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*
[†]*Fraunhofer IESE, Kaiserslautern, Germany*
[*]*Fredhopper B.V., Amsterdam, The Netherlands*
[¶]*University of Oslo, Oslo, Norway*
[§]*Technical University of Madrid, Madrid, Spain*
[‖]*Chalmers University of Technology, Gothenburg, Sweden*

***Abstract*—Trust in software is typically achieved via stabilisation efforts over long periods of use. Adaptation to changing circumstances, however, often requires substantial change of the software. Changing a software system using standard manufacturing processes often results in quality regressions, invalidating trust. Formal methods provide means to guarantee various properties of a software system that increase its trustworthiness. The HATS methodology aims to integrate formal methods that model change through variability in and evolution of software systems, 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 change of the software. In an industrial context that views a software system as a product, change may be viewed in two ways: *anticipated* changes, these may be due to variations to clients requirements and, *unanticipated* changes, these may be due to changes to the market or new technological opportunities. The former defines 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 stabilisation efforts are required to regain the trust of its user.

The "Highly Adaptable and Trustworthy Software using Formal Models" (HATS) project [15] aims to address the issue of producing trustworthy systems in light of the changing requirements. To do this, the HATS project looks 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) [24] 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 modelling of variability. This abstraction fits well with applying formal methods to anticipated change in software systems.
- SPLE addresses the maintenance of the product line artefact base over time. Maintenance includes evolving the artefact base in order to change the possible products of the product line. This facilitates unanticipated changes.

A. Motivation

Motivation for the HATS methodology is fourfold.

- **Incoherent integration of formal methods** Complete methodologies consist of many steps, each addressing different aspects of the product life cycle. Formal methods exist to address most individual aspects, however integrating them together into a development methodology presents a challenge. From 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 make using formal methods more accessible.
- **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 to be usable for industrial applications.
- **High adoption cost** Integrating formal methods into an organisation will usually encounter already established

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).

methods and infrastructure. Supporting the translation or migration from standard less-formal software production methods to incorporate more formal methods represents a significant practical challenge.

Several industrially-proven software product line approaches [9], [23], [24], [27] share similar high-level structure, including family engineering and application engineering concerns, as well as concerns about building maintainable repositories of reusable artefacts. 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.

Several approaches have attempted to put formal methods into SPLE [12], [14], [17], [20]–[22]. Some have already demonstrated applicability in industrial settings [17], while others require additional work to become more applicable in industrial setups [12]. 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 in other steps in a SPLE methodology, and tend to produce tools which can be used in isolation only. By contrast, HATS aims to address the complete life cycle 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 developing and validating the methodology is taken.

- We adapt existing successful industrial methods. This increases the likelihood of acceptance of the HATS methodology by the industry.
- We inject formal methods into various steps of the method. The HATS project aims to provide a well-rounded selection of formal methods to support each step in producing high-quality software. This is achieved by either developing new formalisms or tailoring existing ones to a SPLE methodology. 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 process 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 Behavioural Specification* language (referred to hereafter as ABS) — a formal executable language for specifying software product line artefacts.[1] ABS is based on CREOL [19], a high-level executable modelling language based on asynchronously communicating concurrent objects which, in particular, supports compositional reasoning [2], [10] and runtime code evolution [18].
- We aim to supply a tool chain supporting the HATS methodology. The formalisation of artefacts 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 [**?**]. We will focus on validating each individual step and the integration of different steps examining the quality and cost of production of the resulting software artefacts (models, components, product).

The HATS methodology bases its overall process flow on existing successful methodologies to increase the likelihood of industrial acceptance compared to a methodology built from the ground. 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 the coherence among them.

D. Outline

Section II introduces the current HATS methodology with its general flow; Section III presents the family engineering flow, and Section IV presents the application engineering flow. A more detailed description of the HATS methodology is described in a companion technical report [1]. 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 formalisation.
- 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 in the HATS methodology 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 state explicitly our vision only.

Section V presents how the HATS methodology governs the evolution of software products. Section VI presents results

[1]ABS is used to formally describe specification, design and executable artefacts. It does not consider informal requirements or natural-language based documentation.

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 flow. Section VII summarises the paper.

II. HATS Development methodology

This section describes the main flow of the HATS methodology and highlights where formal methods may be 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).[2] 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 [9], [23], [24], [27] 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 the formal executable language, ABS, for specifying product line artefacts. ABS supports formal specification from as early as Product Line Requirement Analysis to Generic Component Design phases. ABS aims to provide capabilities for modelling SPLE variability, as well as reasoning about concurrency, security, resources guarantees and evolvability. ABS consists of a core language and a number of extensions, and is part of ongoing work in the HATS project [1]. Section VI illustrates how the core language extended with μTVL[3] and delta modelling [8], [25], [26] may be used to model variability.
- Introducing formal approaches to artefact 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 Realisation.
- The HATS methodology aims to support continuous development of the product line as well as 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.

A. 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 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 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 very large opportunity for formal development. In order to leverage the contributions quality and impact, 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) identifies commonalities and variabilities of the product line and builds reusable artefacts for the product line artefact base. The workflow of this process is shown in the lower half of Figure 1.

A. Product Line Requirement Analysis

In the Product Line Requirement Analysis phase, we analyse 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 modelling languages exist [28]. The main purpose of μTVL is to add feature description capabilities to ABS. Underlying μTVL is a formal semantics of features with notions of abstraction, refinement and views, based on existing semantics [16]. Using this abstract model one may resolve ambiguities within the informal requirements of the product line as well as reason about 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.[4]

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 behaviour and

[2]We assume readers are familiar with the concepts of FE and AE, details on current approaches to FE and AE may be found in existing literature.

[3]μTVL is a trimmed down version of TVL [6].

[4]Besides variability, there are informal product line requirement artefacts which may be documented in various ways, for example, use cases, workflows descriptions, etc.

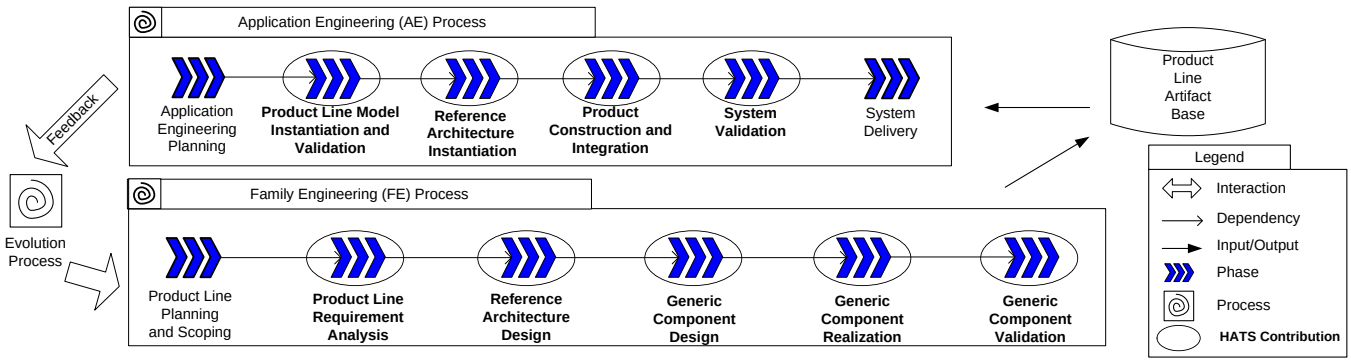

Figure 1. Product Line Lifecycle in the HATS Development Methodology

system's variability. Here we aim to provide theories, techniques and tools to support the description of variability over components, as well as component functionality, system-level invariants and cross-cutting, non-functional properties such as resource guarantees. For brevity, we focus on variability description and resource guarantees; see report [1] for more.

1) Variability Distribution: It is important to ensure that the components cover all variation points and do not invalidate any variation constraints defined during 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 modelling* [8], [25], [26]. Delta modelling connects the features in μTVL to design artefacts, in this context, to the core architecture and its constituent components. Delta models define changes to the core architecture 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 artefacts. Other formal approaches to represent feature-based variability exist, such as AHEAD [5]; see report [13] for a comparison. As with μTVL, delta modelling is designed to be integrated into ABS. In Section VI we illustrate the application of μTVL and delta modelling as an integrated method.

2) Resource Guarantees: The reference architecture also specifies resource constraints, such as execution costs, security requirements, etc. For execution costs, we will develop cost models 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. Using such information as constraints, one may verify that a change (evolution) of the reference architecture guarantees members of the product line run within some given available resources. While this is still work in progress, we aim to leverage existing results on static resource analysis, namely Albert et al.'s COSTA system [4]. Currently COSTA allows obtaining of *safe symbolic upper bounds* on the resource usage of JAVA programs. In Section VI we illustrate the application of COSTA and also discuss the challenges to integrate it with the ABS language.

C. Generic Component Design

In this step, we make a detail internal design of each component based on the artefacts of the reference architecture. Specifically, in this phase an executable model of each component is defined 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 validating 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 artefacts. Two main tasks to be carried out in this phase are feature modelling and feature integration.

1) Feature Modelling: μTVL-based feature models provided the during Product Line Requirement Analysis and Reference Architecture phases formally capture variation points along with all platform-related and other configuration parameters. Along with providing documentation, such models ensure that the generic component designer provides support for all variation points. In addition, 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 configuration 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 modelling [8], [25], [26]. Specifically, deltas formalise the

underlying behaviour 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 modelling helps to resolve conflicts between interdependent features without affecting the behaviour unrelated features.

D. Generic Component Realisation

After designing the generic components, they are *realised* and added to the product line artefact 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 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 navigations along *all* possible execution paths of the component up to a finite depth. Since symbolic execution does not require concrete start states, it is ideal to support the implementation and validation of generic components.

E. Generic Component Validation

After generic components are realised, they are *validated* to ensure that they conform to their specification before being put into the product line artefact base for reuse. By leveraging the compositionally of ABS, it is possible to carry out validation of generic components during the reference architecture design and the generic component design phase. In this phase the validation process may be partitioned into two formal testing and formal verification. For reason of space, we briefly overview the latter. In HATS techniques based on symbolic execution have been investigated (such as [7]) as means to achieve scalable formal verification. These techniques will be integrated into ABS to allow verification of behavioural and functional aspects of generic components.

IV. Application Engineering

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

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 specialising the feature model, making choices and selecting values for attributes. Similar to TVL, resolving variability of a μTVL-based feature model is done using constraint satisfaction [6]. 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.[5]

B. Reference Architecture Instantiation

Given a precise description of the product requirements, the reference architecture may be instantiated. 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 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 [8], and then proving product specific requirements by reusing and composing proof artefacts 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 modelling to mechanically resolve variation points in this variability model. Specifically, required code-changes to specific products are applied to the generic components directly, while very specialised changes, even those affecting only a single product, may be written in 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

[5]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.

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 artefacts 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 [7]. 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 between AE and the FE for better reuse. For brevity we present only the latter. A reuse problem in AE occurs when evaluating 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 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 to improve or recreate a generic artefact than to develop the artefact only specific for that application. In the former case the result is re-injected into the ongoing application engineering process, so that the creation of the specific artefacts can start. In the latter case, a change request is sent to the FE where the concrete impact for changing of the product line can be realised.

The HATS methodology aims to support the evolution process during family engineering. Specifically, we aim to support evolution for the reference architecture and generic component designs. For the reference architecture we aim to apply behavioural interfaces to specify component behaviour in terms of attribute grammars [11]. Interfaces help specify behavioural 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 deriving 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 consider an example based on a text editor to illustrate the following: How to specify, implement and resolve variability of a SPLE using the HATS methodology. In Section VI-B we consider an existing formal method tool for analysing program resources—the COSTA system [4]. We illustrate its application with a simple example and highlight its relevance to HATS and suggest how to integrate it into the HATS methodology.

A. Feature Modelling and Integration

Following the classical SPLE approach, the text editor product line consists of common and variable requirements. For 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 to render the model to the screen, to access the model, to get 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 Realisation phases. This resulting class is a product line artefact, stored in the artefact 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 {
    opt SH {}
    opt SPELL {}
    opt TT { int sense; 0 <= sensitivity <= 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 `sensitivity` 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 reason of space our example μTVL only describes variability.

During Generic Component Design and Generic Component Realisation phases, the variability of the product line are implemented using software deltas. These deltas can be applied to the core 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 the applied to the core during the Reference Architecture Instantiation and Product Construction and Integration phases.

In this example six deltas implement the above three features: SH implements syntax highlighting; SPELL implements spell checking; TT1 implements tool tips for `sensitivity` > 500; TT2 implements tool tips for `sensitivity` ≤ 500 (using a different algorithm); P1 patches the core so that syntax highlighting and spell checking work together, namely, to highlight the spelling errors, and P2 patches core so that spell checking and tool tips work together, namely, to use 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.sensitivity)
  when TT and TT.sensitivity > 500 {
  modifies class Editor {
    adds int sensitivity = TT.sensitivity;
    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 applied in a pre-determined order to ensure that no conflicts arise [8]. 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, the effect of applying delta TT1, based on feature selection TT with attribute `TT.sensitivity` $= 750$, to 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 [18]. Consequently, the operations are specific enough to support both the static feature selection in software product lines and the 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 it consists of two phases.

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

In the second step, COSTA [3] 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 [4].

1) Relevance: In the HATS methodology the COSTA system has several uses: (a) *verification of resource usage requirement*: here resources usage requirements are provided at the level of ABS models and verified either at the level of ABS models or 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 case that an evolution step violates the resource usage requirements, try to identify the smallest part of the system responsible for this violation; and (c) *directing the feature selection process* towards an optimal (from cost point of view) feature selection. This is useful when the number of features combination is large, and our interest is in selecting the features that minimise resource consumption.

2) Challenges: Currently, the COSTA system is able to analyse JAVA bytecode programs, and has support for several cost models. It can be used in 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 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 analyser dedicated to ABS models. In all these alternatives an important issue is to support concurrency, as currently COSTA lacks support for this feature. Support for concurrency should use the ABS concurrency model. This is essential to make COSTA widely applicable in the context of the HATS methodology.

VII. Conclusion

This paper reports the current status of the HATS methodology. The HATS methodology derives 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 modelled in the HATS methodology. Second, we considered the ongoing challenges of integrating an existing tool for analysing 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.

References

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

[2] W. Ahrendt and M. 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.

[3] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 2010. To appear.

[4] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Resource usage analysis and its application to resource certification. In A. Aldini, G. Barthe, and R. Gorrieri, editors, *FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *LNCS*. PUB-SV, 2009.

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

[6] Q. Boucher, A. Classen, P. Faber, and P. Heymans. Introducing TVL, a text-based feature modelling language. In *Proc. of VaMoS'10*. University of Duisburg-Essen, January 2010.

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

[8] D. Clarke, M. Helvenstijn, and I. Schaefer. Abstract delta modeling. Submitted, 2010.

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

[10] F. S. de Boer, D. Clarke, and E. B. Johnsen. A Complete Guide to the Future. In *Proc. of ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*. Springer-Verlag, Mar. 2007.

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

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

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

[14] A. Gruler, M. Leucker, and K. Scheidemann. Calculating and modeling common parts of software product lines. In *Proc. of 12th Software Product Line Conference (SPLC 2008)*.

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

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

[17] A. Hubaux, A. Classen, and P. Heymans. Formal modelling of feature configuration workflows. In *Proc. of 13th Software Product Line Conference (SPLC 2009)*, 2009.

[18] E. B. Johnsen, M. Kyas, and I. C. 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, Nov. 2009.

[19] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and System Modeling*, 6(1):35–58, Mar. 2007.

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

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

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

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

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

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

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

[27] 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.

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