
Project N°: **FP7-231620**

Project Acronym: **HATS**

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

Instrument: **Integrated Project**

Scheme: **Information & Communication Technologies**

**Future and Emerging Technologies**

# Deliverable D1.1B
# Report on the Core ABS Language and Methodology: Part B

Due date of deliverable: (T12)

Actual submission date: 1st March 2010

Revision date: 30th March 2010


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

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

Revised version

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

# Executive Summary:
## Report on the Core ABS Language and Methodology: Part B

This document summarises deliverable D1.1B 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 reports the development of the HATS methodology as part of Task 1.1. Specifically this report describes a high level view of the HATS methodology, identifies in each phase in the method its relevant work tasks and shows how work tasks' contributions relate to the phases.

## List of Authors

Einar Broch Johnsen (UIO)
Richard Bubel (CTH)
Ralf Carbon (FRG)
Dave Clarke (KUL)
Mads Dam (KTH)
Frank de Boer (CWI)
Nikolay Diakov (FRH)
Samir Genaim (UPM)
Reiner Hähnle (CTH)
Michiel Helvensteijn (CWI)
Bjarte M. Østvold (NR)
Arnd Poetzsch-Heffter (UKL)
Davide Sangiorgi (BOL)
Balthasar Weitzel (FRG)
Yannick Welsch (UKL)
Peter Wong (FRH)

# Contents

# Chapter 1

# Introduction

The HATS Deliverable D1.1 is presented in two parts.

- **D1.1A** *Report on the the Core ABS Language and Methodology: Part A* reports on the core ABS language, and
- **D1.1B** *Report on the the Core ABS Language and Methodology: Part B* reports on the HATS methodology.

This is Deliverable D1.1B and describes the HATS methodology based on software product line engineering processes.

## 1.1 Methodological Requirements: An Overview

During the *requirement elicitation* process we have established several high-level requirements that are relevant to the HATS methodology [8, Chapter 2]. Below we provide an overview of these requirements. Note that for brevity we group related requirements with a brief description. Detail descriptions of individual requirements can be found in Deliverable 5.1 [8, Chapter 2].

### 1.1.1 Product Line Engineering

Product line engineering (PLE) provides the basis on which the HATS methodology is developed. We have identified requirements from the point of view of a PLE researcher. In particular, we have identified the following requirements to the HATS methodology [8, Section 2.1].

- **MR1**: HATS should support product line engineering.
- **MR2**: HATS should support existing application engineering approaches.
- **MR3, MR4, MR5, MR6**: HATS should facilitate the specification, generation and quality assurance of reusable artifacts.

### 1.1.2 Organization: Fraunhofer IESE perspective

Fraunhofer IESE, as an applied research organization focusing on technology transfer, has identified the following requirements from the perspective of managers and software developers of an organization that would adopt the HATS methodology [8, Section 2.2].

- **MR7**: HATS should support incremental introduction.
- **MR8**: HATS should support partial adoption.

- **MR9**: HATS should be empirically evaluated.
- **MR10** HATS should scale to provide quality results on large systems.
- **MR11, MR12**: HATS should be easy to learn and to use,
- **MR13**: HATS should reduce manual effort.

### 1.1.3 Industrial Applicability: Fredhopper perspective

Fredhopper, as an industrial partner to the HATS consortium, has identified the following requirements of the HATS methodology [8, Section 2.3].

- **MR14, MR15**: HATS should support iterative development. For example, if HATS provides technology for code generation, then model mining should also be possible. This means HATS should also support iterative updates of the test system.
- **MR16**: HATS should provide tools to support structural code profiling so that performance metrics of the code may be characterized by the code's decomposition.
- **MR17**: HATS should provide analytical tools for protocol analysis during service-oriented software deployment phase.
- **MR18**: HATS should support usage in an integrated (development) environment.

### 1.1.4 End-User Panel

By interviewing the end-user panel of the HATS [4], the following requirements [8, Section 2.4] to the HATS methodology have been identified.

- **MR19**: HATS should be able to import requirement and design models constructed using existing modeling languages, such as UML.
- **MR20**: HATS should specify the extension points of an ABS model, as well as the core of the model. This requirement facilitates model extension.
- **MR21**: HATS should support service-orientation such as specifying properties of individual services as well as their composition.
- **MR22**: HATS should support abstraction from specific middleware technologies.
- **MR23**: HATS should support the specification of architectural variations.

## 1.2 Approach

Our approach to developing the HATS methodology is based on existing literature [2, 5, 6] and best practices [12] in product line engineering (PLE). We study various PLE methodologies [2, 5, 6] and determine the typical phases in a PLE methodology. We then tailor each phase from the methodology in order to arrive to a formal approach to product line engineering. This approach immediately addresses Requirements MR1 and MR2.

In this document we describe our vision of the HATS methodology at the current stage of the HATS project. We also provide a preliminary mapping highlighting how we expect the technical contributions delivered by work tasks in the HATS project to correspond to steps in the methodology.

## 1.3 Structure

The structure for the remaining document is as follows: Chapter 2 introduces the core concepts of the product line engineering processes.We describe in detail the HATS methodology and how each work task in the HATS project contributes to the method in Chapter 3. In Chapter 4 we conclude this document and describe our outlook on both this document and the development method.

# Chapter 2

# Product Line Engineering: An Overview

In this chapter we give an overview of concepts in product line engineering (PLE) [5, 2, 6] and best practices [12]. We studies one of the existing PLE methodologies: Fraunhofer PuLSE [5].

Product line engineering splits the overall development lifecycle into application engineering (AE) and family engineering (FE) (see Figure 2.1 [5] for an overview and Figure 2.2 for details). FE builds reusable artifacts that are stored in a product line artifact base. This provides the scope to high-level requirements that are to be fulfilled by FE. The product line artifacts provided by FE are generic, i.e. they may contain variation points. AE builds products for specific customers based on reuse from the product line artifact base. To come up with the customer specific products, variation points in reusable artifacts are resolved and product specific extensions are made

In general, managing variability is one of the major challenges in product line engineering. All product line artifacts can contain variability. The overall variability is described in an orthogonal variability model [6]. The variability model is called orthogonal because it models the variability outside of development artifacts such as requirements, the architecture, or components. Links from the orthogonal variability model to the development artifacts describe how the variability affects such development artifacts. The following issues

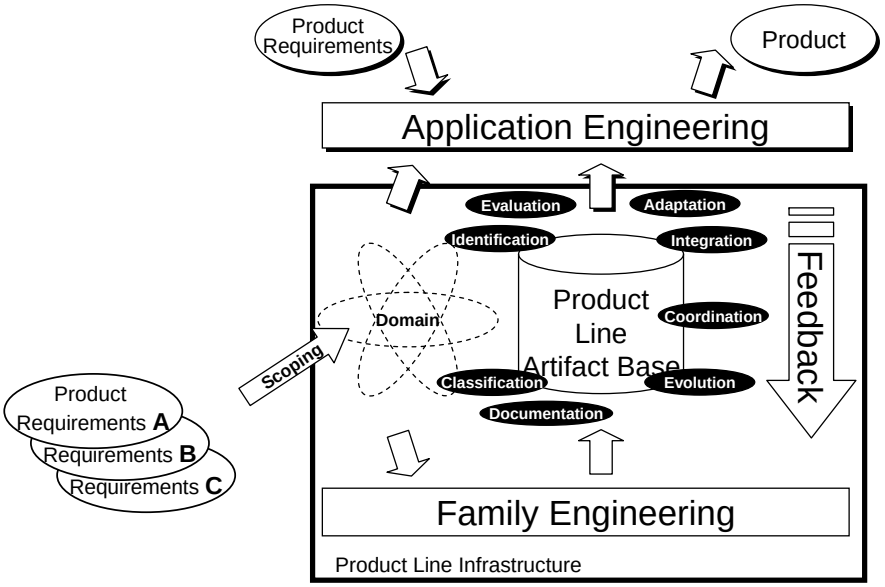


Figure 2.1: An overview of the Product Line Lifecycle

are related to the product line artifact base that need to be dealt with in the context of software product lines. Any sound PLE method must provide solutions to these issues, as mentioned in [5].

**Documentation** Every artifact that is placed in the artifact base for reuse must be documented in such a way, so that potential reusers can work with the artifact.

**Classification** There needs to be a classification scheme that structures the reusable artifacts and their documentation, so that they are accessible in an optimal way. Classifying artifacts means that there is an abstract classification model available that defines attributes for artifacts. If a new artifact is put into the artifact base, values are assigned to these attributes. Guided by the values, a concrete artifact can be identified, as described further below. An optimal classification scheme minimizes both the effort for classifying an artifact by providing values for its classification model and the effort for identifying it again.

**Identification** To enable efficient reuse of artifacts during the application engineering process there needs to be a systematic technique that identifies potential reuse candidates. This technique is build on the classification scheme mentioned above. In the ideal case, the technique would return one concrete artifact. In reality however, most cases result in a set of potential reusable artifacts.

**Evaluation** After identifying a set of potential reuse candidates, they need to be evaluated with respect to their adaptation and development effort. The actual method for estimating this effort needs to be systematic and repeatable.

**Adaptation** In most cases a reuse candidate needs some adaption or configuration effort to make it fit into application-specific requirements. It is clear that the effort for reusing the artifact has to be less than the effort for developing it from scratch. Furthermore, the actual effort must be in the same cost range as the cost estimation done during the evaluation of the artifact for reuse.

**Coordination** When several application engineering projects may run concurrently, the enterprise that owns the product line infrastructure needs to coordinate the reuse activities of these projects to ensure that no identical adaptations would be carried out twice.

**Integration** After adapting the reusable artifact to the application specific context, it needs to be integrated into its new environment. For that reason any assumptions that are taken for that artifact need to be checked to see if they still hold in the setting of the new environment.

**Evolution** Maintaining artifacts that are reused in several applications, is a complex process. It is necessary to reduce the effort spend in maintaining the different adaptations of the artifact. In the ideal case the this maintenance process also imposes and optimizes artifact base.

## 2.1 Variabilities

Variability is handled throughout the product line lifecycle by the so-called Variability Management Process. Variability is modeled on different levels of abstraction. External variability is modeled on the level of the product line *scope* and product line *requirements*. Both of them are visible to the customer using the product. Internal variability is modeled in all other development artifacts like, for instance, the reference architecture, generic components, generic test cases, etc. Variability management includes the selection of mechanisms to realize variability in development artifacts, decisions on the resolution time of variability, etc.

## 2.2 Family Engineering

The Family Engineering Process as depicted in Figure 2.2 starts with planning and scoping the product line. The scope forms the basis for a more detailed analysis of requirements. The resulting refined requirements are input to the design of the reference architecture. The generic components identified during architectural

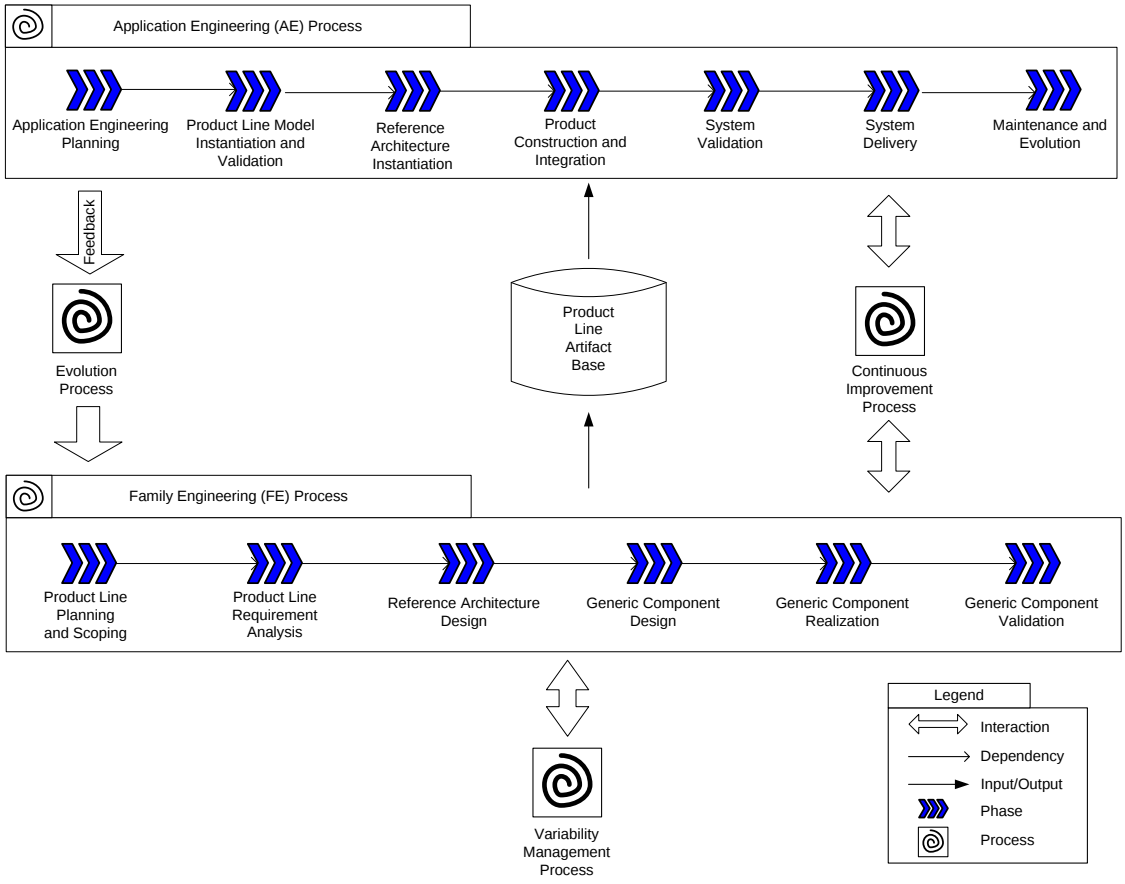


Figure 2.2: Product Line Lifecycle

design can then be designed in detail adhering to the rules that are prescribed by the reference architecture. Generic components are then realized and a validation is performed. The activities in Family Engineering do not need to be executed strictly sequentially, but can also be executed partially in parallel. The validation of certain generic components, for instance, can start before all generic components are realized.

## 2.3 Application Engineering

The Application Engineering Process starts with a planning step of the project to build a specific customer's product as shown in Figure 2.2. The product line model describing the product line requirements can be instantiated and validated. After that the product requirements are fixed, possibly including new customer specific requirements. Based on the product requirements, the reference architecture is instantiated, validated, and, if necessary, customer specific requirements are adapted into it. Based on the instantiated reference architecture the product is constructed and integrated: this includes instantiation and adaptation of generic components, development of product specific components, testing of components, etc. Finally, the overall system is validated and can be delivered. After delivery it enters a product specific maintenance and evolution phase.

## 2.4 Evolution

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. Furthermore, evolution can be initiated by the overall company strategy. If the scope needs to be changed based on, for instance, overall changes of the customer demands, new products of competitors, or opportunities to enter new markets.

Note that evolution is a major challenge in product line engineering. Figure 2.2 provides a more detailed view of the product line lifecycle shown in Figure 2.1. It forms the basis for the HATS methodology.

## 2.5 Continuous Improvement

The 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.

# Chapter 3

# HATS Development Methodology

## 3.1 Overview

This section describes in detail each step in the HATS methodology and highlights how each work task in the HATS project contributes to each step in the methodology. Specifically we derive the HATS development methodology from the existing industrial product line engineering (PLE) method described in Section 2. Figure 3.1 shows the product line lifecycle in the HATS development methodology and highlights the phases where HATS contributes to it. In addition Figure 3.3 on Page 23 presents the general mapping between contributions of the HATS project at task level and the HATS product line life cycle. Table 3.1.1 relates each work task number to its task name. Below we overview the differences between the existing PLE method and the HATS methodology.

- The HATS methodology emphasizes on a formal software product line engineering approach. For this purpose, in the application engineering process HATS specifically extends the Product Line Model Instantiation and Validation activity and the System Validation activity. The former extension adds formal verification activities as early in the process as possible; the latter extension leads to the adoption of both testing and verification before the product is delivered.

- The HATS project aims to deliver a formal compositional modeling language for specifying components in a software product line. For this purpose we aim to move some of the testing and verification activities onto the family engineering process. Specifically, we extend the Generic Component Validation phase to include both testing and verification activities. Note that we envisage that both Generic Component Validation may be carried out in parallel with Generic Component Design and Generic Component Realization; Section 3.2 explains the concurrent nature of the development method in more detail.

- The HATS methodology aims to support continuous development for the long-living product line as well as for the individual family members. This is achieved by developing theories and techniques for handling continuous changes (evolution) to software systems. To reflect this support methodologically, we expanded the Maintenance and Evolution phase with HATS contributions. The iterative nature of the HATS development method is explained in Section 3.2.

### 3.1.1 Scoping

Note that the preliminary mapping that we have provided in Figure 3.3, reflects how the expertise of consortium members contributes to the different phases in a PLE methodology. Nevertheless, providing complete formal support for some of the more work intensive phases (e.g., the Reference Architecture Design phase) may prove overly ambitious and very challenging from a formal scientific perspective. Therefore, we focus contributions by scoping our work in the following manner:

- Some phases in the industrial PLE include informal processes with customers, and therefore HATS contributes only to the phases to which we can apply formal development techniques. Specifically, HATS does not contribute to Product Line Planning and Scoping, Application Engineering Planning and System Delivery phases.
- Some phases in the industrial PLE, such as the Reference Architecture Design phase, contain huge amount of technical work items. In order to leverage the contributions quality and impact with the available HATS resources, each work task only focuses on particular technical aspects of the phases in the methodology. We focus on aspects that have the highest scientific impacts. but are also most amenable to the development of tool support and the integration in a development framework together with the contributions from the other work tasks. Later in this chapter, for each work task, we define the focus and scoping in the context and terminology of the particular academic contribution.
- Some tasks provide basic framework and tools to support for the contribution of other tasks to the PLE phases, and as such do not directly appear in the mapping in Figure 3.3. Specifically, we do not explicitly provide mapping for Tasks 1.1, 1.3, 1.5, 3.1 and 4.4 with phase in the PLE.

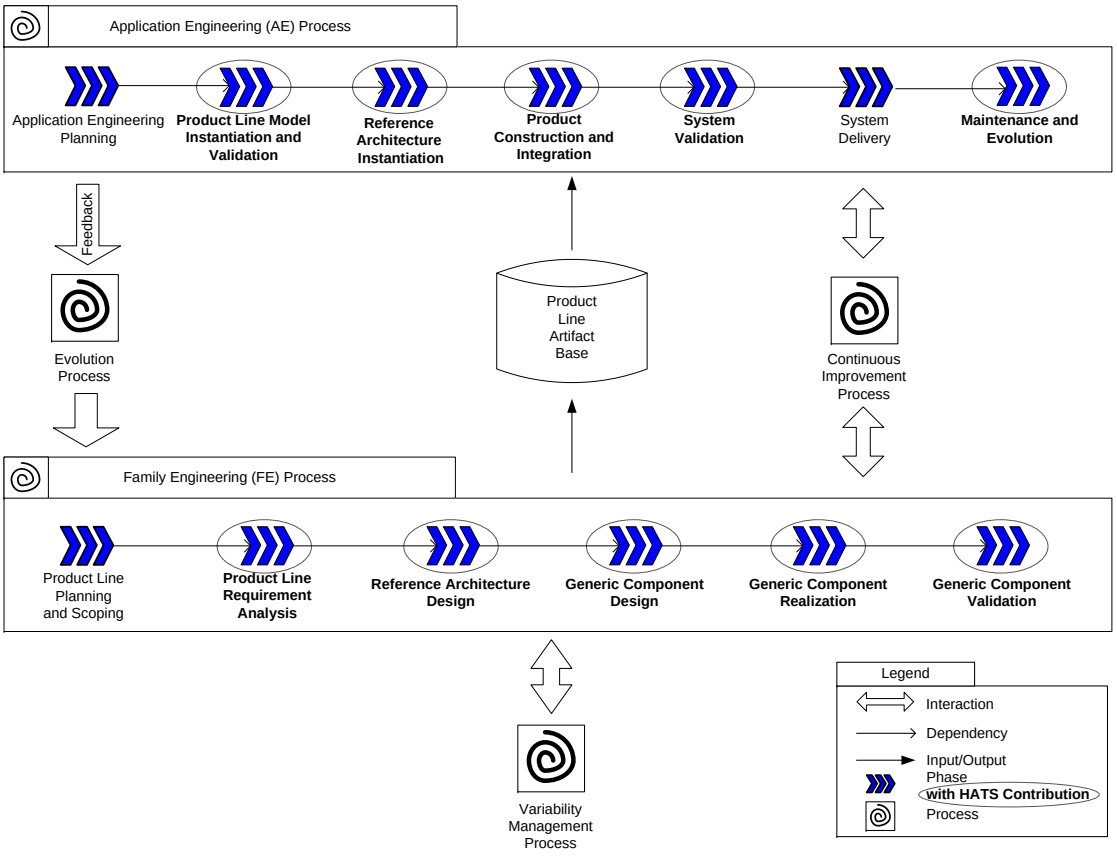


Figure 3.1: Product Line Lifecycle in the HATS Development Methodology

### 3.1.2 Structure

The rest of this section is structured as follows: We first describe the iterative, incremental, and concurrent nature of the HATS methodology in Section 3.2; Section 3.3 describes how different variability models constructed in a product line are managed; Section 3.4 describes the family engineering process in the HATS methodology; Section 3.5 describes the application engineering process in the HATS methodology, and

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

Table 3.1: A table relating work task numbers to their names

Section 3.6 describes the evolution process that connects the family engineering process with the application engineering process;

## 3.2 Iterative, Incremental and Concurrent Development

The HATS methodology should be iterative (MR14, MR15). This is achieved by a combination of evolutionary and continuous improvement processes. Details about these processes are provided in Section 3.6.

The HATS methodology supports incremental development. In the family engineering process, we will leverage the compositionality of the ABS language to allow generic components to be designed, realized and validated in small steps. Furthermore, HATS' contribution in the area of software evolution should provide the necessary technology to support gradual introduction of variabilities and requirements into the product line.

The HATS methodology also supports concurrent development. In family engineering, by exploiting the compositionality of the ABS language, generic components may be validated, while separate new components are being designed and realized by independent teams in parallel. Specifically, the HATS methodology supports concurrent development between the *Generic Component Validation* phase and both *Generic Component Design* and *Generic Component Realization* phases. Furthermore, as soon as a sufficient number of generic components have been realized and validated, they may be instantiated for constructing members of the software family. Therefore, the HATS methodology supports concurrent development between the *Application Engineering* and the *Family Engineering* processes.

## 3.3 Variability Management Process

The Variability Management Process controls and coordinates all activities related to modeling and realizing variability in the product line. Variability appears in the HATS methodology at different abstraction levels and in different product line artifacts like, for instance, requirements, architecture, component designs, code, etc. Variability management provides means to express and realize variability in the product line artifacts and decides when, how, and by whom variability is resolved during application engineering (or even during runtime). There are several variability models instrumented in both family engineering and application engineering. In general, the variability is introduced in family engineering, independently outside the development artifacts. The resolution of the variability is done in the application engineering process when reusing an artifact. Resolving variability can be a multi-staged process. At the topmost level of the product line model there is variability that is externally visible to the customer. Decisions taken at that level trigger other decisions that are resolved internally by architects or developers. Every level of resolution demands some specific knowledge and is in general performed by different specialists in different roles.

## 3.4 Family Engineering

The family engineering process (FE) identifies commonalities and variabilities of the product line and builds reusable artifacts for the product line artifact base. The workflow of this process is shown at the bottom of Figure 3.1.

### 3.4.1 Product Line Requirement Analysis

In the Product Line Requirement Analysis phase, we analyze in detail variabilities and requirements of the product line defined during the Product Line Planning and Scoping phase. In particular, we apply the feature modeling technique to produce feature models describing common and variable features and their constraints. In the HATS project, we aim to provide the theory and the language for modeling and studying features as well as specifying constraints between features. They help to resolve ambiguities within the informal requirements of the product line. Models developed in this phase could then be used in later phases to guide design and validation. The work tasks that aim to develop technologies for this phase are Tasks 2.1, 2.2 and 2.4. Note that contributions from these tasks help to address Requirements MR4.

Task 1.2 aims to develop a high level formal feature description language for expressing variation points in the product line. For example, variation will include details of platforms, which are expressed as attributes in feature models, as well as other configuration parameters. In addition to the language construct provided, the language is equipped with abstraction and refinement mechanism, to assist the development of the feature models as well as to provide different viewpoints to the feature models.

Given a feature model, specification of constraints and dependencies between variation points may be formalized as *behavioral types*. These types specification may then be used to guide the construction of the reference architecture; behavioral types will be studied in Task 2.4. In addition, delta modeling may be employed in this phase to provide insight for other high level requirements of the product line; delta modeling will be studied in Task 2.2

Besides variabilities descriptions, there exist informal product line requirement artifacts and they may be documented in various ways, e.g., use cases, workflows descriptions, etc. In this phase the variabilities described in feature models are linked precisely to variation points in these requirements artifacts.

### 3.4.2 Reference Architecture Design

In the Reference Architecture Design phase a common architecture is defined for all product line members. This is called the *reference architecture*. The reference architecture is documented by means of different architectural views containing information about component interfaces and interactions, overall system behavior and system's variabilities. The reference architecture also specifies rules to which the design

of *generic component* must adhere. This is so that component designs do not corrupt the overall system quality. In this phase HATS aims to provide the theories, techniques and tools support for the distribution of variabilities over components as well as the specification of component functionalities, system-level invariants and cross-cutting, non-functional properties. The contributing tasks to this phase are Tasks 1.2, 2.1, 2.2, 2.4, 2.5, 3.3, 4.1, 4.2 and 4.3. Techniques developed for defining the reference architecture help to address Requirement MR23.

**Variability Distribution**

It is important to ensure that the components in the architecture cover all variation points and do not invalidate any of the variation constraints. The feature description language developed in Task 1.2 provides hooking mechanism for incorporating variation points into the reference architecture at component level, thereby assisting the distribution of variability. Coupled with the feature description language is delta modeling developed in Task 2.2 [10, 9]. Delta modeling connects the features in the feature description language to design artifacts, in this context to the reference architecture and its contained components. The architecture of one product is taken as the core product. Delta models define changes to this core architecture to implement further products. The application condition of a delta model determines for which combination of features the changes are applied to the core architecture, linking features to design artifacts.

**Generic Component Specifications**

Since variation points are assigned with behavioral types during the Product Line Requirement Analysis phase, this information may be used when designing the reference architecture to specify functionalities of components of the architecture. In particular, behavioral types help specifying the functionalities that components offer to the environment; the dependencies of components, that is, what the deployment environment has to provide for the components to function; and contracts on the expected interaction between the component and its environment. Techniques to apply behavioral types at both the analysis and design phases are investigated in Task 2.4.

**Interface Specification**

Behavioral interfaces are a complementary approach to the specification of component behaviors. Interfaces specify abstractly the behavioral requirements and expectations of the reference architecture at each variation point and for each component. Since interfaces help specifying behavioral constraints of the component interactions, this information facilitates well-formed composition and enables safe evolution to be checked statically and dynamically. The techniques for specifying external behaviors of ABS components via interfaces are developed in Task 1.2, while static and dynamic checkings on evolving systems are investigated in Task 3.3. Note that the contributions on behavioral types for interface specification partially address Requirements MR17 and MR21. In particular, interface specification have already been studied in the context of service oriented architecture in the form of contracts [1].

**System-level Specifications**

The reference architecture also records system-level information that normally cross-cuts many components in the system. This information includes variabilities, invariants and resource guarantees:

- **Cross-cutting variabilities**: Cross-cutting variabilities include the specification of different deployment features (concurrency, distribution and failures). These variabilities need to be resolved to ensure that the interfaces of connecting components agree on the *failures model* during deployment, and that components can be instantiated accordingly. Techniques for capturing and resolving cross-cutting variabilities for deployment are considered in Task 2.1. The notion of a failures model helps abstracting underlying middleware and hardware, thereby addressing Requirement MR22 in Section 1.1.

- **Invariants**: System-level invariants specify properties that are to be provided by the environment. These properties can be assumed (system inherent) or need to be maintained by the different components within the architecture. In the specification phase we aim to make formal deductive verification techniques available for verifying invariants. These techniques are investigated in Task 2.5. Furthermore, Task 2.5 develops a calculus for reasoning about invariant properties; Note that the specification of variation points and the realization of compositionality on the calculus level can influence each other.

- **Resource Guarantees**: The reference architecture specifies resource constraints such as execution cost and security requirements of product. To this end, abstract specification interfaces are used to enable modular reasoning of these cross-cutting aspects. The development of abstract specification interfaces is carried out in Task 4.1. While abstract specification interfaces focus on the specification of security requirements, a series of cost models is developed in Task 4.2 for specifying the upper and lower bounds of execution costs. We aim to specify this information at the level of variation points of the reference architecture. This means a safe evolution of the reference architecture satisfying these constraints guarantee members of the product line to run within some given available resources. A formal characterization of resource consumption encourages tools development for analyzing resources, thereby helps addressing Requirement MR16 in Section 1.1.

#### Evaluation of Formal Techniques

A formal specification of the reference architecture allows for further steps in the validation on concrete systems. Nevertheless, the reference architecture includes features such as variability, which are unconventional in existing classical software architectures. It is a major challenge to formally capture and reason about these features, because ordinary models and formal methods do not take them consideration. Hence, it is very hard, at present, to predict which techniques are best suited for analysis. Task 4.3 aims to address these challenges by evaluating various formal techniques, e.g. operational, coinductive, logic-based and automata-based, in order to understand how to best cope with these features.

### 3.4.3 Generic Component Design

In the Generic Component Design phase, we design in detail the components identified in the reference architecture design process. While properties and constraints that are common to all components have been specified in the reference architecture, a detail internal design of each component is provided in this phase. Specifically, in this phase an executable model of each component is defined using the ABS language. Each component's model specifies and integrates both component-specific and cross-cutting variabilities. Note that components designed in this phase are called *generic components*, because these components may contain variation points that are resolved later during the application engineering process.

Furthermore, having a precise reference architecture and the design models of generic components, one may validate the correctness of the component designs against the reference architecture to ensure consistency is maintained at all levels of abstractions. This encourages both incremental and concurrent development as described in Section 3.2. Activities in the Generic Component Design phase may be partitioned into three categories: variability integration, evolution supports and component model validation, and the contributing tasks to this phase are Tasks 1.2, 2.1, 2.2, 2.3, 2.5, 3.2, 3.3, 4.1, 4.2 and 4.3. Technical contributions for this phase address Requirements MR3, MR4, MR5 and MR6 in Section 1.1. We relegate the discussion of evolution supports to Section 3.6.

#### Variability Integration

It is important to have scalable techniques to develop generic component design models using the language of ABS. Such techniques produce models that are consistent with respect to the variability model and the reference architecture. The main work tasks responsible for investigating such techniques are Tasks 1.2, 2.1 and 2.2 .

- **Feature Modeling**: Task 1.2 aims to develop a formal feature description language. In particular, we aim to use the feature description language for specifying feature models. These feature models formally capture variation points along with all platform related and other configuration parameters. Such models serve to document and guide the generic component designer to ensure that sufficient implementation support for all variation is provided within the generic components. Furthermore, cross-cutting variabilities concerning deployment configuration are captured using an abstract failure model developed in Task 2.1. Using the failure model, generic component design model defined in the ABS language may be instantiated to support failure handling.

- **Feature Integration**: Given high-level feature models, Task 2.2 aims to develop a scalable technique known as delta modeling [10, 9] to compose features at the level of the ABS language. Specifically, deltas help to formalize the underlying behavioral semantics of individual features as well as of combinations of features. Each delta is annotated with an application condition, indicating the feature configurations for which it applies. During feature integration, delta modeling helps to resolve conflicts between interdependent features without affecting the behaviors of any non-related features.

**Component Model Validation**

It is important to ensure each generic component design model is consistent with respect to the reference architecture. During the Generic Component Design phase, we aim to carry out some validation activities for the design models. The validation process for the generic component design is investigated in Tasks 2.3, 2.5, 4.1, 4.2 and 4.3.

- **Symbolic Execution**: While generic component design models are defined using the executable ABS language, they may not be executed directly because they are abstractions of the actual components and contain unresolved variabilities. We aim to address this challenge by investigating and developing technologies based on *symbolic execution*. This is carried out in Tasks 2.3 and 2.5. In Task 2.5, a program logic calculus based on symbolic execution is developed. Specifically this calculus helps realizing a symbolic execution engine, which is employed for simulating the execution of the ABS models. For example, in Task 2.3 a visualization tool based on symbolic execution of ABS models is provided. As a consequence, it becomes possible to animate the dynamic behavior of *generic* and *incomplete* components without the need to construct concrete implementations. This helps consolidating the generic component design models. Furthermore, some of the approaches explored in the above mentioned tasks provide suitable treatment for handling variation points under symbolic execution. Note that under symbolic execution, it should be possible to verify certain ABS fragments against their specification under the assumptions that the code inserted at the variation points adheres to specified restrictions.

- **Formal Verification**: Given a formal design model of the generic components, we aim to carry out some verification activities to ensure that the design models satisfy the specified behavioral and resource guarantees. Formal verification activities are investigated in Tasks 4.1, 4.2 and 4.3. Task 4.1 provide a formal basis for parametric specification that is used for reasoning about generic components design models. The parametricity in specification helps adapting formal verification to the variability of the environments in which the software component can be instantiated. Task 4.2 investigates the verification of resource usage requirements. In particular, we aim to verify resource usage at the level of the design models with respect to the constraints specified in both the variability model and the reference architecture. Task 4.3 considers the general notion of correctness of the generic components, and in particular, different verification methods are studied and suitable methods for various behavioral guarantees are identified.

### 3.4.4 Generic Component Realization

Having the design of the generic components at hand, the goal of the Generic Component Realization phase is to realize them according to their intended design, so that they can be added to the product line artifact base for reuse. As mentioned above, generic components contain variation points that are resolved during application engineering. All variation points for a generic component are consolidated in a variability model for that component. This provides the necessary link between the implementation of the generic component and the variabilities it supports. This link enables the application engineer to reuse generic components by resolving the specified variation points and to instantiate them as concrete components. The contributing tasks to this phase are Tasks 1.4, 2.3, 2.5, 4.1 and 4.2. In particular Tasks 1.4, 2.3 and 2.5 focus on the development of theories and tools for specifying and debugging generic components. Tasks 4.1 and 4.2 focus on the validation of generic components. Task 4.1 also studies the mechanisms for automatic code generation, thereby reduces manual effort during implementation. Automatic code generation helps to address Requirement MR13. We relegate the discussion about generic component validation to Section 3.4.5.

Task 1.4 aims to provide tool support for constructing specification and defining preferences for different variation points. This helps guiding the process of generating executable code from design models. Task 2.3 aims to develop debugging tools based on symbolic execution. In contrast to conventional debugging, a symbolic debugging session does not require concrete start states. It provides both forward and backward navigations along *all* possible execution paths up to a finite depth. Since implementation of generic components are not directly executable, symbolic execution debuggers are ideal to support the implementation and validation of generic components. The underlying formal treatment of symbolic execution is investigated in Task 2.5.

### 3.4.5 Generic Component Validation

After generic components are realized, they are *validated* to ensure that they are correct with respect to their specification. Note that by leveraging the compositionality of the ABS language, it is possible to carry out validations on generic components during the reference architecture design and the generic component design phases. In this phase the validation process may be partitioned into two technical arenas: formal testing and formal verification.

Formal testing is investigated in Task 2.3. Specifically in Task 2.3, various formal black-box and white-box test case generation methods are considered. Black-box methods are aimed at validating not yet implemented components for which only a contract exists. On the other hand, white-box methods are based on symbolic execution of concurrent ABS models and hence create self-contained unit test cases with at least feasible statement coverage.

Formal verification is investigated in Task 2.5. Specifically this task aims to provide a program logic for modeling the ABS language faithfully such that it allows behavioral and functional aspects of generic components to be verified. Note that different approaches to achieve reuse and compositionality at the later stages during the application engineering will have influence on the kind of proof-obligations to be verified.

A complementary approach to validate the correctness of a realized generic component is via *model mining*. Instead of validating the implementation of the generic component against the component's ABS model, this approach uses model mining to derive an alternative (partial) model of the component and compares the derived model with the component's ABS model. This is particularly useful when the ABS model is written manually and the generic component cannot be automatically realized. This model mining approach is investigated in Task 3.2.

After passing this validation process, generic components are ready for reuse and are put into the product line artifact base. During the application engineering process, generic components, their specifications as well as *verified properties* can then be reused.

Task 4.1 aims to provide mechanisms to check whether a particular implementation of the generic component is consistent with the validation results obtained from the abstract design model. To this end, it is important to verify each instantiation of the variation points against the conditions required over the

generic component parameters. Task 4.2 aims to provide technologies for verifying the resource consumption on the implementation of a generic component against the specification of its design model. This means that the implementation must be inspected by means of static analysis techniques, in order to check that the resource usage falls within the constraints which were posed by the design model. Moreover, bounds on resource usage on variation points must be integrated into the implementation of the generic component as well as its documentation, so that the application engineer may refer to this information to resolve variations points when reusing generic components.

## 3.5 Application Engineering

The application engineering process (AE) builds products based on reusing artifacts from the product line artifact base. When reusing artifacts, their variabilities are resolved. Note that new customer specific requirements are also taken into account when resolving variabilities. The workflow of this process is shown at the top of Figure 3.1.

### 3.5.1 Product Line Model Instantiation and Validation

During the Product Line Model Instantiation and Validation phase the external product line variability is resolved. This is achieved by applying feature modeling techniques developed in Task 1.2. The feature model, which is constructed using the feature description language during the Product Line Requirement Analysis phase, specifies all variation points available to the customer. Feature selection at this level occurs by specializing the feature model, making choices and selecting values for attributes. Note that not all variability defined in the feature model may be resolved by the customer. There are variation points that require the knowledge of a system architect, and hence such variation points are resolved internally; these internal decisions should not alter the system's external properties.

The process of resolving the feature model ends when variabilities are resolved and a ground feature model is obtained. A ground feature model corresponds to a single product that satisfies all the constraints specified in the feature model. To ensure that the resulting product represents the ground feature model, Task 2.6 provides the means to realize automatic consistency checking for feature selection and system derivation. To this end, a refinement theory with respect to variation points, is provided for checking implementation correctness.

Besides resolving the external variability, this phase also includes requirements analysis for customer specific requirements that cannot be mapped to variation points in the feature model.

### 3.5.2 Reference Architecture Instantiation

Given a precise description of the product requirements, the reference architecture may be instantiated in the Reference Architecture Instantiation phase. This means the internal variability model for the reference architecture needs to be resolved. Furthermore, the resulting product architecture needs to be validated against the product requirements. It is often necessary to make changes to adapt customer-specific product requirements. This means changing either the product architecture or the reference architecture to include these new requirements. The decision about changing the product architecture directly or changing the reference architecture, is taken in the context of the evolution process described in Section 3.6. The contributing task to this phase is Task 4.3.

After internal variabilities are resolved, it is important to prove their correctness with respect to the reference architecture. Techniques for this are developed in Task 4.3. The resulting correctness proof takes into account the specifications of the reference architecture as well as the product requirements. Similarly, the correctness of the appropriate adaptation steps for the product architecture is studied in Task 4.3.

### 3.5.3 Product Construction and Integration

During the application engineering process, generic components are instantiated and reused according to the product's architecture. As mentioned in Section 2 there are several issues to be taken care of when reusing components. After identifying the necessary reusable components, they are either adapted to fit the product requirements, or new product specific components are developed instead.

If the decision about adapting existing components or building new ones cannot be taken in the application engineering process, it is forwarded to the evolution process as described in Section 3.6.

To enable efficient reuse, every generic component is associated with a variability model. The variation points in the variabilities model are resolved in this phase. The delta modeling technique, to be developed in Task 2.2, provides a mechanical procedure to resolve variability. Given adequate preparation when designing the generic components, this procedure may be automated. Specifically, required code-changes to specific products are applied to the generic components directly, while very specialized changes, even those affecting only a single product, may be written in a precisely targeted delta.

Task 2.4 considers type system features, such as subtyping and generic polymorphism. Such type system features can help to ensure that the mechanisms adopted for resolving variability comply with the initial requirements of the product. Type systems also guides the resolution of variability, and facilitate the adaptation and integration of the components that have been identified for reuse.

The correctness of variability resolution in generic components has to be established. This is addressed in Task 4.3. Specifically, one major issue is verifying correctness on the composition of the selected components. The difficulty of this task further increases if there are explicit behavioral requirements on the composition (e.g., constraints on the occurrences of certain interactions).

Task 1.4 aims to provide the required tools for translating generic components into executable code. These tools will follow a semantic-based approach based on writing an evaluator for generic components in some (rich enough) intermediate language, the tools then use a well-known partial evaluation technique which, given the evaluator, automatically translates the generic components into programs written in the intermediate language. The intermediate language can be then compiled to the target language (e.g., Java). By using an intermediate language, we alleviate the needs to write (partial) evaluation component for any target languages.

Task 3.4 investigates code injection techniques. Known techniques, such as monitor inlining, offer a way to provide strong property guarantees for production code, including legacy code, without the need for full formal analysis. In this way, it is often possible to adapt existing code to a wide range of requirements, with a low development effort. Using proof-carrying code, formal correctness proofs can be provided in a fully automatic way, for use both at time of integration and at runtime.

### 3.5.4 System Validation

The product that was constructed during the application engineering process needs to be validated for consistency with respect to the product line requirements and any product-specific requirements. The contributing tasks to this phase are Tasks 2.3, 2.5, 4.1, 4.2 and 4.3.

Task 2.5 aims to develop the theories and techniques for conducting proofs for functional correctness of the constructed product. Task 2.5 provides measures to compile such proofs efficiently by reusing and composing proof artifacts constructed at earlier stages of the life cycle, e.g. during the family engineering process.

Furthermore, a number of techniques aimed at ensuring the correctness of the product with respect to all initial requirements, are studied in Task 4.3. These analyses can be carried out both statically and dynamically. Combinations of techniques and approaches may have to be considered as well. Finally, the possibility of machine-aided validation is also considered, that is, we consider both interactive and (semi)-automatic validation techniques.

Task 4.1 aims to develop a candidate description language for specifying security requirement. This language is designed to accommodate transformation of specifications due to code adaption. Similarly, this

language is associated with the enforcement mechanism that could be adapted during the validation process in case the specification is transformed.

Task 2.4 considers techniques to validate the application against the resource usage requirements. First, the resolution of variation points and instantiation for a concrete application is required not to invalidate any proofs of resource usage of the generic component. On the other hand the produced application also has to satisfy the resource usage behavior that was elicited during the Product Line Model Instantiation phase and documented in its ground feature model.

Task 2.3 aims to provide test case generation methods that take into account deployment issues such as scheduling. Specifically, we provide test cases that resolve non-determinism due to multi-core and/or distributed architectures in a predictable way, which makes tests *replayable*.

### 3.5.5 Maintenance and Evolution

This phase is concerned with managing changes to released products. Such changes happen because either product-specific requirements have changed, the product line requirements have changed or fixes are necessary. The contributing tasks to this phase are Tasks 3.4 and 3.5.

Task 3.4 studies and develops dynamic code injection mechanisms, that can be used to dynamically adapt running code to changing requirements with strong correctness guarantees. Currently, the focus is on monitor inlining and security policy enforcement, but the idea is applicable to a wider range of systems properties.

Task 3.5 considers self-adaptation techniques, which can be used to automatically adapt a running system to some types of changes in requirements, environment, and code base. Examples are automatic adaptation to changes in available hardware, automatic fault recovery, and automatic deployment of software components, e.g. runtime monitors, to adapt to changing requirements.

## 3.6 Evolution Process

The Evolution Process builds a link between the application engineering and the family engineering in order to enable efficient handling of reuse issues, as depicted in Figure 3.2. A reuse problem in application engineering occurs when the evaluation of potential reuse candidates reveals that the adaptation of each of the candidates needs more effort than building it from scratch. Since it is not possible for the application engineer alone to decide the solution to that problem, an evolution request would be triggered and handled in the evolution process. An evolution request is handled independently and in parallel to the application engineering process. The main decision that is resolved there is if it is more efficient to improve or recreate a generic artifact, than to develop the artifact only specific for that application. In the former case the result of the decision is sent back to the ongoing application engineering process, so that the creation of the specific artifacts can start. In the latter case, a change request is sent to the family engineering process where the concrete impact for changing of the product line is realized.

Beside supporting evolution during the Maintenance and Evolution phase in the application engineering process, the HATS methodology aims to support the evolution process at both architectural and component levels in the family engineering process. Specifically, we aim to support evolution during Reference Architecture Design and Generic Component Design.

- **Reference Architecture Design**: We use behavioral interface to specify component behavior at the architectural level. Since interfaces help to specify behavioral constraints of the interacting components, this information facilitates well-formed composition and enables safe evolution to be checked statically and dynamically. The techniques for specifying external behaviors of ABS components via interfaces are developed in Task 1.2, while static and dynamic checking of evolving systems are investigated in Task 3.3.

- **Generic Component Design**: Software evolution techniques for generic components are investigated in Tasks 3.2 and 3.3. In Task 3.2, model mining techniques are considered. Model mining

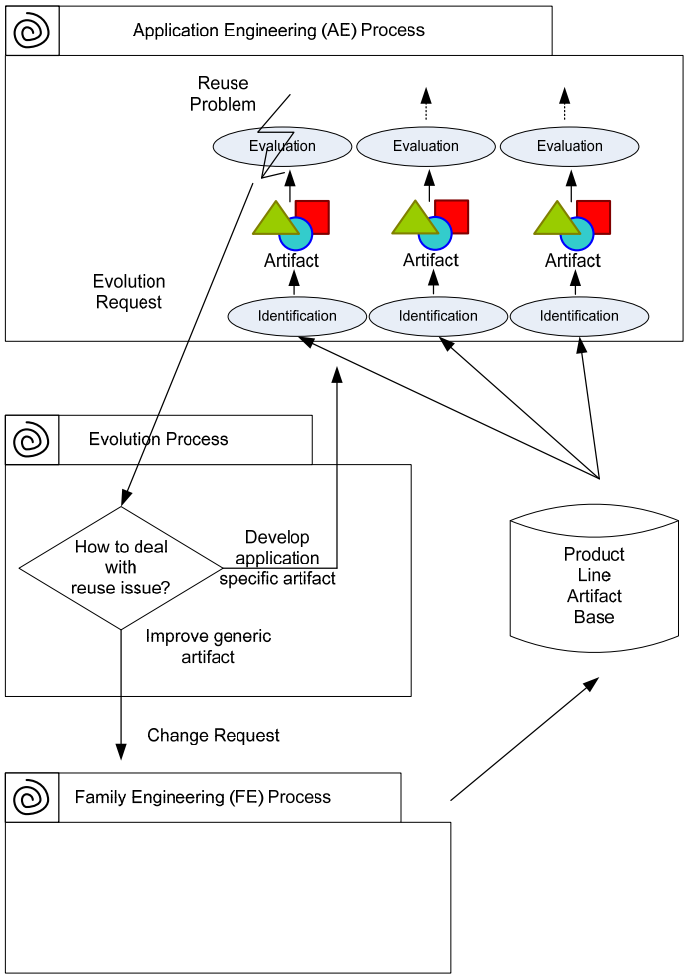


Figure 3.2: Product Line Evolution in the HATS Development Methodology

helps deriving partial models of an application from the application's code base. As a result, given an existing implementation of a component, one could use model mining techniques to formally inspect and revise the corresponding generic component design model. Task 3.3 investigates the application of interface contracts for evolving generic component. Specifically one could specify behavioral requirements and expectations of generic components as interface contracts. Contract specification should be compositional and hence encourage the development of safe evolutions. Furthermore, for each safe evolution, the contract of resulting generic components may be checked either statically or dynamically against the reference architecture and other generic components with which the resulting generic component will be composed.

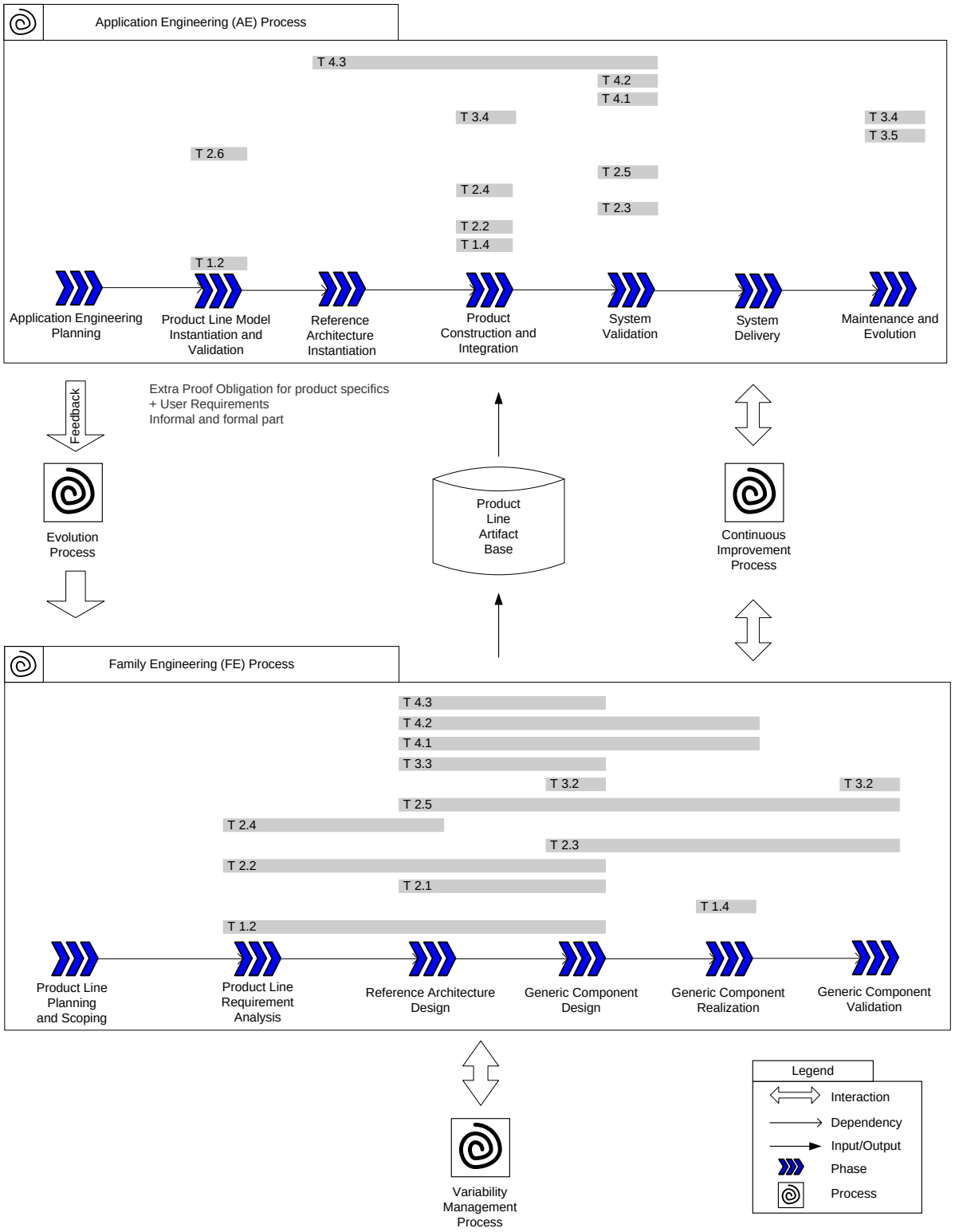


Figure 3.3: Mapping between Work Tasks and HATS Product Line Life Cycle

# Chapter 4

# Conclusion

This document described the HATS methodology. We have shown how we have derived the HATS methodology from a proven industrial strength product line engineering method by applying formal methods to various phases of the method. We have provided an overview of each phase of the methodology. We then identified relevant work tasks to each phase and highlighted how the contributions of these tasks are related to individual phases. We believe that having a top level picture of how contributions of work tasks fit in the development methodology helps the project to achieve its goals successfully. Table 4.1 gives an overview of how we aim to address methodological requirements.

As we are in an early stage of the project, some overarching areas of the methodology have not been discussed in this document. Furthermore, some methodological requirements have yet to be addressed. In this section we describe these areas and how they are addressed by different work tasks in the project.

**An Integrated Framework** An important question in the methodology is how theories, techniques and associated tools delivered by different work tasks can be integrated. This particular question is addressed in Task 1.3. In particular, part of this task aims to develop an integrated, coherent conceptual interface to the various analysis techniques, including language constructs and customization support for properties to be examined. The design of the interface should be user-centric and workflow driven, allowing the user to employ and combine the various analysis techniques effectively [7, Page 32].

The deliverable results of Task 1.3 are served as part of the inputs to Task 1.5. The goal of Task 1.5 is to develop an architecture and hence a prototypical tool platform that supports the software development, deployment, and maintenance following the ABS approach [7, Page 33]. We envisage this tool platform to be integrated into an existing integrated development environment, thereby addressing Requirement MR18 in Section 1.1.

**Evaluation** Requirement MR9 in Section 1.1 specifies that the HATS methodology must be empirically evaluated. Here we give the full definition of the requirement:

> **Requirement** (MR9)**.** *[8, Section 2.2.1] The HATS methodology must be empirically evaluated to demonstrate that the expected benefits like higher quality in terms of trustworthiness, lower effort and cost to achieve such quality, and less time required can be achieved. The benefits need to be quantified by means of concrete measures and the context in which the empirical evaluation has been conducted needs to be thoroughly documented.*

Empirical evaluation of the HATS methodology is carried out in Tasks 5.2, 5.3 and 5.4 [7, Pages 51–52]. Specifically Task 5.2 concerns with the evaluation of the core framework, that is, this task tests the expressive power of the existing parts of the ABS language and the HATS methodology in the form of case studies. Task 5.3, on the other hand, evaluates modeling techniques developed in Work Packages 1, 2 and 3 via case studies. Task 5.4 evaluates the tools and techniques developed in Work Packages 1 and 4 to support the application of the HATS methodology. By actively employing the

HATS methodology in these tasks, we would evaluate the usability and scalability of the method as well as the method's compatibilty with existing software development methods. Empirical evaluation helps addressing Requirements MR2, MR7, MR8, MR10, MR11, MR12 and MR19.

**Methodological Requirements** During the requirement elicitation process, we have identified abstract methodological requirements in the context of "organization" and "end-user panel" [8, Sections 2.2 and 2.4]. Because we are still in an early stage of the project, some of these requirements have yet to be scoped.

For requirements in the context of the organization perspective, we envisage their scoping during the validation process that is to be carried in Tasks 5.2, 5.3 and 5.4. For Requirements MR20 and MR23 in the context of the end-user panel perspective, we defer their scoping to a later stage in the project. Moreover, we only consider Requirement MR21 partially, its full consideration is deferred to a later stage in the project.

| **Identifiers** | **Labels** | **How is it addressed?** |
|---|---|---|
| *Product Line Engineering* | | |
| MR1 | Integrating Product Line Engineering | Design of the HATS methodology |
| MR2 | Integrating application engineering | |
| MR3 | Testing reusable artifacts | Technical contributions for modeling, implementing and validating generic components |
| MR4 | Providing language support for PLE | Application of a formal feature description language and the delta modeling for modeling and resolving variabilities at various levels of abstractions |
| MR5 | Specifying reusable contracts | Application of the ABS language with associated analysis techniques for describing and reasoning about generic contract-based specification during the family engineering process |
| MR6 | Defining reusable artifacts and variation points | Application of a formal feature description language, the delta modeling and the ABS language during the family engineering process |
| *Organization* | | |
| MR7 | Tailorability | Validation in Tasks 5.2, 5.3 and 5.4 |
| MR8 | Incremental adoptability | |
| MR9 | Empirical evaluation | |
| MR10 | Scalability | |
| MR11 | Learnability | |
| MR12 | Usability | |
| MR13 | Reducing manual effort | Automatic code generation for generic components and tools support for various analysis techniques |
| *Industrial Applicability* | | |
| MR14 | Iterative formal modeling | Design of the HATS methodology |
| MR15 | Test system evolution | Evolution process aims to handle changes to all model-based and code-based artifacts |

| | | |
|---|---|---|
| MR16 | Resource guarantees | Application of formal resource analysis techniques to the design of the reference architecture |
| MR17 | Protocol analysis | Application of interface specification techniques to the design of the reference architecture |
| MR18 | Integrated environment support | Development of an integrated framework in Task 1.5 using results from Task 1.3 and application of the framework to the HATS methodology |
| *End-User Panel* | | |
| MR19 | Support existing modeling techniques | Extent to which this is addressed is determined in Tasks 5.2, 5.3 and 5.4 |
| MR20 | ABS extensibility | Deferred |
| MR21 | Service orientation | Application of interface specification techniques during the Reference Architecture Design phase |
| MR22 | Middleware abstraction | Development of an abstract failures model and the application of the model to the design of the reference architecture |
| MR23 | Architectural style | Deferred |

Table 4.1: Methodological requirements

# Bibliography


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

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

[3] Sophia Drossopoulou, editor. *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, volume 5653 of *Lecture Notes in Computer Science*. Springer, 2009.

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

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

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

[7] Highly Adaptable and Trustworthy Software using Formal Models. Project Proposal.

[8] Requirement elicitation, August 2009. Deliverable 5.1 of project FP7-231620 (HATS), available at http://www.cse.chalmers.se/research/hats/sites/default/files/Deliverable5.1.pdf.

[9] I. Schaefer. Variability Modelling for Model-Driven Development of Software Product Lines. In *Intl. Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010)*, 2010.

[10] I. Schaefer, A. Worret, and A. Poetzsch-Heffter. A Model-Based Framework for Automated Product Derivation. In *Proc. of Workshop in Model-based Approaches for Product Line Engineering (MAPLE 2009)*, 2009.

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

[12] F. J. van der Linden, K. Schmid, and E. Rommes. *Software product lines in action: the best industrial practice in product line engineering*. Springer, 2007.

# Glossary

## Terms and Abbreviations

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

**AE** See Application engineering

**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.

**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.

**PLE** See Product line engineering.

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

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

**Reference architecture** The reference architecture is the common architecture is 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.

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