HATS: Highly Adaptable and Trustworthy Software using Formal Models

Reiner Hähnle*, Mads Dam, Arnd Poetzsch-Heffter
Ina Schaefer, Jan Schäfer
http://www.hats-project.eu

1 Problem Description

Many software development projects make strong demands on the *adaptability* and *trustworthiness* of the developed software, while the contradictory goal of ever faster time-to-market is a constant drum beat. *Adaptability* includes two aspects: anticipated *variability* as well as *evolvability*, i.e., unanticipated change. The first is addressed, to a certain extent, in development methods such as software product line engineering. However, increasing product complexity, e.g., coming from a large number of possible product features or different deployment options, is starting to impose serious limitations. Evolvability over time is an even more difficult problem that is far from any satisfying solution, in particular in the context of system diversity.

Current development practices do not make it possible to produce highly adaptable *and* trustworthy software in a large-scale and cost-efficient manner. Adaptability and trustworthiness are not easily reconciled: unanticipated change, in particular, requires freedom to add and replace components, subsystems, communication media, and functionality with as few constraints regarding behavioral preservation as possible. Trustworthiness, on the other hand, requires that behavior is carefully constrained, preferably through rigorous models and property specifications since informal or semi-formal notations lack the means to describe precisely the *behavioral* aspects of software systems: concurrency, modularity, integrity, security, resource consumption, etc.

Existing notations for system specification at the modeling level, such as architectural languages, visual design languages, or feature description languages, are mainly *structural* and lack adequate means to specify detailed behavior including datatypes, compositionality, or concurrency. But without a formal notation for the behavior of distributed, component-based systems, it is impossible to automate behavioral verification, enforcement of security, trusted code generation, resource analysis, test case generation, specification mining, etc.

At the same time, formal specification and reasoning about executable programs on the implementation level is by now well understood even for com-

*Corresponding author; contact address: Chalmers University of Technology, Department of Computer Science and Engineering, 41 296 Gothenburg, Sweden, reiner@chalmers.se

mercial languages such as JAVA, C, or C#, and reasonably well supported by tools. The size and complexity of these languages, however, makes specification and verification extremely expensive. In addition, re-use of specification and verification efforts is very hard to realize. In conclusion, there is a gap between highly abstract modelling formalisms and implementation-level tools.

The HATS project develops a formal method for the design, analysis, and implementation of highly *adaptable* software systems that are at the same time characterized by a high demand on *trustworthiness*. The core of the method is an object-oriented, executable modeling language for adaptable, concurrent software components: the *Abstract Behavioral Specification* (ABS) language. Its design goal is to permit formal specification of concurrent, component-based systems on a level that abstracts away from implementation details, but retains essential behavioral properties, thus, closing the mentioned gap, see Fig. 1. In Sect. 3 we provide more details.

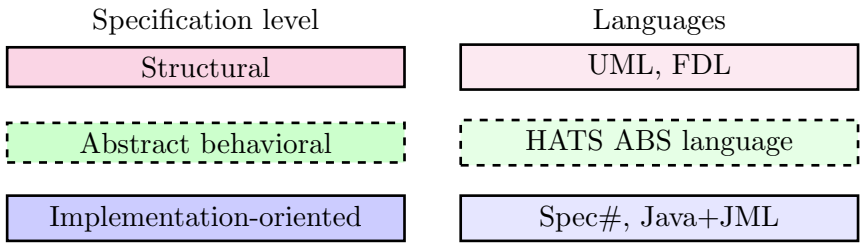

Figure 1: Positioning of the HATS ABS language.

2 Target Audience

The target audience comprises anyone with a professional interest in model-centric development of large-scale distributed, object-oriented systems. Both, academic and industrial participants, are equally in the focus of the presentation which will be accessible to a wide audience:

1. *Academic researchers* working in concurrent programming, modeling languages, formal specification/verification, and software evolution.

2. *Industrial researchers and practitioners* dealing with highly configurable software that must be adapted to varying scenarios, specifically, people who are working in software product line engineering and people from industries where high trustworthiness of software is an issue. The ABS language and tool set will also be interesting for developers and users of tools for model-centric development.

3. *Academic instructors* who wish to teach a modern, OO concurrent programming language that is more abstract and less complex than, e.g., Java or C++. ABS has a uniform formal semantics and comes with a compiler, code generators, simulators, and a debugger, which are all usable either as standalone tools or as Eclipse plugins.

3 Details of the Solution

A characteristic feature of the HATS project is that it *integrates* recent advances from various research communities. The HATS consortium brings together leading research groups from programming languages, distributed systems, formal verification, and software modeling. The HATS method consists of the following three ingredients:

1. The object-oriented, executable modeling language ABS for adaptable, concurrent software components. It permits formal specification of concurrent, component-based systems on a level that abstracts away from implementation details, but captures essential behavioral and data aspects such as the concurrency and object model, the component structure, execution histories, or algebraic datatypes. The ABS language has a formal operational semantics which is the basis for unambiguous usage and rigorous analysis methods.

2. A tool suite for analysis and development of ABS models. On the one hand, this includes *analytic* methods, such as functional verification, behavioral verification, resource analysis, feature consistency, runtime assertion checking, type checking, test case generation, or visualization. On the other hand, HATS also develops *generative* methods including code generation, model mining, or monitor inlining. Methods from both, the *formal* and the *informal* end of the spectrum of development tools, are considered.

   One decisive aspect of the HATS project is to develop methods and tools *hand in hand* with the ABS language to ensure their *feasibility* and *scalability*.

3. A methodological and technological framework that integrates the HATS tool architecture and the ABS language.

As a main challenge in the development of the ABS language and the HATS tool suite, we identified (in addition to the technical difficulties) the need to make the project results *relevant* for industrial practice. Therefore, to keep the project firmly grounded, we orient the design of the ABS language and the HATS methodology along an empirically highly successful *informal* software development paradigm. In software product line engineering (SPLE), *Family Engineering*, including feature modeling and development of reusable artifacts, is separated from *Application Engineering*, where code is derived via artifact selection, instantiation and composition.

In HATS we turn software product line-based development into a rigorous approach based on formal specification.

Constructing a software product line requires architecting both the *commonality*, that is, features, properties, and resources that are common to all products a family, as well as the *adaptability*, that is, the varying aspects across the software family, in order to exploit them during the customization and system derivation process. As explained above, adaptability encompasses both

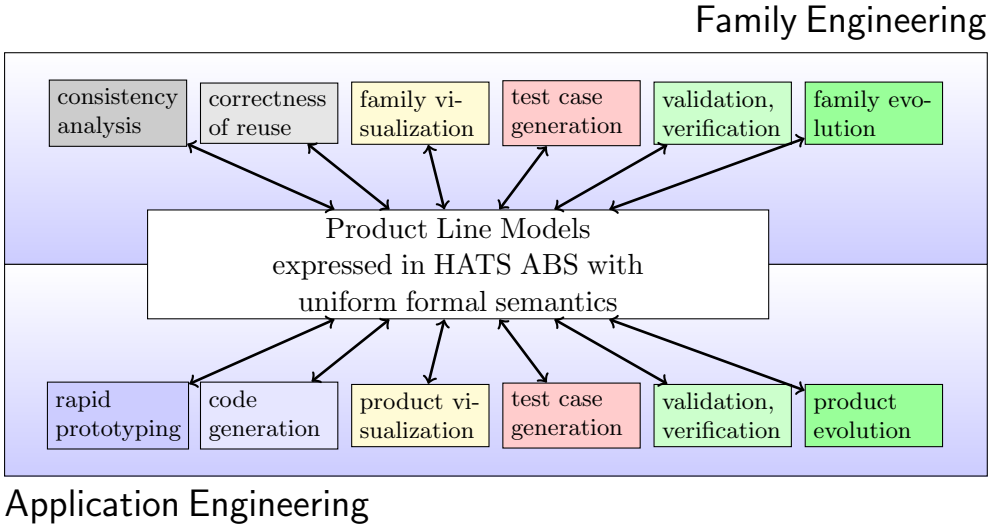

Figure 2: HATS ABS as an integrative technology for Software Product Lines.

anticipated differences among products in the family (*variability*), as well as the different products which appear over time due to evolving requirements (*evolvability*).

Ultimately, a model-centric development process for software product lines based on a uniform formal framework such as HATS ABS can evolve into a *single-source technology* for engineering software product lines, see Figure 2.[1] The integration of different modeling formalisms on the basis of a common formal semantics allows analyzing the *consistency* of different modeling concerns. Language concepts for artifact *reuse* provide a formal semantics for reuse such that consistency and correctness can be ensured by formal analysis techniques. Executable models of the product line as well as of individual products enable simulation and *visualization* techniques during all phases of family and application engineering. This helps to discover and correct defects early and permits *rapid prototyping* of products for communication with stakeholders.

Formal models allow automatic generation of *test cases* on the family level as well as on the product level. This is essential for reusability and maintainability. On the product level, test cases can be reused between different products that share common components. It is even possible to perform formal *verification* of critical system requirements already at early development stages. Evidence for the certifiability of derived products can be immediately provided. The modular and hierarchic structure of the product line model allows efficient validation and verification based on incremental and compositional reasoning. *Code generation* from formal models means to decrease time-to-market without sacrificing quality. Code generators can be verified and certified. The model-centric development approach supports product line maintenance and *evolution*.

[1] A more detailed exposition of this idea is in: I. Schaefer and R. Hähnle: Formal Methods in Software Product Line Engineering, *IEEE Computer*, 44(2), 82–85, Feb. 2011.

4 Presentation Format

The presentation is in tutorial style and self-contained. We refrain from excessive usage of technical and formal jargon. Anyone with basic knowledge of concurrent and OO programming will be able to follow. We divided the presentation into two equal blocks of 90 minutes. The first of these gives a general overview of the HATS project and the ABS language while the second goes into selected specific topics including architectural components, verification, and security.

Part I
General Introduction into the Abstract Behavioral Specification Language

Introduction to ABS, 60min This is a tutorial on the ABS language. We will use examples, partly from industrial case studies, to illustrate the main language concepts: abstract data types, object model, concurrency model, component model, feature modeling, contract-based behavioral assertion language.

Demo of ABS tools, 30min The compiler, code generation, simulation, editing support, automatic resource analysis, and graphical rendering of dynamic execution are presented. The tools are integrated into Eclipse. Participants will be invited to try part of the tools hands-on on their own computers (Eclipse version 3.6 and high-speed internet access are required). The demos and the presentation of ABS concepts (see previous paragraph) will be interleaved.

Part II
Components, Verification, Security

A Component Model for the ABS Language, 30min The component model provides conceptual support for ports, interfaces, and dynamically reconfigurable connections between objects and components. A calculus provides the formal foundation for reasoning about the evolution of ABS components.

Compositional Verification of Product Lines, 30min For large and evolving product lines, it is in general infeasible to verify each product variant in isolation. We present compositional verification techniques for product lines that allow reusing verification effort between different anticipated product variants and in case of product line evolution.

Evolution and Monitoring, 30min We present techniques to maintain consistency and security during systems evolution. We focus on monitor inlining which has been analyzed in depth in the HATS project. We show how monitor inlining can be used to rewrite a Java bytecode application in order to enforce a given security policy. Practical examples are provided.