The KeY Approach: Integrating Object Oriented Design and Formal Verification

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

[1] University of Karlsruhe, Institute for Logic, Complexity and Deduction Systems, D-76128 Karlsruhe, Germany, `http://i12www.ira.uka.de/~key`

[2] Department of Computing Science, Chalmers University of Technology S-41296 Gothenburg, `reiner@cs.chalmers.se`

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 (the majority of producers of integrated circuits are routinely using BDD-based model checking packages for design and validation), usage of formal methods in software development is currently confined essentially to academic research projects. There are industrial applications of formal software development [8], but they are still exceptional [9].

The limits of applicability of formal methods in software design are not defined by the potential range and power of existing approaches. Several case studies clearly demonstrate that computer-aided specification and verification of realistic software is feasible [18]. 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.

While this is viable for projects with high safety and low secrecy demands, it is unlikely that formal software specification and verification will become a routine task in industry under these circumstances.

The future challenge for formal software specification and verification is to make the considerable potential of existing methods and tools 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 formal 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").

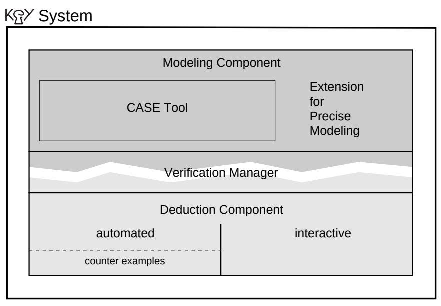
In the principal use case of the KeY system there are actors who want to implement a software system that complies with given requirements and formally verify its correctness. The system is responsible for adding formal detail to the analysis model, for creating conditions that ensure the correctness 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) [24] 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 [22].
- 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's TOGETHER 4.0.
- Formal verification is based on an axiomatic semantics of the *real* programming language JAVA CARD [29] (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 [15,17]. 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 (see the Figure on the right):

[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.

- 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 Together and the system IBIJa [16] as (intercative) deduction component (it has limited capabilities and lacks the verification manager). Work on the full KeY system is in progress.

2 Designing a System with KeY

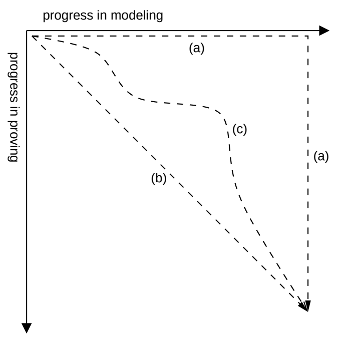
2.1 The Modeling Process

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 *modeling 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 [19] 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* modeling 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.

We do not assume any dependencies between the increments in the development process and the verification of proof obligations. On the right, progress in modeling is depicted along the horizontal axis and progress in verifying proof obligations on the vertical axis. The overall goal is to proceed from the upper left corner (empty model, nothing proven) to the bottom right one (complete model, all proof obligations verified). There are two extreme ways of doing that:



- First complete the whole modeling and coding process, only then start to verify (line (a)).
- Start verifying proof obligations as soon as they are generated (line (b)).

In practice an intermediate approach is chosen (line (c)). How this approach does exactly look is an important design decision of the verification process with strong impact on the possibilities for reuse and is the topic of future research.

2.2 Specification with the UML and the OCL

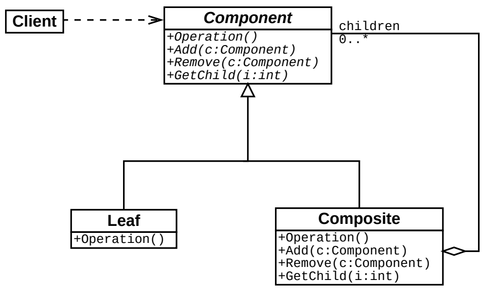
The diagrams of the Unified Modeling Language provide, in principle, an easy and concise way to formulate various aspects of a specification, however, as Steve Cook remarked [31, foreword]: "[...] there are many subtleties and nuances of meaning diagrams cannot convey by themselves."

This was a main source of motivation for the development of the Object Constraint Language (OCL), part of the UML since version 1.3 [24]. 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 provide common patterns that come complete with predefined OCL constraint schemata. They are flexible and allow the user to generate well-adapted constraints for the different instances of a pattern as easily as one uses patterns alone. 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].

As an example, consider the *composite* pattern, depicted on the right [11, p. 163ff]. This is a ubiquitous pattern in many contexts such as user interfaces, recursive data structures, and, in particular, in the model for the address book of an email client that is part of one of our case studies.



The concrete `Add` and `Remove` operations in `Composite` are intuitively clear but leave some questions unanswered. Can we add the same element twice? Some implementations of the *composite* pattern allow this [14]. If it is not intended, then one has to impose a constraint, such as:

```
context Composite::Add(c:Component)
post: self.children→select(p | p = c)→size = 1
```

This is a postcondition on the call of the operation `Add` in OCL syntax. After completion of the operation call, the stated postcondition is guaranteed to be true. Without going into details of the OCL, we give some hints on how to read this expression. The arrow "→" indicates that the expression to its left represents a collection of objects (a set, a bag, or a sequence), and the operation to its right is to be applied to this collection. The dot "." is used to navigate within diagrams and (here) yields those objects associated to the item on its left via the role name on its right. If C is the multiset of all children of the object `self` to which `Add` is applied, then the `select` operator yields the set $A = \{p \in C \mid p = \texttt{c}\}$ and the subsequent integer-valued operation `size` gives the number of elements in A. Thus, the postcondition expresses that after adding `c` as a child to `self`, the object `c` occurs exactly once among the children of `self`.

There are a lot of other useful (and more complex) constraints, e.g., the constraint that the child relationship between objects of class *Component* is acyclic.

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.

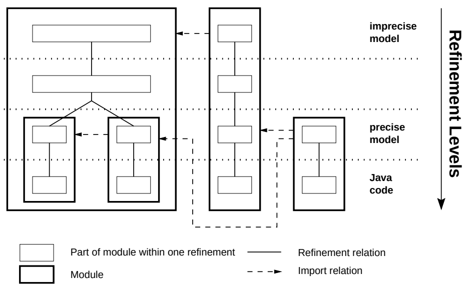
Besides sub-modules and model components, a module contains the refinement relations between components that describe the same part of the modeled system in two consecutive levels of refinement. The verification problem associated with a module is to show that these refinements are correct (see Section 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).[2]

[2] In conventional verification systems that do not use an iterative modeling process [25,27], only these final two models exist (see also the following subsection). In such

Below is a schematic example for the levels of refinement and the modules of a system model (the visibility aspect of modules is not represented here). Stronger refinement may require additional structure via (sub-)modules, hence the number of modules may increase with the degree of refinement.



Although the import and refinement relations are similar in some respects, there is a fundamental difference: by way of example, consider a system component being (imprecisely) modeled as a class DataStorage in an early iteration. It may later be *refined* to a class DataSet, which replaces DataStorage. On the other hand, the module containing DataSet could *import* a module DataList and use lists to implement sets, in which case lists are not a refinement of sets and do not replace them.

Relation of KeY Modules to other Approaches The ideas of refinement and modularization in the KeY module concept can be compared with (and are partly influenced by) the KIV approach [27] and the B Method [1].

In KIV, each module (in the above sense) corresponds to exactly two refinement levels, that is to say, a single refinement step. The first level is an algebraic data type, the second an imperative program, whose procedures intentionally implement the operations of the data type. The import relation allows the algebraic data type operations (not the program procedures!) of the imported module to appear textually in the program of the importing module. In contrast to this, the JAVA code of a KeY module directly calls methods of the imported module's JAVA code. Thus, the object programs of our method are pure JAVA programs. Moreover, KeY modules in general have more than two refinement levels.

The B Method offers (among other things) multi-level refinement of abstract machines. There is an elaborate theory behind the precise semantics of a refinement and the resulting proof obligations. This is possible, because both, a machine and its refinement, are *completely formal*, even if the refinement happens to be *less abstract*. That differs from the situation in KeY, where all but the last refinement levels are UML-based, and a refined part is typically *more formal* than its origin. KeY advocates the integrated usage of notational paradigms as opposed to, for example, prepending OOM to abstract machine specification in the B Method [21].

systems, modules consist of a specification and an implementation that is a refinement of the specification.

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.

Example 1. Let class Registry contain a method `seen(o:Object):Boolean` that maintains a list of all the objects it has "seen". It returns `false`, if it "sees" an object for the first time, and `true`, otherwise. In this example, we add the function `state():Set(Object)` allowing to observe the state of an object of class Registry by returning the set of all seen objects. The behavior of `seen` can now be specified in the OCL as follows:

```
context Registry::seen(o:Object)
post: result  = state@pre()→includes(o) and
      state() = state@pre()→including(o)
```

The OCL key word `result` refers to the return value of `seen`, while `@pre` gives the result of `state()` before invocation of `seen`, which we denote by *oldstate*. The OCL expression `state@pre()→includes(o)` then stands for $\mathtt{o} \in \mathit{oldstate}$ and `state@pre()→including(o)` stands for $\mathit{oldstate} \cup \{\mathtt{o}\}$.

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.

3.1 Proof Obligations

We use constraints in two different ways: first, they can be part of a model (the default); these constraints do not generate proof obligations by themselves. Second, constraints can be given the status of a proof obligation; these are not part of the model, but must be *shown* to hold in it. Proof obligations may arise indirectly from constraints of the first kind: by checking consistency of invariants, pre- and postconditions of a superclass and its subclasses, by checking consistency of the postcondition of an operation and the invariant of its result type, etc. Even more important are proof obligations arising from iterative refinement steps. To prove that a diagram D' is a sound refinement of a diagram D requires to check that the assertions stated in D' entail the assertions in D. A particular refinement step is the passage from a fully refined specification to its realization in concrete code.

3.2 Dynamic Logic.

We use Dynamic Logic (DL) [20]—an extension of Hoare logic [2]—as the logical basis of the KeY system's software verification component. We believe that this is a good choice, as deduction in DL is based on symbolic program execution and simple program transformations, being close to a programmer's understanding of Java Card. For a more detailed description of our Java Card DL than given here, see [5].

DL is successfully used in the KIV software verification system [27] for an imperative programming language; and Poetzsch-Heffter and Müller's definition of a Hoare logic for a Java subset [26] shows that there are no principal obstacles to adapting the DL/Hoare approach to OO languages.

DL can be seen as a modal predicate logic with a modality $\langle p \rangle$ for every program p (p can be any legal Java Card program); $\langle p \rangle$ refers to the successor worlds (called states in the DL framework) reachable by running the program p. In classical DL there can be several such states (worlds) because the programs can be non-deterministic; here, since Java Card programs are deterministic, there is exactly one such world (if p terminates) or there is none (if p does not terminate). The formula $\langle p \rangle \phi$ expresses that the program p terminates in a state in which ϕ holds. A formula $\phi \to \langle p \rangle \psi$ is valid, if for every state s satisfying precondition ϕ a run of the program p starting in s terminates, and in the terminating state the postcondition ψ holds.

The formula $\phi \to \langle p \rangle \psi$ is similar to the Hoare triple $\{\phi\}p\{\psi\}$. In contrast to Hoare logic, the set of formulas of DL is closed under the usual logical operators: In Hoare logic, the formulas ϕ and ψ are pure first-order formulas, whereas in DL they can contain programs. DL allows programs to occur in the descriptions ϕ resp. ψ of states. With is feature it is easy, for example, to specify that a data structure is not cyclic (it is impossible in first-order logic). Also, all Java constructs (e.g., `instanceof`) are available in DL for the description of states. So it is not necessary to define an abstract data type *state* and to represent states as terms of that type (like in [26]); instead, DL formulas can be used to give a (partial) description of states, which is a more flexible technique and allows to concentrate on the relevant properties of a state.

In comparison to classical DL (that uses a toy programming language), a DL for a "real" OO programming language like Java Card has to cope with some complications: (1) A program state does not only depend on the value of (local) program variables but also on the values of the attributes of all existing objects. (2) Evaluation of a Java expression may have side effects, so there is a difference between expressions and logical terms. (3) Such language features as built-in data types, exception handling, and object initialisation must be handled.

3.3 Syntax and Semantics of Java Card DL.

We do not allow class definitions in the programs that are part of DL formulas, but define syntax and semantics of DL formulas wrt a given Java Card program (the context), i.e., a sequence of class definitions. The programs in DL formulas are executable code and comprise all legal Java Card statements, including: (a) expression statements (assignments, method calls, `new`-statements, etc.); (b) blocks and compound statements built with `if-else`, `switch`, `for`, `while`, and `do-while`; (c) statements with exception handling using `try-catch-finally`; (d) statements that redirect the control flow (`continue`, `return`, `break`, `throw`).

We allow programs in DL formulas (not in the context) to contain logical terms. Wherever a Java Card expression can be used, a term of the same type as the expression can be used as well. Accordingly, expressions can contain terms (but not vice versa). Formulas are built as usual from the (logical) terms, the predicate symbols (including the equality predicate $\doteq$), the logical connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$, the quantifiers $\forall$ and $\exists$ (that can be applied to logical variables but not to program variables), and the modal operator $\langle p \rangle$, i.e., if p is a program and ϕ is a formula, then $\langle p \rangle\phi$ is a formula as well.

The models of DL consist of program states. These states share the same universe containing a sufficient number of elements of each type. In each state a (possibly different) value (an element of the universe) of the appropriate type is assigned to: (a) the program variables, (b) the attributes (fields) of all objects, (c) the class attributes (static fields) of all classes in the context, and (d) the special object variable `this`. Variables and attributes of object types can be assigned the special value *null*. States do not contain any information on control flow such as a program counter or the fact that an exception has been thrown.

The semantics of a program p is a state transition, i.e., it assigns to each state s the set of all states that can be reached by running p starting in s. Since Java Card is deterministic, that set either contains exactly one state or is empty. The set of states of a model must be closed under the reachability relation for all programs p, i.e., all states that are reachable must exist in a model (other models are not considered).

We consider programs that terminate abnormally to be non-terminating: nothing can be said about their final state. Examples are a program that throws an uncaught exception and a `return` statement outside of a method invocation. Thus, for example, $\langle$`throw x;`$\rangle\phi$ is unsatisfiable for all ϕ.[3]

3.4 A Sequent Calculus for Java Card DL.

We outline the ideas behind our sequent calculus for Java Card DL and give some of its basic rules (actually, simplified versions of the rules, e.g., initialisation of objects and classes is not considered). The DL rules of our calculus operate on

[3] It is still possible to express and (if true) prove the fact that a program p terminates abnormally. For example, $\langle$`try{`p`}catch{Exception e}`$\rangle(\neg$ `e` $\doteq$ `null`$)$ expresses that p throws an exception.

$$\frac{\Gamma \vdash \mathit{cnd} \doteq \texttt{true} \qquad \Gamma \vdash \langle \pi\ \mathit{prg}\ \texttt{while}\,(\mathit{cnd})\,\mathit{prg}\ \ \omega\rangle\phi}{\Gamma \vdash \langle \pi\ \texttt{while}\,(\mathit{cnd})\,\mathit{prg}\ \ \omega\rangle\phi} \tag{1}$$

$$\frac{\Gamma \vdash \mathit{cnd} \doteq \texttt{false} \qquad \Gamma \vdash \langle \pi\omega\rangle\phi}{\Gamma \vdash \langle \pi\ \texttt{while}\,(\mathit{cnd})\,\mathit{prg}\ \ \omega\rangle\phi} \tag{2}$$

$$\frac{\Gamma \vdash \mathit{instanceof}(\mathit{exc}\,,T) \qquad \Gamma \vdash \langle \pi\ \texttt{try\{}\mathit{e}\texttt{=}\mathit{exc}\,\texttt{;}\ \ \mathit{q}\texttt{\}finally\{}\mathit{r}\texttt{\}}\ \ \omega\rangle\phi}{\Gamma \vdash \langle \pi\ \texttt{try\{throw}\ \mathit{exc}\,\texttt{;}\ \ \mathit{p}\texttt{\}catch(}T\ \ \mathit{e}\texttt{)\{}\mathit{q}\texttt{\}finally\{}\mathit{r}\texttt{\}}\ \ \omega\rangle\phi} \tag{3}$$

$$\frac{\Gamma \vdash \neg\mathit{instanceof}(\mathit{exc}\,,T) \qquad \Gamma \vdash \langle \pi\ \ \mathit{r}\,\texttt{;}\ \texttt{throw}\ \mathit{exc}\,\texttt{;}\ \ \omega\rangle\phi}{\Gamma \vdash \langle \pi\ \texttt{try\{throw}\ \mathit{exc}\,\texttt{;}\ \ \mathit{p}\texttt{\}catch(}T\ \ \mathit{e}\texttt{)\{}\mathit{q}\texttt{\}finally\{}\mathit{r}\texttt{\}}\ \ \omega\rangle\phi} \tag{4}$$

$$\frac{\Gamma \vdash \langle \pi\ \ \mathit{r}\ \ \omega\rangle\phi}{\Gamma \vdash \langle \pi\ \texttt{try\{\}catch(}T\ \ \mathit{e}\texttt{)\{}\mathit{q}\texttt{\}finally\{}\mathit{r}\texttt{\}}\ \ \omega\rangle\phi} \tag{5}$$

Table 1. Some of the rules of our calculus for Java Card DL.

the first *active* command p of a program $\pi p\,\omega$. The non-active prefix π consists of an arbitrary sequence of opening braces "`{`", labels, beginnings "`try{`" of `try-catch` blocks, etc. The prefix is needed to keep track of the blocks that the (first) active command is part of, such that the commands `throw`, `return`, `break`, and `continue` that abruptly change the control flow are handled correctly. (In classical DL, where no prefixes are needed, any formula of the form $\langle p\ q\rangle\phi$ can be replaced by $\langle p\,\rangle\langle q\,\rangle\phi$. In our calculus, splitting of $\langle \pi p q \omega\rangle\phi$ into $\langle \pi p\,\rangle\langle q\,\omega\rangle\phi$ is not possible (unless the prefix π is empty) because πp is not a valid program; and the formula $\langle \pi p\,\omega\rangle\langle \pi\, q\,\omega\rangle\phi$ cannot be used either because its semantics is in general different from that of $\langle \pi p q\,\omega\rangle\phi$.)

As examples, we present the rules for `while` loops and for exception handling. The rules operate on sequents $\Gamma \vdash \phi$. The semantics of a sequent is that the conjunction of the DL formulas in Γ implies the DL formula ϕ. Sequents are used to represent proof obligations, proof (sub-)goals, and lemmata.

Rules (1) and (2) in Table 1 allow to "unwind" `while` loops. They are simplified versions that only work if (a) the condition cnd is a logical term (i.e., has side effects), and (b) the program prg does not contain a `continue` statement. These rules allow to handle loops if used in combination with induction schemata. Similar rules are defined for `do-while` and `for` loops.

Rules (3)–(5) handle `try-catch-finally` blocks and the `throw` statement. Again, these are simplified versions of the actual rules; they are only applicable if (a) exc is a logical term (e.g., a program variable), and (b) the statements `break`, `continue`, `return` do not occur. Rule (3) applies, if an exception exc is thrown that is an instance of exception class T, i.e., the exception is caught; otherwise, if the exception is not caught, rule (4) applies. Rule (5) applies if the `try` block is empty and terminates normally.

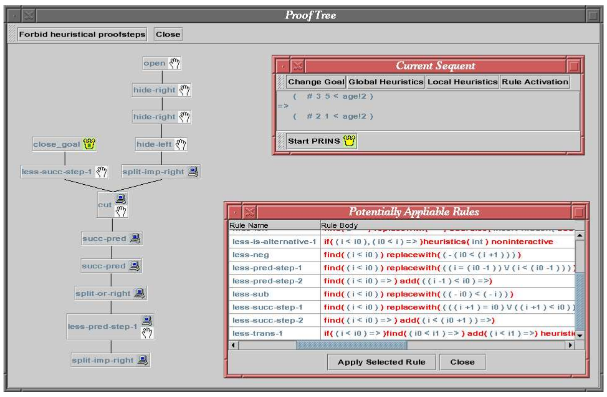
3.5 The KeY Deduction Component

The KeY system comprises a deductive component, that can handle KeY-DL. This KeY prover combines interactive and automated theorem proving techniques. Experience with the KIV system [27] has shown how to cope with DL proof obligations. The original goal is reduced to first-order predicate logic using DL rules, as demonstrated in the previous section. First-order goals can be proven using theory specific knowledge about the used data types.

We developed a language for expressing knowledge of specific theories—we are thinking here mainly of theories of abstract data types—in the form of proof rules. We believe that this format, stressing the operational aspect, is easier to understand and simpler to use than alternative approaches coding the same knowledge in declarative axioms, higher-order logic, or fixed sets of special proof rules. This format, called *schematic theory specific rules*, is explained in detail in [16] and has been implemented in the interactive proof system IBIJa (`i11www.ira.uka.de/~ibija`). In particular, a schematic theory specific rule contains: (a) Pure logical knowledge, (b) information on how this knowledge is to be used, and (c) information on when and where this knowledge should be presented for interactive use.

Nearly all potential rule applications are triggered by the occurrence of certain terms or formulas in the proof context. The easy-to-use graphical user interface of IBIJa supports invocation of rule applications by mouse clicks on the relevant terms and formulas. The rule schema language is expressive enough to describe even complex induction rules. The rule schema language is carefully designed in such a way that for every new schematic theory specific rule, IBIJa automatically generates proof obligations in first-order logic. Once these obligations are shown to be true the soundness of all applications of this rule is guaranteed. Hence, during each state of a proof, soundness-preserving new rules can be introduced.

To be practically useful, interactive proving must be enhanced by automating intermediate proof steps as much as possible. Therefore, the KeY prover combines the IBIJa with automated proof search in the style of analytic tableaux. This integration is based on the concepts described in [12,13]. A screen shot of a typical situation as it may arise during proof construction with our prototype is shown below. The user may either interactively apply a rule (button "Apply Selected Rule") or invoke the automated deduction component (button "Start PRINS").



```
  (   # 3 5 < age!2 )
=>
  (   # 2 1 < age!2 )
```

Rule Name	Rule Body
less-is-alternative-1	if((i < i0), (i0 < i) =>)heuristics(int) noninteractive
less-neg	find((i < i0)) replacewith((- (i0 < (i +1))))
less-pred-step-1	find((i < i0)) replacewith(((i = (i0 -1)) V (i < (i0 -1)))
less-pred-step-2	find((i < i0) =>) add(((i -1) < i0) =>)
less-sub	find((i < i0)) replacewith(((- i0) < (- i)))
less-succ-step-1	find((i < i0)) replacewith((((i +1) = i0) V ((i +1) < i0))
less-succ-step-2	find((i < i0) =>) add((i < (i0 +1)) =>)
less-trans-1	if((i < i0) =>)find((i0 < i1) =>) add((i < i1) =>) heuristi

In a real development process, resulting programs often are bug-ridden, therefore, the ability of *disproving* correctness is as important as the ability of *proving* it. The interesting and common case is that neither correctness nor its negation are deducible from given assumptions. A typical reason is that data structures are underspecified. We may, for example, not have any knowledge about the behavior of, say, `pop(s:Stack):Stack` if `s` is empty. To recognize such situations, which often lead to bugs in the implementation, we develop special deductive techniques. They are based on automatically constructing *interpretations* (of data type operations) that fulfill all assumptions but falsify the hypothesis.

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.

A thorough mathematical analysis of Java using Abstract State Machines has been given in [6]. Following another approach, a precise semantics of a Java sublanguage was obtained by embedding it into Isabelle/HOL [23]; there, an axiomatic semantics is used in a similar spirit as in the present paper.

The COGITO project [30] 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 [28] 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].

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

In this paper 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.

Acknowledgements

Thanks to S. Klingenbeck and J. Posegga for valuable comments on earlier versions of this paper. We also thank our former group members T. Fuchß, R. Preiß, and A. Schönegge for their input during the preparation of the project. KeY is supported by the Deutsche Forschungsgemeinschaft (grant no. Ha 2617/2-1).

References

1. J.-R. Abrial. *The B Book - Assigning Programs to Meanings.* Cambridge University Press, Aug. 1996.
2. K. R. Apt. Ten years of Hoare logic: A survey – part I. *ACM Transactions on Programming Languages and Systems*, 1981.
3. T. Baar. Experiences with the UML/OCL-approach to precise software modeling: A report from practice. Submitted, see i12www.ira.uka.de/~key, 2000.
4. T. Baar, R. Hähnle, T. Sattler, and P. H. Schmitt. Entwurfsmustergesteuerte Erzeugung von OCL-Constraints. In G. Snelting, editor, *Softwaretechnik-Trends*, Informatik Aktuell. Springer, 2000.
5. B. Beckert. A dynamic logic for Java Card. In *Proc. 2nd ECOOP Workshop on Formal Techniques for Java Programs, Cannes, France*, 2000. See i12www.ira.uka.de/~key/doc/2000/beckert00.pdf.gz.
6. E. Börger and W. Schulte. A programmer friendly modular definition of the semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS 1523, pages 353–404. Springer, 1999.
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, *Proc Workshop on Precise Semantics for Object-Oriented Modeling Techniques at ECOOP'97.* Techn. Univ. of Munich, Tech. Rep. 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, *Proc. Algebraic Methodology and Software Technology (AMAST), Berlin, Germany*, LNCS 1349. Springer, 1997.
11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
12. M. Giese. Integriertes automatisches und interaktives Beweisen: Die Kalkülebene. Diploma Thesis, Fakultät für Informatik, Universität Karlsruhe, June 1998.
13. M. Giese. A first-order simplification rule with constraints. In *Proc. Int. Workshop on First-Order Theorem Proving, St. Andrews, Scotland*, 2000. See i12www.ira.uka.de/~key/doc/2000/giese00a.ps.gz.
14. M. Grand. *Patterns in Java*, volume 2. John Wiley & Sons, 1999.
15. S. B. Guthery. Java Card: Internet computing on a smart card. *IEEE Internet Computing*, 1(1):57–59, 1997.
16. E. Habermalz. Interactive theorem proving with schematic theory specific rules. See i12www.ira.uka.de/~key/doc/2000/stsr.ps.gz, 2000.
17. U. Hansmann, M. S. Nicklous, T. Schäck, and F. Seliger. *Smart Card Application Development Using Java.* Springer, 2000.
18. M. G. Hinchey and J. P. Bowen, editors. *Applications of Formal Methods.* Prentice Hall, 1995.
19. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.* Object Technology Series. Addison-Wesley, 1999.
20. 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.
21. K. Lano. *The B Language and Method: A guide to Practical Formal Development.* Springer Verlag London Ltd., 1996.
22. J. Martin and J. J. Odell. *Object-Oriented Methods: A Foundation, UML Edition.* Prentice-Hall, 1997.
23. T. Nipkow and D. von Oheimb. Machine-checking the Java specification: Proving type safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS 1523, pages 119–156. Springer, 1999.
24. Object Management Group, Inc., Framingham/MA, USA, www.omg.org. *OMG Unified Modeling Language Specification, Version 1.3*, June 1999.
25. L. C. Paulson. *Isabelle: A Generic Theorem Prover.* Springer, Berlin, 1994.
26. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *Proc. Programming Languages and Systems (ESOP), Amsterdam, The Netherlands*, LNCS 1576, pages 162–176. Springer, 1999.
27. 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.
28. 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.
29. Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.1 Application Programming Interfaces, Draft 2, Release 1.3*, 1998.
30. O. Traynor, D. Hazel, P. Kearney, A. Martin, R. Nickson, and L. Wildman. The Cogito development system. In M. Johnson, editor, *Proc. Algebraic Methodology and Software Technology, Berlin*, LNCS 1349, pages 586–591. Springer, 1997.
31. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modelling with UML.* Object Technology Series. Addison-Wesley, 1999.