HATS—A Formal Software Product Line Engineering Methodology*

Dave Clarke*, Nikolay Diakov†, Reiner Hähnle‡,
Einar Broch Johnsen§, Germán Puebla¶, Balthasar Weitzel‖ and Peter Y. H. Wong†

*Katholieke Universiteit Leuven, Heverlee, Belgium
†Fredhopper B.V., Amsterdam, The Netherlands
‡Chalmers University of Technology, Gothenburg, Sweden
§University of Oslo, Oslo, Norway
¶Technical University of Madrid, Madrid, Spain
‖Fraunhofer IESE, Kaiserslautern, Germany

***Abstract*—Trust in software is typically achieved via stabilization efforts over long periods of use. Adaptation to changing circumstances, however, often requires substantial changes to the software. Changing a software system using standard manufacturing processes often results in quality regressions, invalidating trust. Formal methods provide a means for guaranteeing various properties of a software system that increase its trustworthiness. The HATS methodology aims to integrate formal methods for modeling changes of software systems in terms of variability and evolution, while preserving trustworthiness properties. This paper outlines how different formal methods are extended and integrated to build an industrially viable Software Product Line Engineering method for manufacturing highly adaptable and trustworthy software.**

***Keywords*-software product lines; methodology; formal methods**

I. Introduction

Finance and health care are two of the many examples where long-lived, trustworthy software systems have a large impact on modern society. Long-lived systems typically show the benefit of trustworthiness by proving their usefulness over a long period of time.

Software systems must regularly adapt to changing circumstances in society. Adaptation to changing circumstances, however, often requires substantial changes to the software. In an industrial context that views a software system as a product, change may be viewed in two ways: *anticipated* changes, which may be due to variations in client requirements, and *unanticipated* changes, which may be due to changes to the market or new technological opportunities. The former defines the variability of a product in relation to its clients, while the latter refers to the evolution of the product.

Changing a software system, for whatever reason, using standard manufacturing processes typically results in quality regressions. For example, after a change in a software system, time and stabilization efforts need to be made in order to regain the trust of its users.

*This research is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (http://www.hats-project.eu).

The "Highly Adaptable and Trustworthy Software using Formal Models" (HATS) project* aims to address the issue of producing trustworthy systems in light of changing requirements. The HATS project achieves this by looking at a software system not merely as a product offering some service, but as a framework around the product. This framework governs how one can modify a software product to meet changing requirements while preserving trustworthiness guarantees.

HATS takes an empirically successful, yet mostly informal, software development paradigm for *software product line engineering* (SPLE) [1] and places it on a more formal basis. The SPLE approach is adopted for two main reasons:

- SPLE considers a product as belonging to a family of products that includes explicit modeling of variability. This abstraction fits well with applying formal methods to anticipated changes in software systems.
- SPLE supports the maintenance of the product line artifact base over time. This support includes evolving the artifact base in order to change the possible products of the product line. This facilitates unanticipated changes.

A. Motivation

The motivation for using the HATS methodology is fourfold.

- **Incoherent integration of formal methods** Complete methodologies consist of many steps, each addressing different aspects of the product lifecycle. Formal methods exist to address most individual aspects; however, integrating them into a development methodology presents a challenge. From a methodological perspective, one needs to reliably propagate verified properties through the entire development work flow.
- **Lack of industrial-level tools** Learning formal methods requires a large investment in time. To reduce costs and increase productivity, tools are required that facilitate the use of formal methods.
- **Low scalability of existing formal methods** Industrial applications are large in terms of combinations of aspects and concerns. It is therefore important to scale formal

methods up in order to make them usable for industrial applications.

- **High adoption cost** When formal methods are to be integrated into an organization, one will usually encounter already established methods and infrastructure. Supporting the translation or migration from standard less-formal software production methods in order to incorporate more formal methods represents a significant practical challenge.

Several industry-proven software product line approaches [1]–[4] share a similar high-level structure, including family engineering and application engineering concerns, as well as concerns about building maintainable repositories of reusable artifacts. HATS uses these proven product line abstractions as the backbone of all intended extensions. Doing so ensures that the newly developed method will benefit from the experience of many practitioners in this area.

Much research has already been carried out to apply individual formal methods into SPLE, including, but not limited to, those documented in [5]–[10]. Some have already demonstrated their applicability in industrial settings [5], while others require additional work in order to become more applicable in industrial setups [6]. These approaches typically focus on enhancing specific steps in SPLE development, without demonstrating a broader ambition of being a whole-process methodology. While these approaches provide the means to focus on particular fundamental aspects of software systems, the approaches themselves do not offer the necessary traceability from formal models to resulting software systems that are end products. These limitations reduce the likelihood of industrial adoption, do not help integration with formal methods into other steps in a SPLE methodology, and tend to produce tools that can be used in isolation only. By contrast, HATS aims to address the complete lifecycle of a software product line, resulting in a holistic approach to the development of adaptable and trustworthy software systems.

B. Goals

The HATS project aims to bring about the following:

- Integration of advanced software tools based on formal models into SPLE development processes;
- High usability in an industrial context by a) designing tool interfaces suitable for SPLE designers, and b) providing scalability of the underlying methods.

C. Approach

The following approach to developing and validating the methodology is taken.

- We adapt existing successful industrial methods. This increases the likelihood of that the HATS methodology will be accepted by the industry.
- We inject formal methods into various steps of the methodology. The idea is to provide a well-rounded collection of formal methods to support each step in order to produce high-quality software. The idea is to provide these formal methods by either tailoring some of the existing ones for SPLE or developing new formalisms to enable better integration support. When augmenting an existing methodological step with formal methods, we will ensure that the step integrates well with the other (adjacent) steps in the method work flow, so that these steps can exchange inputs and outputs, that tools can interoperate, and so on.
- Coupled with the development method is the *Abstract Behavioral Specification* language (referred to hereafter as ABS) — a formal executable language for specifying software product line artifacts. ABS is used to formally describe specification, design and executable artifacts. It does not consider informal requirements or natural-language-based documentation. ABS is based on CREOL [11], a high-level executable modeling language based on asynchronously communicating concurrent objects, which, in particular, supports compositional reasoning [12], [13] and runtime code evolution [14].
- We aim to supply a tool chain supporting the HATS methodology. Formalization of the artifacts and requirements of a system encourages the development of mechanical and automatic procedures for the methodology.
- To validate our results, we apply the methodology to three independent case studies: one academic and two industrial projects [15]. We will focus on validating each individual step and integrating the different steps examining the quality and cost of production of the resulting software artifacts (models, components, product).

The HATS methodology bases its overall work flow on existing successful methodologies to increase the likelihood of industrial acceptance compared to a methodology built from scratch. In addition, a complete industrial evaluation of a product family methodology requires the application of the method over a family of similar products of significant size and over a long period of time. Therefore, the validation effort within HATS is limited to individual steps and to proving coherence among them.

D. Outline

Section II introduces the current HATS methodology with its work flow; Section III presents the family engineering work flow, and Section IV presents the application engineering work flow. A more detailed description of the HATS methodology is described in a companion technical report [16]. The presentation of each methodological step is structured as follows:

- We motivate the formal methods used in each step by identifying deficiencies in the state of the art and/or particular needs for formalization.
- We explain whether a new formal method is to be developed or whether existing methods and techniques can be adapted.

- We discuss how to integrate the formal methods from different steps into the HATS methodology work flow.

We report concrete results depending on the work progress within the HATS project regarding the particular methodological step. If the project has not yet addressed a particular step, we explicitly state our vision on how to address it.

Section V presents how the HATS methodology governs the evolution of software products. Section VI presents results already achieved by the HATS project. These results are structured as examples of the application of formal methods in several steps of the HATS methodology. We discuss the kind of formalisms used, the adaptations made for SPLE, and the integration with adjacent steps in the HATS methodology work flow. Section VII summarizes the paper.

II. HATS Development methodology

This section describes the main work flow of the HATS methodology and highlights where formal methods are applied. Figure 1 shows the product line lifecycle in the HATS development methodology. The HATS methodology adopts the traditional SPLE approach by splitting the overall development lifecycle into family engineering (FE) and application engineering (AE). FE and AE are described in detail in Sections III and IV, respectively. The HATS development methodology is derived by extending and adapting existing industrial software product line engineering (SPLE) methods [1]–[4] in the following ways.

- The HATS methodology emphasizes 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 activities. The former adds formal verification activities as early in the process as possible, and the latter allows for testing and verification of the ultimately generated product.
- Coupled with the development method is a formal executable language, ABS, for specifying product line artifacts. ABS supports formal specification from the start of the Product Line Requirement Analysis phase to the end of the Generic Component Design phase. ABS aims to provide capabilities for modeling SPLE variability, as well as reasoning about concurrency, security, resource guarantees, and evolvability. ABS consists of a core language and a number of extensions, and is part of ongoing work in the HATS project [16]. Section VI illustrates how the core language extended with μTVL, a trimmed-down version of TVL [17] and delta modelling [18]–[20] can be used to model variability.
- Introducing formal approaches to artifact development in the product line enables much of the testing and verification efforts to be moved to the family engineering process. Methodologically, we extend the Generic Component Validation phase to include both testing and verification activities. Furthermore, we envisage that Generic Component Validation may be carried out in parallel with Generic Component Design and Generic Component Realization.
- The HATS methodology aims to support continuous development of the product line itself as well as of individual family members. This is achieved by developing theories and techniques for handling continuous evolution of software systems. The Evolution Process (EP) of the HATS methodology supports this, see Section V.

Scope: 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 scientific perspective. Therefore, we focus our contributions by scoping our work. Specifically, informal processes such as those involving customers in industrial SPLE are not considered. As a consequence, we do not make a formal contribution to the Product Line Planning and Scoping, Application Engineering Planning, and System Delivery phases. On the other hand, phases such as the Reference Architecture Design phase, provide a large opportunity for formal development. In order to leverage the quality and impact of the contributions, the HATS methodology focuses on particular technical aspects of the relevant phases that will have the highest scientific impact, but are also most amenable to the development of tool support and the integration into a development framework.

III. Family Engineering

The family engineering process (FE), as depicted in the lower half of Figure 1, identifies commonalities and variabilities of the product line and builds reusable artifacts for the product line artifact base.

A. Product Line Requirements Analysis

In the Product Line Requirements Analysis phase, we analyze variability in detail. The requirements of the product line are defined during the Product Line Planning and Scoping phase. In particular, μTVL is used to specify feature models describing common and variable features and their constraints. Other text-based feature modeling languages exist, such as [21]. The main purpose of μTVL is to add feature description capabilities to ABS. In addition, μTVL has formal semantics, which is based on existing semantics [22]. It is coupled with notions of abstraction, refinement, and views for specifying and comparing features. Using this abstract model, one may resolve ambiguities within the informal requirements of the product line as well as reason about the compatibility and reconciliation of feature model views. Models developed in this phase could then be used in later phases to guide design and validation, while compatibility and reconciliation are important during the Product Model Instantiation and Validation phase in AE. Besides variability, there are informal product line requirement artifacts, which may be documented in various ways, for example as use cases, work flows descriptions, etc.

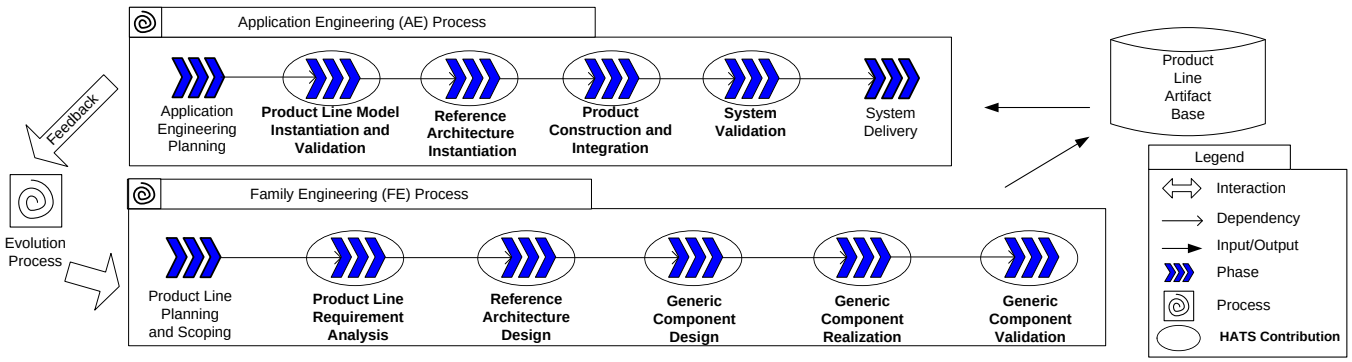


Figure 1. Product Line Lifecycle in the HATS Development Methodology

B. Reference Architecture Design

In the Reference Architecture Design phase, a common *reference architecture* is defined for all product line members. The reference architecture is documented by means of different architectural views containing information about component interfaces, their interactions, overall system behavior, and system variability. Here we aim to provide theories, techniques, and tools to support the description of variability across components, as well as component functionality, system-level invariants, and cross-cutting, non-functional properties such as resource guarantees. For the sake of brevity, we focus on variability description and resource guarantees; see our technical report [16] for more details.

1) Variability Distribution: It is important to ensure that the components cover all variation points and do not invalidate any variation constraints defined during the Product Line Requirement Analysis phase. Specifically, during Product Line Requirement Analysis, feature models defined using μTVL are provided. Using these models, appropriate hooks for incorporating variation points into the core architecture at the component level may be provided, thereby assisting the distribution of variability. Coupled with μTVL is *delta modeling* [18]–[20]. Delta modeling connects the features in μTVL to design artifacts; in this context, to the core architecture and its constituent components. Delta models define changes to the core architecture in order to implement the various 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. Other formal approaches for representing feature-based variability exist, such as AHEAD [23]; see report [24] for a comparison. As with μTVL, delta modeling is designed to be integrated into ABS. In Section VI, we illustrate the application of μTVL and delta modeling as an integrated method.

2) Resource Guarantees: The reference architecture also specifies resource constraints, such as execution costs, security requirements, etc. For the execution costs, we will develop cost models for specifying their upper and lower bounds. We aim to specify this information at the level of variation points of the reference architecture. Using such information as constraints, one may verify that a change (evolution) of the reference architecture guarantees that members of the product line only use the available resources. While this is still work in progress, we aim to leverage existing results on static resource analysis, namely the COSTA system [25]. Currently, COSTA allows obtaining *safe symbolic upper bounds* on the resource usage of JAVA programs. In Section VI, we illustrate the application of COSTA and discuss the challenges we face when integrating it with the ABS language.

C. Generic Component Design

In this step, we make a detailed internal design of each component based on the artifacts of the reference architecture. Specifically, an executable model of each component is defined in this phase using ABS. Each component's model specifies and integrates both component-specific and cross-cutting variabilities. 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. As a result, every generic component is associated with a variability model, that is, a combination of the high-level feature models described in μTVL and deltas defined in the Product Line Requirement Analysis and Reference Architecture Design phases.

Furthermore, having a precise reference architecture and design models of generic components helps to validate the correctness of component designs against the reference architecture to ensure consistency at all levels of abstraction. This encourages both incremental and concurrent development of product line artifacts. Two main tasks to be carried out in this phase are feature modeling and feature integration.

1) Feature Modeling: μTVL-based feature models provided during the Product Line Requirement Analysis and Reference Architecture phases formally capture variation points along with all platform-related and other configuration parameters. In addition to providing documentation, such models ensure that the generic component designer provides support for all variation points. Furthermore, we aim to integrate an abstract

failure model into ABS to capture cross-cutting variability in platform configurations. The failure model allows generic component design models to be instantiated with different platform configurations that support different levels of failure handling.

2) Feature Integration: Given μTVL-based feature models, features are composed at the level of ABS using delta modeling [18]–[20]. In particular, deltas formalize the underlying behavior of individual features as well as their combinations. 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 behavior of unrelated features.

D. Generic Component Realisation

After the generic components have been designed, they are *realized* and 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, which is a combination of high-level μTVL-based feature models and deltas. This provides the necessary link between the implementation of the generic component and the variability it supports. This link enables the application engineer to reuse generic components by resolving variation points and instantiating them as concrete components. Other considerations in this phase include: developing the techniques and the tool support for specifying and debugging generic components, and automatic code generation. In particular, these techniques will be based on symbolic execution. Symbolic execution provides both forward and backward navigation along *all* possible execution paths of the component up to a finite depth. Since symbolic execution does not require concrete start states, it is ideally suited to support the implementation and validation of generic components.

E. Generic Component Validation

After generic components have been realized, they are *validated* to ensure that they conform to their specification before being put into the product line artifact base for reuse. By leveraging the compositionality of ABS, it is possible to carry out the validation of generic components during the reference architecture design and the generic component design phases. In the Generic Component Validation phase the validation process may be partitioned into two, formal testing and formal verification. For reason of space, we provide just a brief overview of the latter. In HATS, techniques based on symbolic execution have been investigated (such as [26]) as a means for achieving scalable formal verification. These techniques will be integrated into ABS to allow verification of behavioral and functional aspects of generic components.

IV. Application Engineering

The application engineering process (AE), as shown at the top of Figure 1, builds products based on the reuse of generic artifacts from the product line artifact base. When artifacts are reused, their variability is resolved. New customer-specific requirements are also taken into account when resolving variability.

A. Product Line Model Instantiation and Validation

During the Product Line Model Instantiation and Validation phase, the external product line variability is resolved. The μTVL-based feature model constructed 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. Similar to TVL, resolving the variability of a μTVL-based feature model is done using constraint satisfaction [17]. Using both constraint solving and the refinement theories that underlie the feature model's semantics, we aim to provide automatic consistency checking for feature selection and system derivation. Some internal variation points may require the knowledge of a system architect, and hence such variation points are resolved without altering the system's external properties. Besides resolving external variability, this phase also includes requirements analysis for customer-specific requirements that cannot be mapped to variation points in the feature model.

B. Reference Architecture Instantiation

Given a precise description of the product requirements, the reference architecture may be instantiated. This means that 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 the product requirements to adapt to customer-specific requirements. This means changing either the product architecture or the reference architecture to include these new requirements. This decision will be made in the context of the evolution process, which is described in Section V. Resolving variability at this level means selecting the correct platform configuration as well as the correct set of deltas defined during the Reference Architecture Design phase. We then prove this selection to be correct by first resolving any conflict over selected deltas [18], and then proving product-specific requirements by reusing and composing proof artifacts constructed during the family engineering process.

C. Product Construction and Integration

During AE, generic components are instantiated and reused according to the product's architecture. After identifying the necessary generic components, they are either adapted to fit the product requirements, or new product-specific components are developed instead. After identifying the correct set of generic

components, we employ delta modeling to mechanically resolve variation points in this variability model. In particular, 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 into an additional precisely targeted delta. Other considerations during this phase include verifying the correctness of the composition of the selected components, and translating the verified composition into executable code.

D. System Validation

The product that was constructed during AE needs to be validated for correctness against the product line requirements and any customer-specific requirements. HATS aims to develop the theories and techniques for conducting proofs for functional correctness of the constructed product. This will be achieved by efficiently reusing and composing proof artifacts constructed at earlier stages of the life-cycle, such as during family engineering. While it is still ongoing work, verification techniques based on symbolic execution have been investigated [26]. We will also consider test case generation to address deployment issues such as scheduling and platform configurations.

V. Evolution Process

The aim of the evolution process is twofold: a) to manage changes to released products and, b) to link AE and FE for better reuse. For the sake of brevity we present only the latter. A reuse problem in AE occurs when the evaluation of potential reuse candidates reveals that adapting each candidate requires more effort than building the desired component from scratch. Since it is not possible for the application engineer alone to decide upon the solution to this problem, an evolution request would be triggered and handled in the evolution process. An evolution request is handled independently and in parallel to AE. The main decision to be resolved is whether it is more efficient improving or recreating a generic artifact than developing the specific artifact only for that application. In the former case, a change request is sent to the FE where the concrete change of the product line is realized. In the latter case, the result is re-injected into the ongoing application engineering process so that the creation of the specific artifacts can start.

The HATS methodology aims to support the evolution process during family engineering. Specifically, we aim to support evolution for the reference architecture and for generic component designs. For the reference architecture, we aim to apply behavioral interfaces to specify component behavior in terms of attribute grammars [27]. Interfaces help specify behavioral constraints of the interacting components, which facilitates well-formed composition and enables safe evolution to be checked statically and dynamically. For generic components we aim to use model mining. Model mining helps to derive partial models of an application from its code base. As a result, given an existing implementation of a component, model mining techniques can be used to formally inspect and revise the corresponding generic component design model. Techniques discussed in this section are still ongoing work in the HATS project.

VI. Example

In this section, we consider two examples of the application of formal methods in several steps of the HATS methodology. In Section VI-A we present an example based on a text editor to illustrate how to specify, implement, and resolve the variability of a SPL using the HATS methodology. In Section VI-B, we look at an existing formal method tool for analyzing program resources—the COSTA system [25]. We illustrate its application with a simple example, highlight its relevance to HATS, and suggest how to integrate it into the HATS methodology.

A. Feature Modeling and Integration

Following the classical SPLE approach, the text editor product line consists of common and variable requirements. For the sake of brevity, we provide the following code fragment of the text editor to illustrate the product line's commonality.

```
class Editor {
  Model model;
  void draw () { ... }
  Font font(int c) { ... }
  void onMouseOver(Coordinate c) { ... } }
```

Specifically, the `Editor` class holds a reference to the underlying model being edited. It has methods for rendering the model on the screen, for accessing the model, for getting access to the default font, and an event handler for when the mouse hovers over some location in the text editor. The interface types of these methods would have been defined during the Reference Architecture Design phase, while the (generic) implementation of these methods would be provided during the Generic Component Design and the Generic Component Realization phases. This resulting class is a product line artifact, stored in the artifact base for reuse during AE.

We model variability of the product line using feature models expressed in μTVL. The following feature model describes the variability of the text editor.

```
root Editor { group [0..3] {
    opt SH {}, opt SPELL {},
    opt TT { int sense; 0 <= sense <= 1000; }}}
```

This model expresses the following additional features: SH, a syntax highlighting module; SPELL, a spell checker; and TT, offering tool-tip functionality. In addition, the TT feature takes an integer parameter reflecting the desired sensitivity level, which is a value between 0 and 1000 milliseconds. This micro-variability is reflected in the attribute `sense` in the feature model. Normally, both high-level commonality and variability of a product line are expressed in μTVL-based feature models during the Product Line Requirement Analysis phase; for reasons of space, our example μTVL only describes variability.

During the Generic Component Design and Generic Component Realisation phases, the variability of the product line is implemented using software deltas. These deltas can be applied to the core in order to modify it by adding, removing, or modifying classes, methods, and fields. Deltas are stored in the artefact base as generic components for reuse during AE. During AE, variability is resolved by first selecting the required set of features during the Product Line Model Instantiation and Validation phase. The corresponding deltas are then applied to the core during the Reference Architecture Instantiation and Product Construction and Integration phases.

In this example, six deltas implement the three features above: SH implements syntax highlighting; SPELL implements spell-checking; TT1 implements tool tips for `sense` > 500; TT2 implements tool tips for `sense` ≤ 500 (using a different algorithm); P1 patches the core so that syntax highlighting and spell-checking work together to highlight spelling errors, and P2 patches the core so that spell-checking and tool tips work together using tool tips to suggest alternative spellings.

For illustration purposes, we provide the definition of TT1 delta. The first line in TT1 expresses the name of the delta and a list of the attributes imported from the feature selection made from the feature model. The second line gives an application condition stating when the delta is applicable, based on the features selection (in this case, TT) and values passed in as the attributes. These attributes can also be used within the body of a delta.

```
delta TT1 (attr int TT.sense)
  when TT and TT.sense > 500 {
  modifies class Editor {
    adds int sensitivity = TT.sense;
    modifies void onMouseOver(Coordinate c)
    { S } // the method body in TT1 } }
```

A delta may modify multiple classes and a class may be modified by more than one delta. Deltas need to be applied in a pre-determined order to ensure that no conflicts arise [18]. Besides adding a new class to a program, ABS supports three operations to support class modifications: the *addition* of a new interface to an existing class; the *redefinition* (or addition) of fields and methods in an existing class; and the *removal* of fields and methods from an existing class. To illustrate this: The effect of applying delta TT1 based on feature selection TT with the attribute `TT.sense` $= 750$ to the class `Editor` yields the following class definition:

```
class Editor {
  Model model; int sensitivity = 750;
  void draw () { ... }
  Font font(int c) { ... }
  void onMouseOver(Coordinate c) { S } }
```

These operations are flexible enough to capture the modification given in terms of deltas when projected down to single classes. Furthermore, these operations have been shown to support the type-safe runtime redefinition of classes in concurrent distributed systems [14]. Consequently, the operations are specific enough to support both static feature selection in software product lines and runtime reconfiguration of variation points in deployed products.

B. Resource Guarantee

Typical resource usage (or cost) measures of a program include execution time, executions steps, memory usage, amount of data transmitted over the network, etc. The COSTA system can obtain *closed-form upper bounds* on resource usages of JAVA bytecode programs (and therefore JAVA), parametric on the notion of *resource* (cost model). Consider the following JAVA implementation of the binary search method:

```
int bi(int[] t, int v, int l, int u) {
  int m;
  while (l <= u) {
    m = (l+u)/2; if (t[m] == v) return m;
    if (t[m] > v) u = m-1; else l = m+1; }
  return -1; }
```

COSTA infers an arithmetic expression that is an upper bound on the number of execution steps when the method `bi` is called. The calculated upper bounds are parametric on the input values and not specific for given concrete input values. COSTA follows the classical approach to static resource analysis and consists of two phases.

In the first step, several static analyses and a cost model are applied to a given program in order to generate a *cost relation system* that represents the program's cost w.r.t. the given cost model.

In the second step, COSTA [28] solves the cost relations and obtains a closed-form upper-bound (i.e, an expression without recursion). For example, for the above cost relation it obtains $bi(t,v,l,u) = 24 * \lceil log_2(nat(u-l)+1) \rceil + 40$ where $nat(a) = max(a,0)$. The full details of this example may be found in [25].

1) Relevance: In the HATS methodology, the COSTA system has several uses: (a) *Verification of resource usage requirements*: Here, resource usage requirements are provided at the level of ABS models and verified either at the level of ABS models or at the level of the generated concrete code (e.g., JAVA), depending on the resource of interest; (b) *tracking the resource usage evolution of a given system* and in the event that an evolution step violates the resource usage requirements, trying to identify the smallest part of the system responsible for this violation; and (c) *directing the feature selection process* towards an optimal (from a cost point of view) feature selection. This is useful when the number of feature combinations is large, and our interest is to select those features that minimize resource consumption.

2) Challenges: Currently, the COSTA system is able to analyze JAVA bytecode programs and has support for several cost models. It can be used in the HATS methodology in one of the following ways: (a) apply it to JAVA programs generated from ABS models. This requires a language for specifying resource constraints at the level of ABS; (b) compile ABS models into the intermediate language used in COSTA, and then apply COSTA directly to the compiled models. This requires

developing a translator from ABS models to this intermediate language; and (c) reuse the technologies developed in COSTA to develop a cost analyzer dedicated to ABS models. In all of these alternatives an important issue is to support concurrency, as COSTA currently lacks support for this feature. Support for concurrency should use the ABS concurrency model. This is essential to making COSTA widely applicable in the context of the HATS methodology.

VII. Conclusion

This paper reports on the current status of the HATS methodology. The HATS methodology is derived from industrial-strength software product line engineering methods by applying formal methods to various phases of the method. We have provided an overview of each phase in the methodology and identified specific places where formal methods are applied. We have presented two applications of formal methods in the HATS methodology. First, we illustrated how variability of a product line is modeled in the HATS methodology. Second, we considered the ongoing challenges of integrating an existing tool for analyzing resource usage in a product line.

The HATS project has yet to validate the intended methodological benefits for large-scale information systems in industrial settings. This evaluation remains future work.

Acknowledgements We would like to thank the anonymous referees for useful suggestions and comments.

References

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

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

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

[4] 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.

[5] Arnaud Hubaux, Andreas Classen, and Patrick Heymans. Formal modelling of feature configuration workflows. In *Proc. of 13th Software Product Line Conference (SPLC 2009)*, 2009.

[6] Alessandro Fantechi and Stefania Gnesi. Formal modeling for product families engineering. In *Proc. of 12th Software Product Line Conference (SPLC 2008)*, 2008.

[7] Alexander Gruler, Martin Leucker, and Kathrin Scheidemann. Calculating and modeling common parts of software product lines. In *Proc. of 12th Software Product Line Conference (SPLC 2008)*, 2008.

[8] Tomoji Kishi and Natsuko Noda. Formal verification and software product lines. *Communications of the ACM*, 49(12):73–77, 2006.

[9] Tomoji Kishi, Natsuko Noda, and Takuya Katayama. Design verification for product line development. In *Proc. of 9th Software Product Line Conference (SPLC 2005)*, 2005.

[10] Mike Mannion. Using first-order logic for product line model validation. In *Proc. of SPLC 2*, LNCS. Springer-Verlag, 2002.

[11] Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. *Software and System Modeling*, 6(1):35–58, March 2007.

[12] Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A Complete Guide to the Future. In *Proc. of ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*. Springer-Verlag, March 2007.

[13] Wolfgang Ahrendt and Maximilian Dylla. A verification system for distributed objects with asynchronous method calls. In *ICFEM'09*, volume 5885 of *Lecture Notes in Computer Science*. Springer-Verlag, 2009.

[14] Einar Broch Johnsen, Marcel Kyas, and Ingrid Chieh Yu. Dynamic classes: Modular asynchronous evolution of distributed concurrent objects. In *Proc. of FM'09*, volume 5850 of *Lecture Notes in Computer Science*. Springer-Verlag, November 2009.

[15] Requirement Elicitation, August 2009. Deliverable 5.1 of project FP7-231620 (HATS), available at http://www.cse.chalmers.se/research/hats/sites/default/files/Deliverable51_rev2.pdf.

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

[17] Quentin Boucher, Andreas Classen, Paul Faber, and Patrick Heymans. Introducing TVL, a text-based feature modelling language. In *Proc. of VaMoS'10*. University of Duisburg-Essen, January 2010.

[18] Dave Clarke, Michiel Helvenstijn, and Ina Schaefer. Abstract Delta Modeling. In *Proceeding of GPCE'10*, October 2010. To appear.

[19] I. Schaefer. Variability Modelling for Model-Driven Development of Software Product Lines. In *Proc. of VaMoS 2010*, 2010.

[20] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *Proc. of SPLC 2010*, September 2010. To appear.

[21] A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–18, 2002.

[22] P. Heymans, P.Y. Schobbens, J.C. Trigaux, Y. Bontemps, R. Matulevicius, and A. Classen. Evaluating formal properties of feature diagram languages. *Software, IET*, 2(3):281–302, 2008.

[23] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Software Eng.*, 30(6), 2004.

[24] First Report on Feature Selection and Integration, March 2010. Deliverable 2.2a of project FP7-231620 (HATS), available at http://www.hats-project.eu.

[25] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Resource usage analysis and its application to resource certification. In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors, *FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *LNCS*. PUB-SV, 2009.

[26] Richard Bubel, Reiner Hähnle, and Ran Ji. Interleaving symbolic execution and partial evaluation. In *Post Conf. Proc. FMCO2009*, LNCS. Springer-Verlag, 2010.

[27] Stijn de Gouw, Jurgen Vinju, and Frank de Boer. Prototyping a tool environment for run-time assertion checking in JML with Communication Histories. In *Proc. of FTfJP 2010*, 2010. To appear.

[28] Elvira Albert, Puri Arenas, Samir Genaim, and German Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 2010. To appear.