FASE System Description

The KeY System: Integrating Object-Oriented Design and Formal Methods*

Wolfgang Ahrendt[2], Thomas Baar[1], Bernhard Beckert[1], Martin Giese[1], Reiner Hähnle[2], Wolfram Menzel[1], Wojciech Mostowski[2], and Peter H. Schmitt[1]

[1] Universität Karlsruhe
Inst. f. Logik, Komplexität und Dedukt.-Syst.
D-76128 Karlsruhe, Germany
[2] Chalmers University of Technology
Department of Computing Science
S-41296 Gothenburg, Sweden

Abstract. This paper gives a brief description of the KeY system, a tool written as part of the ongoing KeY project[1], which is aimed at bridging the gap between (a) OO software engineering methods and tools and (b) deductive verification. The KeY system consists of a commercial CASE tool enhanced with functionality for formal specification and deductive verification.

1 Introduction

The goal of the ongoing KeY project is to make the application of *formal methods* possible and effective in a *real-world* software development setting. The incentive for this project is the fact that formal methods for software development – i.e. formal software specification and verification – are hardly used in practical, industrial software development [5]. As the analysis in [1] shows, the reason is not a lack of maturity or capacity of existing methods. Our work in the KeY project is based on the assumption that the primary reasons are as follows:

- The application of formal methods is not integrated into the iterative and incremental software development processes employed in real-world projects.
- The tools for formal software specification and verification are not integrated into the CASE tools used to support these processes. Indeed, the target language of verification tools, in which the programs to be verified have to be written, is hardly ever a 'real' programming language used in industrial software development.

* The KeY project is supported by the Deutsche Forschungsgemeinschaft (grant no. Ha 2617/2-1).
[1] URL: `http://i12www.ira.uka.de/~key/`

– Users of verification tools are expected to know syntax and semantics of one or more complex formal languages. Typically, at least a tactical programming language and a logical language are involved. Even worse, to make serious use of many tools, intimate knowledge of employed logic calculi and proof search strategies is necessary.

Accordingly, a main part of the KeY project is the design and implementation of a software development tool, the KeY system, that strives to improve upon previous tools for formal software development in these respects.

In the principal use case of the KeY system there are actors who want to implement a software system that complies with given requirements and formally verify its correctness. In this scenario, the KeY system is responsible for adding formal detail to the analysis model, for creating conditions that ensure the correctness of refinement steps (called proof obligations), for finding proofs showing that these conditions are satisfied by the model, and for generating counter examples if they are not. Special features of KeY are:

– We concentrate on object-oriented analysis and design methods (OOAD) – because of their key role in today's software development practice –, and on Java as the target language. In particular, we use the Unified Modeling Language (UML) [7] for visual modeling of designs and specifications and the Object Constraint Language (OCL) for adding further restrictions. This choice is supported by the fact, that the UML (which contains OCL) is not only an OMG standard, but has been adopted by all major OOAD software vendors and is featured in recent OOAD textbooks.
– We use a commercial CASE tool as starting point and enhance it by additional functionality for formal specification and verification. The tool of our choice is TogetherCC (Together Control Center).[2]
– Formal verification is based on an axiomatic semantics of Java. More precisely, we confine ourselves to the subset of Java known as Java Card.
– Through direct contacts with software companies we check the soundness of our approach for real world applications.

2 The KeY System

The overall structure of the KeY system is illustrated in Fig. 2. We extend the UML/Java-based CASE tool TogetherCC by adding features to support the annotation of UML diagrams with OCL constraints. A Verification Component is responsible for generating formulae in dynamic logic [3] that express properties of the specification or the relation between specification and implementation. Finally, the Deduction Component can be used to prove or disprove these formulae.

[2] `http://www.togethersoft.com/`

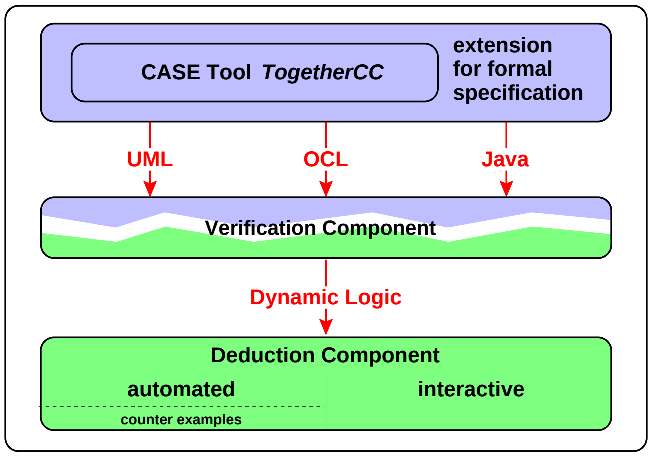

Fig. 1. Structure of the KeY system

2.1 Specification Services

Support for authoring formal specifications is an important part of the KeY project, not only because it is the basis for any ensuing verification, but also because many problems in software development can be avoided with a precise specification, even without doing a formal verification. The KeY system supports the annotation of UML diagrams by constraints written in the OCL language. These can be used to express pre- and postconditions of methods as well as invariants of classes. The specification of OCL constraints is fully integrated into the TOGETHERCC user interface.

As certain types of constraints tend to recur in many applications, we implemented a mechanism, based on TOGETHERCC's pattern instantiation, to automatically generate OCL constraints for common situations. For instance, an OCL expression stating that a method adds its argument to a set of associated objects may be generated automatically. Another example would be a non-cyclicity constraint added to a composite pattern when it is instantiated.

On demand, the system gathers OCL constraints, parses them and performs type checking with respect to the UML diagram. It is typical of the KeY approach that syntactic correctness of constraints is not necessarily enforced, permitting the user to start with semi-formal specifications and refining them when needed.

Future development will include scanning OCL constraints for certain common mistakes, implausibilities and inconsistencies [2]. We are also planning to support an authoring tool for formal and informal specification in OCL and natural language [6].

2.2 Deduction Services

After OCL constraints have been checked syntactically, the user can trigger the generation of proof obligations. These either capture relationships between the

OCL constraints of the diagram, e.g. that the invariant of some class is implied by the invariant of one of its subclasses, or they assert relationships between the OCL constraints and the implementation, e.g. that a method makes its post-condition true. Again, note that the generation of proof obligations is only done on demand to let the user benefit from the specification services of the system even when verification is not desired.

The Verification Component generates proof conditions by transforming the information from the UML diagrams, the OCL constraints and the implementation provided by the user into formulae of JavaDL, a dynamic logic for Java [3]. Details of the transformation are covered in [4].

The JavaDL formulae are passed to the Deduction Component to permit the user to show their validity. The Deduction Component is an integrated automated and interactive theorem prover for JavaDL. It features a graphical user interface and is tailored to make the discharging of proof obligations generated by the Verification Component as intuitive as possible. The proof rules follow the principle of symbolic execution, so the structure of the proofs follows the structure of the involved programs.

As verification is intended to be a tool to discover errors in a specification or implementation, it will often be the case that the generated proof obligations are not provable. We are currently integrating a component for counter-example search that will make it easier to identify errors from failed proof attempts.

We are also planning to include a sophisticated proof management system in the Verification Component that allows users to keep track of which aspects of their development have been formalized or verified to which extent.

References

[1] W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, and P. H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. In M. Ojeda-Aciego, I. P. de Guzmán, G. Brewka, and L. M. Pereira, editors, *Proc. 8th European Workshop on Logics in AI (JELIA), Malaga, Spain*, volume 1919 of *LNCS*, pages 21–36. Springer-Verlag, Oct. 2000.

[2] T. Baar. Experiences with the UML/OCL-approach to precise software modeling: A report from practice. In *Proc. Net.ObjectDays, Erfurt, Germany*, 2000. `http://i12www.ira.uka.de/~key/doc/2000/baar00.pdf.gz`.

[3] B. Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041, pages 6–24. Springer-Verlag, 2001.

[4] B. Beckert, U. Keller, and P. H. Schmitt. Translating the object constraint language into first-order predicate logic. Submitted to FASE 2002, available from `http://i12www.ira.uka.de/~projekt/publicat.htm`.

[5] D. L. Dill and J. Rushby. Acceptance of formal methods: Lessons from hardware design. *IEEE Computer*, 29(4):23–24, Apr. 1996.

[6] R. Hähnle and A. Ranta. Connecting OCL with the rest of the world. In J. Whittle, editor, *Workshop on Transformations in UML at ETAPS, Genova, Italy*, Apr. 2001.

[7] Object Modeling Group. *Unified Modelling Language Specification, v1.4*, Sept. 2001.