionality: Feature Model Analysis - The tool provides a command line interface that allows analysing the feature model.
   - Provided Functionality: AST Flattening/Delta Application - The tool provides the functionality to "flatten" the AST, generating the code for a valid product of the SPL. This flattened AST can then serve as input to one of the various compiler backends (Java, Maude, etc.), which will generate an executable software product.

2. *View Processes in Context* The tool is an integral part of the ABS Frontend, with command line and IDE switches that activate its provided functionality.

3. *View Data and Functions at Runtime* The tool manipulates the AST representation of ABS programmes; no other external data is used or provided.

   - Data: Abstract Syntax Tree - AST representing ABS code.

4. *View Processes at Runtime* The tool provides its functionality in the ABS compiler chain as indicated by the *Rewriter* box in Figure 2.4.

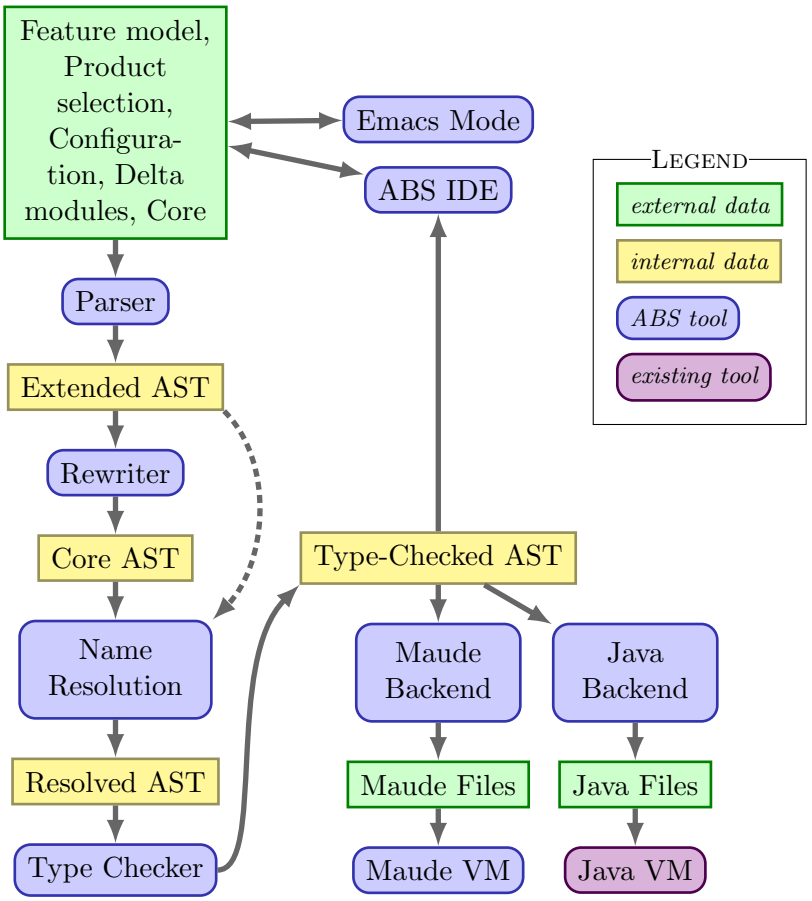

Figure 2.4: Overview of the ABS compiler framework.

5. *View Technologies at Runtime*

   - Java Runtime Environment - The tool is executed on the JRE.

6. *View Technologies at Development Time*

   - Java - is used as the implementation language.
   - Beaver - is used as parser generator.
   - JastAdd - is used to generate the AST representation.

2.4.3 PEKeY

Goals

- Perform deductive compilation of ABS models into Java.

Use Cases

- The basic use case is to use the tool as an additional ABS-to-Java compiler. The code can then be used to execute ABS models on any Java platform.

Important Architectural Views

1. *View Data and Functions in Context* The tool uses the same infrastructure as the KeY-ABS tool. For further information on related tools, please see Section 2.4.5. We mention here only the additional input and output formats in detail.

   - Input: ABS model - The tool starts a verification attempt for a specially crafted proof-obligation formula to construct a suitable proof object.
   - Input: KeY Proof Representation - The internal proof object constructed by KeY during a proof attempt.
   - Output: Text file with Java code - Text file containing the Java code for the compiled ABS model.
   - Used Functionality: KeY-ABS - The tool uses the KeY-ABS tool to construct a proof tree on which it performs the compilation process.
   - Provided Functionality: ABS-to-Java compiler - The tool provides an automatic ABS-to-Java compiler.

2. *View Data and Functions at Runtime*

   - Data: Proof Object - Internally, the proof object (basically a tree) is processed to perform the compilation.
   - Functionality: ABS to Java - A converter from the ABS model definition to a textual Java programme.

3. *View Technologies at Runtime*

   - Java Runtime Environment - The tool is executed on the JRE and requires at least Java SE7 to run.

2.4.4 Eclipse Plugin

Goals

- The main goal is to facilitate access to the HATS tool suite by providing an integrated development environment to model ABS programmes, run and debug said programmes, and run various analyses. The IDE is realised as an Eclipse plugin.

Use Cases

- Modelling - The IDE provides a user experience to the ABS modeller in a similar way as for writing regular Java/Scala/C programmes, e.g., a module explorer, outline view, syntax and semantic highlighting, content assist, code navigation, code completion, ...

- Execution - The IDE provides a simple yet powerful way to configure and run ABS programmes. Furthermore, a visual debugger is integrated into the IDE, which allows inspecting the runtime programme state.

Important Architectural Views

1. *View Processes in Context* The Eclipse plugin can simply be installed and updated with the use of an Eclipse Update site (containing all the dependencies, frontend, backends, ...).
2. *View Data and Functions at Runtime* The ABS Java runtime environment allows running the generated Java code.
   - Used Functionality: Frontend - The AST representation of the ABS programme is used for the different views and functionalities, e.g., outline view, module explorer, code navigation, code completion, ...
   - Provided Functionality: Menus and Views - The plugin allows hooking into the various menu items and view options.
3. *View Technologies at Runtime*
   - Java Runtime Environment - The tool is executed on the JRE.
   - Eclipse platform - The tool runs as an Eclipse plugin.
4. *View Technologies at Development Time*
   - Eclipse platform - The tool is realised as an Eclipse plugin.

2.4.5 KeY-ABS

Goals

- KeY-ABS is a deductive software verification tool derived from the KeY verification system for Java Card. Its main goal is to verify that an ABS model adheres to a given state-based functional specification. The specification follows the design-by-contract paradigm.
- The implemented logic calculus is based on symbolic execution. This allows also to use the tool as a symbolic execution engine for other tools like test generation or symbolic state debugging.

Use Cases

- The user specifies the ABS model by providing invariants and operation contracts using e.g., the attribute grammar based specification language from CWI or the trace-based language from UKL. The contracts are then translated into the KeY dynamic logic for ABS, which is the native format for the KeY prover. Verification is performed compositionally and done for each method in isolation. After loading, the KeY prover can be run using the automatic strategies. If the strategies succeed, the ABS method under consideration has been proven correct, otherwise the user needs to investigate whether interactions are necessary to close the proof or if the reason is a bug in the specification or in the ABS model.

Important Architectural Views

1. *View Data and Functions in Context*
   - Input: ABS model - The tool uses the ABS backend to parse the ABS model and supports all the input formats that the backend supports.
   - Input: High-Level Specification - For parsing, the tool relies on the backend for parsing specification languages. As it only uses the internal programmatic representation of the parsed specifications, it supports all the formats that the respective backends support.

- Input / Output: KeY problem files - KeY-ABS supports also its native ASCII text based format for specifying properties using dynamic logic. The same format is used to save proofs.

The tool can be used as an automatic prover from the command line or as a semi-automated prover using the graphical user interface.

2. *View Processes in Context* The tool is intended to support the step of proving that the ABS model satisfies user-specified state-based properties.
3. *View Data and Functions at Runtime*
   - Data: Abstract Syntax Tree - The specification and ABS model are translated into internal data structures of KeY. These data structures are basically abstract syntax trees. A speciality is that most of the ASTs are immutable.
4. *View Processes at Runtime* The work-flow of the tool is as follows: The ABS code is loaded together with the property to be verified. To check that the code satisfies the stated property, the user can start the automatic proof search strategy. If the proof cannot be closed automatically, the user can browse the created proof tree in order to understand the current proof situation. Depending on the situation, the user can either identify a bug in the specification or programme or interact with the system by performing some interactive proof steps before continuing with the automatic proof search.
5. *View Technologies at Runtime*
   - Java Runtime Environment - The tool is executed on the JRE and requires at least Java SE7 to run.
6. *View Technologies at Development Time*
   - Eclipse - The tool is developed using the Eclipse IDE (see `http://www.eclipse.org/`) and provides a compatible project structure. However, other IDEs like Netbeans can also be used.

2.4.6 aPET

aPET is a tool for the automatic generation of glass-box test cases for ABS programmes based on symbolic execution in Constraint Logic Programming (CLP).

Goals

- Symbolic execution of ABS in CLP - Design and implementation of a CLP-based framework for the symbolic execution of ABS programmes.
- Generation of test cases - Implementation of a test case generation framework based on symbolic execution in CLP.
- ABSUnit code generation - Implementation of a code generator that produces ABSUnit test cases.

Use Cases

- Given an ABS programme and a set of selected methods/functions, the user can generate a set of test cases for each selected method/function in textual form in order to observe the methods'/functions' input-output behaviour.
- Given an ABS programme and a set of selected methods/functions, the user can generate ABSUnit tests.
- Given the set of ABSUnit tests generated by aPET, the user can check whether the obtained test oracles are as expected, or edit them otherwise, and run the tests in the ABSUnit test executor.

Important Architectural Views

1. *View Data and Functions in Context* aPET takes as input a set of ABS files and outputs a set of test cases, both in a textual format and in the ABSUnit format. Optionally, it can take other parameters to control its functionality. aPET can be used both from a commandline and from the ABS Tool Suite Eclipse plugin.

   - Input: ABS Files - aPET takes as input a set of ABS files.
   - Input: Parameters - aPET allows controlling several parameters of the test case generation process: (1) the coverage criterion, with independent limits on both the number of task interleavings allowed and the number of loop unrollings performed in each parallel component; (2) the scheduling policy; (3) the maximum number of tasks in the initial object's queue; (4) whether to get concrete test cases or path constraints, etc.
   - Output: Text - One output of aPET is a set of test cases in textual format.
   - Output: ABSUnit - Another output of aPET is a set of test cases in the ABSUnit format.
   - Used Functionality: Compiler frontend - The input ABS files are first processed by the Compiler frontend.
   - Provided Functionality: ABSUnit Test Executor - aPET generates ABSUnit test cases that can be run using the ABSUnit test executor.

2. *View Processes in Context* Within the ABS Tool Suite, the user can select a set of methods and functions from the outline view of the opened ABS file, and ask aPET to generate a set of test cases for them. The generated test cases can be seen in the console view in textual format, and are also stored in XML format. In the context of testing with ABSUnit, this XML file is parsed and the test cases are generated into the ABSUnit format, so that they can be edited or run with the test executor of ABSUnit.

3. *View Data and Functions at Runtime*

   - Data: Intermediate representation - The input ABS programmes are transformed into an intermediate representation shared with COSTABS, which is suitable for programme analysis and transformations.
   - Data: CLP-transformed programme - The input ABS programmes are transformed (via the above intermediate representation) into an equivalent CLP programme, which can be run and/or symbolically executed in CLP.
   - Data exchange: Test cases in XML format - The test cases are represented in XML format to be parsed by the ABSUnit code generator.
   - Functionality: Generation of test cases - aPET generates a set of test cases which are output in textual form and stored in XML format.
   - Functionality: Generation of ABSUnit tests - The generated test cases, represented in XML format, are parsed, and the corresponding ABSUnit tests are generated.

4. *View Technologies at Runtime*

   - Prolog - aPET requires SWI-Prolog in order to execute. However, on Linux, it is available as a standalone executable that includes SWI-Prolog.

5. *View Technologies at Development Time*

   - Prolog - aPET is written in Prolog, in particular SWI-Prolog, and uses its finite domain constraints solving library *clpfd*.
   - XML parsing - The parser of the XML file uses the Document Object Model (DOM), which is a component of the Java API for XML Processing.

2.4.7 LBTest

Goals

- The main goal of the LBTest tool is to perform fully automated black-box requirements testing of reactive systems (including ABS models) using the method of *learning-based testing*. Since LBTest implements black-box testing, it is not specifically tied to the ABS language. In principle, any executable ABS model can provide a *system under test* (SUT) for testing with LBTest.

  User requirements on the system must be expressed as linear temporal logic (LTL) formulas. Through the use of learning and model checking methods, LBTest searches for a violation of a user requirement (LTL formula) as a failed test case. Note that LBTest automatically generates its own test cases either by (i) model checking, (ii) model inference or (iii) randomly.

  The tool returns a report on the status of testing for each individual requirement formula. For an individual requirement formula, the main information returned is: (i) the total number of test cases executed, and (ii) the test verdict: pass/fail/warning.

Use Cases

- GetRequirements - This allows the end-user to enter a set of user requirements into the system from different sources.
- GetSUT - This allows the end-user to link testing to an external system under test.
- SetTestOptions - This allows the end-user to choose various options for the testing process.
- ExecuteLBT - This invokes the learning-based testing procedure once all preconditions for testing have been established (set of user requirements, link to SUT, etc).
- Report - This produces reports subsequent to testing:
  - ReportRegression - In the case of a 100% passed test set, these can be saved in a file for test reporting and future regression testing.
  - ReportDiagnosis - In the case of one or more failures or warnings, these can be saved in a file for test reporting and fault diagnosis.
  - ReportModel - For enhanced fault diagnosis, coverage analysis and SUT code documentation, the inferred model can be exported in graphical format.

Important Architectural Views

1. *View Data and Functions in Context* The tool takes a set of user requirements and a SUT, and then constructs test cases in an attempt to show that one or more user requirements fail. It actively searches for such failed test cases using computational learning and model-based testing techniques, and reports on the results. It can be invoked via a graphical user interface.
   - Input: User requirement - Currently, we use the language of *propositional linear temporal logic* (PLTL) to formally represent user requirements. On the level of abstract syntax, there is agreement in the model checking community on how to write such formulas. However, currently no metalanguage (e.g., XML) is used in LBTest for PLTL requirements. Instead, we use the concrete syntax of NuSMV to define and parse PLTL formulas.
   - Input: SUT - Currently, the SUT is assumed to be a Java programme. Connection to this SUT is achieved through the Java ProcessBuilder class.
   - Input: SUT Output - The output from the SUT (a Java programme which is assumed to implement a reactive system) is a Boolean array.

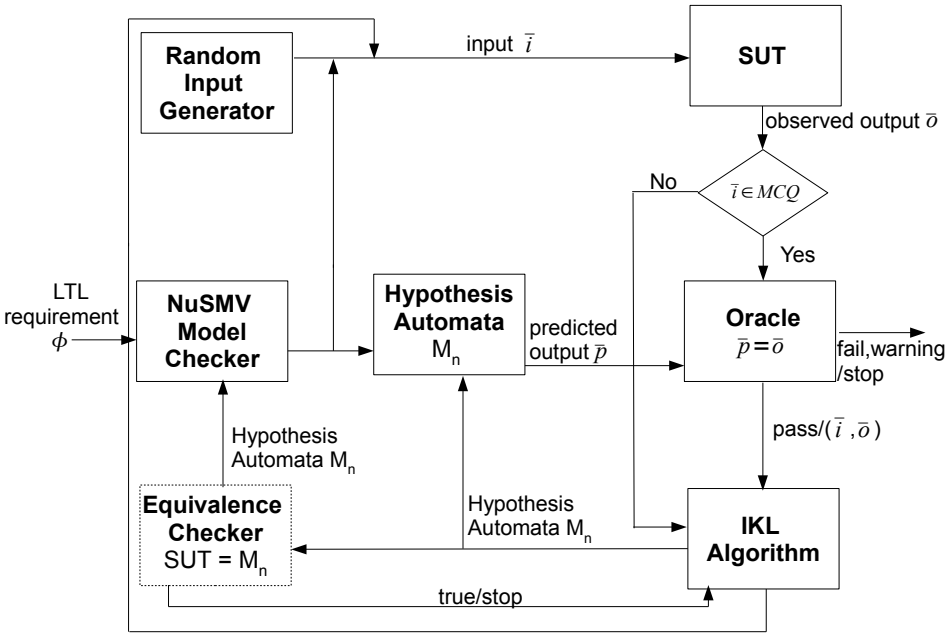

Figure 2.5: The architecture of LBTest.

- Output: SUT Test Case - A test case that is to be executed on the SUT is sent as an input to the SUT in the form of a Java string over the current input alphabet.
- Output: Test Report - Currently, we are formulating structures in XML format for exporting test reports for: (a) regression testing and (b) fault diagnosis.
- Output: Graphical Model - The inferred model obtained by testing can be exported in .dot format for visualisation using external tools.
- Used Functionality: Model Inference - The ExecutionLBT use case invokes a learning algorithm for Kripke structures to infer a model from test cases.
- Provided Functionality: Model Checking - The ExecutionLBT use case invokes a model checker to construct test cases.
- Provided Functionality: Test Case Generation - The ExecutionLBT use case invokes automated test case generation.
- Provided Functionality: Oracle Step/Verdict Construction - The ExecutionLBT use case invokes automated verdict construction with pass/warning/fail verdicts.
- Provided Functionality: Report - The Report use case automates report construction after testing is finished.

2. *View Data and Functions at Runtime*

The global architecture of LBTest is given in Figure 2.5.

- Data: Hypothesis Automaton - Internally, a deterministic Kripke structure is represented by its input alphabet Σ, state set Q, state transition function $\delta : Q \times \Sigma \to Q$, and output function $\lambda : Q \times \Sigma \to B^k$
- Data: input - A test case (aka input) is a sequence of input elements over Σ.
- Data: output - Predicted and observed outputs are sequences of Boolean vectors of fixed length k.

- Data: verdict - A test verdict is a string.
- Data: SUT Observation Table - Inputs and outputs are recorded in a SUT observation table which is used by the IKL learning algorithm to synthesise a Kripke structure as the hypothesis automaton.
- Functionality: IKL Algorithm - Used to synthesise a Kripke structure as the hypothesis automaton. This algorithm has been formally proved to correctly learn given sufficient data.
- Functionality: NuSMV Model Checker - This system checks each hypothesis automaton M_i, for $i = 1, 2, \ldots$ against the current PLTL requirement formula ϕ being tested. Note that only one user requirement formula is under analysis at any one time. Any counterexample to the correctness of ϕ can be used as the next test case.
- Functionality: Equivalence Checker - This component is only used for tool bench-marking to determine when the learning process has finished. It is switched off under normal circumstances.
- Functionality: Oracle - This component automates verdict construction by comparing a predicted output with an observed output using a simple equality test.
- Functionality: Random Input Generator - In the case that no counterexample is found by model checking, a random test case is generated in order to continue refining the current model M_i. From the point of view of learning theory, this can be seen as a stochastic solution to the equivalence checking problem.

3. *View Processes at Runtime* The workflow of the tool is to first execute use cases 1, 2 and 3 to set up a testing experiment. After successful setup, use case 4 executes the test experiment. After a brief interactive inspection of the outcome, use cases 5.1, 5.2 and 5.3 can be invoked for reporting as appropriate on an individual requirement basis. Then the testing experiment is finished. Typically the tool will be run in the background overnight, as an experiment may take several hours to complete. So interaction with the tool is currently confined to the initial and final phases of a testing experiment.

4. *View Technologies at Runtime*

   - Java Runtime Environment - The LBTest tool and the SUT are both executed on the JRE and require at least Java SE7 to run.
   - Graphviz - For visualisation of the inferred models, the open source graph visualisation tool Graphviz is used (see `http://www.graphviz.org/`).
   - NuSMV - For model checking, the NuSMV model checker is used (see `http://nusmv.fbk.eu/`).

5. *View Technologies at Development Time*

   - ABS Tool Set - We assume an ABS model is developed elsewhere in the ABS tool set. There is currently no interaction with this activity. Feedback from testing back into the ABS model development process is achieved by report artifacts. Any XML visualisation technology could be applied to close this loop.

     In principle, failed/warned user requirements could be diagnosed either by manual inspection and/or by using a debugger.
   - Verification Tools - Failed/warned user requirements (particularly failed liveness requirements) could possibly also be diagnosed/confirmed using verification/theorem proving techniques. We have not explored these options.

2.4.8 CVPP: Compositional Verification of Programmes with Procedures

Goals

- To verify *temporal safety properties* of procedural programmes in a *modular* fashion, thus supporting spatial and temporal *variability*. The properties concern safe sequences of method invocations. Modularity is achieved by relativisation of the global temporal property on local temporal assumptions on the modules' behaviour.

Use Cases

- Typically, the user annotates all modules with local temporal assumptions. Note that this does not require having the actual code. From these assumptions, *maximal programme models* (in terms of *flow graphs*) are constructed and composed. The composed programme model is model checked against the global property. Independently, from all modules for which code is available, programme models (again: flow graphs) are extracted and model checked against their respective local specifications (assumptions) using push-down system representations of flow graph behaviour.
- In the case of *software product lines*, a *hierarchical variability model* is constructed first in the solution space (code). Then, in addition to the specifications mentioned in the first use case, all *variation points* are also annotated. These specifications serve as local properties for the current *module* of the hierarchical model, but get inherited by their children to serve as global properties for the children's modules. So, every module defines a separate verification task as outlined in the first step of the previous use case.

Important Architectural Views

1. *View Data and Functions in Context*
   - Input: annotated Java programme - Java programme annotated with global and local specifications.
   - Used Functionality: SAWJA - An external static analysis tool for Java byte-code, used by CVPP for the extraction of flow graphs.
   - Used Functionality: MOPED - An external model checker for push-down systems.
   - Provided Functionality: Verification - The tool performs the verification of the local specifications and the check of entailment of the global property by the local specifications. All results are stored; when the programme or the specifications change, the least effort is performed that is needed to re-verify the system.
2. *View Data and Functions at Runtime*
   - Data: Flow Graph - The programme model, extracted from code and constructed from temporal formulas.
   - Data: Formula - Various formats for specifying temporal safety properties at both the structural and the behavioural level.
   - Functionality: Flow Graph Extraction and Construction - The tool performs the extraction of flow graphs from Java (byte)code, as well as the construction of (maximal) flow graphs from the local specifications.
   - Functionality: Verification of Local Specifications - The extracted flow graphs are model checked against their respective local specifications.
   - Functionality: Verification of Global Specifications - The constructed (maximal) flow graphs are composed and model checked against the global specification.

3. *View Technologies at Development Time*

    - OCaml - Apart from the external tools, almost all tools comprising CVPP are written in OCaml.

2.4.9 ABS Product Configurator

Goals

- Support Quality Aware Configuration - The configurator provides a mechanism to support the configuration of products from Software Product Lines (SPL) in ABS considering performance and security.

Use Cases

- Feature Model Visualisation - Visualise the μTVL, the feature model of an ABS SPL.
- User Concern Elicitation - User specifies required features and other quality requirements.
- Real-time Guidance - Check the current selection of features with respect to the feature model.
- Functional Configuration - Configure and correct incomplete and erroneous configurations with minimal changes.
- Quality Aware Configuration - Configure optimal product based on prioritised quality attributes.
- Product Specification Generation - Generate the specification (SPL) for the output configuration selected by the user.

Important Architectural Views

1. *View Data and Functions in Context* The tool takes the AST of an ABS programme (which represents an SPL) and generates a PSL representing a particular valid product of that SPL.

    - Provided Functionality: Feature Model Visualisation - The tool visualises the feature model graphically, which allows the user to specify her configuration requirements.
    - Provided Functionality: Functional Configuration - The tool provides the functionality to configure the product that is partially configured by the user and erroneous.
    - Provided Functionality: Quality-Aware Configuration - The tool provides the functionality to configure the product based on the optimisation of multiple quality attributes.

2. *View Processes in Context* The tool is developed as an Eclipse plugin; with the provided GUI, the user can activate its provided functionality.

3. *View Data and Functions at Runtime* The tool uses the AST representation of ABS programmes (more specifically μTVL and its quality extensions); users also provide the quality requirements as input.

    - Data: Abstract Syntax Tree - AST representing μTVL.
    - Data: Quality requirements - prioritisation among various quality attributes.

4. *View Technologies at Runtime*

    - Java Runtime Environment - The tool is executed on the JRE.

5. *View Technologies at Development Time*

    - Java - is used as the implementation language.
    - FeatureIDE - is used to visualise the feature model.
    - JastAdd - is used for aspect orientation.

2.4.10 COSTABS

COSTABS is a tool for inferring, verifying, and understanding the resource usage consumption of ABS programmes.

Goals

- Inference of resource usage upper bounds for ABS programmes.
- Verification of resource usage upper bounds for ABS programmes.

Use Cases

- Given an ABS programme, the user can infer upper bounds on the resource usage of the different methods of the ABS programme.
- The user can verify that the inferred upper bounds meet the expected ones.
- The user can infer the upper bounds for the different components, which help to understand the resource usage distribution among different objects (or groups).

Important Architectural Views

1. *View Data and Functions in Context* COSTABS takes as input a set of ABS files and outputs a set of upper-bound functions, in textual format. Optionally, it can take other parameters to control its functionality. COSTABS can be used from a command line, a web interface and an Eclipse plugin.
    - Input: ABS Files - COSTABS takes as input a set of ABS files.
    - Input: Parameters - COSTABS allows controlling several parameters of the analysis: (1) the form of the upper bounds, e.g., asymptotic or non-asymptotic; (2) the resource we want to measure, e.g., memory, execution steps, user defined, etc; (3) the way the size of a data structure is measured; (4) the granularity of the upper bounds, e.g., object level, class level, application level, etc.
    - Output: Text - The output of COSTABS is a set of upper-bound functions in textual format.
    - Used Functionality: Compiler frontend - The input ABS files are processed first by the Compiler frontend.
    - Provided Functionality: inference of upper bounds - The tool provides a command line interface that allows inferring the resource usage of ABS programmes. It also provides the same functionality through a web interface and an Eclipse plugin.
2. *View Data and Functions at Runtime*
    - Data: Intermediate representation - The input ABS programmes are transformed into an intermediate representation which is suitable for programme analysis.
    - Functionality: Programme analyses - COSTABS applies several programme analyses to understand how data changes when moving from one part of the programme to another.
    - Functionality: Generating Cost Relations - COSTABS generates a set of recurrence relations that describe the resource usage of the different parts of the ABS programme. These relations are then solved by a dedicated solver, called PUBS, which is part of COSTA.

3. *View Technologies at Runtime*

   - Prolog - COSTABS requires SWI-Prolog in order to execute; however, on Linux, it is available as a standalone executable that includes SWI-Prolog.
   - PPL - COSTABS requires the Parma Polyhedra Library (PPL) for manipulating linear constraints, which is used in the underlying static analysis. PPL must be installed in order to execute COSTABS; however, in Linux, it is included in the standalone executable (see previous point).

4. *View Technologies at Development Time*

   - Prolog - COSTABS is written in Prolog, in particular SWI-Prolog.

2.4.11 Scala Backend

Goals

- Compiler Backend - Generate Scala code from ABS programmes
- ABS Scala Runtime - Provide a runtime environment for ABS in Scala

Use Cases

- Distributed Execution - The ABS Scala backend allows the execution of concurrent object groups on different (physical) nodes.
- Integration - The tool outputs Scala source code, which may be integrated with other Scala code.
- Debugging - The communication history between objects can be logged and used for debugging purposes.

Important Architectural Views

1. *View Data and Functions in Context* The tool takes the Java AST representation of the ABS programme from the ABS Frontend.
2. *View Data and Functions at Runtime* The compiler backend takes the Java representation of the ABS programme and generates Scala source code. The Scala runtime library allows the compilation of the generated code with the standard Scala compiler, and is used during the execution of the programme.
3. *View Technologies at Runtime*

   - Java Runtime Environment - The tool is executed on the JRE and requires at least Java SE 6 to run.
   - Scala libraries - The tool requires Scala 2.9 and the continuations plugin for the Scala compiler.
   - Akka actor libraries - The tool requires Akka 2 libraries.

2.4.12 Maude Backend

Goals

- Compiler Backend - Generate Maude input from ABS programme
- ABS Java Runtime - Provide an interpreter for ABS in Maude

Use Cases

- Observation - The ABS Maude interpreter gives a whole-system state in textual form, either at system termination time or after a number of reductions.
- Simulation - The ABS Maude interpreter can be used to simulate timed and untimed ABS models.
- Resource Modelling - Resource-aware models (with statement costs and deployment scenarios) can be inspected for effect.

Important Architectural Views

1. *View Data and Functions in Context* The tool takes the Java AST representation of the ABS programme from the ABS Frontend and generates the corresponding Maude code.
   - Simulation output - The ABS Maude interpreter provides a textual representation of the simulation results and the whole-system object state.
2. *View Processes in Context* Both code generation and simulation on the Maude platform can be run in the following ways:
   - Stand-alone, from the command line
   - From the Eclipse plugin
   - From the Emacs text editor
3. *View Data and Functions at Runtime* The code generator of the Maude backend converts the AST from the parser and type-checker into Maude code as a file to be run in the Maude rewriting engine.

   The ABS Maude interpreter allows running the generated Maude code, and produces a text-only whole-system view of the system state.
4. *View Processes at Runtime* The code generator is integrated into the ABS compiler – parsing, type-checking and code generation is seen as one step from the user's perspective.

   The ABS Maude interpreter can be used either integrated into the Eclipse platform, integrated into the Emacs text editor, or stand-alone in a terminal.
5. *View Technologies at Runtime* Code generation needs no extra tools. Execution of models is done on top of the Maude rewrite engine (`http://maude.cs.uiuc.edu/`).

2.4.13 Java Backend

Goals

- Compiler Backend - Generate Java code for ABS programme
- ABS Java Runtime - Provide a runtime for ABS in Java

Use Cases

- Observation API - The ABS Java runtime provides hooks for inspecting the running ABS programmes (e.g., for debugging purposes or drawing sequence diagrams).
- Scheduling API - The ABS Java runtime provides the possibility to run ABS programmes with different schedulers.
- Foreign Function Interface - The ABS Java runtime provides the possibility for ABS code to interact with Java code.

Important Architectural Views

1. *View Data and Functions in Context* The tool takes the Java AST representation of the ABS programme from the ABS Frontend and generates the corresponding Java code.
   - Provided Functionality: Observation API - The ABS Java runtime provides interfaces for writing custom observers.
   - Provided Functionality: Scheduling API - The ABS Java runtime provides interfaces for writing custom schedulers.
   - Provided Functionality: Foreign Function Interface - The Java backend generates stubs that allow overriding with legacy Java code.
2. *View Data and Functions at Runtime* The ABS Java runtime allows running the generated Java code.
   - Provided Functionality: Observation API - The ABS Java runtime allows registering observers (e.g., the debugger). During a run of the ABS programme, the runtime then generates events corresponding to typical runtime actions (method called, object created, ...) of ABS programmes and sends them to the observers.
   - Provided Functionality: Scheduling API - The ABS Java runtime allows setting schedulers (e.g., random scheduler, FIFO scheduler, Eclipse interactive scheduler, ...). During a run of the ABS programme, the runtime then schedules the COGS and tasks according to the scheduler that was set.
   - Provided Functionality: Foreign Function Interface - The ABS Java runtime allows communication with legacy Java code.
3. *View Technologies at Runtime*
   - Java Runtime Environment - The tool is executed on the JRE.

2.4.14 JMS2ABS

JMS2ABS is a tool that automatically extracts ABS models from the byte-code of JMS (Java Messaging Service) applications. In particular, the tool targets the JMS Publish/Subscribe model.

Goals

- Assist developers in building ABS models of their JMS applications.
- Help developers understand the properties of their JMS applications by facilitating the application of HATS tools.

Use Cases

- Annotate a JMS application and run JMS2ABS on it. Then proceed to analyse the obtained ABS models using HATS tools.

Important Architectural Views

1. *View Data and Functions in Context*
   - Input - JMS Publish/Subscribe application.
   - Output - ABS Model.
   - Functionality provided - The tool obtains an ABS model from the input JMS application.

- Interaction with other HATS Tools - The resulting ABS model can be analysed and simulated with the existing tools for ABS, e.g., COSTABS, the Maude simulator.
- Usability - The tool is usable from the command line.

2. *View Data and Functions at Runtime*

   - Data: Intermediate Rule-based Representation - Obtained from the Java bytecode of the input application and stored as Prolog terms. This translation is reused from COSTA, a COSt and Termination Analyser for Java bytecode developed by UPM.
   - Data: Abstract Syntax Tree - The main translation phase of JMS2ABS writes the ABS model in AST syntax.
   - Data: ABS code - As a last step, JMS2ABS transforms the AST syntax into pretty-printed executable ABS files.

3. *View Technologies at Runtime*

   - SWI-Prolog - SWI-Prolog (http://www.swi-prolog.org/) is required to execute JMS2ABS. However, a standalone executable version of the tool is available for Linux systems.

4. *View Technologies at Development Time*

   - Prolog - JMS2ABS is written in Prolog, using the open source implementation SWI-Prolog.

2.4.15 RascalABS

RascalABS allows meta-programming on ABS models. It is a framework that allows rapid development of ABS parser extensions, model analyses and programme transformations as well as ABS compiler backends. This has been achieved through the adoption of the meta-programming language Rascal, which provides powerful parsing and pattern matching techniques.

Goals

- To enable various *static* analyses directly on the level of the ABS model.

Use Cases

- Code Analysis - Extracting relevant information directly from the source code of an ABS model. Examples are code metrics, generation of documentation, construction of an abstract or concrete syntax tree of an ABS programme, syntax-highlighting and extracting a call graph of an ABS programme. Extracted information is stored in an internal representation (Rascal data types specifically developed for this purpose), which can be used as input for code transformation or code generation.

- Code Transformation - Transform an ABS model (input) into another ABS model (output). Examples include the elimination of syntactic sugaring (and more generally, pre-processors) to produce a pure Core ABS abstract syntax tree, refactorings of ABS code and aspect-oriented programming, which can be implemented by instrumenting (also known as weaving) source code at appropriate user-defined places (pointcuts).

- Code Generation - Code generation of a different language from and to ABS. This can be used to add support of a domain-specific languages (DSL) to ABS (the input language is the DSL, the output language is ABS). As an example where ABS is translated into another language, RascalABS can be used to implement the code generation part for the various ABS backends in a systematic manner.

Important Architectural Views

1. *View Data and Functions in Context*
   - Input: An ABS model and a meta-programme which acts on the parse tree of the ABS model.
   - Output: Depends on the meta-programme. Ranges from code metrics (as a simple example, the total number of classes in the ABS model) to a refactored ABS model (see the above use cases for details).
2. *View Data and Functions at Runtime*
   - Data: In general, everything is stored in appropriate Rascal types. As an example, ABS models are stored in an intermediate representation (a Rascal ParseTree, to be precise).
   - Functionality: RascalABS allows writing meta-programmes that operate on parse trees of ABS models. The exact format of the output of the meta-programme varies depending on the task.
3. *View Technologies at Runtime*
   - Rascal - SAGA is implemented as a meta-programme in the meta-language Rascal.
   - Java SDK 1.6 or higher - Rascal itself needs at least the Java SDK 1.6 or higher.
   - Eclipse for RCP and RAP Developers Indigo (optional) RascalABS - For full functionality, Eclipse is needed.

2.4.16 SAGA

SAGA is a tool for runtime verification on ABS models. SAGA allows combining protocol-oriented properties, such as orderings of messages sent between COGs, with data-oriented properties, such as assertions over the data sent in these messages (i.e., the actual parameters of asynchronous method calls). Attribute grammars extended with assertions are used as specifications for COGs.

Goals

- To ensure that the behaviour of a COG, as observed by its interactions with the environment (other COGs), is safe. When an execution violates the given specification of the behaviour, unsafe and undefined behaviour should be prevented by means of an assertion failure.

Use Cases

- Testing and Debugging - SAGA allows users to test whether an actual execution of a COG in an ABS model satisfies the intended behaviour. The results of these tests are highly useful for debugging purposes, especially for locating errors.
- As a (benign) side-effect, SAGA also generates a parser generator for (simple) attribute grammars with ABS as the target language. This allows using, for example, regular expressions inside ABS programmes.

Important Architectural Views The architecture of SAGA is depicted in Figure 2.6.

1. *View Data and Functions in Context*
   - Input: The input consists of three ingredients
     - A *communication view*, which lists and names the asynchronous calls to methods that can potentially be used in specifications.

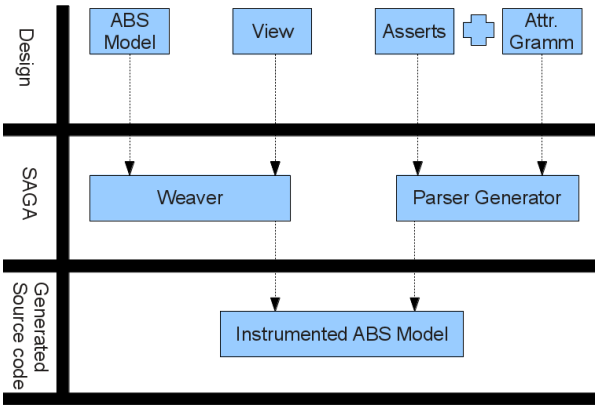

Figure 2.6: SAGA architecture.

  - An ordinary ABS model with annotations to indicate which of the asynchronous calls should actually be captured (note: this should be a subset of the methods listed in the communication view).
  - An *attribute grammar* extended with assertions that specify the legal orderings of the asynchronous calls and assertions that specify the legal values of the data sent in these calls.
- Output: The ABS model instrumented with updates to the communication history of a COG, including an ABS parser which throws an assertion error if the message is not allowed to be sent according to the specifications of the involved COGs.

2. *View Data and Functions at Runtime*

   - Data: From the input ABS model, SAGA builds its parse tree and stores it in a convenient ParseTree data type (a type provided by the meta-programming language Rascal).
   - Functionality: SAGA instruments the parse tree of the ABS programme with additional statements to update the history. Additionally, SAGA generates an ABS parser for the high-level attribute grammar.

3. *View Technologies at Runtime*

   - Rascal - SAGA is implemented as a meta-programme in the meta-language Rascal.
   - RascalABS - SAGA uses RascalABS to parse the input ABS model.

2.4.17 SDA

SDA is a tool to analyse deadlocks and livelocks in an ABS programme. More precisely, it studies the synchronisation between objects, generated by **await**, **get** and synchronous method calls, and checks if the graph of these dependencies has a cycle (thus proving that there is a lock in the analysed programme). Because of the undecidable nature of deadlock detection, SDA is overprotective, and may signal a cycle in the dependency graph while the programme has no deadlocks.

Goals

- Deadlock Detection - Given an ABS programme, the tool detects if it is deadlock-free or not.

Use Cases

- Given an ABS programme, we can detect if it is deadlock-free.

- Given an ABS programme, the programmer can test if the possible deadlocks found by the tool are actual deadlocks. If no true deadlocks are found, the programme is deadlock-free.
- The tool can be used to analyse the dependencies between objects at runtime, which can improve the understanding of the programme and its dynamics at runtime.

Important Architectural Views

1. *View Data and Functions in Context*
   SDA takes as input ABS source code and produces an output stating whether a deadlock or a livelock has been detected.
   - Input: ABS Files - SDA takes as input a set of ABS source files.
   - Output: Text - The textual output is a statement about the (possible) presence or absence of deadlocks in the input programme.
   - Used Functionality: Compiler frontend - The input ABS files are processed first by the Compiler frontend.
   - Provided Functionality: deadlock detection - The tool automatically executes to detect deadlocks in the input programme.
2. *View Processes in Context*
   The tool is an optional part of the ABS Frontend, which can be controlled via the command line.
3. *View Data and Functions at Runtime*
   - Data: Abstract Syntax Tree - The analysis takes as input the ABS AST.
   - Data: Contracts - The tool generates *Contracts* from an AST, capturing the behaviour of the ABS code, on which it performs the analysis.
   - Data: Dependency Sets - The analysis works by inferring sets of dependencies between objects, based on the computed contracts.
   - Functionality: Behaviour Description - Contracts capture dependencies created by method calls or synchronisation statements in the code of a method.
   - Functionality: Loop Detection - The Dependency Set is generated so we can detect a loop in the set of dependencies between objects.
4. *View Processes at Runtime*
   The workflow, divided in two parts, takes place within the ABS compilation tool chain, after the run of the type analysis on the AST. Using that AST, the first part of the SDA workflow generates a contract for every method in the input programme. The second part then uses these contracts to generate the sets of dependencies and detect deadlocks. Both parts are fully automatic and invisible to the user.
5. *View Technologies at Runtime*
   - Java Runtime Environment - The tool is executed on the JRE.
6. *View Technologies at Development Time*
   - Java - is used as the implementation language.
   - Beaver - is used as parser generator.
   - JastAdd - is used to generate the AST representation.

2.5 Towards an Industrial Tool Architecture

Most HATS tools were initially developed independently. However, a significant amount of effort has been spent to achieve integration, as described in Section 2.2. Currently any ABS developer who wants to build or verify his/her model in the ABS tool suit will feel that he/she is working in an integrated tool platform. But we wanted to go beyond that and define the steps towards an integrated tool platform that is highly scalable, efficient and maintainable. The current relationships among a partial set of tools are shown in Figure 2.7. This figure shows that there are numerous connections between the tools, so preserving interfaces will be a challenge in case of evolution. It also points out that the ABS Frontend has very high coupling with other tools. All these issues (interface preservation, high coupling and so on) are addressed in this section.

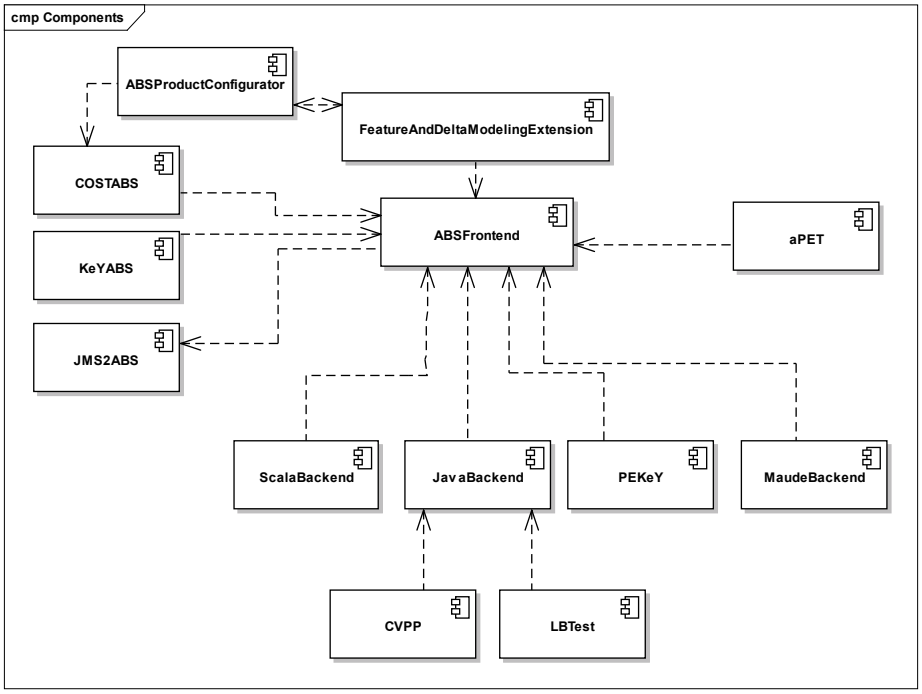

Figure 2.7: Current dependency relationships between different tools.

To design the architecture of an integrated tool platform, we first need to establish architectural requirements for such a platform. This helps us to determine whether the proposed architecture can meet those requirements. The most important architectural signification requirements (ASR) are described below in Tables 2.3, 2.4, 2.5, 2.6 using the scenario description template [6].

A number of functional and quality scenarios can be imagined for the ABS tool platform. Here we are addressing the most important ones. To achieve the mentioned architectural requirements, we developed the conceptual view of an integrated architecture, which is shown in Figure 2.8. The figure shows the main components of the to-be ABS tool platform.

The main idea behind the integrated architecture is that all activities (i.e., tool functionalities) will be made atomic so that they can be composed into a compound activity. For any two activities, if the output artifact of one activity matches with the input artifact of another activity, they can be composed with each other. Each activity will be described with its input and output artifact names. The architecture will take input from the user regarding his input and output artifacts, and with the help of the *Rule Engine* it can determine the sequence of activities that need to be performed to get the desired output artifact. Activities will be loaded dynamically based on the rule output. Dynamic loading will be performed through the use of the factory pattern.

The conceptual view shows that the developer will interact with the *UI Manager* only and no longer with the independent tools. The developer will be able to do any activity that is supported by HATS by using

Table 2.3: ASR Usability.

Scenario	One-point access to the developer		
Quality	Usability		
Stakeholders	ABS Developers		
Environment	ABS Developer is using ABS tool set		
Stimulus	ABS developer wants to perform some activity		
Response	The architecture takes the input from a common interface and provides the service, hiding the internal heterogeneous set of tools		
Response Measure	The architecture provides a common GUI for all activities		
Architectural Decisions	Risks	Sensitivity Points	Discarded Alternatives
Common GUI	GUI can be complicated	Separation between user interface and application logic	

Table 2.4: ASR Scalability.

Scenario	Any number of tools can be added in the future		
Quality	Scalability		
Stakeholders	ABS IDE Developers		
Environment	A set of tools are already integrated and running		
Stimulus	A new tool is developed either for development, verification, testing or any other purpose		
Response	The architecture integrates this new tool without significant effort		
Response Measure	Any new tool can be integrated and tested within one person-day		
Architectural Decisions	Risks	Sensitivity Points	Discarded Alternatives
Abstract Activity		Making each activity/artifact an instance of the abstract activity/artifact	Making all the tools different plugins and integrating them through extension points and extensions

Table 2.5: ASR Openness.

Scenario	Additional functionalities can be added in the platform		
Quality	Openness		
Stakeholders	3rd-Party Tool Developers		
Environment	The integrated tool is running under normal conditions		
Stimulus	Any developer or 3rd-party organisation wants to bring new functionalities or extend existing functionalities in the ABS development environment		
Response	The architecture provides an option for 3rd-party developers or organisations to deploy new functionality		
Response Measure	New functionality can be deployed and run in an integrated way		
Architectural Decisions	Risks	Sensitivity Points	Discarded Alternatives
Eclipse Plugin		Extension points of the plugin	

Table 2.6: ASR Interoperability.

Scenario	Member tools should be able to interoperate with each other		
Quality	Interoperability		
Stakeholders	ABS Developers		
Environment	ABS Developer is using ABS tool set		
Stimulus	ABS developer wants to perform some compound activity		
Response	The architecture makes sure that participating sub-activities can interoperate with each other to provide the high-level activity without bothering the user		
Response Measure	ABS developer performs some compound activity and can get the final result without problems		
Architectural Decisions	Risks	Sensitivity Points	Discarded Alternatives
Atomic Functionality		Making all the functionalities atomic and their input/ output standardised	

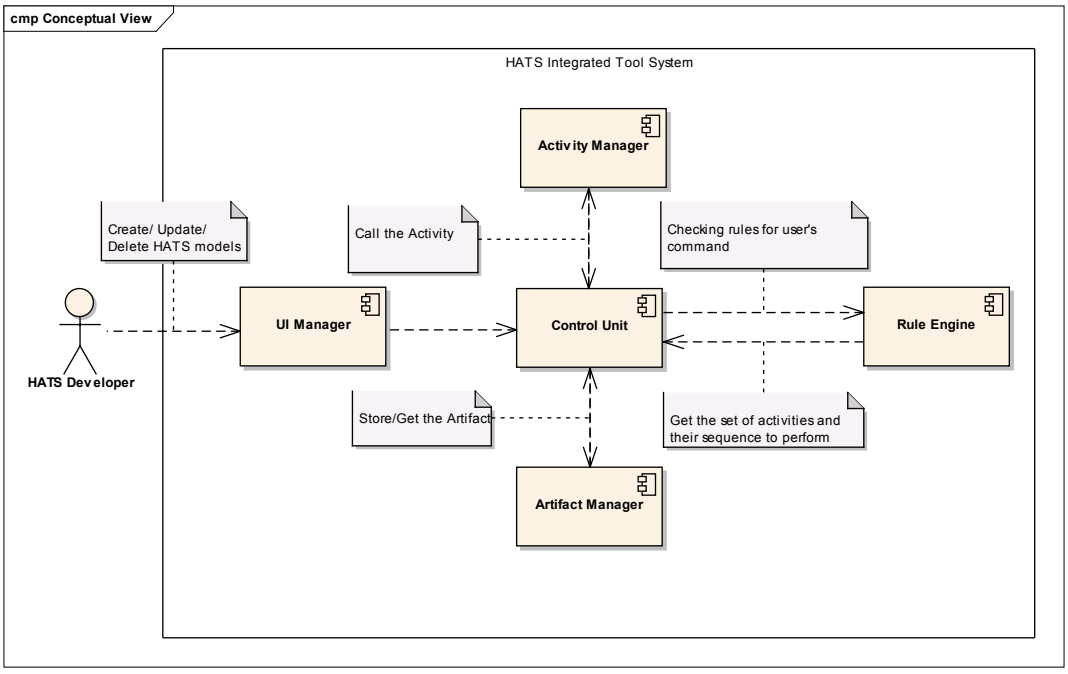

Figure 2.8: Conceptual view of the integrated platform.

this integrated UI. The developer's request will be sent to the *Control Unit*. In order to identify the set of activities and their sequence, the *Control Unit* will use the *Rule Engine*. Based on that, the *Control Unit* will know which activities to perform and in which order, including their artifacts. Appropriate activities and artifacts will be invoked by the *Activity Manager* based on the runtime requirements. The main components are described below.

- UI Manager: This component provides the user interface that is accessed by the HATS developers. It receives the request from the users, i.e., the HATS developers, and responds by communicating with the *Control Unit*. One example request could be: the developer has an ABS programme and he or she wants to generate test cases for it.
- Control Unit: This is the main mediator unit of the application which interacts with all the components involved in this application. It receives the activity to be performed and checks the rule by means of the *Rule Engine*, which identifies the sequence of activities that need to be performed. Figure 2.9 shows the internal conceptual structure of this component. It shows that the Invoker class aggregates a set of activities based on the activity list it gets from the *Rule Engine*. As it also has the activity sequence, it invokes all those activities sequentially. The activities can be called anonymously as they all are instantiations of the abstract activity. This component handles the artifacts in a similar way.
- Rule Engine: This component consists of the rules associated with all HATS activities. Every activity is described, including its input and output artifacts. It checks the rules and determines whether any valid composition of activities is possible. If composition is possible, it determines the activities need to be performed and in which order.
- Activity Manager: This component is responsible for managing all activities. It realises the Factory pattern to generate the desired concrete activity object during runtime.
- Artifact Manager: This component is responsible for managing all artifacts. It realises the Factory pattern to generate the desired concrete artifact object during runtime.

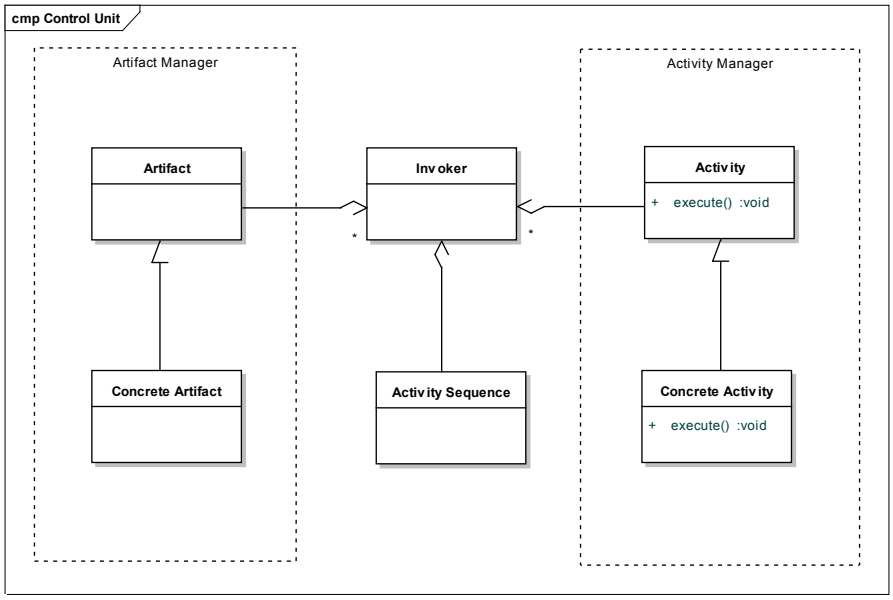

Figure 2.9: Control Unit.

To make the execution of the to-be integrated tool platform clear, we present the correspondent sequence diagram in Figure 2.10. It depicts how the components involved in the tool platform cooperate over time to respond to a user's (i.e., HATS developer's) action.

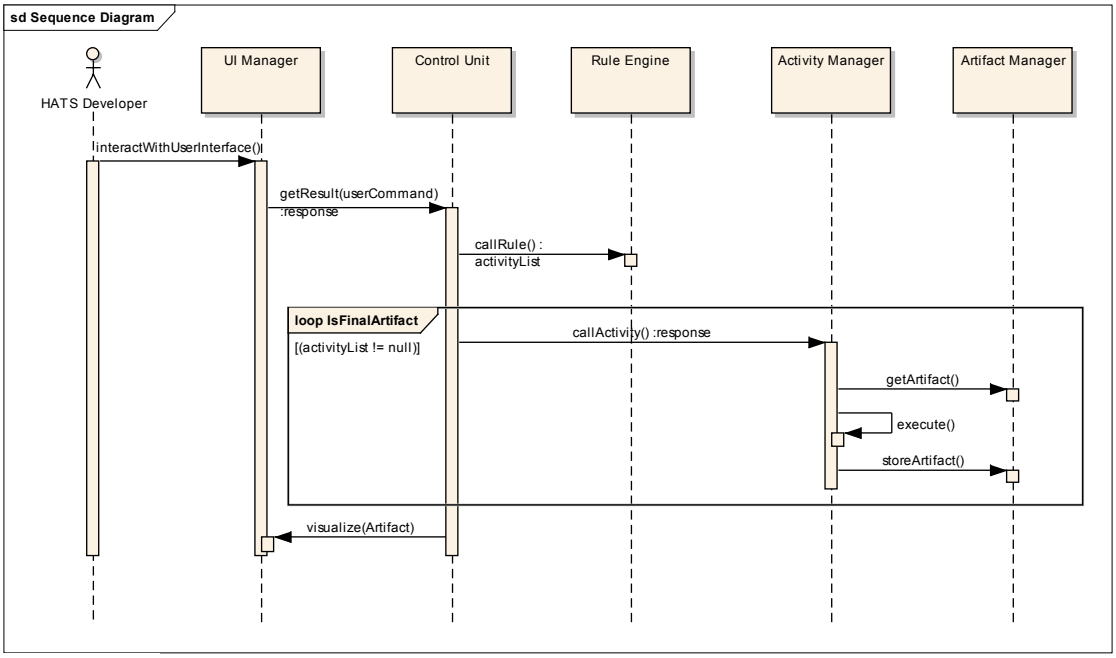

Figure 2.10: Sequence diagram of the integrated architecture.

1. The HATS developer accesses the User Interface of the *Integrated Tool Platform* application and commands the system to perform some HATS activities (e.g., Specification, Testing, Implementation, Verification, etc.)

2. The call is received by the *UI Manager*. The *UI Manager* sends the request to the *Control Unit*. The request can be any atomic activity, or the user can just specify the input and output artifacts.

3. Then the *Control Unit* asks the *Rule Engine*, which looks for the association of the activities with the specified artifacts and tries to compose a compound activity out of the sequence of registered atomic activities. The *Rule Engine* returns the set of activities along with their order of execution.

4. The *Control Unit* invokes the concrete activity with the respective artifact as input and gets the response.

5. Step 4 is executed until all activities are finished.

6. The *Control Unit* sends a response to the *UI Manager*.

2.5.1 Steps Needed for Migration

Analysing the current integration status described in Section 2.2 and the industrial level integration vision presented in Section 2.5, we identified steps to allow smooth migration. Those steps should also be followed by tools to be developed in the future.

- Plugin-oriented development: Most of the tools are already built as or converted into Eclipse plugins. Nevertheless, some tools (e.g., CVPP) are still standalone executable components. Those tools as well as all future tools need to be developed as Eclipse plugins.

- Wrapping incompatible tools: Some components are not built in Java. In this case, to avoid rework, a wrapper needs to be built in Java to hide the heterogeneity of the component, so that other plugins can easily communicate with this component.

- One atomic activity - one plugin: To enhance reuse and minimise high coupling, any tool that performs multiple activities needs to be split into multiple individual plugins. On the other hand, the part of the tools that use or call other components can just be removed. They can assume that the output of those calls will already be available as input. This is already handled in the proposed architecture.
- Exploiting extension points: To achieve openness, tools should open up their plugins through extension points and other tools should contribute by providing extensions. This mechanism will also make the platform customisable to any user's context.
- Developing the core platform: To bring all the developed plugins into action, we need to develop the proposed architecture described in Figure 2.8. This will handle the user interaction, provide an integrated look and feel of the overall tool suite, and also act as a mediator for communication between multiple plugins.

Concrete migration paths for each of the tools need to be derived from the above-mentioned steps. The migration path can then be used to extract concrete change requests, which can be implemented by the individual tool developers.

Chapter 3

HATS Development Methodology

This chapter presents the HATS development methodology. The methodology describes how to use the HATS tools to build software product lines. The methodology description is divided into two processes, the Family Engineering Process, described in Section 3.1, and the Application Engineering Process, described in Section 3.2. Each process is divided into phases and each phase is again divided into activities. Associated with an activity are stakeholders, artifacts involved in the activity, and a procedure for carrying out the activity. There is only a procedure for activities where the HATS project has a contribution. The other activities are included so that the methodology makes it clear what additional activities have to be performed.

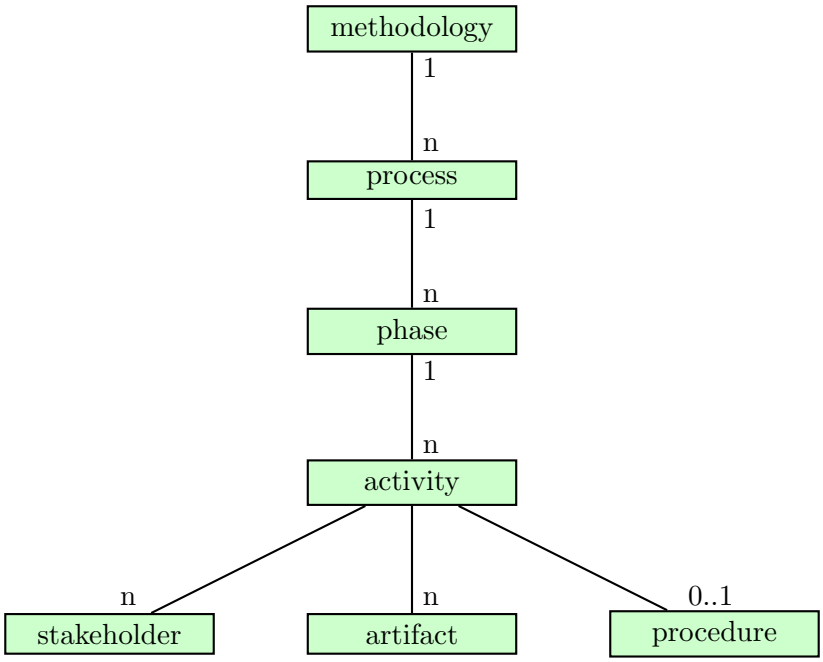

Figure 3.1: Conceptual overview of the HATS methodology.

Tables 3.2, 3.3 and 3.4 list artifacts required or produced by the methodology, while Table 3.1 does the same for stakeholders. Figure 3.2 provides a graphical representation of the HATS methodology at the level of phases, and Figures 3.3 and 3.4 detail the Family Engineering and Application Engineering Processes in terms of activities. Phases, activities, and artifacts are numbered for cross-reference.[1]

Some activities in the methodology are *optional*, as one can see in Figures 3.3 and 3.4. In addition, some activities produce artifacts that are qualified versions of their input artifacts, for example: [extended] PL Architectural Model (A11) and [verified] External Feature Model (A9). A qualification means that the

[1] In a PDF file, a click on an occurrence of a phase, activity, or artifact shows the place where it is defined.

activity extends or modifies the artifact in some way, for example a verified output artifact may include a formal proof that an input artifact has some property.

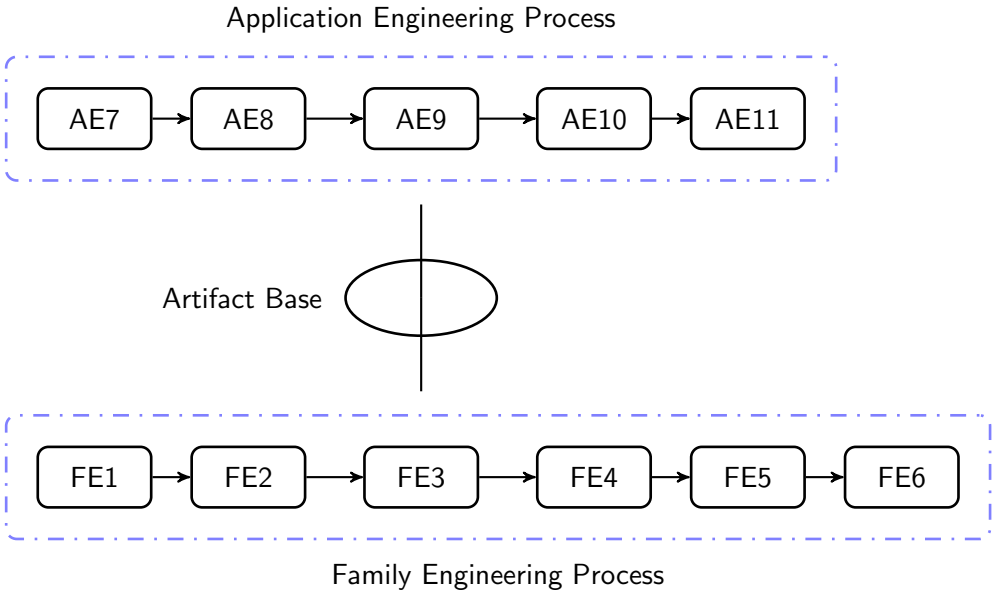

Figure 3.2: HATS methodology.

3.1 Family Engineering

In the Family Engineering process, the Product Line (PL) infrastructure is created, from which several similar applications will be instantiated afterwards according to the customers' requirements.

Phase FE1: PL Scoping

The purpose of this phase is to decide the set of products and features to be included in the PL infrastructure, so that the systematic reuse of this infrastructure brings economical benefits.

Activity FE1.1: Identify Products

Description:	Identify appropriate products for the product line and describe their potential market, high level features, and the rough release plan
Stakeholder(s):	Scoping Expert (S8), PL Manager (S5), Domain Expert (S4)
Input Artifact(s):	PL Proposal (A1)
Output Artifact(s):	Product Description (A2) for each product

Activity FE1.2: Identify Features

Description:	Refine the high level features already provided in the Product Description (A2) and identify further innovative features.
Stakeholder(s):	Scoping Expert (S8), PL Manager (S5), Domain Expert (S4)
Input Artifact(s):	PL Proposal (A1), Product Description (A2) for each product, Documentation of Existing Products (A3)

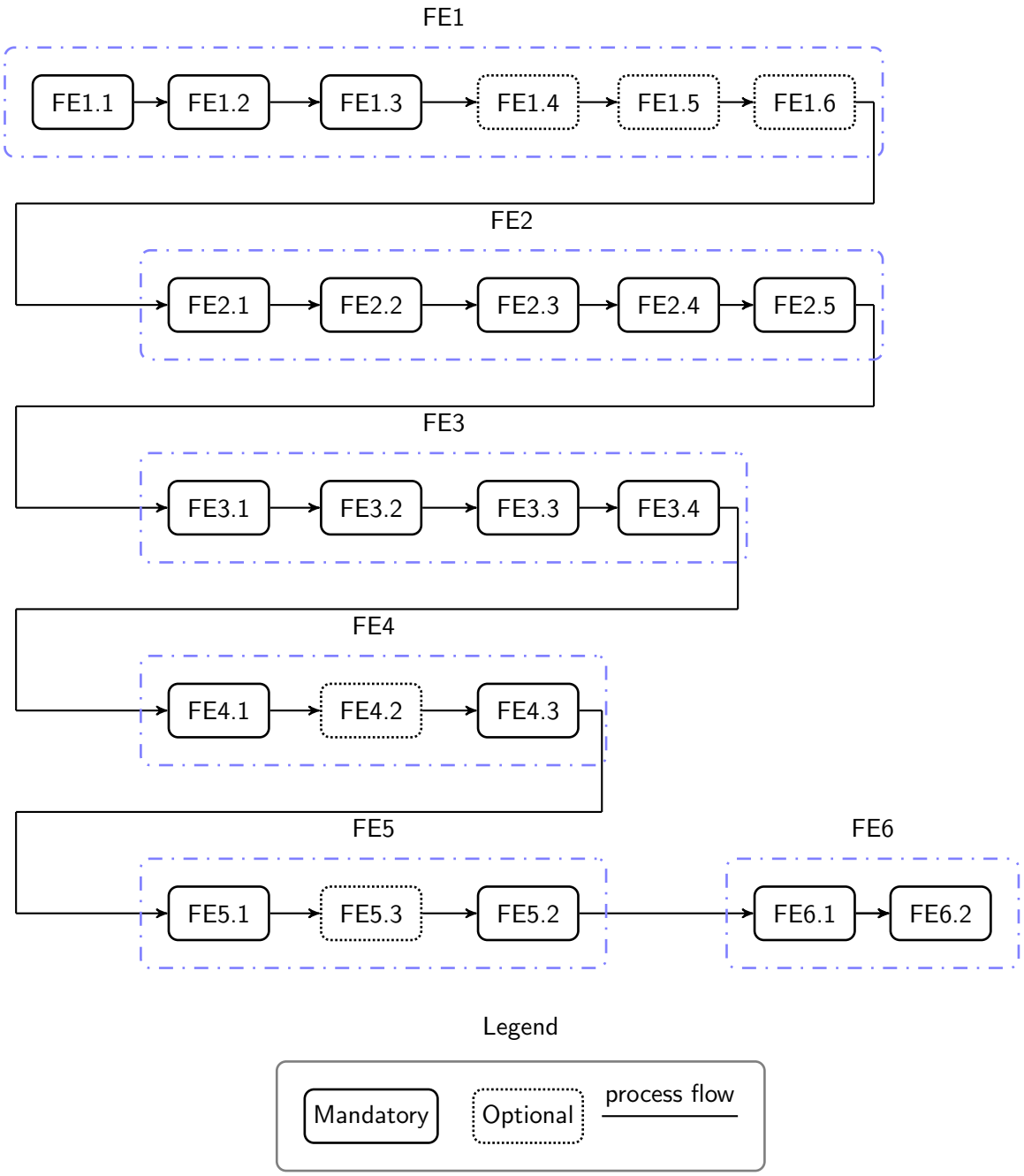

Figure 3.3: Family Engineering process.

Stakeholder	*Description*
Architect (S1)	Responsible for the high-level design of a product or family of products.
Customer (S2)	The one whose activities should be supported by a product or family of products.
Developer (S3)	Responsible for the low-level design and implementation of a product or family of products.
Domain Expert (S4)	Experts on the application domain of the product line, who can have either a technical viewpoint or a market viewpoint, and belong either to a development unit or to a customer organisation.
PL Manager (S5)	Responsible for planning, developing and maintaining an effective PL infrastructure from which the several products can be instantiated.
Product Manager (S6)	Manager of a product, whose main concern is to assure that the product meets its customers' requirements over time.
Requirements Engineer (S7)	Responsible for eliciting and specifying the requirements of a product or family of products.
Scoping Expert (S8)	Responsible for guiding the domain experts through the PL scoping activities, from the selection of products and features to the concrete plan of product releases.
Tester (S9)	Responsible for testing a product or family of products.

Table 3.1: Stakeholders of the Family Engineering and Application Engineering processes.

Output Artifact(s): Feature List (A4)

Activity FE1.3: Create Product Feature Matrix

Description: Group features that are closely related in areas of functionality or domains, and specify for each feature the products that are expected to provide this feature
Stakeholder(s): Scoping Expert (S8), PL Manager (S5), Domain Expert (S4)
Input Artifact(s): Product Description (A2) for each product, Feature List (A4)
Output Artifact(s): Product Feature Matrix (A5)

Activity FE1.4: Plan Product Releases

Description: Produce an overview of the releases scheduled for the products based on the information provided in each Product Description (A2).
Stakeholder(s): Scoping Expert (S8), PL Manager (S5), Domain Expert (S4)
Input Artifact(s): PL Proposal (A1), Product Description (A2) for each product, Product Feature Matrix (A5)
Output Artifact(s): Product Release Plan (A6)

Activity FE1.5: Assess Features

Description: Assess and prioritise individual features according to the satisfaction/dissatisfaction of the customer, implementation cost, and implementation risk. For this it is helpful to know existing artifacts that can be used as basis for the PL assets.
Stakeholder(s): Scoping Expert (S8), PL Manager (S5), Domain Expert (S4)

Artifact	*Description*
PL Scoping (FE1)	
PL Proposal (A1)	Document that proposes the creation of a PL. It includes motivation based on existing or planned products, goals, and the assignment of a PL Manager.
Product Description (A2)	Document that provides the high level features of a product, its potential market and a rough release plan.
Documentation of Existing Products (A3)	Set of all available documents, which include models and code, about the existing products.
Feature List (A4)	List of features that should be considered for inclusion in the PL. It is normally provided as part of the Product Feature Matrix (A5).
Product Feature Matrix (A5)	Matrix that shows which features should be available in which products.
Product Release Plan (A6)	Map that shows when the different products should be made available in the market. The rows are product categories and the columns are time periods.
Domain Assessment Report (A7)	Report that provides reuse recommendations for the main areas of functionality or domains captured in the Product Feature Matrix (A5).
PL Requirement Analysis (FE2)	
Requirements Elicitation Documents (A8)	Set of all documents used for eliciting requirements.
External Feature Model (A9)	Model that captures common and variable features from the customer's viewpoint.
Family Requirements Models (A10)	Models that capture both common and variable requirements, but keep track of what is common and what is variable.
Reference Architecture Design (FE3)	
PL Architectural Model (A11)	Formal description of the structural composition of, the coordination among, and the behavioural guarantee by elements that can be used to compose the products, as well as the high-level functional description of each individual element. Note that each architectural element consists of one or more instances of Generic Component Model (A16).
Architectural Feature Model (A12)	Feature model that describes the variation points in the PL Architectural Model (A11) that need to be resolved during the Application Engineering process.
Generic Component Interface (A13)	Set of interfaces to be behaviourally modelled by the Generic Component Model (A16). Interfaces include method signatures, inheritance descriptions as well as behavioural properties about method invocations.
Interface Feature Model (A14)	Feature model that describes the variation points at the Generic Component Interface (A13) that need to be resolved during the Application Engineering process.
Generic Interface Test (A15)	Test case that exercises methods of Generic Component Interface (A13) that are modelled by Generic Component Model (A16), and checks whether the invocations satisfy their behavioural properties, also defined by Generic Component Interface (A13).

Table 3.2: Artifacts required and produced by the Family Engineering process (a), grouped by phase.

Artifact	*Description*
Generic Component Design (FE4)	
Generic Component Model (A16)	Behavioural model of Generic Component Interface (A13).
Generic Component Feature Model (A17)	Feature model of a Generic Component Model (A16).
Existing Code (A18)	Existing product code (e.g., Java), not generated from ABS models.
Generic Component Realisation (FE5)	
Generic Component Code (A19)	Code generated from PL Architectural Model (A11), Architectural Feature Model (A12), and Generic Component Model (A16).
External Code (A20)	External (or legacy) with which Generic Component Code (A19) must be integrated.
Glue Code (A21)	Code to glue other kinds of code together.
Unit Test (A22)	Unit test for all code, generated or not.
Generic Component Testing and Verification (FE6)	
Generic Integration Test (A23)	Integration test for generic component code, external code, and glue code.

Table 3.3: Artifacts required and produced by the Family Engineering process (b), grouped by phase.

Input Artifact(s): Product Feature Matrix (A5), Product Release Plan (A6), Documentation of Existing Products (A3)
Output Artifact(s): [prioritised] Product Feature Matrix (A5)

Activity FE1.6: Assess Domains

Description: Assess the areas of functionality or domains captured in the Product Feature Matrix (A5) in order to give reuse recommendations concerning each area of functionality or domain. The aspects to be considered are: maturity, stability, constraints, market potential, commonalities and variabilities, coupling and cohesion, and existing assets.
Stakeholder(s): Scoping Expert (S8), PL Manager (S5), Domain Expert (S4)
Input Artifact(s): PL Proposal (A1), Product Feature Matrix (A5), Documentation of Existing Products (A3)
Output Artifact(s): Domain Assessment Report (A7)

Phase FE2: PL Requirement Analysis

The purpose of this phase is to identify and model features and requirements of the family of products within the scope of the product line, and analyse the commonalities and systematic variabilities in those.

Activity FE2.1: Elicit Family Requirements

Description: Elicit from different sources (existing artifacts, people, etc.) the requirements to be fulfilled by the set of products and identify further requested or implicit features. Products to be supported from the beginning or in the near future should have priority. The focus is on common requirements or requirements with systematic variability.

Stakeholder(s):	Requirements Engineer (S7), Domain Expert (S4)
Input Artifact(s):	Product Feature Matrix (A5), Product Description (A2) for each product
Output Artifact(s):	Requirements Elicitation Documents (A8)

Activity FE2.2: Create External Feature Model

Description:	Create a feature model based on the set of features captured in the Product Feature Matrix (A5) and their allocation to products, and extend this feature model with further variabilities identified during the activities "Elicit Family Requirements" (FE2.1) and "Create Family Requirements Models" (FE2.3).
Stakeholder(s):	Requirements Engineer (S7)
Input Artifact(s):	Product Feature Matrix (A5), Requirements Elicitation Documents (A8), Family Requirements Models (A10)
Output Artifact(s):	External Feature Model (A9)

Procedure
In this activity, the Requirements Engineer (S7) should use the HATS Feature Modelling Language (μTVL) to create the External Feature Model (A9). This model should make commonalities and systematic variabilities from the customer's viewpoint clear and include constraints (if in: Expression and if out: Expression) and dependencies between features (require Feature ID and exclude Feature ID). The Requirements Engineer (S7) should use the ABS Product Configurator to visualise the feature model.

In addition, the Requirements Engineer (S7) should formalise the Product Feature Matrix (A5) by using the HATS Product Selection Language (PSL), which means to use the language to specify the features to be included in each product. The Requirements Engineer (S7) should also enrich the mapping between products and features captured in PSL with attribute values identified during the activity "Elicit Family Requirements" (FE2.1). The Requirements Engineer (S7) should use the ABS Product Configurator to automatically find out the mapping between products and features based on functional and quality requirements and constraints.

Besides the mapping between products and features captured in PSL, the Requirements Engineer (S7) can start creating the adaptation/reconfiguration rules for that product by using PSL. The rules define to which other configuration the product can be reconfigured.

This activity and the activity "Create Family Requirements Models" (FE2.3) are expected to influence each other.

Activity FE2.3: Create Family Requirements Models

Description:	Create family requirements models from the Requirements Elicitation Documents (A8). During this activity, parts of the requirements models that vary among products are linked to the External Feature Model (A9).
Stakeholder(s):	Requirements Engineer (S7)
Input Artifact(s):	Product Feature Matrix (A5), Requirements Elicitation Documents (A8), External Feature Model (A9)
Output Artifact(s):	Family Requirements Models (A10)

Activity FE2.4: Verify Family Requirements Models

Description:	Verify the External Feature Model (A9) and Family Requirements Models (A10) against the Product Feature Matrix (A5) and the set of Product Description (A2), in order to assure that they are consistent.
Stakeholder(s):	Requirements Engineer (S7)

Input Artifact(s):	Product Feature Matrix (A5), Product Description (A2), External Feature Model (A9), Family Requirements Models (A10)
Output Artifact(s):	[verified] External Feature Model (A9) and [verified] Family Requirements Models (A10)

Activity FE2.5: Validate Family Requirements Models

Description:	Validate together with domain experts and/or customer representatives the models created during this phase.
Stakeholder(s):	Requirements Engineer (S7), Domain Expert (S4), Customer (S2)
Input Artifact(s):	Product Feature Matrix (A5), External Feature Model (A9), Family Requirements Models (A10)
Output Artifact(s):	[validated] Product Feature Matrix (A5), [validated] External Feature Model (A9), and [validated] Family Requirements Models (A10)

Procedure
The Requirements Engineer (S7) should show the External Feature Model (A9) in the HATS Feature Modelling Language (μTVL) or an equivalent graphical representation to domain experts and/or customer representatives, and ask whether there are missing features or there are features with wrong characterisation (mandatory, optional etc.).

The Requirements Engineer (S7) should also ask about missing requirements and the correctness of the requirements.

Phase FE3: Reference Architecture Design

The purpose of this phase is to define the *reference architecture*, which is a common architecture for all products in the product line. The reference architecture is documented by means of different architectural views that contain information about component interfaces, overall product behaviour, and variabilities. The reference architecture also specifies rules which the design of *generic components* must adhere to, so that component designs do not corrupt the overall product quality.

Activity FE3.1: Define Architectural Model

Description:	Define architectural elements that can be used to compose the PL products and how they relate to each other.
Stakeholder(s):	Architect (S1)
Input Artifact(s):	External Feature Model (A9), Family Requirements Models (A10)
Output Artifact(s):	PL Architectural Model (A11), Architectural Feature Model (A12)

Procedure
The Architect (S1) defines the PL Architectural Model (A11) using the ABS Component Modelling Language and integrates the external variation points from the External Feature Model (A9) defined in μTVL. Whenever required, the Architect (S1) adds additional variation points to be captured in the Architectural Feature Model (A12) defined in μTVL. The Architect (S1) ensures that the components in the architecture take into consideration all variation points and do not invalidate any of the variation constraints.

Activity FE3.2: Specify System-level Properties

Description:	Specify desired system-level properties to be ensured by the PL Architectural Model (A11).
Stakeholder(s):	Architect (S1)
Input Artifact(s):	External Feature Model (A9), Family Requirements Models (A10), PL Architectural Model (A11), Architectural Feature Model (A12)
Output Artifact(s):	[extended] PL Architectural Model (A11)

Procedure
The Architect (S1) defines a set of system-level properties[2] using the HATS Assertion Language and/or the behavioural interface specification[3]. The Architect (S1) defines behavioural interfaces as ABS interfaces accompanied by specification over communication histories using attribute grammar based or interaction pattern based techniques. The Architect (S1) annotates these properties at the level of components. Properties may include invariants[4], and lower/upper bounds on the resource consumption (such as memory, execution steps, etc.).

Activity FE3.3: Define Component Interfaces

Description:	Define interfaces to be modelled by generic component models, and distribute PL variation points across the interfaces.
Stakeholder(s):	Architect (S1)
Input Artifact(s):	External Feature Model (A9), Family Requirements Models (A10), [extended] PL Architectural Model (A11), Architectural Feature Model (A12)
Output Artifact(s):	Generic Component Interface (A13), Interface Feature Model (A14)

Procedure
The Architect (S1) defines a set of behavioural interfaces using the combination of ABS interfaces and either the attribute grammar based or the interaction pattern based specification techniques; interfaces specify abstractly the behavioural requirements and expectations of the reference architecture at each variation point and for each component. The Architect (S1) models variation points using the HATS Delta Modelling Language.

Activity FE3.4: Define Interface Properties

Description:	Define desired system-level properties to be ensured by interfaces. These properties come in three flavours: method level contracts, interface-level invariants, and protocols (communication histories).
Stakeholder(s):	Architect (S1)
Input Artifact(s):	External Feature Model (A9), Family Requirements Models (A10), Generic Component Interface (A13), Interface Feature Model (A14) [extended] PL Architectural Model (A11), Architectural Feature Model (A12)
Output Artifact(s):	[extended] Generic Component Interface (A13), [extended] Interface Feature Model (A14), Generic Interface Test (A15)

[2] A system-level property is a property that cross-cuts several architectural components and may be orthogonal to the functional behaviour of the products themselves.

[3] A behavioural interface is an interface that defines the type information such as method signatures, but also the sequence of method invocations that are allowed.

[4] An invariant is a property about the state of the product that is true at certain points, e.g., suspension points.

Procedure
When possible, the Architect (S1) infers interface-level invariants from system-level properties and as such ensures that interface-level invariants are consistent with respect to system-level properties. Invariants may include lower/upper bounds of the cost (i.e., resource consumption) of invoking a method and location types. The Architect (S1) generates unit test cases from behavioural interfaces; these test cases can be used to partially guard (without full verification) against invalid implementations of these interfaces.

Phase FE4: Generic Component Design

The purpose of this phase is to design in detail the PL components identified in the Reference Architecture Design (FE3) phase. While properties and constraints that are common to all components are specified in the Reference Architecture Design (FE3) phase, a detailed internal design of each component is provided in this phase. Each component model specifies and integrates both component-specific and cross-cutting variabilities.

Activity FE4.1: Define Component Models

Description:	Define models for the Generic Component Interface (A13) using information provided by Interface Feature Model (A14), define properties about these models.
Stakeholder(s):	Developer (S3)
Input Artifact(s):	[extended] PL Architectural Model (A11), Architectural Feature Model (A12), [extended] Generic Component Interface (A13), Interface Feature Model (A14)
Output Artifact(s):	Generic Component Model (A16), Generic Component Feature Model (A17)

Procedure
The Developer (S3) provides ABS models to behavioural interfaces defined in the Generic Component Interface (A13). ABS models define an executable behavioural model to interfaces that they implement. Using ABS, the Developer (S3) can abstract away from environments and concrete data. The Developer (S3) ensures that the behavioural model satisfies properties at the level of both interfaces and components. The Developer (S3) also uses the HATS Assertion Language to specify properties about class implementations including method level contracts, class-level invariants as well as location types and deployment configuration (deployment components, deadlines, and durations). The Developer (S3) defines a product selection in the PL using the HATS Product Selection Language. In addition, the Developer (S3) can create adaptation/reconfiguration rules for that product. The rules define to which other configuration each product can be reconfigured. The Developer (S3) provides behavioural models of the PL variabilities in terms of deltas using the HATS Delta Modelling Language following the HATS Delta Development Workflow. Deltas are part of the Generic Component Model (A16). The Developer (S3) connects PL variabilities defined in the Generic Component Feature Model (A17) with deltas using the HATS Product Line Configuration Language. The PL configuration also governs runtime adaptation. With the adaptation rules in mind, it is possible to declare deltas that are specifically designed for adapting a product to a different product at runtime.

Activity FE4.2: Mine Generic Components

Description:	Extract a (partial) model of a generic component from existing code.
Stakeholder(s):	Developer (S3)
Input Artifact(s):	Existing Code (A18)

Output Artifact(s): [partial] Generic Component Model (A16)

Procedure
If applicable, the Developer (S3) runs necessary tooling for model extraction (mining) on available Existing Code (A18) in Java to get a partial Generic Component Model (A16) in the HATS Abstract Behavioural Specification Language (ABS). One such experimental tool is JMS2ABS.

Activity FE4.3: Test and Verify Component Models

Description:	Test and verify the set of Generic Component Model (A16) against properties specified in the [extended] PL Architectural Model (A11) and [extended] Generic Component Interface (A13).
Stakeholder(s):	Developer (S3), Tester (S9)
Input Artifact(s):	[extended] PL Architectural Model (A11), Architectural Feature Model (A12), [extended] Generic Component Interface (A13), Interface Feature Model (A14), Generic Interface Test (A15), Generic Component Model (A16), Generic Component Feature Model (A17)
Output Artifact(s):	[verified] Generic Component Model (A16), [verified] Generic Component Feature Model (A17)

Procedure
The Tester (S9) checks that deltas are developed according to the HATS Delta Development Workflow, thereby ensuring that all features are implemented and there would be no ambiguity in the Generic Component Model (A16) in the ABS. The Tester (S9) tests and verifies individual artifacts in the Generic Component Model (A16) in the ABS. A suite of HATS validation tools is available for simulating, testing and verifying behaviours of classes, functions and deltas against class-level, interface-level, and component-level specifications. If there is an error in the model, the Tester (S9) uses one or more of the simulators to debug the ABS models, or uses the visual debugger to perform symbolic execution of the models to uncover bugs. The Developer (S3) then fixes the bugs.

Note that this activity shares the verification workload with the Generic Component Testing and Verification (FE6) phase by focusing on smaller units such as individual methods, classes, and components, while the Generic Component Testing and Verification (FE6) phase focuses on the testing and verification of their integrations.

PET is a test-case generation tool which aims at being a generic platform for CLP-based test-case generation of different languages. It has been recently extended, into a tool called aPET, to generate test-cases for concurrent ABS programs. This is done by compiling an ABS program into CLP equivalent programs and extending the symbolic execution engine of PET with the concurrency primitives of ABS.

Phase FE5: Generic Component Realisation

The purpose of this phase is to realise the generic components, that is to code them using a suitable programming language. This code should then be added to the PL Artifact Base for reuse.

Activity FE5.1: Generate Generic Code

Description:	Generate generic component code from models by using code generators.
Stakeholder(s):	Developer (S3)

Input Artifact(s):	PL Architectural Model (A11), Architectural Feature Model (A12), Generic Component Model (A16), Generic Component Feature Model (A17)
Output Artifact(s):	[possibly manually modified] Generic Component Code (A19)

Procedure
The Developer (S3) runs one or more code generators, providing the ABS models of the Generic Component Model (A16) and their PL Architectural Model (A11) as arguments. Each code generator outputs code in some programming language. The Developer (S3) may modify the generated code should that prove necessary. This activity may be carried out in parallel with activity "Integrate External Code" (FE5.3).

Activity FE5.2: Test Generic Code

Description:	Test the code parts individually by writing and running unit test.
Stakeholder(s):	Developer (S3)
Input Artifact(s):	Generic Component Code (A19), External Code (A20), Glue Code (A21)
Output Artifact(s):	Unit Test (A22)

Procedure
The Developer (S3) writes tests for the generated code, for the external code, and for the glue code, testing each part in isolation. The Developer (S3) does this either manually or uses the aPET test case generator. Should a test fail, the Developer (S3) debugs the test and the code to locate the source of the problems. For this a regular code debugger may be used.

Activity FE5.3: Integrate External Code

Description:	Integrate generic component code with external (or legacy) code.
Stakeholder(s):	Developer (S3)
Input Artifact(s):	Generic Component Code (A19), External Code (A20)
Output Artifact(s):	Glue Code (A21)

Phase FE6: Generic Component Testing and Verification

The purpose of this phase is to test and/or verify system-level properties of generic component code, external code, and glue code. The verification involves checking the code against contracts.

Activity FE6.1: Test Generic Integration

Description:	Write and run integration test of the generic component code, the external code and the glue code.
Stakeholder(s):	Tester (S9)
Input Artifact(s):	Generic Component Code (A19), External Code (A20), Glue Code (A21)
Output Artifact(s):	Generic Integration Test (A23)

Activity FE6.2: Verify Generic Components

Description:	Verify that the generic component code satisfies the contracts.

Stakeholder(s):	Developer (S3)
Input Artifact(s):	Generic Component Code (A19), External Code (A20), Glue Code (A21), [extended] Generic Component Interface (A13), [extended] PL Architectural Model (A11)
Output Artifact(s):	[verified] Generic Component Code (A19)

Procedure
The Developer (S3) runs a verification tool that checks the Generic Component Code (A19) against the contracts specified at the [extended] Generic Component Interface (A13) and the [extended] PL Architectural Model (A11). The SDA tool can be used for inferring the interaction behaviour, namely the method invocations and synchronisation operations, from the Generic Component Code (A19). This behaviour can then be checked to be compliant to the specified [extended] Generic Component Interface (A13). The upper/lower bound assertions can be checked by COSTABS which incorporates an option to compare two bounds. Concretely, given two bounds, the system checks if one of them is greater (or smaller) than the other. The checking is based on a subsumption test applied to the different subexpression.

3.2 Application Engineering

In the Application Engineering process, a specific application or product is engineered according to the specific customer's requirements and based on the assets of the PL infrastructure.

Phase AE7: Product Scoping

The purpose of this phase is to define the product features in order to provide a scope for the next Application Engineering phases.

Activity AE7.1: Create/Revise Product Description

Description:	Create a description of the product if the product was not considered during Family Engineering, or revise the Product Description (A2) obtained from the PL infrastructure. The description should provide potential market, high level features, and release plan.
Stakeholder(s):	Product Manager (S6), Requirements Engineer (S7), Customer (S2)
Input Artifact(s):	Product Proposal (A24), Product Description (A2)
Output Artifact(s):	Product Description (A2)

Activity AE7.2: Derive Product Scope

Description:	Select the features of the External Feature Model (A9) that define the scope of the specific product at hand and add features that are product-specific.
Stakeholder(s):	Requirements Engineer (S7)
Input Artifact(s):	External Feature Model (A9), Product Description (A2)
Output Artifact(s):	Product Scope Specification (A25)

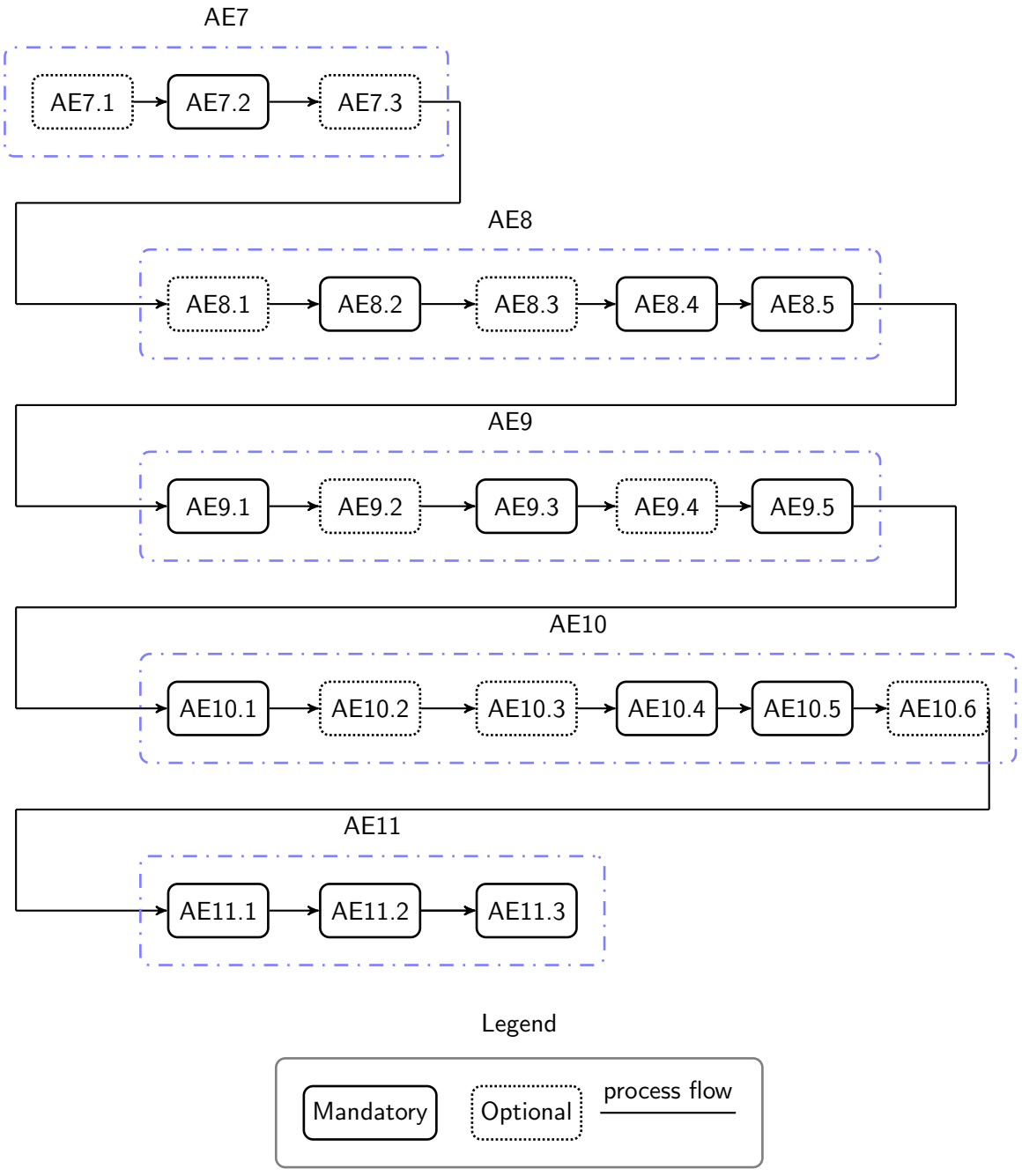

Figure 3.4: Application Engineering process.

Artifact	*Description*
Product Scoping (AE7)	
Product Proposal (A24)	Document that proposes a new product. It includes motivation based on non-fulfilled market needs, target group for the new product, and the assignment of a Product Manager.
Product Scope Specification (A25)	Document that defines the scope of a specific product to be derived from the PL infrastructure. The scope is specified in terms of features that are of interest to the customer(s).
PL Matrix Update Request (A26)	Document that communicates the scope of a product to the organisation responsible for the PL infrastructure so that they can be aware of it and react accordingly.
PL Requirements Model Instantiation (AE8)	
Product Requirements Models (A27)	Models that specify the requirements of a specific product.
Reference Architecture Instantiation (AE9)	
Product Architectural Model (A28)	Formal description of the structural composition of, the coordination among, and the behavioural guarantee by elements that compose the product. This description is obtained by resolving all variation points in the PL Architectural Model (A11) according to the Architectural Feature Model (A12).
Product Component Interface (A29)	Set of interfaces to be behaviourally modelled by Product Component Model (A31). Interfaces are obtained by resolving all variation points in Generic Component Interface (A13) according to the Interface Feature Model (A14).
Product Interface Test (A30)	Test that exercises methods of Product Component Interface (A29) modelled by Product Component Model (A31), and checks whether the invocations satisfy their behavioural properties, also defined by Product Component Interface (A29). Test is obtained by resolving all variation points in Generic Interface Test (A15) according to the Interface Feature Model (A14).
Product Construction and Integration (AE10)	
Product Component Model (A31)	Behavioural model of Product Component Interface (A29), obtained by resolving all variation points in Generic Component Model (A16) according to the Generic Component Feature Model (A17).
Product Code (A32)	Executable code generated from Product Component Model (A31).
Product Contracts (A33)	Properties against which the product code is to be checked.
Product Testing and Verification (AE11)	
Product Integration Tests (A34)	Integration test for a product as a whole
Product Test Report (A35)	Test report for product as a whole

Table 3.4: Artifacts required and produced by the Application Engineering process, grouped by phase.

Procedure
The Requirements Engineer (S7) should use the HATS Product Selection Language (PSL) to select the features from the External Feature Model (A9) that apply for the product at hand. If the product to be engineered was considered during the Family Engineering process, an initial product specification in PSL should be available. Besides the selection of features, the Requirements Engineer (S7) can start creating the adaptation/reconfiguration rules for that product by using the HATS Product Selection Language (PSL). The rules define to which other configuration each product can be reconfigured.

Activity AE7.3: Update Product Feature Matrix

Description:	Communicate the Product Scope Specification (A25) to the organisation responsible for the PL infrastructure so that they can update the Product Feature Matrix (A5), if necessary, and be aware of the engineering effort.
Stakeholder(s):	Requirements Engineer (S7)
Input Artifact(s):	Product Scope Specification (A25)
Output Artifact(s):	PL Matrix Update Request (A26)

Phase AE8: PL Requirements Model Instantiation

The purpose of this phase is to obtain a requirements model for the specific product being engineered and assure that no constraints specified in the External Feature Model (A9) and Family Requirements Models (A10) are violated.

Activity AE8.1: Elicit Product Requirements

Description:	Elicit from different sources (existing artifacts, people, etc.) the requirements to be fulfilled by the specific product. The collected information should be enough to resolve any external variability.
Stakeholder(s):	Requirements Engineer (S7), Domain Expert (S4), Customer (S2)
Input Artifact(s):	Product Scope Specification (A25), External Feature Model (A9), Family Requirements Models (A10)
Output Artifact(s):	Requirements Elicitation Documents (A8)

Activity AE8.2: Instantiate Family Requirements Models

Description:	Resolve all variation points in the Family Requirements Models (A10), and any remaining variability in the Product Scope Specification (A25).
Stakeholder(s):	Requirements Engineer (S7)
Input Artifact(s):	Product Scope Specification (A25), External Feature Model (A9), Family Requirements Models (A10), Requirements Elicitation Documents (A8)
Output Artifact(s):	Product Requirements Models (A27), [extended] Product Scope Specification (A25)

Procedure
The Requirements Engineer (S7) should use the Product Scope Specification (A25) to instantiate the Family Requirements Models (A10). The Requirements Engineer (S7) may have to select further features in the External Feature Model (A9) in order to complete/instantiate the Family Requirements Models (A10).

These further selection of features should be captured in the Product Scope Specification (A25) by using the HATS Product Selection Language (PSL). In any case, s/he should provide values for the attributes associated to the selected features.

Activity AE8.3: Extend Product Requirements Models

Description:	According to the information collected in the activity "Elicit Product Requirements" (AE8.1), add requirements that are product-specific, which means they cannot be mapped to variation points in the Family Requirements Models (A10).
Stakeholder(s):	Requirements Engineer (S7)
Input Artifact(s):	Product Requirements Models (A27), Family Requirements Models (A10), Requirements Elicitation Documents (A8)
Output Artifact(s):	[extended] Product Requirements Models (A27)

Activity AE8.4: Verify Product Requirements Model

Description:	Verify the Product Requirements Models (A27) against the External Feature Model (A9) and Family Requirements Models (A10), in order to assure that no constraints are violated.
Stakeholder(s):	Requirements Engineer (S7)
Input Artifact(s):	External Feature Model (A9), Family Requirements Models (A10), Product Requirements Models (A27)
Output Artifact(s):	[verified] Product Requirements Models (A27)

Activity AE8.5: Validate Product Requirements Model

Description:	Validate with domain experts and/or customer representatives that the specified product is exactly what the customers need to support their activities.
Stakeholder(s):	Requirements Engineer (S7), Domain Expert (S4), Customer (S2)
Input Artifact(s):	External Feature Model (A9), Product Scope Specification (A25), Product Requirements Models (A27)
Output Artifact(s):	[validated] Product Requirements Models (A27)

Procedure
The Requirements Engineer (S7) should show the External Feature Model (A9) in the HATS Feature Modelling Language (μTVL) and the Product Scope Specification (A25) in the HATS Product Selection Language (PSL), or equivalent graphical representations, to domain experts and/or customer representatives, and ask whether there are missing features in the product.

The Requirements Engineer (S7) should also ask about missing requirements, the correctness of the requirements, and correctness of the adaptation/reconfiguration rules if any exists.

Phase AE9: Reference Architecture Instantiation

The purpose of this phase is to instantiate the reference architecture taking into consideration the product requirements. It is often necessary to adapt the reference architecture to customer-specific product requirements, which gives rise to a specific product architecture.

Activity AE9.1: Instantiate Architectural Model

Description:	Resolve all variation points in the PL Architectural Model (A11).
Stakeholder(s):	Architect (S1)
Input Artifact(s):	Product Scope Specification (A25), External Feature Model (A9), Product Requirements Models (A27), Architectural Feature Model (A12), [extended]PL Architectural Model (A11)
Output Artifact(s):	Product Architectural Model (A28), [extended] Product Scope Specification (A25)

Procedure
The Architect (S1) should use the Product Scope Specification (A25) and the Product Requirements Models (A27) to resolve the variation points in the PL Architectural Model (A11). These variation points, which are documented in the Architectural Feature Model (A12), capture variability in the set of ABS components or in the specification of the ABS components.

The Architect (S1) should select features from the Architectural Feature Model (A12) according to the product requirements in order to instantiate the PL Architectural Model (A11). This selection of features should be captured in the Product Scope Specification (A25) by using the HATS Product Selection Language (PSL).

The Architect (S1) ensures that the selected features and the correspondent resolution of variability cause no conflict with respect to both the Product Requirements Models (A27) and the Architectural Feature Model (A12).

Activity AE9.2: Extend Architectural Model

Description:	Extend architectural models according to customer specific requirements
Stakeholder(s):	Architect (S1)
Input Artifact(s):	Product Architectural Model (A28), [extended] Product Requirements Models (A27)
Output Artifact(s):	[extended] Product Architectural Model (A28)

Procedure
The Architect (S1) extends the Product Architectural Model (A28) with new components, connections or interfaces, taking into account the [extended] Product Requirements Models (A27). The Architect (S1) extends system-level properties at the architectural level to ensure that the extended model do not invalidate the desired properties.

Activity AE9.3: Instantiate Interfaces

Description:	Resolve all variation points in the Generic Component Interface (A13).
Stakeholder(s):	Architect (S1)
Input Artifact(s):	[extended]PL Architectural Model (A11), Architectural Feature Model (A12), [extended] Product Architectural Model (A28), Generic Component Interface (A13), Interface Feature Model (A14), Generic Interface Test (A15)
Output Artifact(s):	Product Component Interface (A29), Product Interface Test (A30)

Procedure
The Architect (S1) should select features from the Interface Feature Model (A14) according to the product requirements and architecture in order to instantiate the Generic Component Interface (A13). This selection of features should be captured in the Product Scope Specification (A25) by using the HATS Product Selection Language (PSL).

Besides, the Architect (S1) can create adaptation/reconfiguration rules for that product by using the HATS Product Selection Language (PSL). The rules define to which other configurations the product can be reconfigured. The Architect (S1) ensures that the selected features and the correspondent resolution of variability cause no conflict with respect to both the PL Architectural Model (A11) and the Interface Feature Model (A14). The Architect (S1) generates from the Generic Interface Test (A15) and the Interface Feature Model (A14) concrete unit test cases for the instantiated interface specifications.

Activity AE9.4: Extend Interfaces

Description:	Extend interfaces according to customer specific requirements
Stakeholder(s):	Architect (S1)
Input Artifact(s):	[extended] Product Requirements Models (A27), [extended] Product Architectural Model (A28), Product Component Interface (A29), Product Interface Test (A30)
Output Artifact(s):	[extended] Product Component Interface (A29), [extended] Product Interface Test (A30)

Procedure
The Architect (S1) extends the Product Component Interface (A29), taking into account the [extended] Product Requirements Models (A27) and the [extended] Product Architectural Model (A28). The Architect (S1) extends system-level properties at the level of the behavioural interfaces to ensure the extended behavioural interfaces do not invalidate the desired properties. The Architect (S1) specifies these properties using the HATS Assertion Language.

Activity AE9.5: Verify Product Architecture

Description:	Verify the Product Architectural Model (A28) and the Product Component Interface (A29) against system-level properties.
Stakeholder(s):	Tester (S9)
Input Artifact(s):	[extended] Product Architectural Model (A28), [extended] Product Component Interface (A29), Product Interface Test (A30)
Output Artifact(s):	[verified] Product Architectural Model (A28), [verified] Product Component Interface (A29)

Procedure
The Tester (S9) verifies that the Product Architectural Model (A28) and the Product Component Interface (A29) are consistent with respect to the system-level properties by using HATS tools and verification techniques. The Tester (S9) generates proof obligations for implementations of behavioural interfaces.

Phase AE10: Product Construction and Integration

The purpose of this phase is to instantiate and reuse generic components according to the product's architecture. After identifying the necessary reusable components, they are either adapted to fit the product requirements, or new product specific components are developed instead.

Activity AE10.1: Instantiate Component Models

Description:	Resolve all variation points in the PL Architectural Model (A11) and ensure that resolution causes no conflict with respect to the Product Requirements Models (A27).

Stakeholder(s):	Developer (S3)
Input Artifact(s):	[verified] Product Requirements Models (A27), [verified] Product Architectural Model (A28), [verified] Product Component Interface (A29), [verified] Generic Component Model (A16), [verified] Generic Component Feature Model (A17), Product Interface Test (A30)
Output Artifact(s):	Product Component Model (A31)

Procedure
The Developer (S3) resolves variation on ABS models by making product selections defined using the HATS Product Selection Language. The Developer (S3) ensures that the resolution causes no conflict with respect to the Generic Component Feature Model (A17), and that the resolved ABS models are well-typed.

Activity AE10.2: Mine Product Components

Description:	Extract a (partial) model of a product from existing product code.
Stakeholder(s):	Developer (S3)
Input Artifact(s):	Existing Code (A18)
Output Artifact(s):	[partial] Product Component Model (A31)

Procedure
If applicable, the Developer (S3) runs the tool for model extraction (mining) on existing product code (in Java) to get a partial product model (in ABS). This activity is similar to FE4.2, but the goal here is to obtain a model of a specific product component, not a model of a generic component. Should the Developer (S3) wish to change a running system, without stopping it, she may write new deltas to describe the desired changes, including changes to classes, objects, and synchronisation constraints. The deltas ensure that the changes are deployed at the right time. The extensions are written using MetaABS and deployed to the running system via a special interface in the ABS run-time. Changes will then take place gradually by applying the MetaABS code to the run-time system. The changes that are deployed could be new delta modules or components.

Activity AE10.3: Extend Component Models

Description:	Adapt the Product Component Model (A31) according to custom product requirements.
Stakeholder(s):	Developer (S3)
Input Artifact(s):	[verified] Product Requirements Models (A27), [verified] Product Architectural Model (A28), [verified] Product Component Interface (A29), [verified] Generic Component Model (A16), [verified] Generic Component Feature Model (A17), Product Interface Test (A30) Product Component Model (A31)
Output Artifact(s):	[extended] Product Component Model (A31)

Procedure
The Developer (S3) updates the ABS model of the Product Component Model (A31) according to customer-specific product requirements. This might involve implementing new delta modules to be applied to the ABS model.

Activity AE10.4: Test and Verify Product Model

Description:	Test and verify the Product Component Model (A31) against desired properties at the Product Architectural Model (A28) and the Product Component Model (A31).
Stakeholder(s):	Tester (S9)

Input Artifact(s):	[verified] Product Architectural Model (A28), [verified] Product Component Interface (A29), Product Component Model (A31)
Output Artifact(s):	[verified] Product Component Model (A31)

Procedure
The Tester (S9) tests and verifies the ABS model of the Product Component Model (A31) against properties defined at the Product Architectural Model (A28) and the Product Component Model (A31). This might involve the combination of model checking and interactive/automated theorem proving; tool support for which are provided by the HATS tool suite.

Activity AE10.5: Generate Product Code

Description:	Generate executable Product Code (A32) from the Product Component Model (A31). Generate Product Contracts (A33) that would contain properties specified in the Product Component Model (A31).
Stakeholder(s):	Developer (S3)
Input Artifact(s):	[verified] Product Architectural Model (A28), [verified] Product Component Interface (A29), [verified] Generic Component Model (A16), [verified] Generic Component Feature Model (A17), [verified] Product Component Model (A31)
Output Artifact(s):	Product Code (A32), Product Contracts (A33)

Procedure
The Developer (S3) generates executable Product Code (A32) using HATS ABS back-ends (e.g., Java, Scala) from ABS models of the Product Component Model (A31), and translates properties from the model level into the Product Contracts (A33) such that they can be reused for verifying and testing generated code.

Activity AE10.6: Integrate Product Code

Description:	Integrate the Product Code (A32) with external (and/or legacy) code.
Stakeholder(s):	Developer (S3)
Input Artifact(s):	[verified] Product Architectural Model (A28), [verified] Product Component Interface (A29), Product Code (A32), Product Contracts (A33)
Output Artifact(s):	[integrated] Product Code (A32)

Procedure
The Developer (S3) integrates/connects the Product Code (A32) with external code using the HATS ABS Foreign Function Interface (ABS-FFI).

Phase AE11: Product Testing and Verification

The purpose of this phase is to test and verify the constructed product against product line requirements and any product-specific requirements.

Activity AE11.1: Test Product

Description:	Test the whole product using handwritten or automatically generated tests.
Stakeholder(s):	Tester (S9)
Input Artifact(s):	Product Code (A32), Product Requirements Models (A27), Product Contracts (A33)
Output Artifact(s):	Product Integration Tests (A34)

Procedure
The aPET tool is used to automatically generate test cases which ensure a high degree of code average (e.g, number of iterations each loop is executed), as well as concurrency scenarios coverage (e.g., the number of switching between different tasks).

The LBTest tool automatically generates and executes black box requirements tests on a system under test (SUT). It also constructs a verdict on the outcome of each test case, and reports these to the Tester (S9). The use of LBTest for black box requirements testing can be divided into four main phases: Phase 1: Data Type Modelling. ABS data types are abstracted to finite data types by data type partitioning. Phase 2: Requirements Modelling. A set of $PLTL(\Sigma)$ requirements are derived using Phase 1 output and the Product Requirements Models (A27) and Product Contracts (A33). Phase 3: Automated Testing. LBTest automatically generates test cases based on the requirements model, executes these on the SUT and constructs a verdict for each test case. Phase 4: Test Reporting. The tool exports test results for integration into the Product Test Report (A35).

Activity AE11.2: Verify Product

Description:	Verify the correctness of the product against a given contract.
Stakeholder(s):	Developer (S3)
Input Artifact(s):	Product Code (A32), Product Contracts (A33)
Output Artifact(s):	[verified] Product Code (A32)

Procedure
The Developer (S3) and Tester (S9) run a verification tool that checks the Product Code (A32) against the Product Contracts (A33). For the case of upper/lower bounds on the resource consumption, COSTABS is able to infer a correct upper/lower bound for the Product Code (A32) and compare then against the Product Contracts (A33). The SDA tool can be used to infer the interaction behaviour of a Product Code (A32), which can then be checked to be compliant to the specified Product Contracts (A33). Moreover the code can be checked for liveness properties.

Activity AE11.3: Validate Product

Description:	Validate with domain experts and/or customer representatives that the product code does exactly what the customers need to support their activities.
Stakeholder(s):	Developer (S3), Domain Expert (S4), Customer (S2)
Input Artifact(s):	Product Code (A32)
Output Artifact(s):	[validated] Product Code (A32)

3.3 The HATS Methodology and ABS Tool Support

Here we give an overview of the HATS methodology and its tool support, and we list the parts of the methodology that have been tried in HATS case studies.

Table 3.5 shows all activities in the methodology that have procedure descriptions. For each activity, the names of the tools used in the activity are listed.[5] In some cases, a general class of alternative tools are listed, e.g., "Tools for verification". Here we list the members of these classes:

- Tools for verification: KeY-ABS, Compositional Verification of Programs and Procedures (CVPP), SDA.

[5]The tool SAGA is implied when a procedure mentions the Attribute Grammar Specification Language.

- Tools for test case generation: aPET, LBTest.
- Tools for code generation: Scala, Maude, and Java backends for general code, and IM-PROSA for protocol code.
- Tools for validation: simulation tools, tools for verification, and tools for test case generation.

For most activities the ABS Language and Core Tools apply, but we do not list them unless there are no other applicable tools. These tools are: ABS Frontend, PEKeY, and the Eclipse plug-in.[6]

The case studies carried out in the HATS project have involved carrying out work similar to the methodology phases and activities. Here we list, for each deliverable, and for each tool, the activities that the were carried out:

- Case studies in D5.2 [7] and D5.3 [10]:

Tool name	*Activities*
ABS language, Location Type System	FE2.2–FE2.5, FE4.1, FE5.1, AE10.5, AE10.6
Feature Modelling Framework[7]	FE2.2–FE2.5
Delta Modelling Language, Delta Modelling Development Workflow	FE4.1, FE5.1
ABS Product Configurator	FE5.1, AE10.5, AE10.6

- Case studies in D5.4 [12]:

Tool name	*Activities*
COSTABS	AE10.1, AE10.3–AE10.5, AE11.2
aPET	AE10.1, AE10.3–AE10.5, AE11.1
ABS Component Model	FE3.3, FE3.4, FE4.1, FE4.3, FE5.1–FE5.3, AE10.4–AE10.5, AE11.1
SAGA	AE9.3–AE9.4, AE10.1, AE10.3–AE10.6, AE11.1
KeY-ABS	AE10.4, AE11.2
RT-ABS	FE3.3, FE3.4, FE4.1, FE4.3, FE5.1–FE5.3, AE10.4–AE10.5, AE11.1
SDA	AE10.1, AE10.3–AE10.5, AE11.2
LBTest	AE10.1, AE10.3–AE10.6, AE11.1

Appendix B contains a table that relates the methodology to the exploitable items from deliverable D6.2b [11].

[6] The tool RascalABS is for working with the ABS language itself and therefore is not listed in the methodology.

[7] Feature Modelling Framework means the Feature Modelling Language, the Delta Modelling Language and the Product Line Configuration Language.

Act.	*Tool name/class or formalism*
FE2.2	Feature Modelling Language Product Selection Language ABS Product Configurator
FE2.5	Feature Modelling Language
FE3.1	ABS Component Modelling Language Feature Modelling Language
FE3.2	Assertion Language SAGA Interaction Pattern Specification Language
FE3.3	SAGA Interaction Pattern Specification Language Delta Modelling Language
FE3.4	Location Type System SAGA *Tools for test case generation*
FE4.1	Assertion Language Location Type System Product Selection Language Delta Modelling Language Delta Modelling Development Workflow Product Line Configuration Language SAGA
FE4.2	JMS2ABS
FE4.3	Delta Modelling Development Workflow aPET SAGA *Tools for validation*
FE5.1	*Tools for code generation*
FE5.2	aPET
FE6.2	*Tools for verification* COSTABS SDA
AE7.2	Product Selection Language
AE8.2	Product Selection Language
AE8.5	Feature Modelling Language
AE9.1	Product Selection Language
AE9.2	*ABS Language and Core Tools*
AE9.3	Product Selection Language *Tools for test case generation*
AE9.4	Assertion Language
AE9.5	*Tools for verification*
AE10.1	Product Selection Language
AE10.2	JMS2ABS MetaABS Delta Modelling Framework
AE10.3	Delta Modelling Framework
AE10.4	*Tools for testing and verification*
AE10.5	*Tools for code generation*
AE10.6	ABS Foreign Function Interface
AE11.1	aPET LBTest SAGA
AE11.2	*Tools for verification* COSTABS SDA

Table 3.5: Methodology activities and their tool support.

Chapter 4

ABS Tutorial

This brief chapter lists tutorial and reference material for the ABS language. The following papers give a comprehensive account of the ABS.

- *ABS: A Core Language for Abstract Behavioral Specification* [17] introduces and motivates the ABS language and considers its type system and formal semantics.
- *Abstract delta modeling* [5] presents a mathematical account of the delta modelling part of the ABS and how to resolve conflict between deltas.
- *Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language* [4] provides a tutorial perspective of how to use ABS to model an industrial-scale web merchandising system.
- *HATS Abstract Behavioral Specification: The Architectural View* [15] overviews the architectural aspects of the ABS, including how to specify deployment architectures and the ABS component model.

Slides for the following tutorials are available at the HATS web site[1]:

- *Abstract Behavioral Specification of Distributed Object-Oriented Systems* (2012).
- *Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language* (2011).
- *Modelling Software Product Lines with the HATS Abstract Behavioural Modelling Language* (2011).

In addition, the *HATS International School on Formal Models for Objects and Components* [3, 13] features tutorial lectures on software engineering and formal methods. The majority of the lectures present material from HATS [1, 14, 16, 18, 19]. The slides from the lectures are at the same site[2].

Furthermore, the HATS web site also includes a tools section that explains how to get started using the ABS Tool Suite[3]. The ABS language specification may be obtained at the same site.[4]

[1] `http://www.hats-project.eu/node/162`
[2] `http://www.hats-project.eu/node/214`
[3] `http://tools.hats-project.eu/`
[4] `http://tools.hats-project.eu/download/absrefmanual.pdf`

Chapter 5

Summary

This deliverable has two main contributions: First, it presents an integrated tool architecture, that is, an architecture for the set of tools developed in the HATS project. The architecture is split into three dimensions: the context dimension, or how the tool interacts with the environment; the runtime dimension, or what are the runtime entities of the tool; and development-time dimension, or how is the tool implemented. Each dimension is split further into viewpoints: data, functions, allocation, processes, and technologies. For each individual tool these dimensions are documented, and in addition there is a migration plan to achieve more tool integration.

The second main contribution of the deliverable is the HATS methodology, a process that describes how to build product line software using these tools. The methodology is a process description split into two processes: the family engineering process and the application engineering process. These two processes are again divided into phases and each phase is further divided into activities. Associated with an activity are stakeholders, artifacts involved in the activity, and a procedure for carrying out the activity using HATS tools if relevant. The HATS methodology presented here is an update of a previous version from deliverable D1.1b, and it goes beyond what was promised in the HATS Description of Work. Its intention is to demonstrate that the HATS approach can be worked out into a fully fledged software product line process with rich tool support.

Bibliography

[1] Elvira Albert. Automatic inference of bounds on resource consumption. In Marcello Bonsangue, Frank de Boer, Elena Giachino, and Reiner Hähnle, editors, *International School on Formal Models for Components and Objects: Post Proceedings*, Lecture Notes in Computer Science. Springer-Verlag, 2013.

[2] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice - Third Edition*. Addison Wesley, 2012.

[3] Marcello Bonsangue, Frank de Boer, Elena Giachino, and Reiner Hähnle, editors. *International School on Formal Models for Components and Objects: Post Proceedings*, Lecture Notes in Computer Science. Springer-Verlag, 2013.

[4] Dave Clarke, Nikolay Diakov, Reiner Hähnle, Einar Broch Johnsen, Ina Schaefer, Jan Schäfer, Rudi Schlatte, and Peter Y. H. Wong. Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language. In M. Bernardo and V. Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*, pages 417–457. Springer-Verlag, 2011.

[5] Dave Clarke, Michiel Helvensteijn, and Ina Schaefer. Abstract delta modeling. Accepted to Special Issue of MSCS, To appear.

[6] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord, and Judith Stafford. *Documenting Software Architectures Views and Beyond - Second Edition*. Addison Wesley, 2010.

[7] Evaluation of Core Framework, August 2010. Deliverable 5.2 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[8] Report on the Core ABS Language and Methodology: Part B, March 2010. Part of Deliverable 1.1 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[9] Full ABS Modeling Framework, March 2011. Deliverable 1.2 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[10] Evaluation of Modeling, March 2012. Deliverable 5.3 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[11] Exploitation Strategy, September 2012. Deliverable 6.2b of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[12] Evaluation of Tools and Techniques, March 2013. Deliverable 5.4 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[13] Evolvability Final Report, March 2013. Deliverable 3.6 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[14] Reiner Hähnle. The Abstract Behavioral Specification language: A tutorial introduction. In Marcello Bonsangue, Frank de Boer, Elena Giachino, and Reiner Hähnle, editors, *International School on Formal Models for Components and Objects: Post Proceedings*, Lecture Notes in Computer Science. Springer-Verlag, 2013.

[15] Reiner Hähnle, Michiel Helvensteijn, Einar Broch Johnsen, Michael Lienhardt, Davide Sangiorgi, Ina Schaefer, and Peter Y. H. Wong. HATS abstract behavioral specification: the architectural view. In Bernhard Beckert, Ferruccio Damiani, Frank de Boer, and Marcello M. Bonsangue, editors, *Proc. 10th International Symposium on Formal Methods for Components and Objects (FMCO 2011), Torino, Italy*, volume 7542 of *Lecture Notes in Computer Science*, pages 109–132. Springer-Verlag, 2013.

[16] Einar Broch Johnsen. Integrating deployment decisions in application-level models. In Marcello Bonsangue, Frank de Boer, Elena Giachino, and Reiner Hähnle, editors, *International School on Formal Models for Components and Objects: Post Proceedings*, Lecture Notes in Computer Science. Springer-Verlag, 2013.

[17] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *LNCS*, pages 142–164. Springer-Verlag, 2011.

[18] Arnd Poetzsch-Heffter. Verification of open concurrent object systems. In Marcello Bonsangue, Frank de Boer, Elena Giachino, and Reiner Hähnle, editors, *International School on Formal Models for Components and Objects: Post Proceedings*, Lecture Notes in Computer Science. Springer-Verlag, 2013.

[19] Marco van Dooren, Dave Clarke, and Bart Jacobs. Subobject-oriented programming. In Marcello Bonsangue, Frank de Boer, Elena Giachino, and Reiner Hähnle, editors, *International School on Formal Models for Components and Objects: Post Proceedings*, Lecture Notes in Computer Science. Springer-Verlag, 2013.

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.

Activity In the HATS methodology, an activity describes how stakeholders use artifacts to perform a PLE task, possibly applying hats exploitable items. See also *Procedure*.

AE See *Application engineering*.

Application condition A condition in propositional logic indicating to which feature configurations a delta is applicable.

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.

Continuous improvement process Continuous improvement process takes care of keeping the overall product line consistent over time. The current status of the product line is monitored, issues are detected, and improvement measures are initiated and controlled.

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

Delta model Generally, a means for expressing the semantics of features within product lines. In the new delta modeling approach, a delta model is defined more specifically as a partially ordered set of deltas. See also *Delta*.

Evolution process Evolution of the product line is coordinated by the so-called Evolution Process. Evolution can be triggered by application engineering projects that explicitly require the adaptation of product line artifacts or new product line artifacts based on customer specific requirements.

Exploitable item An exploitable item is a tool or another project result that can be exploited after the end of HATS.

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

FE See *Family engineering*.

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

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

Phase In the HATS methodology, a phase is a set of related activities within one of the processes 'Family engineering' or 'Application engineering'.

PLE See *Product line engineering*.

Procedure In the HATS methodology, a procedure describes how to carry out an activity, in particular, how to apply exploitable items.

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

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

Reference architecture The reference architecture is the common architecture defined for all product line members.

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

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

SWPF See *Software product family*.

Stakeholder A person or organisation involved with the software, either using it, making it, or ordering it.

Variability management process The variability management process controls and coordinates all activities related to modeling and realizing variability in the product line

Appendix A

Questionnaire

A.1 Goals and Typical Use Cases of the Tool

What is the main goal the users are trying to achieve with the tool? What are some of the typical use cases where the tool is employed? At least three to five use cases should be mentioned.
[Motivation - Classification of the tool, understanding the context]

A.2 History of the Tool

To understand the structure of the tool it is helpful to know its history. Especially scientific prototypes are originating from various projects and are merged into new ones. A brief overview is sufficient, nevertheless any reference is helpful for documentation of the history.
[Motivation - Responsibilities for the parts, possibility of migrating specific parts]

A.3 Involved Parties

During the realization of the tool, what parties (external sites or HATS partners) were involved? Which one is the main responsible? How was the work divided? References to the history of the tool are helpful.
[Motivation - HATS internal responsibilities for the parts]

A.4 Related Tools

Which tools in the HATS project are related? How are they connected currently?
[Motivation - Targets for migration]

A.5 Tool Architecture

This section documents the architecture of the tool according to the principles mentioned in the file "Guidelines for Collection of Architectural Information". It is structured in a way that each view can be described in its own section. The mandatory ones are: A.5.1, A.5.3 and A.5.5. If additional diagrams can be provided to illustrate the description, please include them.
[Motivation - Planning of possible migration steps]

A.5.1 *View* Data and Functions at Context

mandatory
What data is exchanged with other tools? Which format does this data have? Which functions are used by connected systems or tools? Which functionality is provided to other tools or systems?
[Motivation - Compatibility of data, dependencies on others]

A.5.2 *View* Processes at Context

optional
How is the tool integrated in the processes of its environment, especially with respect to the HATS methodology?
[Motivation - Seamless tool support for the overall HATS methodology]

A.5.3 *View* Data and Functions at Runtime

mandatory
What data is used in the tool? When is it exchanged for what purpose? Which functions are provided by the elements of the tool? How is the data structured?
[Motivation - Reuse and unification of internally used data and functionality]

A.5.4 *View* Processes at Runtime

optional
How is the workflow that should be supported by the tool realized by runtime entities? Such entities can be visible to the user or internal ones. This workflow might be the overall HATS methodology or a more fine grained one concerning only the work with the tool.
[Motivation - Organization of detailed workflow support with the tool, avoidance of redundant steps in an integrated environment]

A.5.5 *View* Technologies at Runtime

mandatory
Which technologies are used while the tool is running? How are the technologies used to provide the intended functionality? If there are references to these technologies available, please provide them.
[Motivation - Technical constraints for integration, unification of dependencies on technologies]

A.5.6 *View* Technologies at Development Time

optional
What technologies are used at development time for realizing the tools artifacts? Which programming languages are used? The usage of a specific IDE is only relevant if it has influences on the implementation itself. Are there technologies used to generate code? If there are references to these technologies available, please provide them.
[Motivation - Technical constraints for migration of code and maintenance of integrated platform]

Appendix B

Methodology related to exploitable items

Here we briefly relate the methodology to the exploitable items of deliverable D6.2b [11]. Table B.1, based on a similar table in D6.2b, contains one row for each exploitable item (from D6.2b) with number, name, HATS partner responsible for the item (owner), which HATS deliverable that made the item, and what phases and/or activities of the methodology the item is relevant for.

No.	*Exploitable item name*	*Owner(s)*	*Deliverables*	*Phases/Activities*
4.2	ABS Language and Core Tools	All	D1.1a, D1.1b, D1.2	All
4.3	Location Type System	UKL	D1.2	FE3.4, FE3.2, FE4.1, AE9.3, AE10.1
4.4	Deductive Compilation	TUD	D1.3, D1.4, D2.5	FE5.1, AE10.5
4.5	Provably Correct Compiler	IoC	D1.4	FE5.1, AE10.5
4.6	Feature Modelling Framework	KUL	D1.2	FE2.2, FE2.3, FE3.1, FE4.1, AE8.2, AE8.4, AE8.5, AE9.1, AE9.3, AE9.4
4.7	Delta Modelling Framework	KUL, TUD and others	D2.2	FE2.3, FE3.1, FE4.1, AE8.2, AE8.4, AE8.5, AE9.1, AE9.3, AE9.4
4.8	Delta Modelling Development Workflow	KUL, TUD and others	D2.2	FE2.3, FE3.1, FE4.1, AE8.2, AE8.4, AE8.5, AE9.1, AE9.3, AE9.4
4.9	Models and constructs for components	BOL, UKL	D2.1, D3.1.a, D3.1.b	FE3.3, FE4, FE5, FE6
4.10	Behavioural specification framework	UKL, CWI	D1.2, D2.5, D2.6	FE3.3, FE4, FE5, FE6, FE3.2
4.11	HATS Methodology	All	D1.1b, D1.5	All
4.12	Integrated Development Environment	UKL, UIO	D1.1, D1.2, D1.3, D1.4, D1.5	All
4.13	Formal verification tool	TUD, UIO	D1.3, D2.5, D4.3	FE4.3, FE6.2, AE8.4, AE9.5, AE10.4, AE11.2
4.14	Interactive simulator	UKL	D1.1-D1.5, D2.3	FE4.3, FE5.1, AE10.4, AE11.1
4.15	Partial evaluation-based test case generator	UPM	D2.3	FE4.3, FE5.2, FE6.1, AE9.3, AE10.4, AE11.1
4.16	Monitor inlining tool	KTH	D3.4	AE10.5, FE3.2

4.17	Visual debugging tool	TUD	D1.3, D2.3	FE4.3, FE5.3, FE5.2, AE11.1
4.18	LBTest	KTH	D2.3	AE11.1
4.19	SPL verification tool	KTH	D2.5, D4.3	AE11.2
4.20	ABS Product Configurator	FRG, UPM, NR	D4.4	FE2.2
4.21	Static analysis techniques for deadlock freedom	BOL	D2.4, D2.5	FE3.2, FE6.2, AE11.2
4.22	Cost and termination analyzer	UPM	D4.2	FE3.4,FE3.2, FE4.3, FE6.2, AE11.2
4.23	Automated derivation and verification of resource bounds	UPM	D1.3, D4.2	FE3.2, FE6.2, AE11.2
4.24	Timed Resource Consumption Analysis and Simulation	UIO	D2.1	FE3.2, FE6.2, AE11.2, FE5.1, AE11.2
4.25	MetaABS	KUL	D3.3	AE10.2
4.26	Model mining algorithm and tools	KTH, NR, UPM	D3.2	FE4.2, AE10.2
4.27	Product adaptation framework	FRG	D3.5	FE2.2, FE4.1, AE7.2, AE8.5, AE9.3

Table B.1: Exploitable items and methodology activities.