The KeY Approach: Integrating Object Oriented Design and Formal Verification

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

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

Abstract. This paper reports on the ongoing KeY project aimed at bridging the gap between (a) OO 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. Although there are industrial applications of formal software development [7], these are still exceptional [8].

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 [27, 24]. 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 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. 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, a *formal software specification and verification process* is needed.

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 will assist with and document the different work-flows of this process: requirements, analysis, design, and implementation. In addition there will be the work-flow called verification. It 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 practice, and on JAVA as the target language. In particular, we use the Unified Modeling Language (UML) [21] for visual modeling of designs and specification 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 [19].

- 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 Sterling's COOL:JEX.
- Formal verification is based on an axiomatic semantics of the *real* programming language JAVA CARD [28] (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 a programming language [14, 15]. JAVA smart cards make an extremely suitable target for a case study:
  - As an object-oriented language, JAVA CARD is well suited for OOAD;
  - the JAVA CARD language 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 (although 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.

The KeY system consists of three main components (see Fig. 1):

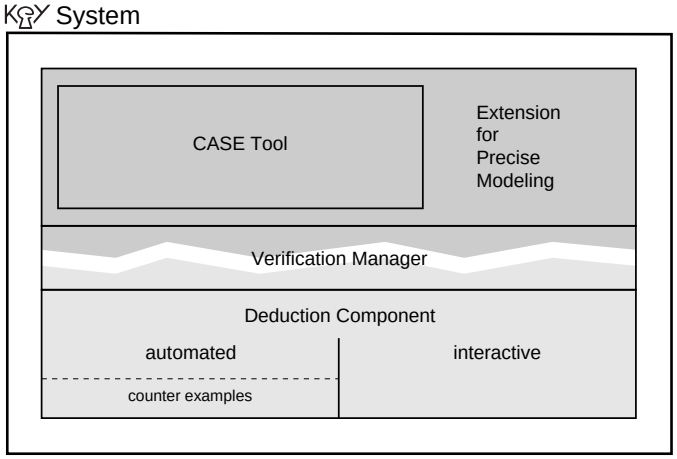

Fig. 1. Architecture of the KeY system.

- 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.

[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 *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 the 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 for 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, KeY is a fully integrated system with a uniform user interface.

It is worth pointing out that we do not assume any dependencies between the increments in the development process and the verification of proof obligations. In Fig. 2 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 proved) 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 (Fig. 2(a)).
– Start verifying proof obligations as soon as they are generated (Fig. 2(b)).

In practice one chooses an intermediate approach (Fig. 2(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.

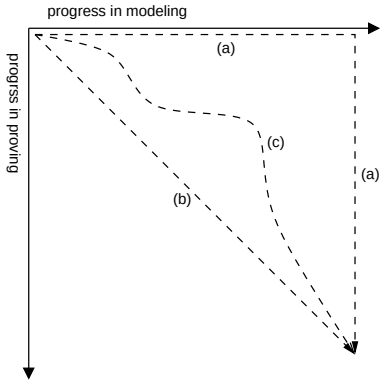

Fig. 2. Two dimensions: modeling and verification.

2 Designing a System with KeY

2.1 Specification with the UML including the OCL

When designing a system with KeY, one first develops a UML model using our integrated CASE tool as usual (see the following subsection for process methodology).

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 [30, 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 [21]. 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. It is easy to extract the signature to be used in OCL expressions automatically from the class diagrams of a model.

The second step in designing a system with KeY is thus to make the UML model more precise by adding OCL constraints (making the UML more precise is also on the agenda of the *precise UML group* whose goals are laid down in [9], see also `www.cs.york.ac.uk/puml/`). For that purpose, the KeY system provides menu and dialog driven input possibility to assist the user. 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 fully automated, 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.

Another possibility to bring (OCL) constraints into a UML model is by enriched design patterns. In the KeY system we will provide common patterns that come complete with predefined constraints or constraint schemata. The user needs not write formal specifications from scratch, but only to adapt and complete them.

As an example, consider the *composite* pattern [11, p. 163ff], depicted in Fig. 3. 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.

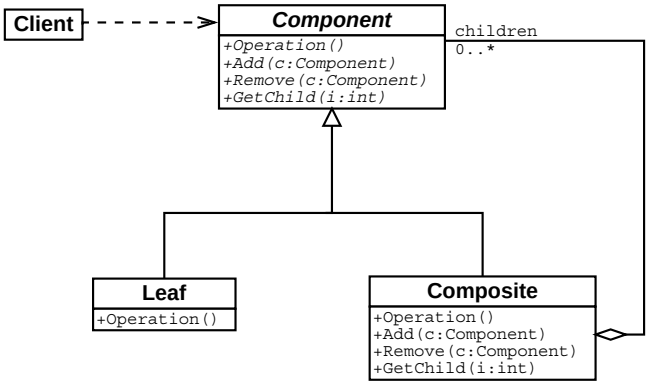

Fig. 3. The *composite* pattern.

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 that [13]. 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 "$\rightarrow$" indicates that the expression to its left represents a collection of objects (a set, a multiset, 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.

2.2 The Modeling Process

In addition to a suitable language to express (formal) models—in KeY this is the UML including the OCL—a *methodology* guiding the modeling process must be provided. Most methodologies described in the OOAD literature, for example, OOD [5] or the Rational Unified Process [17], have two important features in common: They are *iterative* and *incremental*.

These features have been adopted for the methodology used in KeY: A project is divided into *iterations*. In each iteration the user develops a complete model; the increments achieved within the iterations are that the models get more and more precise. The model of iteration $i + 1$ *refines* the model of iteration i (a detailed description of the refinement relation is given below).

In all refinements—except the final one—models are expressed in the UML (including the OCL), however, in later iterations the models become more detailed and, in addition, contain more OCL constraints, which provide for a higher degree of precision. The final refinement step is the implementation, in other words, we consider the realization of a system in JAVA code to be a particular (and very precise) model of that system.

2.3 The KeY Module Concept

As said before, the process of modeling a system consists of several iterations. In addition, the KeY system supports modularization of the model. 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: 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 other 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/OCL vs. JAVA).[2]

Fig. 4 shows 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.

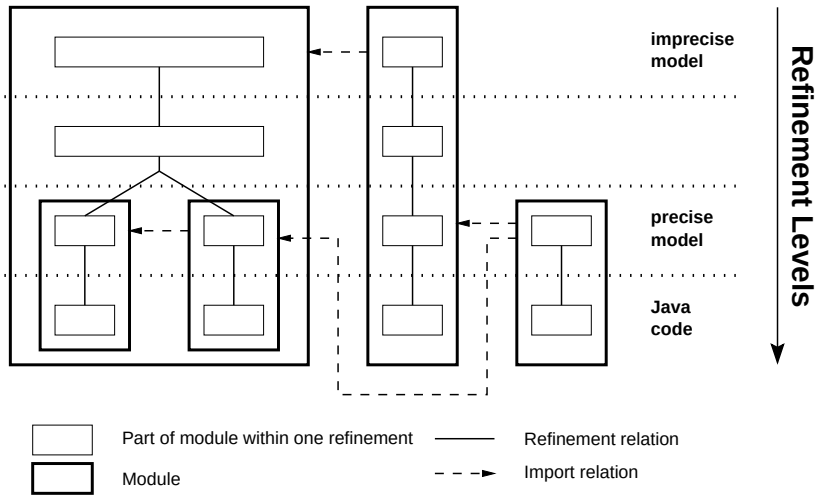

Fig. 4. Example for levels of refinement and modules of a system model.

[2] In conventional verification systems that do not use an iterative modeling process [22, 25], only these final two models exist (see also the following subsection). In such systems, modules consist of a specification and an implementation that is a refinement of the specification.

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.

2.4 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 [25] 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 [18].

2.5 Modeling the Internal State of Objects

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 model 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. Consider a class Registry containing 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 would 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` specifies the expected 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 $o \in oldstate$ and `state@pre()→including(o)` for $oldstate \cup \{o\}$.

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/unspecified parts) or strengthens preconditions (adding assumptions about unimplemented parts). Data encapsulation and structuredness of OO designs should 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 [4] and interactive theorem provers [25].

In a real development process, resulting programs often are bug-ridden, therefore, *disproving* the correctness of programs is as important as *proving* it. The interesting and common case is that neither correctness nor its negation are deducible from given assumptions, often because these assumptions do not fully specify the data structures modified by the program. As a simple example, we might not have any knowledge about the behavior of, say, `pop(s: Stack): Stack` if `s` is empty. We are developing deductive techniques to automatically exhibit bugs, in particular caused by underspecification, within the verification process.

Due to space limitation, a full description of the deductive component will be given elsewhere.

3.1 Proof Obligations

The basis for reasoning about properties of programs in KeY is dynamic logic (DL) [16], an extension of Hoare logic [3]. In contrast to Hoare logic, the set of formulas of dynamic logic is closed under the usual logical operations. Typical buildings blocks of DL formulas are schemata $P \rightarrow \langle\alpha\rangle Q$, which are true if for every state satisfying precondition P a run of the program α starting in such a state terminates, and in the terminating state the postcondition Q holds. DL has been successfully used in the KIV system [25]. It was shown [23] that there are no principal obstacles to adapt the DL/Hoare approach to typed object-oriented languages. DL is stronger than first-order logic, and allows, for example, to characterize cyclicity of data structures.

Typically, the statements to be proven arise from OCL constraints in UML models. The OCL (a) has no formal semantics and (b) has no means to connect constraints to target programs. It is, therefore, not directly usable for automated deduction and, because of (a), one has to translate OCL constraints into DL formulas. Details of this interesting subtask of the KeY project will be addressed in a separate publication. Here, we merely say a few words on the origin of proof obligations.

We employ *design by contract* [20] as a guiding principle with the same restriction as [30]: we completely ignore run-time aspects of this concept. Constraints occur as pre- and postconditions of operations, and as invariants of classes, to mention the most frequent cases.

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 The KeY Program Logic

The basic building blocks for correctness statements in DL have the form $\langle\alpha\rangle Q$, representing the weakest condition, whose validity in a state s guarantees that execution of the program α terminates in a state satisfying Q. We decided to take a bold step and allow any legal JAVA CARD program to occur in the place of α in our DL formulas.

We assume that programs and, in particular, expressions in programs are parsed already. Thus, the calculus needs not to know about operator priorities etc., and we can use notions like "immediate sub-expression" in the definition of our rules. A full description of KeY-DL, the dynamic logic used in KeY, will be given elsewhere. Here, we try to convey the basic spirit of our approach. The usual assignment rule of DL[3]

$$\frac{(P^y_{\mathtt{x}} \wedge \mathtt{x} \doteq \mathtt{t}^y_{\mathtt{x}}) \rightarrow Q}{P \rightarrow \langle \mathtt{x = t}\rangle Q} \qquad \text{where } y \text{ is new} \tag{1}$$

has to be modified and extended, because the evaluation of the JAVA CARD expression `t` (and even of `x`) may have side effects. The logic has to "know" about the control flow during evaluation of expressions.

Example 2. Let us consider the formula $F \equiv (P \rightarrow \langle\alpha\rangle Q)$ with

$$\begin{aligned} P &\equiv \mathtt{i} \doteq 3 \wedge \mathtt{v}[1] \doteq 4 \wedge \mathtt{j} \doteq 4 \\ \alpha &\equiv \mathtt{v[i++] = j++ * j;} \\ Q &\equiv \mathtt{i} \doteq 4 \wedge \mathtt{v}[1] \doteq 4 \wedge \mathtt{v}[3] \doteq 20 \wedge \mathtt{j} \doteq 5 \end{aligned}$$

We want to show that F is a valid formula: the execution of α in a state, where precondition P holds, terminates in a state where postcondition Q holds.

The program α contains the postfix increment operator `++`. According to the JAVA language specification [12, Sect. 15.13.2], `i++` may be used to refer to the variable `i`. As a side effect, the value of `i` is increased by one afterwards. This is reflected by a KeY-DL rule that handles `i++`. Applied to formula F, it yields:

$$(P \wedge \mathtt{x} \doteq \mathtt{i}) \rightarrow \langle \mathtt{i = i+1;}\rangle\langle \mathtt{v[x] = j++ * j;}\rangle Q \tag{2}$$

Application of the assignment rule (1) to (2) then gives:

$$(P^y_{\mathtt{i}} \wedge \mathtt{x} \doteq y \wedge \mathtt{i} \doteq y+1) \rightarrow \langle \mathtt{v[x] = j++ * j;}\rangle Q$$

Treating `j++` in the same way we get the next two steps in the evaluation:

$$\begin{aligned} &(P^y_{\mathtt{i}} \wedge \mathtt{x} \doteq y \wedge \mathtt{i} \doteq y+1 \wedge \mathtt{z} \doteq \mathtt{j}) \rightarrow \langle \mathtt{j = j+1;}\rangle\langle \mathtt{v[x] = z*j;}\rangle Q \\ &(P^{y,u}_{\mathtt{i,j}} \wedge \mathtt{x} \doteq y \wedge \mathtt{i} \doteq y+1 \wedge \mathtt{z} \doteq u \wedge \mathtt{j} \doteq u+1) \rightarrow \langle \mathtt{v[x] = z*j;}\rangle Q \end{aligned}$$

How to treat assignments to array variables in program logics is well known [3]; in the present case, note that `v[1]` occurs in P and thus in the premiss of the implication, and the two cases that `v[1]` is/is not changed by the assignment have to be considered:

$$\begin{aligned} &((\mathtt{x} \doteq 1 \wedge P^{y,u,w}_{\mathtt{i,j,v[1]}} \wedge \cdots \wedge \mathtt{j} \doteq u+1 \wedge \mathtt{v}[1] \doteq \mathtt{z} * \mathtt{j}) \rightarrow Q) \ \wedge \\ &((\neg(\mathtt{x} \doteq 1) \wedge P^{y,u}_{\mathtt{i,j}} \wedge \cdots \wedge \mathtt{j} \doteq u+1 \wedge \mathtt{v[x]} \doteq \mathtt{z} * \mathtt{j}) \rightarrow Q) \end{aligned} \tag{3}$$

[3] The formula F^y_x arises from the formula F by replacing all free occurrences of x by y.

The result (3) does not contain any JAVA code. Simplification of (3) using the definition of P now yields:

$$((\mathtt{x} \doteq 1 \land y \doteq 3 \land \mathtt{x} \doteq y \land \cdots) \to Q) \ \land$$
$$((y \doteq 3 \land \mathtt{v}[1] \doteq 4 \land u \doteq 4 \land \cdots \land \mathtt{i} \doteq 4 \land \mathtt{z} \doteq 4 \land \mathtt{j} \doteq 5 \land \mathtt{v}[3] \doteq 20) \to Q)$$

It is easy to check that this indeed is a valid formula and our present theorem proving tools [25, 4] have no difficulties to show this automatically.

It is important to note that the postfix incremental operator `++` is not just a fancy construct we must deal with to complete the picture. Such an operator, whether important by itself or not, serves as a concise example for a general phenomenon in a language like JAVA: expressions can have both, a value and an effect. In particular, JAVA allows to call (non-`void`) methods, possibly changing the object's state, inside a value-returning expression. Therefore, the calculus must be able to *execute* an expression stepwise, as illustrated by the above example.

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 [29] 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 standalone.
- 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 [26] 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 [6].
- The PROSPER (www.dcs.gla.ac.uk/prosper/index.html) project's goal was to provide the means to deliver the benefits of mechanized formal specification and verification to system designers in industry. The difference to the KeY project is that the dominant goal is hardware verification; the software part only involves 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 are due 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 KeY project. The KeY project is supported by the Deutsche Forschungsgemeinschaft under grant no. Ha 2617/2-1.

References

1. J.-R. Abrial. *The B Book – Assigning Programs to Meanings.* Cambridge University Press, 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. K. R. Apt. Ten years of Hoare logic: A survey—Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, Oct. 1981.
4. B. Beckert, R. Hähnle, P. Oel, and M. Sulzmann. The tableau-based theorem prover $_3T^AP$, version 4.0. In *13th International Conference on Automated Deduction, New Brunswick/NJ, USA*, volume 1104 of *LNCS*, pages 303–307. Springer-Verlag, 1996.
5. G. Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin Cummings, Redwood City, 2nd edition, 1994.
6. R. Breu, R. Grosu, F. Huber, B. Rumpe, and W. Schwerin. Towards a precise semantics for object-oriented modeling techniques. In J. Bosch and S. Mitchell, editors, *Object-Oriented Technology, ECOOP'97 Post Conference Workshop Reader, Jyväskylä, Finland*, volume 1357 of *LNCS*. Springer-Verlag, 1997.
7. E. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.
8. 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.
9. A. S. Evans, S. Cook, S. Mellor, J. Warmer, and A. Wills. Panel paper: Advanced methods and tools for a precise UML. In B. Rumpe and R. B. France, editors, *2nd International Conference on the Unified Modeling Language*, volume 1732 of *LNCS*. Springer-Verlag, 1999.
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, *Algebraic Methodology and Software Technology (AMAST), Berlin, Germany*, volume 1349 of *LNCS*. Springer-Verlag, 1997.
11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
12. J. Gosling, B. Joy, and G. Steele, editors. *The Java Language Specification.* Addison Wesley, 1996.
13. M. Grand. *Patterns in Java*, volume 2. John Wiley & Sons, 1999.
14. S. B. Guthery. Java Card: Internet computing on a smart card. *IEEE Internet Computing*, 1(1):57–59, 1997.
15. U. Hansmann, M. S. Nicklous, T. Schäck, and F. Seliger. *Smart Card Application Development Using Java.* Springer-Verlag, 1999, to appear.

16. D. Harel. Dynamic logic. In *Handbook of Philosophical Logic*, volume II: Extensions of Classical Logic, pages 497–604. Reidel, 1984.
17. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.* Object Technology Series. Addison-Wesley, 1999.
18. K. Lano. *The B Language and Method: A guide to Practical Formal Development.* Springer-Verlag London Ltd., 1996.
19. J. Martin and J. J. Odell. *Object-Oriented Methods: A Foundation, UML Edition.* Prentice-Hall, 1997.
20. B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Englewood Cliffs, second edition, 1997.
21. Object Modeling Group. *Unified Modeling Language Specification, version 1.3*, June 1999. URL: `uml.shl.com:80/docs/UML1.3/99-06-08.pdf`.
22. L. C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *LNCS*. Springer-Verlag, 1994.
23. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *Programming Languages and Systems (ESOP)*, volume 1576 of *LNCS*, pages 162–176. Springer-Verlag, 1999.
24. S. Qadeer and N. Shankar. Verifying a self-stabilizing mutual exclusion algorithm. In *IFIP Working Conference on Programming Concepts and Methods (PROCOMET), Shelter Island/NY, USA*. Chapman & Hall, 1998.
25. 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*, volume 1009 of *LNCS*. Springer-Verlag, 1995.
26. O. Slotosch. QUEST: Overview over the project. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Applied Formal Methods — FM-Trends 98 — International Workshop on Current Trends in Applied Formal Methods, Boppard, Germany*, volume 1641 of *LNCS*, pages 346–350. Springer-Verlag, 1999.
27. K. Stenzel. A Verified Access Control Model. Technical Report 26/93, Fakultät für Informatik, Universität Karlsruhe, 1993.
28. Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.1 Platform API Specification*, 1998. URL: `java.sun.com/products/javacard/JavaCard21API.pdf`.
29. O. Traynor, D. Hazel, P. Kearney, A. Martin, R. Nickson, and L. Wildman. The Cogito development system. In M. Johnson, editor, *Algebraic Methodology and Software Technology, Berlin, Germany*, volume 1349 of *LNCS*, pages 586–591. Springer-Verlag, 1997.
30. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Object Technology Series. Addison-Wesley, 1999.