The KeY Approach: Integrating Object-oriented Design and Formal Verification[*]

Wolfgang Ahrendt[†] Thomas Baar[†] Bernhard Beckert[†] Martin Giese[†]
Elmar Habermalz[†] Reiner Hähnle[‡] Wolfram Menzel[†] Peter H. Schmitt[†]

[†] Universität Karlsruhe
Inst. f. Logik, Komplexität und Dedukt.-Syst.
D-76128 Karlsruhe, Germany

[‡] Chalmers University of Technology
Department of Computing Science
S-41296 Gothenburg, Sweden

i12www.ira.uka.de/~key

Abstract

This paper reports on the ongoing KeY project aimed at bridging the gap between (a) object-oriented software engineering methods and tools and (b) deductive verification. A distinctive feature of our approach is the use of a commercial CASE tool enhanced with functionality for formal specification and deductive verification.

1 Introduction

1.1 Analysis of the Current Situation

While formal methods are by now well established in hardware and system design, usage of formal methods in software development is still (and in spite of exceptions [8], [9]) more or less confined to academic research. This is true though case studies clearly demonstrate that computer-aided specification and verification of realistic software is feasible [14]. The real problem lies in the excessive demand imposed by current tools on the skills of prospective users:

1. Tools for formal software specification and verification are not integrated into industrial software engineering processes.

2. User interfaces of verification tools are not ergonomic: they are complex, idiosyncratic, and are often without graphical support.

3. 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. And even worse, to make serious use of many tools, intimate knowledge of employed logic calculi and proof search strategies is necessary.

Successful specification and verification of larger projects, therefore, is done separately from software development by academic specialists with several years of training in formal methods, in many cases by the tool developers themselves. It is unlikely that formal software specification and verification will become a routine task in industry under these circumstances.

The future challenge for formal methods is to make their considerable potential feasible to use in an industrial environment. This leads to the requirements:

1. Tools for formal software specification and verification must be integrated into industrial software engineering procedures.

2. User interfaces of these tools must comply with state-of-the-art software engineering tools.

3. The necessary amount of training in formal methods must be minimized. Moreover, techniques involving formal software specification and verification must be teachable in a structured manner. They should be integrated in courses on software engineering topics.

To be sure, the thought that full formal software verification might be possible without any background in formal methods is utopian. An industrial verification tool should, however, allow for *gradual* verification so that software engineers at any (including low) experience level with formal methods may benefit. In addition, an integrated tool with well-defined interfaces facilitates "outsourcing" those parts of the modeling process that require special skills.

Another important motivation to integrate design, development, and verification of software is provided by modern software development methodologies which are *iterative* and *incremental*. *Post mortem* verification would enforce the antiquated waterfall model. Even worse, in a linear model the extra effort needed for verification cannot be parallelized and thus compensated by greater work force. Therefore, delivery time increases considerably and would make formally verified software decisively less competitive.

But not only must the extra time for formal software development be within reasonable bounds, the cost of for-

[*] The KeY project is supported by the Deutsche Forschungsgemeinschaft (grant no. Ha 2617/2-1).

mal specification and verification in an industrial context requires accountability:

4. It must be possible to give realistic estimations of the cost of each step in formal software specification and verification depending on the type of software and the degree of formalization.

This implies immediately that the mere existence of tools for formal software specification and verification is not sufficient, rather, formal specification and verification have to be fully integrated into the software development process.

1.2 The KeY Project

Since November 1998 the authors work on a project addressing the goals outlined in the previous section; we call it the KeY project (read "key").

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. The 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) [20] 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 since version 1.3) is not only an OMG standard, but has been adopted by all major OOAD software vendors and is featured in recent OOAD textbooks [18].
- We use a commercial CASE tool as starting point and enhance it by additional functionality for formal specification and verification. The current tool of our choice is TogetherSoft LLC's TOGETHERJ.
- Formal verification is based on an axiomatic semantics of the *real* programming language JAVA CARD [23] (soon to be replaced by Java 2 Micro Edition, J2ME).
- As a case study to evaluate the usability of our approach we develop a scenario using smart cards with JAVA CARD as programming language [12, 13]. JAVA smart cards make an extremely suitable target for a case study:
  - As an object-oriented language, JAVA CARD is well suited for OOAD;
  - JAVA CARD lacks some crucial complications of the full JAVA language (no threads, fewer data types, no graphical user interfaces);
  - JAVA CARD applications are small (JAVA smart cards currently offer 16K memory for code);
  - at the same time, JAVA CARD applications are embedded into larger program systems or business processes which should be modeled (though not necessarily formally verified) as well;
  - JAVA CARD applications are often security-critical, thus giving incentive to apply formal methods;
  - the high number (usually millions) of deployed smart cards constitutes a new motivation for formal verification, because, in contrast to software run on standard computers, arbitrary updates are not feasible.[1]
- Through direct contacts with software companies we check the soundness of our approach for real world applications (some of the experiences from these contacts are reported in [3]).

The KeY system consists of three main components:

- The *modeling component*: this component is based on the CASE tool and is responsible for all user interactions (except interactive deduction). It is used to generate and refine models, and to store and process them. The extensions for precise modeling contains, e.g., editor and parser for the OCL. Additional functionality for the verification process is provided, e.g., for writing proof obligations.
- The *verification manager*: the link between the modeling component and the deduction component. It generates proof obligations expressed in formal logic from the refinement relations in the model. It stores and processes partial and completed proofs; and it is responsible for correctness management (to make sure, e.g., that there are no cyclic dependencies in proofs).
- The *deduction component*. It is used to actually construct proofs—or counter examples—for proof obligations generated by the verification manager. It is based on an interactive verification system combined with powerful automated deduction techniques that increase the degree of automation; it also contains a part for automatically generating counter examples from failed proof attempts. The interactive and automated techniques and those for finding counter examples are fully integrated and operate on the same data structures.

Although consisting of different components, the KeY system is going to be fully integrated with a uniform user interface.

A first KeY system prototype has been implemented, integrating the CASE tool TOGETHERJ and a deductive component (it has only limited capabilities and lacks the verification manager component). Work on the full KeY system is under progress.

[1]While JAVA CARD applets on smart cards can be updated in principle, for security reasons this does not extend to those applets that verify and load updates.

2 Designing a System with KeY

2.1 The Modeling Process

Software development is generally divided into four activities: analysis, design, implementation, and test. The KeY approach embraces verification as a fifth category. The way in which the development activities are arranged in a sequential order over time is called software development *process*. It consists of different phases. The end of each phase is defined by certain criteria the actual model should meet (milestones).

In some older process models like the waterfall model or Boehm's spiral model no difference is made between the main activities—analysis, design, implementation, test—and the process phases. More recent process models distinguish between phases and activities very carefully; for example, the Rational Unified Process [15] uses the phases inception, elaboration, construction, and transition along with the above activities.

The KeY system does neither support nor require the usage of a *particular* process. However, it is taken into account that most modern processes have two principles in common. They are *iterative* and *incremental*. The design of an iteration is often regarded as the refinement of the design developed in the previous iteration. This has an influence on the way in which the KeY system treats UML models and additional verification tasks (see Section 2.3). The verification activities are spread across all phases in software development. They are often carried out after test activities.

2.2 Specification with the UML and the OCL

The diagrams of the Unified Modeling Language provide, in principle, an easy and concise way to formulate various aspects of a specification, however [25, foreword]: "[...] there are many subtleties and nuances of meaning diagrams cannot convey by themselves." This was a main source of motivation for the development of the Object Constraint Language (OCL), part of the UML since version 1.3 [20]. Constraints written in this language are understood in the context of a UML model, they never stand by themselves. The OCL allows to attach preconditions, postconditions, invariants, and guards to specific elements of a UML model.

When designing a system with KeY, one develops a UML model that is enriched by OCL constraints to make it more precise. This is done using the CASE tool integrated into the KeY system. To assist the user, the KeY system provides menu and dialog driven input possibility. Certain standard tasks, for example, generation of formal specifications of inductive data structures (including the common ones such as lists, stacks, trees) in the UML and the OCL can be done in a fully automated way, while the user simply supplies names of constructors and selectors. Even if formal specifications cannot fully be composed in such a schematic way, considerable parts usually can.

In addition, we have developed a method supporting the extension of a UML model by OCL constraints that is based on enriched design patterns. In the KeY system we will provide common patterns that come complete with predefined constraint schemata. These schemata are formulated in a language that is a slight extension of OCL. They are flexible and allow the user to easily generate well-adapted constraints for the different instances of a pattern. The user needs not write formal specifications from scratch, but only to adapt and complete them. A detailed description of this technique and of experiences with its application in practice is given in [4].

2.3 The KeY Module Concept

The KeY system supports modularization of the model in a particular way. Those parts of a model that correspond to a certain component of the modeled system are grouped together and form a *module*. Modules are a different structuring concept than iterations and serve a different purpose. A module contains all the model components (diagrams, code etc.) that refer to a certain system component. A module is not restricted to a single level of refinement.

There are three main reasons behind the module concept of the KeY system:

Structuring: Models of large systems can be structured, which makes them easier to handle.

Information hiding: Parts of a module that are not relevant for other modules are hidden. This makes it easier to change modules and correct them when errors are found, and to re-use them for different purposes.

Verification of single modules: Different modules can be verified separately, which allows to structure large verification problems. If the size of modules is limited, the complexity of verifying a system grows linearly in the number of its modules and thus in the size of the system. This is indispensable for the scalability of the KeY approach.

In the KeY approach, a hierarchical module concept with sub-modules supports the structuring of large models. The modules in a system model form a tree with respect to the sub-module relation.

Besides sub-modules and model components, a module contains the refinement relations between components that describe the same part of the modeled system in two consecutive levels of refinement. The verification problem associated with a module is to show that these refinements are correct (see Section 3.1). The refinement relations must be provided by the user; typically, they include a signature mapping.

To facilitate information hiding, a module is divided into a public part, its *contract*, and a private (hidden) part; the user can declare parts of *each* refinement level as public or private. Only the public information of a module A is visible in another module B provided that module B implicitly or explicitly *imports* module A. Moreover, a component of module B belonging to some refinement level can only *see* the visible information from module A that belongs to the same level. Thus, the private part of a module can be changed as long as its contract is not affected. For the description of a refinement relation (like a signature mapping) all elements of a module belonging to the initial model or the refined model are visible, whether declared public or not.

As the modeling process proceeds through iterations, the system model becomes ever more precise. The final step is a special case, though: the involved models—the implementation model and its realization in JAVA—do not necessarily differ in precision, but use different paradigms (specification vs. implementation) and different languages (UML with OCL vs. JAVA).

The ideas of refinement and modularization in the KeY module concept can be compared with (and are partly influenced by) the KIV approach [21] and the B Method [1, 17], but still follow different guidelines.

2.4 The Internal State of Objects

The formal specification of objects and their behavior requires special techniques. One important aspect is that the behavior of objects depends on their state that is stored in their attributes, however, the methods of a JAVA class can in general not be described as functions on their input as they may have side effects and change the state. To fully specify the behavior of an object or class, it must be possible to refer to its state (including its initial state). Difficulties may arise if methods for observing the state are not defined or are declared private and, therefore, cannot be used in the public contract of a class. To model such classes, *observer methods* have to be added. These allow to observe the state of a class without changing it.

3 Formal Verification with KeY

Once a program is formally specified to a sufficient degree one can start to formally verify it. Neither a program nor its specification need to be complete in order to start verifying it. In this case one suitably weakens the postconditions (leaving out properties of unimplemented or unspecified parts) or strengthens preconditions (adding assumptions about unimplemented parts). Data encapsulation and structuredness of OO designs are going to be of great help here.

The verification process will be automated as much as possible with the help of deduction techniques based on previous work [2] done in our group on integrating our automated [6] and interactive theorem provers [21].

3.1 Proof Obligations and Program Logic

For obtaining the proof obligations to be justified, we employ design by contract [19] with the same restriction as [25]: run-time aspects are completely ignored.

The logic we use is dynamic logic (DL) [16]. It is a full logic with first-order quantification, built from basic blocks of the form $\langle\alpha\rangle Q$ with the meaning: program α terminates and afterwards formula Q holds. We decided to take a bold step and allow any legal JAVA CARD program to occur in the place of α. A more detailed description of KeY-DL is given in [5]. The central point is, of course, to deal with features of OO languages such as side effects and exception handling.

3.2 The Deduction Component

The KeY system comprises a deductive component that can handle KeY-DL. This KeY prover combines interactive and automated theorem proving techniques. Experiences with the KIV system [21] have shown how to cope with DL proof obligations: The original goal is reduced to first-order predicate logic using DL rules, as described in [5].

Our deductive system uses a technique of *schematic theory specific rules*, which combine purely logical knowledge, information on how this knowledge should be used, and information on when and where this knowledge should be presented for interactive use. This technique has been implemented in the interactive proof system IBIJa[2].

Interactive proving is greatly enhanced by intermediate automated steps based on proof search in the style of analytic tableaux [11]. Also, a component of *disproving* formulas by finding counterexamples is being developed.

4 Related Work

There are many projects dealing with formal methods in software engineering including several ones aimed at JAVA as a target language. There is also work on security of JAVA CARD and ACTIVEX applications as well as on secure smart card applications in general. We are, however, not aware of any project quite like ours. We mention some of the more closely related projects:

- The COGITO project [24] resulted in an integrated formal software development methodology and support system based on extended Z as specification language and Ada as target language. It is not integrated into a CASE tool, but stand-alone.
- The FUZE project [10] realized CASE tool support for integrating the FUSION OOAD process with the formal specification language Z. The aim was to formalize OOAD methods and notations such as the UML, whereas we are interested to derive formal specifications with the help of an OOAD process extension.
- The goal of the QUEST project [22] is to enrich the CASE tool AUTOFOCUS for description of distributed systems with means for formal specification and support by model checking. Applications are embedded systems, description formalisms are state charts, activity diagrams, and temporal logic.
- Aim of the SYSLAB project is the development of a scientifically founded approach for software and systems development. At the core is a precise and formal notion of hierarchical "documents" consisting of informal text, message sequence charts, state transition systems, object models, specifications, and programs. All documents have a "mathematical system model" that allows to precisely describe dependencies or transformations [7].

[2]More information on IBIJa is available at i11www.ira.uka.de/~ibija.

- The goal of the PROSPER project was to provide the means to deliver the benefits of mechanized formal specification and verification to system designers in industry (www.dcs.gla.ac.uk/prosper/index.html). The difference to the KeY project is that the dominant goal is hardware verification; and the software part involves only specification.

5 Conclusion and the Future of KeY

We described the current state of the KeY project and its ultimate goal: To facilitate and promote the use of formal verification in an industrial context for real-world applications. It remains to be seen to which degree this goal can be achieved.

Our vision is to make the logical formalisms transparent for the user with respect to OO modeling. That is, whenever user interaction is required, the current state of the verification task is presented in terms of the environment the user has created so far and not in terms of the underlying deduction machinery. The situation is comparable to a symbolic debugger that lets the user step through the source code of a program while it actually executes compiled machine code.

References

[1] J.-R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, Aug. 1996.

[2] W. Ahrendt, B. Beckert, R. Hähnle, W. Menzel, W. Reif, G. Schellhorn, and P. H. Schmitt. Integration of automated and interactive theorem proving. In W. Bibel and P. Schmitt, editors, *Automated Deduction: A Basis for Applications*, volume II, chapter 4, pages 97–116. Kluwer, 1998.

[3] T. Baar. Experiences with the UML/OCL-approach to precise software modeling: A report from practice. Available at i12www.ira.uka.de/~key, 2000.

[4] T. Baar, T. Sattler, R. Hähnle, and P. H. Schmitt. Entwurfsmustergesteuerte Erzeugung von OCL-Constraints. In *Proceedings, Softwaretechnik 2000, Berlin, Germany*, 2000. To appear. Available at i12www.ira.uka.de/~key.

[5] B. Beckert. A dynamic logic for java card. In *Proceedings, 2nd ECOOP Workshop on Formal Techniques for Java Programs, Cannes, France*, 2000. To appear. Available at i12www.ira.uka.de/~key.

[6] B. Beckert, R. Hähnle, P. Oel, and M. Sulzmann. The tableau-based theorem prover $_3T^AP$, version 4.0. In *Proceedings, 13th International Conference on Automated Deduction (CADE), New Brunswick/NJ, USA*, LNCS 1104, pages 303–307. Springer, 1996.

[7] R. Breu, R. Grosu, F. Huber, B. Rumpe, and W. Schwerin. Towards a precise semantics for object-oriented modeling techniques. In H. Kilov and B. Rumpe, editors, *Proceedings, Workshop on Precise Semantics for Object-Oriented Modeling Techniques at ECOOP'97*. Technical University of Munich, Technical Report TUM-I9725, 1997.

[8] E. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[9] D. L. Dill and J. Rushby. Acceptance of formal methods: Lessons from hardware design. *IEEE Computer*, 29(4):23–24, 1996. Part of: Hossein Saiedian (ed.). *An Invitation to Formal Methods*. Pages 16–30.

[10] R. B. France, J.-M. Bruel, M. M. Larrondo-Petrie, and E. Grant. Rigorous object-oriented modeling: Integrating formal and informal notations. In M. Johnson, editor, *Proceedings, Algebraic Methodology and Software Technology (AMAST), Berlin, Germany*, LNCS 1349. Springer, 1997.

[11] M. Giese. Integriertes automatisches und interaktives Beweisen: Die Kalkülebene. Diploma Thesis, Fakultät für Informatik, Universität Karlsruhe, June 1998. In German. Available at i11www.ira.uka.de/~giese/da.ps.gz.

[12] S. B. Guthery. Java Card: Internet computing on a smart card. *IEEE Internet Computing*, 1(1):57–59, 1997.

[13] U. Hansmann, M. S. Nicklous, T. Schäck, and F. Seliger. *Smart Card Application Development Using Java*. Springer, 2000.

[14] M. G. Hinchey and J. P. Bowen, editors. *Applications of Formal Methods*. Prentice Hall, 1995.

[15] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Object Technology Series. Addison-Wesley, 1999.

[16] D. Kozen and J. Tiuryn. Logic of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 14, pages 789–840. Elsevier, Amsterdam, 1990.

[17] K. Lano. *The B Language and Method: A guide to Practical Formal Development*. Springer Verlag London Ltd., 1996.

[18] J. Martin and J. J. Odell. *Object-Oriented Methods: A Foundation, UML Edition*. Prentice-Hall, 1997.

[19] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, 2nd edition, 1997.

[20] Object Management Group, Inc., Framingham/MA, USA, www.omg.org. *OMG Unified Modeling Language Specification, Version 1.3*, June 1999.

[21] W. Reif. The KIV-approach to software verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, LNCS 1009. Springer, 1995.

[22] O. Slotosch. Overview over the project QUEST. In *Applied Formal Methods, Proceedings of FM-Trends 98, Boppard, Germany*, LNCS 1641, pages 346–350. Springer, 1999.

[23] Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.1 Application Programming Interfaces, Draft 2, Release 1.3*, 1998.

[24] O. Traynor, D. Hazel, P. Kearney, A. Martin, R. Nickson, and L. Wildman. The Cogito development system. In M. Johnson, editor, *Proceedings, Algebraic Methodology and Software Technology (AMAST), Berlin*, LNCS 1349, pages 586–591. Springer, 1997.

[25] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series. Addison-Wesley, 1999.