

Technical Report no. 2003-05

## The KeY Tool<sup>1</sup>

**Wolfgang Ahrendt    Thomas Baar    Bernhard Beckert    Richard Bubel**  
**Martin Giese    Reiner Hähnle    Wolfram Menzel    Wojciech Mostowski**  
**Andreas Roth    Steffen Schlager    Peter H. Schmitt**

**CHALMERS** | GÖTEBORG UNIVERSITY



Department of Computing Science  
Chalmers University of Technology and Göteborg University  
412 96 Göteborg, Sweden

Göteborg, 2003

<sup>1</sup>The research reported here has been partially supported by Vetenskapsrådet, Vinnova, and DFG. It has been jointly carried out here and at the Department of Computer Science, University of Karlsruhe, 76128 Karlsruhe, Germany.



Technical Report in Computing Science at  
Chalmers University of Technology and Göteborg University

Technical Report no. 2003-05  
ISSN: 1650-3023

Department of Computing Science  
Chalmers University of Technology and Göteborg University  
412 96 Göteborg, Sweden

Göteborg, Sweden, 2003

## Abstract

*KeY is a tool that provides facilities for formal specification and verification of programs within a commercial platform for UML based software development. Using the KeY tool, formal methods and object-oriented development techniques are applied in an integrated manner. Formal specification is performed using the Object Constraint Language (OCL), which is part of the UML standard. KeY provides support for the authoring and formal analysis of OCL constraints. The target language of KeY based development is JAVA CARD, a proper subset of JAVA for smart card applications and embedded systems. KeY uses a dynamic logic for JAVA CARD to express proof obligations, and provides a state-of-the-art theorem prover for interactive and automated verification. Apart from its integration into UML based software development, a characteristic feature of KeY is that formal specification and verification can be introduced incrementally.*

**Keywords:** *object-oriented design, formal specification, formal verification, UML, OCL, design patterns*JAVA

## 1 Introduction

KeY is a tool for the development of high quality object-oriented software. The “KeY” idea behind this tool is to provide facilities for formal specification and verification of programs *within* a software development platform supporting contemporary design and implementation methodologies. The KeY tool empowers its users to perform formal specification and verification as part of software development based on the *Unified Modeling Language (UML)*. To achieve this, the system is realised as the extension of a commercial UML-based *Computer Aided Software Engineering Tool (CASE tool)*. As a consequence, specification and verification can be performed within the extended CASE tool itself. Such a deep integration of formal specification and verification into modern software engineering concepts serves two purposes. First, formal methods and object-oriented development techniques become *applicable* in proper combination at all. Second, formal specification and verification become more *accessible* to developers who are already using object-oriented design methodology. Moreover, KeY allows a *lightweight* usage of the provided formal techniques, as both, specification and verification, can be performed at any time, and to any desired degree.

The target language of KeY-driven software development is JAVA. More specifically, the verification facilities of KeY are restricted to code written in JAVA CARD [53, 19]. JAVA CARD is a proper subset of the JAVA programming language, excluding certain features (like threads, cloning or dynamic class loading) and with a much reduced API. The JAVA CARD language [53] and platform [54] are provided by Sun Microsystems to enable JAVA technology to run on smart cards and other devices with limited memory, such as embedded systems.

UML based software development puts an emphasis on the activity of *designing* the targeted system. It is increasingly accepted that the design stage is very much where one actually has the power to prevent a system from failing. This suggests that formal specification and verification should (in different ways) be closely tied to the design phase, to design documents, and to design tools. One way of combining object-oriented design and formal specification is to attach *constraints* to *class diagrams*. An appropriate notation for such a purpose is already offered by the UML: the standard [43] includes the *Object Constraint Language (OCL)*. We briefly point out the three major roles of OCL constraints within KeY:

- The KeY tool supports the *creation* of constraints. While a user is free in general to formulate any desired constraint, he or she can also take advantage of the automatic generation of constraints, a feature which is realised in the KeY tool by extending the CASE tool’s *design pattern instantiation* mechanism.
- The KeY tool supports the *formal analysis* of constraints. The relations between classes in the design imply relations between according constraints, which can be analysed regardless of any implementation.

- The KeY tool supports the *verification* of implementations with respect to the constraints. A theorem prover with interactive and automatic operation modes can check consistency of JAVA implementations with the given constraints.

These mechanisms, and their interaction with the features already provided by the underlying CASE tool, will be described in detail in this paper.

The KeY tool realises full integration of certain formal techniques into more widely spread techniques. Nevertheless, the usage of specification or verification facilities requires additional effort and skill, which has to be motivated. In the software industry, the “residual defect ratio” (the number of bugs that remain in the shipped product) normally lies between 0.5 and 5 defects per thousand lines of non-commented source code [33]. Whether this number justifies to undertake an extra effort or not depends on the damage caused by system failures. Application areas, where this damage is known to be particularly high, include: *safety critical* applications (e.g. railway switches), *security critical* applications (e.g. access control, electronic banking), *cost critical* applications (which, for example, run on a large number of non-administrated devices, such as phone cards), and *legally critical* applications (e.g. falling under digital signature laws).

Such applications are often intended to run on smart cards or similar devices. Therefore, JAVA CARD as the target language of the KeY tool, is highly *significant*. At the same time, the technical restrictions of JAVA CARD make verification of the full language *feasible*. We stress that KeY is not restricted to being used for the development of smart card applications, because many JAVA applications do not use features excluded by JAVA CARD. In general, the KeY tool is particularly valuable, whenever the minimisation of software defects is an important issue.

This article is organised as follows: Sect. 2 describes the general architecture of the KeY system. Different scenarios of using the system are discussed in Sect. 3. In Sect. 4 we introduce an example that is used throughout the rest of this paper. Sect. 5 describes the KeY-specific embedding of formal specifications into a UML-based design process. Then, the formal analysis of the relationship of such specifications to each other (Sect. 6), as well as to a given implementation (Sect. 7), is discussed. Sect. 8 then describes how the resulting proof obligations are processed by interactive and automated theorem proving. After a brief look on implementation issues (Sect. 9), we describe some case studies performed with KeY (Sect. 10). Finally, we summarise the current state of the KeY project (Sect. 11), and in Sect. 12 draw some conclusions. The paper is an updated, extended, and completely rewritten version of [2, 3].

## 2 Architecture of the KeY Tool

The KeY system is built on top of a commercial CASE tool. Integrating our system into an already existing tool has obvious advantages:

1. All features of the existing tool can be used and do not need to be reimplemented.
2. The software developer does not have to become familiar with a new design and development tool. Furthermore the developer is not required to change tools during development, everything that is needed is integrated into one tool.

A CASE tool that is well suited for our purposes has to be easily extensible and the extensions have to fit nicely into the tool providing a uniform user interface. We decided to use *Together Control Center* from TogetherSoft [55], in the following referred to as TogetherCC. Among all the tools on the market this one seems to be most suitable for our purposes. It has state-of-the-art development and UML support (including some very basic support for textual specifications) and can be extended in almost any possible way by JAVA modules—TogetherCC offers access to most of its “internals” by means of a JAVA open API (see Sect. 9). There is however no fundamental obstacle to adding the KeY extensions to other, similar CASE tools. Fig. 1 shows a screenshot of TogetherCC with KeY system extensions.

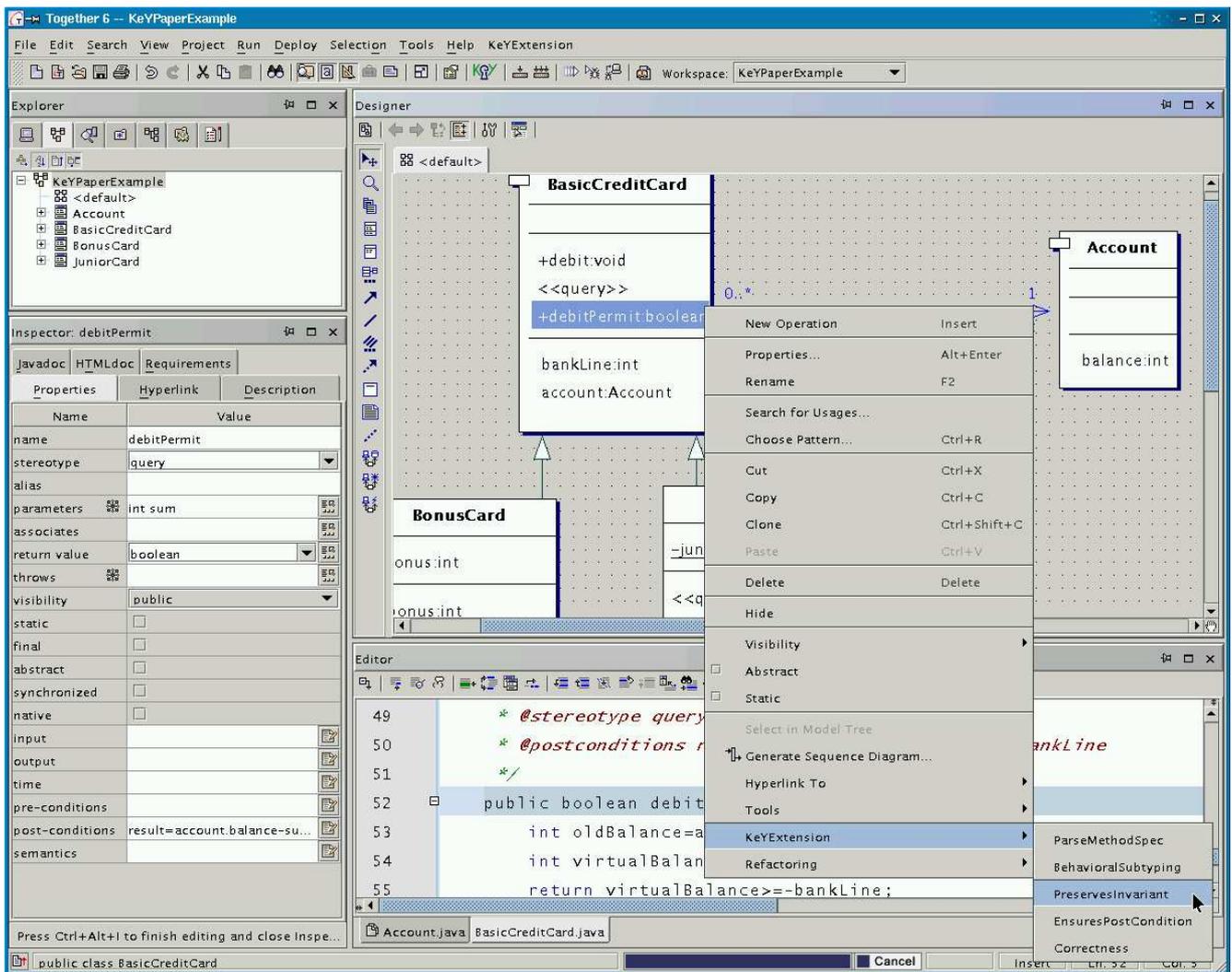
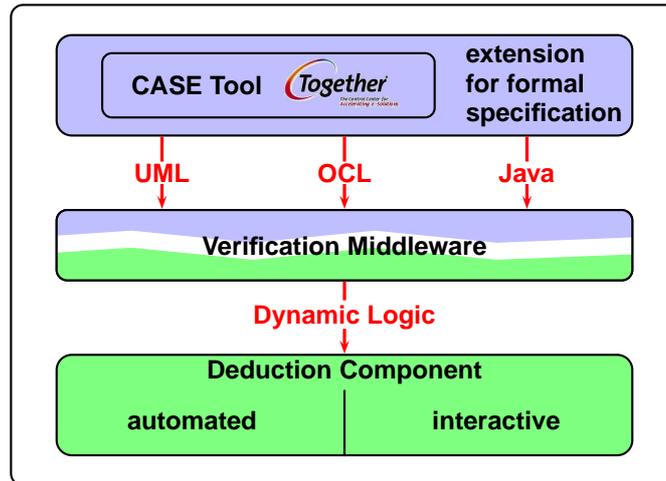


Figure 1. TogetherCC with Key system extensions

The architecture of the KeY system is shown in Fig. 2. In the following, we briefly describe the components and the interactions between them:

1. The *modelling component* (upper part in Fig. 2) consists of the CASE tool with extensions for formal specification. While the CASE tool already allows the software model to contain OCL specifications it does not have any support to create or process them (OCL specifications are simply handled as textual annotations). This is where the extension comes into play. It allows the user to create, process and prepare the OCL specifications (together with the model and its implementation) which can be later processed and passed to the deduction component. Manipulating OCL specifications is done by employing external programs and libraries [30, 23, 21] as well as using TogetherCC's pattern mechanism to instantiate specifications from OCL specification templates [8] (see also Sect. 5). The CASE tool itself provides all the functionality for UML modelling and project development and is responsible for most of the user interactions with the project.
2. The *verification middleware* is the link between the modelling and the deduction component. It translates the model (UML), the implementation (JAVA) and the specification (OCL) into JAVA Dynamic Logic proof



**Figure 2. The architecture of the KeY system**

obligations which are passed to the deduction component. JAVA Dynamic Logic is a program logic used by the KeY prover (deduction component), see Sect. 7. The verification component is also responsible for storing and managing proofs during the development process.

3. The *deduction component* is used to actually construct proofs for JAVA Dynamic Logic proof obligations generated by the verification component. It is an interactive verification system combined with powerful automated deduction techniques. All those components are fully integrated and work on the same data structures.

All components are implemented in JAVA and fully integrated with TogetherCC through its open API resulting in a uniform user interface. In addition, some components of the KeY tool can be used stand alone: the OCL to JAVA Dynamic Logic translator and the prover, see Sect. 9. Uniform implementation in JAVA makes the KeY tool portable. It should also be mentioned that all KeY system extensions can optionally be switched on and off in TogetherCC and thus it is the developer's decision to use them or not.

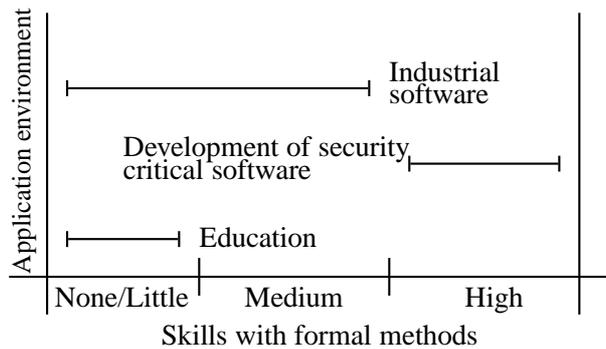
### 3 KeY Tool Use Cases

The KeY tool can be used in various scenarios by people who have widely differing skills with formal methods. In the present section we sketch three main scenarios for the KeY tool: KeY in the development process of industrial software without particular demands on security, KeY in the development of security critical software, and, finally, KeY in education and training. Fig. 3 shows the appropriate level of skill with formal methods for each environment.

As stated in the introduction, the aim of the KeY project is the integration of formal methods into the industrial software development process. Therefore, the most important target user group for the KeY tool are people who are not experts in formal methods. In many cases, users will even have reservations against formal methods.

As a consequence, to reach the goal of the project it is of crucial importance that the KeY tool allows for *gradual* verification, so that software engineers on any (including low) experience level with formal methods may benefit. In particular, the existence of full formal specification is *not* a prerequisite to make productive use of the KeY tool. The software engineer is free to determine the amount of formal methods he or she is willing to utilise.

The main use of KeY in an industrial environment is not necessarily full formal verification, but formal modelling. While implementations undergo frequent alterations and warrant formal verification only in exceptional



**Figure 3. KeY application environments and corresponding skill levels.**

cases, specifications are much less prone to changes. The benefits of formal and, hence, unambiguous specifications, are obvious. Moreover, our experience shows that many bugs are contained in specifications, and mere formalisation exhibits many of them.

To motivate users with few skills in formal methods or who have reservations against these, the KeY tool provides automatic support for creating formal specifications in several ways (see Sect. 5). For example, the KeY tool contains templates for often-needed OCL constraints which we call *KeY idioms*. These KeY idioms can easily be instantiated by the user and the corresponding OCL constraints are then generated automatically. In addition, instantiating *KeY patterns* [8], extensions of certain well-known design patterns, is another possibility to obtain a specification without having to know OCL syntax. Finally, an authoring tool for OCL constraints [30] offers assistance in generating specifications and helps to understand OCL constraints by rendering them automatically in natural language. It is currently being integrated into the KeY tool. We believe that the user support provided by the KeY tool can help to overcome reservations against formal methods and, hopefully, increases the willingness of developers to give formal methods a try.

A second possible field of application for the KeY tool is the development (including formal verification) of security critical software [42]. Here, the high risks that emanate from faulty implementations warrant the effort of formal verification. An interesting possibility is the provision of a formally verified reference implementation.

We do not claim that full formal software verification is possible without any skills in formal methods, so this application scenario pertains to formal methods experts. Since the KeY tool is an integrated system with a uniform user interface for modelling, specification, implementation, and verification of software, it can be used for the whole development process. This is an advantage of the KeY tool over conventional verification tools (for example, [9, 46, 45]). Without integration, several tools with typically incompatible interfaces have to be deployed to cover all steps from design to verification.

Another advantage of an integrated tool is that it enables efficient cooperation between developers whose skills in formal methods differs significantly (between none and high).

The final, but no less important, scenario we mention is the use of KeY in education and training. Its modular architecture allows certain components to be used stand-alone which is of advantage here.

The deduction component, for example, may be used stand-alone for teaching interactive theorem proving in first-order predicate logic or program logic. The authoring tool for OCL constraints and the OCL syntax checker are predestined to support teaching how to write formal specifications. But also the integrated tool can be used in a formal methods course. This has the important effect to emphasise that formal software development can be complementary rather than alien to conventional methods. We are currently developing a teaching unit for undergraduate level formal methods courses based on the KeY tool. The stand-alone components are also suitable for self-study, in particular, the authoring tool for OCL constraints.

## 4 Running Example

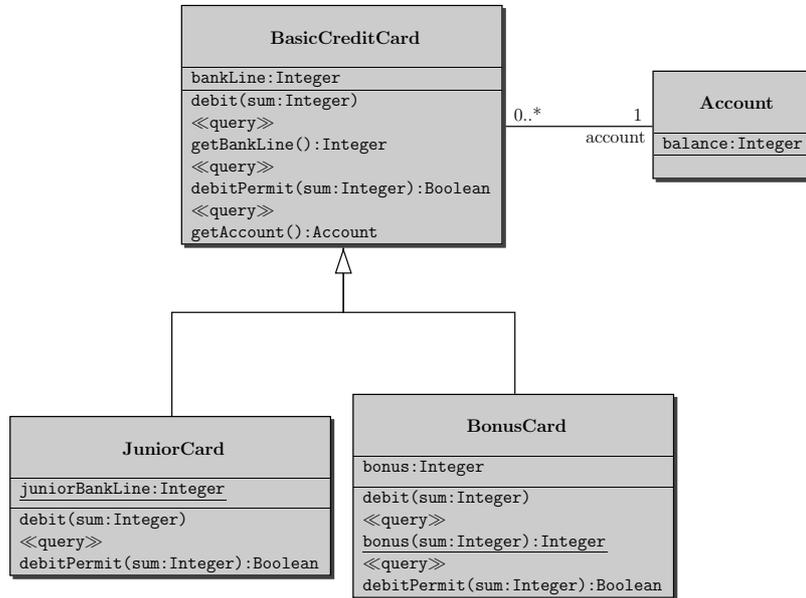


Figure 4. A simple credit card scenario

Throughout the paper we will use a running example, simple enough to concisely illustrate the KeY concepts and mechanisms. It consists of a credit card application, the class diagram of which is depicted in Fig. 4, in its simplest form. We assume some familiarity with reading UML class diagrams. As a quick introduction to UML we recommend [24]. The main feature of the diagram in Fig. 4 is the class *BasicCreditCard*. In the sequel we often use the abbreviation *BCC* for the name of this class. The class offers the operation `debit` to charge a certain `sum` to the credit card. This operation is only permitted as long as the credit limit `bankLine` is not exceeded. Further operations allow to query the attributes `bankLine` and `account` and permissibility of a `debit` operation. These operations do not modify the system state and are, therefore, labelled with the UML stereotype `<<query>>`. To make things a little bit more interesting we have included in the model two subclasses *JuniorCard*, which will have a stricter credit limit, and *BonusCard*, in which the `debit` operation in addition to its usual function may increase the bonus points stored in the `bonus` attribute depending on the result of the operation `bonus`. The *Account* class is modelled only rudimentary.

## 5 Embedding Formal Specification into the Design Process

### 5.1 Process Models

In industrial contexts of software development it became popular to take advantage of mainly graphical modelling notations such as the UML. Modelling notations vary in many aspects and are tailored to special purposes. It turns out that software developers have difficulties in practical application of modelling notations even if developers understood what the notations mean and for which purposes they should be used. To overcome this problem it is seen as best practice to follow certain guidelines according to which notation should be applied by

software developers in each phase of a project. Such guidelines are known as *process models*. Most of these (for example, Extreme Programming (XP) [10], Rational Unified Process (RUP) [35], Boehm's spiral model [15]) include the basic phases of software development: inception/analysis — design — implementation/test — deployment/maintenance. Modern process models tend to cycle through basic phases (iterative model) and to use at each development step all information available from artefacts created in previous development steps. The main concern of a process model is twofold:

1. To increase the productivity of the software developer.
2. To improve the quality of delivered software including the documentation so as to facilitate adaptations in the maintenance phase.

The main goal of the KeY project is the popularisation of formal methods in the industrial setting of software development. In a first step, formalisation is imported in the form of more precise models obtained through usage of the textual OCL. We focus on the application of OCL within class diagrams and we describe at which stage of a process model users can take advantage of OCL constraints.

## 5.2 OCL Constraints in the Domain Model

The result of the inception/analysis phase is a domain model of the target system. The domain model ought to give an overview over concepts identified and the most important relationships among them. For the sake of flexibility and changeability in later phases, the domain model should not be too detailed. On the other hand, certain properties of domain classes become evident already in the first phase. As an example, the class diagram in Fig. 4 does not contain all the information that we want to be included in our model. The meaning of a bank (credit) line may be clear to a human reader, but it is not mirrored in our model so far and thus no analysis tool, that goes beyond syntactic checks, could make use of it. It is exactly for the purpose to express information of this kind that the OCL has been included in UML (see [58] for a quick introduction and [43] for the current language specification of OCL). OCL allows to add invariants such as

```
context BasicCreditCard
inv: self.bankLine >= 0
inv: self.account.balance >= -c.bankLine
```

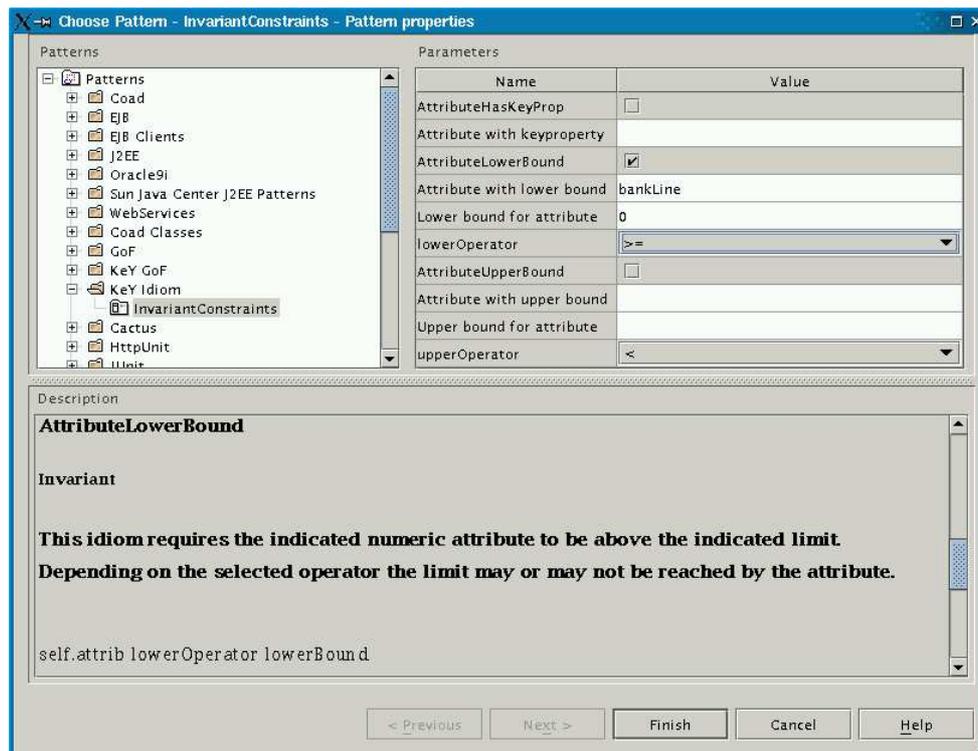
to the class *BasicCreditCard*. The intention is that the constraint should be satisfied in all system states, where the reserved variable `self` is implicitly quantified over all existing objects in the class *BasicCreditCard*.

Our experience with software developers working in an industrial context showed that they are often well aware of such constraints, but they are being documented in a rather informal way (if at all) so that no tool can make use of them [8].

Closer questioning reveals that software developers often have even constraints in mind but they are not used to formulate them in a formal language such as OCL. The KeY tool offers a simple, but powerful mechanism to start authoring formal constraints in a gentle way. Users can generate formal constraints without the immediate need to learn specific syntax or keywords.

## 5.3 KeY idioms

The KeY tool contains a library of predefined constraints called *KeY idioms*. Users may choose an idiom from the library and instantiate it to the current target model by setting idiom-specific parameters. The desired constraint is then automatically generated according to the context of the target model.



**Figure 5. Instantiation of KeY idiom AttributeLowerBound**

The first of the invariants given above for class *BasicCreditCard*, for example, can be generated from an idiom. Calling the KeY idiom library for class *BasicCreditCard* results in the dialog displayed in Fig. 5. Filling in the values as shown, returns exactly the first invariant from above.

Not only invariants, but also pre- and postconditions can be generated in this way. These are attached to operations instead of classes.

The library of idioms is extensible by means of a simple scripting language. Hence, experienced users (or the formal methods expert in a development team) can write project-specific idioms. In addition, the generated OCL constraints can, of course, be manually changed afterwards. Even the generation of constraint skeletons may be useful in some situations.

What the KeY tool does not (yet) offer is a facility to “reverse engineer” OCL constraints, i.e., to find out which idiom a given constrained was generated from. Reverse engineering could provide a correspondence between possibly large and complex OCL constraints and abstract, descriptive, and more understandable idioms. There are two possibilities to attain similar goals. First, one may simply change idioms in such a way that suitable comments (like the name of the current idiom) are generated in addition to OCL constraints. The second possibility is the usage of an authoring tool for simultaneous development of natural language and OCL constraints [30]. It is currently being integrated into the KeY tool. With the help of this tool, the example above is thus rendered in English (German and Swedish are supported as well):

“The following invariant holds for all *BasicCreditCards*:  
the *bankLine* of the *BasicCreditCard* is greater than or equal to zero.”

It is possible to make the textual rendering (“linearisation”) dependent of the type of an object. For example, one could write “bank line” instead of “bankLine” to enhance readability. While automatic translation from arbitrary

natural language texts into OCL is unrealistic, the other direction is feasible. Even if the result is not always stylistically elegant, it is quite helpful, for example, to have an automatic rendering in English after making changes to the OCL. This opens up the possibility of “single source” technology for informal and formal specifications. Without this, we foresee massive synchronisation and maintenance problems for formal specifications of non-trivial size.

KeY idioms help software developers to become faster acquainted with the syntax of a formal language. However, they provide only little help to decide *which* invariants and pre-/postconditions should be added to a model. It remains a task for the software developer to characterise the roles played by the classes in the domain model and the responsibilities they are assigned to. The KeY tool, as well as OCL, generally follow the design-by-contract approach. We refer to [41, Chapter 11] for heuristics to find useful constraints.

## 5.4 KeY patterns

In the design phase, the domain model is transformed into a more detailed model in order to meet new requirements which were intentionally ignored in the first phase. For our running example in Fig. 4, such a requirement could be to change the kind of a credit card dynamically, e.g. a customer applies for the bonus program of the bank and hence his current credit card of type *BasicCreditCard* turns into a card of type *BonusCard*.

The transformation of a sparsely structured domain model into a more fine grained and appropriate model during the design phase is often facilitated by the application of design patterns. In the running example, the new requirements are best captured by application of the Decorator [27, page 175 ff] pattern. In terms of the Decorator pattern, the type change from class *BasicCreditCard* to *BonusCard* for an object is seen as attaching additional responsibilities to this object. Technically, this is done by wrapping it in an object of type *CardDecorator*. The revised model after applying the Decorator pattern is displayed in Fig. 6.

The case tool TogetherCC offers special support for pattern application. When applying a pattern in TogetherCC, the user must indicate the roles of existing classes within the applied pattern. For example, the class *BasicCreditCard* is assigned to role `concrete component` and the class *BonusCard* to role `concrete decorator`. Based on these assignments, TogetherCC automatically generates further classes and even parts of the implementation according to the pattern definition. In the example, the class *CardDecorator* and the implementation of its method `getBankLine` are generated:

```
public int getBankLine () {
    return component.getBankLine ();
}
```

In the KeY tool, the idea of pattern-application support was extended. *KeY patterns* are based on the well-known GoF patterns [27], but they contain constraints written in OCL that formally characterise important aspects of application scenarios. KeY patterns are instantiated in the same way as other patterns in TogetherCC but the user selects in addition appropriate textual constraints which are instantiated<sup>1</sup> as well [8].

Like in the case of KeY idioms, the library of KeY patterns can be extended by the user. The predefined version of the KeY Decorator pattern supports the generation of formal constraints for rather complex properties such as “no object of *BonusCard* has an inner component that has the type *BonusCard*”. In terms of the Decorator pattern, this means that responsibilities can be attached to an object at most once. Here is a simpler example for a generated constraint: a postcondition of the operation `CardDecorator::getBankLine()` to ensure that the implementation and specification of this operation generated during the application of the KeY pattern match each other:

---

<sup>1</sup>The idea of instantiating textual constraints goes back at least to Syntropy [20] and the technique is used successfully in other contexts. One example is the proposed language description of UML 2.0 given in [40] where the process of instantiating constraints is called *stamp out mechanism*.

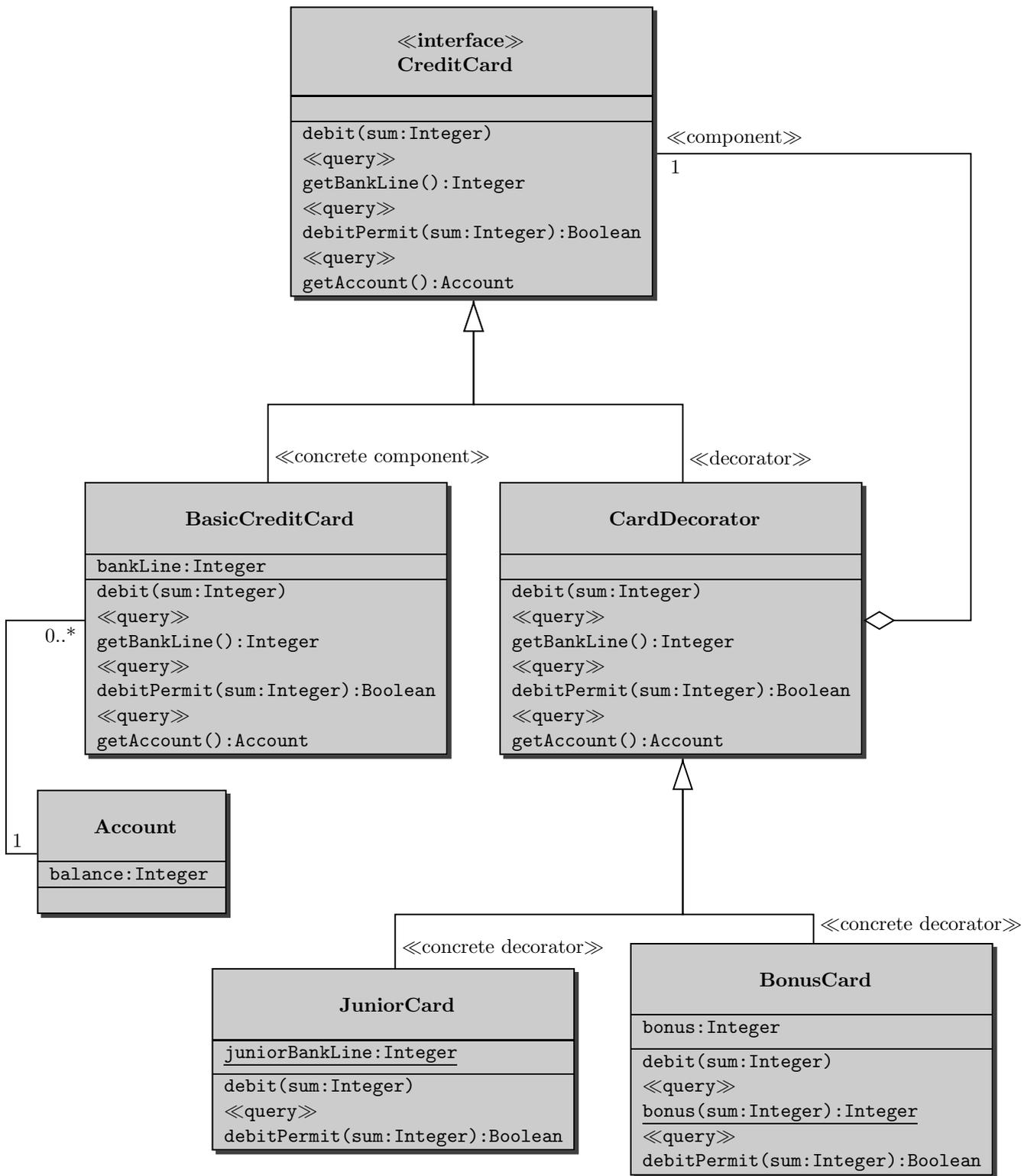


Figure 6. Credit card diagram with Decorator pattern

```
context CardDecorator::getBankLine():Integer
post: result = self.component.getBankLine()
```

The instantiation of KeY patterns facilitates the creation of a specification in two ways. As in case of idioms, the difficulties of using a formal language like OCL are hidden from the user. Even more important, however, is the support in obtaining a complete specification: KeY patterns extend well-known GoF patterns for many scenarios. Listing all the predefined constraints that are useful in a given context reminds the user of aspects that might have been forgotten in the specification so far. In some cases the constraints attached to a pattern might contradict each other, so that choosing all of them would result in an inconsistent design. The KeY tool does not automatically detect such clashes between chosen constraints. However, it provides possibilities for computer-assisted analysis of the resulting constraints (see Sect. 6).

## 6 Analysing Specifications

In this section we look at verification tasks that can be performed on the specification alone without reference to a possible or existing implementation. This is sometimes called *horizontal verification*. In the first subsection we describe what tasks are currently supported and in the following subsection we outline how these tasks are dealt with in the KeY system.

### 6.1 Proof Obligations

Formal modelling as it is supported by the KeY system is based on OCL constraints which allow to characterise precisely the relationship between classes, attributes, and operations. A very simple example for an OCL constraint was already given by the invariants in Sect. 5.2. Now, we proceed on discussing further examples and give an overview to other kinds of constraints expressible in OCL and the most important subtleties of the language OCL.

Formal modelling, as it is supported by the KeY system, is based on OCL constraints which allow to characterise precisely the relationship between classes, attributes, and operations. Simple examples for OCL constraints were already given in the previous section. Now, we proceed by discussing further examples and give an overview of the kinds of constraints expressible in OCL. We also discuss some subtleties of the language OCL. All constraints refer to our running example in Fig. 4.

The subclass *JuniorCard* of *BasicCreditCard* contains the class attribute `juniorBankLine`, i.e., the scope of this attribute is not individual objects but the whole class. The requirements we want this attribute to fulfil are described by these invariants:

```
context c:JuniorCard
inv: JuniorCard.juniorBankLine >= c.bankLine
inv: c.account.balance >= -juniorBankLine
inv: c.bankLine >= 0
```

For every junior card `c` the value of its `bankLine` attribute does not exceed the integer `juniorBankLine`, it is non-negative, and the account of card `c` should not drop below `-juniorBankLine`.

Note that we take advantage of declaring a local variable `c` and use it instead of `self`, which results in a more readable OCL constraint. Besides invariants, OCL also allows to add pre- and postconditions to operations.

```

context c:BasicCreditCard::debit(sum:Integer)
pre:    debitPermit(sum)
post:   c.account.balance = c.account.balance@pre - sum
pre:    not debitPermit(sum)
post:   c.account.balance = c.account.balance@pre
pre:    true
post:   c.bankLine = c.bankLine@pre

```

The OCL construct @pre is only applicable in postconditions and causes the feature it decorates to refer to its value before the start of the operation. It is admissible to use the operation debitPermit in the constraints, because it has been specified as a query in the class diagram (Fig. 4).

The OCL language definition is not quite precise regarding the meaning of multiple pre-/postcondition pairs. We use a constraint with  $n$  pre-/postcondition pairs as a convenient shorthand for  $n$  constraints with respectively one pair each. Furthermore, multiple pre-/postcondition pairs can be equivalently translated into a constraint with just one postcondition and precondition *true*. For the constraint above the following translation could be used:

```

context c:BasicCreditCard::debit(sum:Integer)
pre:    true
post:   c.bankLine = c.bankLine@pre and
         if debitPermit@pre(sum)
         then   c.account.balance = c.account.balance@pre - sum
         else   c.account.balance = c.account.balance@pre

```

The OCL offers the predefined variable result to refer to the possible return value of an operation. This is particularly useful for query operations which are fully specified by fixing their return value.

```

context c:BasicCreditCard::debitPermit(sum:Integer):Boolean
pre:    true
post:   result = (c.account.balance - sum >= -c.bankLine)

```

Note that we need not use the @pre suffix for attributes in this statement, since we know by the query property of debitPermit that pre and post values coincide.

Pre- and postconditions are viewed, as is usual in the design-by-contract paradigm [41, Chapter 11], as two parts of a contract. If the client calling an operation makes sure that its precondition is satisfied, then the supplier of the operation guarantees that it terminates, and upon termination its postcondition holds.

Once a class diagram is supplemented with OCL invariants, pre- and postconditions, it is useful to analyse mutual dependencies among them. The simplest requirement, called *structural subtyping*, is to check whether the conjunction of all invariants of a subclass implies all invariants of its superclass. A quick glance at the above invariants shows that this is true for the subclass *JuniorCard* of *BasicCreditCard* (the invariants of the latter were given in Sect. 5.2).

A design methodology might require that operations preserve invariants, i.e., that for every operation *op* of class *C* the precondition of *op* together with the invariant of *C* logically implies the invariant in the successor state. If there is more than one precondition/postcondition pair to an operation this implication has, of course, to be proved for every pair. The operation debit in class *BasicCreditCard* does indeed preserve the invariant `c.account.balance >= -bankLine`. The computation establishing this is straightforward and done fully automatic with the KeY tool.

Returning to Fig. 4, the debit operation in subclass *BonusCard* has additional functionality—it is supposed to increase the number of bonus points by `bonus(sum)` yielding the constraint:

```

context c:BonusCard::debit(sum:Integer)
pre: debitPermit(sum)
post: c.account.balance = c.account.balance@pre - sum and
        bonus = bonus@pre + bonus(sum)
pre: not debitPermit(sum)
post: c.account.balance = c.account.balance@pre and
        bonus = bonus@pre
pre: true
post: c.bankLine = c.bankLine@pre

```

This allows us to illustrate another condition required by some design methodologies, called *behavioural subtyping*, or sometimes also the Liskov principle. It applies when an operation occurs in a *class*<sub>1</sub> with precondition *pre*<sub>1</sub> and postcondition *post*<sub>1</sub> as well as in a subclass *class*<sub>2</sub> of *class*<sub>1</sub> with precondition *pre*<sub>2</sub> and postcondition *post*<sub>2</sub>.

Behavioural subtyping requires that the implications  $pre_1 \rightarrow pre_2$  and  $post_2 \rightarrow post_1$  be logically valid. These requirements can be justified when one accepts that the subclass relation entails: any object of *class*<sub>2</sub> can be used in any circumstances that an object from *class*<sub>1</sub> could be used. It is a trivial observation that the behavioural subtyping regime holds true for *BasicCreditCard* and its subclass *BonusCard* with respect to the operation *debit*.

Behavioural subtyping and the preservation of invariants are supported by the KeY tool.

## 6.2 Proving Obligations

When the user selects either the structural subtyping or invariant preservation task from the KeY extension menu within TogetherCC a verification condition, formalised in Dynamic Logic, is generated and passed on to KeY's deduction system. To this end, the information contained in the UML class diagram as well as the OCL constraints have to be translated into Dynamic Logic.

This translation fixes a particular semantics for UML/OCL. Quite a number of papers ([17, 22, 25] to name just a few) have been published doing the same, fixing a formal semantics by translating UML diagrams into some known formal system. Despite the often voiced need of a precise semantics for UML the informal semantics description did not lead to major discrepancies (at least for class diagrams and not touching issues of the meta-model). For OCL the situation was a less satisfactory. Most of the trouble arose from its meta-model and the integration into the rest of UML. These issues have been rigorously resolved in the submission [44] for the UML2.0 which is awaiting approval. Its formal semantics is based on the PhD thesis [49].

In the following we describe our translation from UML class diagrams with OCL constraints into typed Dynamic Logic by way of example. A full account can be found in the paper [38]. Summaries of parts of it were published as [51, 12].

The first step in the translation is to fix the vocabulary to be used on the logical side. This is straight forward: for every class in the UML model there will be a type in the logic, built-in OCL types are mapped onto corresponding abstract data types. Attributes, associations and query operations are mapped into functions in the obvious manner. Class attributes (e.g., `juniorBankLine`) turn into constants. We gloss over some details like naming and disambiguating conventions, except for the remark that unlabelled association ends get by default the name of the class they are attached to (e.g.,  $bcc : Account \rightarrow Set_{BCC}$ ). A selection of the vocabulary for the class diagram in Fig. 4 is shown in Fig. 7. The second invariant in Sect. 5.2 reads in logic as follows:

$$\forall x: BCC. (x.account.balance \geq -x.bankLine)$$

We decided to stick also on the logical side with the *dot notation* as opposed to the traditional notation using brackets, in which the above formula would read  $\forall x: BCC. (balance(account(x)) \geq -bankLine(x))$ . This way

**Types:**

<i>category</i>	<i>names</i>
model types	<i>BCC, JuniorCard, BonusCard, Account</i>
OCL basic types	<i>Integer, Boolean,...</i>
OCL collection types	<i>Set<sub>BCC</sub>, Sequence<sub>Integer</sub>,...</i>

**Functions:**

<i>name</i>	<i>signature</i>
<i>bankLine</i>	<i>BCC → Integer</i>
<i>balance</i>	<i>Account → Integer</i>
<i>bonus</i>	<i>BonusCard → Integer</i>
<i>juniorBankLine</i>	<i>Integer</i>
<i>account</i>	<i>BCC → Account</i>
<i>bcc</i>	<i>Account → Set<sub>BCC</sub></i>
<i>bankLine()</i>	<i>BCC → Integer</i>
<i>debitPermit()</i>	<i>BCC × Integer → Boolean</i>
⋮	

**Figure 7. Vocabulary for the simple credit card scenario in Fig. 4**

it is possible to keep track of OCL constraints even when using the interactive theorem prover, see Fig. 8. To establish the structural subtyping property for the subclass *JuniorCard* of *BasicCreditCard* the following formula has to be proved to be a tautology:

$$\begin{aligned} \forall c: JuniorCard. \quad & (juniorBankLine \geq c.bankLine \wedge \\ & c.account.balance \geq -juniorBankLine \wedge \\ & c.bankLine \geq 0 \\ & \rightarrow \\ & c.account.balance \geq -c.bankLine) \end{aligned}$$

Let us look at a new invariant for the class *Account* in Fig. 4:

**context** a:Account

**inv:** a.bcc->select(c | c.bankLine > 1000)->size <= 10

The same account may be used by different credit cards. The constraint says that at most 10 credit cards with credit limit exceeding 1000 can share the same account. Its translation into first-order logic reads

$$\forall a: Account. (a.bcc.select_E.size \leq 10)$$

$select_E : Set_{BCC} \rightarrow Set_{BCC}$  is a new function symbol depending on the expression  $E = select(c | c.bankLine > 1000)$ . If we use  $\emptyset$  (of type  $Set_{BCC}$ ) and  $insert$  (of type  $BCC \times Set_{BCC} \rightarrow Set_{BCC}$ ) as the constructors of the abstract data type  $Set_{BCC}$  then the definition of  $select_E$  reads

$$\begin{aligned} \emptyset.select_E &= \emptyset \\ c.bankline > 1000 &\rightarrow c.insert(s).select_E = c.insert(s.select_E) \\ c.bankline \leq 1000 &\rightarrow c.insert(s).select_E = s.select_E \end{aligned}$$

All these formulas are passed on to the deduction system. The translation of the @pre construct requires more than first-order logic and will be explained in Sect. 7.3.

## 7 Verifying Correctness of Implementations

Besides supporting the analysis of a specification, KeY provides functionality for checking the correctness of a JAVA implementation with respect to a given UML/OCL specification.

In particular, KeY allows (1) to prove that after running a method, the method’s post-condition holds, and (2) to prove that a method preserves a class invariant (program correctness requires that all public methods preserve all invariants).

### 7.1 Dynamic Logic

We use an instance of Dynamic Logic (DL) [31, 32, 39, 47]—which can be seen as an extension of Hoare logic—as the logical basis of the KeY system’s software verification component. Deduction in DL is based on symbolic program execution and simple program transformations and is, thus, close to a programmer’s understanding of JAVA. DL is used in the software verification systems KIV [9] and VSE [34] for (artificial) imperative programming languages. More recently, the KIV system supports also a fragment of the JAVA language [52]. In both systems, DL was successfully applied to verify software systems of considerable size.

DL can be seen as a modal logic with a modality  $\langle p \rangle$  for every program  $p$  (we allow  $p$  to be any sequence of legal JAVA CARD statements);  $\langle p \rangle$  refers to the successor worlds (called states in the DL framework) that are reachable by running the program  $p$ . In standard DL there can be several such states (worlds) because the programs can be non-deterministic; but here, since JAVA programs are deterministic, there is exactly one such world (if  $p$  terminates) or there is no such world (if  $p$  does not terminate). The formula  $\langle p \rangle \phi$  expresses that the program  $p$  terminates in a state in which  $\phi$  holds. A formula  $\phi \rightarrow \langle p \rangle \psi$  is valid if for every state  $s$  satisfying pre-condition  $\phi$  a run of the program  $p$  starting in  $s$  terminates, and in the terminating state the post-condition  $\psi$  holds.

Thus, the formula  $\phi \rightarrow \langle p \rangle \psi$  is similar to the Hoare triple  $\{\phi\}p\{\psi\}$ . But in contrast to Hoare logic, the set of formulas of DL is closed under the usual logical operators: In Hoare logic, the formulas  $\phi$  and  $\psi$  are pure first-order formulas, whereas in DL they can contain programs. DL allows to involve programs in the descriptions  $\phi$  resp.  $\psi$  of states. For example, using a program, it is easy to specify that a data structure is not cyclic, which is impossible in pure first-order logic. Also, all JAVA constructs are available in DL for the description of states (including while loops and recursion). It is, therefore, not necessary to define an abstract data type *state* and to represent states as terms of that type; 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.

### 7.2 Syntax of JAVA CARD DL

As said above, a dynamic logic is constructed by extending some non-dynamic logic with a modal operator  $\langle \cdot \rangle$ . In addition, we use the dual operator  $[\cdot]$ , for which  $[p]\phi \equiv \neg \langle p \rangle \neg \phi$ . The non-dynamic base logic of our DL is typed first-order predicate logic. We do not describe in detail what the types of our logic are (basically they are identical to the JAVA types) nor how exactly terms and formulas are built. The definitions can be found in [11]. Note that terms (which we often call “logical terms” in the following) are different from JAVA expressions; the former never have side effects.

In order to reduce the complexity of the programs occurring in formulas, we introduce the notion of a *program context*. The context can consist of any JAVA CARD program, i.e., it is a sequence of class and interface definitions. Syntax and semantics of JAVA CARD DL formulas are then defined with respect to a given context; and the programs in JAVA CARD DL formulas are assumed not to contain class definitions.

The programs in JAVA CARD DL formulas are basically executable JAVA CARD code. The verification of a given program can be thought of as *symbolic code execution*. As will be detailed below, each rule of the calculus for JAVA CARD DL specifies how to execute one particular JAVA statement, possibly with additional restrictions.

When a loop or a recursive method call is encountered, it is necessary to perform induction over a suitable data structure.

Given that we follow the symbolic execution paradigm for verification, it is evident that a certain amount of runtime infrastructure must be represented in JAVA CARD DL. It would be possible, but clumsy and inefficient, to achieve this by purely logical means. Therefore, we introduced an additional construct for handling of method calls that is not available in plain JAVA CARD. Methods are invoked by syntactically replacing the call by the method's implementation. To handle the `return` statement in the right way, it is necessary (a) to record the object field or variable  $x$  that the result is to be assigned to, and (b) to mark the boundaries of the implementation *body* when it is substituted for the method call. For that purpose, we allow statements of the form `method-frame( $x$ ){body}` to occur in DL programs. Note, that this is a “harmless” extension because the additional construct is only used for proof purposes and never occurs in the verified JAVA CARD programs.

### 7.3 Proof Obligations

Let us now turn to the translation of OCL constraints into JAVA CARD DL proof obligations. To prove that a method  $m(\mathbf{arg}_1, \dots, \mathbf{arg}_n)$  of class  $\mathbf{C}$  satisfies a pre-/post-condition pair, the OCL conditions are first translated into first-order formulas  $pre(\mathit{self}, \mathit{arg}_1, \dots, \mathit{arg}_n)$  and  $post(\mathit{self}, \mathit{arg}_1, \dots, \mathit{arg}_n)$ , respectively (as described in Section 6). From these formulas, KeY constructs the JAVA CARD DL proof obligation

$$pre(\mathit{self}, \mathit{arg}_1, \dots, \mathit{arg}_n) \rightarrow \langle \mathit{self}.m(\mathit{arg}_1, \dots, \mathit{arg}_n); \rangle post(\mathit{self}, \mathit{arg}_1, \dots, \mathit{arg}_n) ,$$

where now  $\mathit{self}$  and  $\mathit{arg}_1, \dots, \mathit{arg}_n$  are program variables, which are implicitly universally quantified w.r.t. their initial value.

For example, the first pre-/postcondition pair for the `debit` operation from Section 6 is transformed into

$$\mathbf{c}.debitPermit(\mathit{sum}) = \text{TRUE} \rightarrow \langle \mathbf{c}.debit(\mathit{sum}); \rangle (\mathbf{c}.account.balance = \mathbf{c}.account.balance@pre - \mathit{sum}) .$$

The call to the operation `debit` is translated into the JAVA CARD DL program “`c.debit(sum);`” that appears within angle brackets in the above formula. Furthermore  $balance@pre$  is a new function symbol with the same signature as  $balance$ . There are several possibilities to ensure that the function  $balance@pre$  has the intended semantics, see [6] for a detailed account. The simplest way is by adding a definition of  $balance@pre$  to the above formula:

$$\forall x: Account. x.balance@pre = x.balance \wedge (\mathbf{c}.debitPermit(\mathit{sum}) = \text{TRUE} \rightarrow \langle \mathbf{c}.debit(\mathit{sum}); \rangle (\mathbf{c}.account.balance = \mathbf{c}.account.balance@pre - \mathit{sum})) .$$

Similarly, to prove that a method  $m(\mathbf{arg}_1, \dots, \mathbf{arg}_n)$  preserves an invariant, the proof obligation

$$(inv(\mathit{self}) \wedge pre(\mathit{self}, \mathit{arg}_1, \dots, \mathit{arg}_n)) \rightarrow \langle \mathit{self}.m(\mathit{arg}_1, \dots, \mathit{arg}_n); \rangle inv(\mathit{self})$$

is constructed, where  $inv(\mathit{self})$  is the first-order translation of the invariant.

### 7.4 Deductive Calculus for Proving Obligations

Since there is (at least) one rule in our JAVA CARD DL calculus for each JAVA CARD programming construct, we cannot present all rules in this paper. Instead we describe some important rules, which are exemplary for their respective class of rules.

### 7.4.1 The Active Statement in Program

The rules of our calculus operate on the first *active* command  $p$  of a program  $\pi p \omega$ . The non-active prefix  $\pi$  consists of an arbitrary sequence of opening braces “{”, labels, beginnings “try{” of try-catch-finally blocks, and beginnings “method-frame(. . .){” of method invocation blocks. The prefix is needed to keep track of the blocks that the (first) active command is part of, such that the abruptly terminating statements `throw`, `return`, `break`, and `continue` can be handled appropriately.<sup>2</sup> The postfix  $\omega$  denotes the “rest” of the program, i.e., everything except the non-active prefix and the part of the program the rule operates on. For example, if a rule is applied to the following JAVA block operating on its first active command “`i=0;`”, then the non-active prefix  $\pi$  and the “rest”  $\omega$  are the indicated parts of the block:

$$\underbrace{l:\{\text{try}\{ i=0; j=0; \} \text{finally}\{ k=0; \}\}}_{\pi}$$

### 7.4.2 The Assignment Rule and Handling State Updates

In JAVA (like in other object-oriented programming languages), different object variables can refer to the same object. This phenomenon, called aliasing, causes serious difficulties for handling of assignments in a calculus for JAVA CARD DL.

For example, whether or not a formula “`o1.a = 1`” still holds after the (symbolic) execution of the assignment “`o2.a = 2;`”, depends on whether or not `o1` and `o2` refer to the same object.

Therefore, JAVA assignments cannot be symbolically executed by syntactic substitution. Solving this problem naively—by doing a case split if the effect of an assignment is unclear—is inefficient and leads to heavy branching of the proof tree.

In our JAVA CARD DL calculus we use a different solution. It is based on the notion of *updates*. These (state) updates are of the form  $\langle loc := val \rangle$  and can be put in front of any formula. The semantics of  $\langle loc := val \rangle \phi$  is the same as that of  $\langle loc = val; \rangle \phi$ . The difference between an update and an assignment is syntactical. The expressions  $loc$  and  $val$  must be simple in the following sense:  $loc$  is (a) a program variable `var`, or (b) a field access `obj.attr`, or (c) an array access `arr[i]`; and  $val$  is a logical term (that is free of side effects). More complex expressions are not allowed in updates.

The syntactical simplicity of  $loc$  and  $val$  has semantical consequences. In particular, computing the value of  $val$  has no side effects. The KeY system uses special simplification rules to compute the result of applying an update to logical terms and formulas not containing programs. Computing the effect of an update to a program  $p$  (and a formula  $\langle p \rangle \phi$ ) is delayed until  $p$  has been symbolically executed using other rules of the calculus. Thus, case distinctions are not only delayed but they can often be avoided completely, because (a) updates can be simplified *before* their effect is computed and (b) their effect is computed when a maximal amount of information is available (namely after the symbolic execution of the program).

The assignment rule now takes the following form ( $\mathcal{U}$  stands for an arbitrary sequence of updates):

$$\frac{\Gamma \vdash \mathcal{U} \langle loc := val \rangle \langle \pi \omega \rangle \phi}{\Gamma \vdash \mathcal{U} \langle \pi loc = val; \omega \rangle \phi}$$

That is, it just adds the assignment to the list of updates  $\mathcal{U}$ . Of course, this does not solve the problem of computing the effect of the assignment. This problem is postponed and solved by rules for simplifying updates.

This assignment rule can, of course, only be used if the expression  $val$  is a logical term. Otherwise, other rules have to be applied first to evaluate  $val$  (as that evaluation may have side effects). For example, these rules replace

<sup>2</sup>In DL versions for simple artificial programming languages, where no prefixes are needed, any formula of the form  $\langle pq \rangle \phi$  can be replaced by  $\langle p \rangle \langle q \rangle \phi$ . In our calculus, splitting of  $\langle \pi pq \omega \rangle \phi$  into  $\langle \pi p \rangle \langle q \omega \rangle \phi$  is not possible (unless the prefix  $\pi$  is empty) because  $\pi p$  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  $\langle \pi pq \omega \rangle \phi$ .

the formula  $\langle x = ++i; \rangle \phi$  with  $\langle i = i+1; x = i; \rangle \phi$ . One can view these rules as on-the-fly program transformations. Their effect is always local and fairly obvious, so that the user's understanding of the proof is not obfuscated.

### 7.4.3 The Rule for if-else

As a first example for a rule with more than one premiss, we present the rule for the if statement.

$$\frac{\Gamma, \mathcal{U}(b = \text{TRUE}) \vdash \mathcal{U}\langle \pi p \omega \rangle \phi \quad \Gamma, \mathcal{U}(b = \text{FALSE}) \vdash \mathcal{U}\langle \pi q \omega \rangle \phi}{\Gamma \vdash \mathcal{U}\langle \pi \text{if}(b) p \text{ else } q \omega \rangle \phi}$$

The two premisses of this rule correspond to the two cases of the if statement. The semantics of rules is that, if all the premisses are true in a state, then the conclusion is true in that state. In particular, if the premisses are valid, then the conclusion is valid.

In practice, rules are applied from bottom to top: From the old proof obligation, new proof obligations are derived. As the if rule demonstrates, applying a rule from bottom to top corresponds to a symbolic execution of the program to be verified.

### 7.4.4 The Rule for while Loops

The following rule “unwinds” while loops. Its application is the prerequisite for symbolically executing the loop body. These “unwind” rules allow to handle while loops if used together with induction schemata for primitive and user defined data types.

$$\frac{\Gamma \vdash (\langle \pi \text{if}(c) l' : \{ l'' : \{ p' \} l_1 : \dots : l_n : \text{while}(c) \{ p \} \} \omega \rangle \phi)}{\Gamma \vdash (\langle \pi l_1 : \dots : l_n : \text{while}(c) \{ p \} \omega \rangle \phi)} \quad (\text{R1})$$

where

- $l'$  and  $l''$  are new labels,
- $p'$  is the result of (simultaneously) replacing in  $p$ 
  - (a) every `break`  $l_i$  (for  $1 \leq i \leq n$ ) and every `break` (with no label) that has the `while` loop as its target by `break`  $l'$ , and
  - (b) every `continue`  $l_i$  (for  $1 \leq i \leq n$ ) and every `continue` (with no label) that has the `while` loop as its target by `break`  $l''$ .<sup>3</sup>

The list “ $l_1 : \dots : l_n$ ” usually is empty or has only one element, but in general a loop can have more than one label.

In the “unwound” instance  $p'$  of the loop body  $p$ , the label  $l'$  is the new target for `break` statements and  $l''$  is the new target for `continue` statements, which both had the `while` loop as target before. This results in the desired behaviour: `break` abruptly terminates the whole loop, while `continue` abruptly terminates the current instance of the loop body.

A `continue` with or without label is never handled by a rule directly, because it can only occur in loops, where it is always transformed into a `break` by the loop rules.

---

<sup>3</sup>The target of a `break` or `continue` statement with no label is the loop that immediately encloses it.

### 7.4.5 The Rules for try/throw

The following rules allow to handle **try-catch-finally** blocks and the **throw** statement. These are simplified versions of the actual rules that apply to the case where there is exactly one **catch** clause and one **finally** clause.

$$\frac{\Gamma \vdash \text{instanceof}(exc, T) \quad \Gamma \vdash (\langle \pi \text{ try}\{e=exc; q\} \text{ finally}\{r\} \omega \rangle \phi)}{\Gamma \vdash (\langle \pi \text{ try}\{\text{throw } exc; p\} \text{ catch}(T e)\{q\} \text{ finally}\{r\} \omega \rangle \phi)} \quad (\text{R2})$$

$$\frac{\Gamma \vdash \neg \text{instanceof}(exc, T) \quad \Gamma \vdash (\langle \pi r; \text{throw } exc; \omega \rangle \phi)}{\Gamma \vdash (\langle \pi \text{ try}\{\text{throw } exc; p\} \text{ catch}(T e)\{q\} \text{ finally}\{r\} \omega \rangle \phi)} \quad (\text{R3})$$

The predicate  $\text{instanceof}(exc, T)$  has the same semantics as the **instanceof** operator in **JAVA**. It evaluates to true if the value of  $exc$  is assignable to a program variable of type  $T$ , i.e., if its dynamic type is a sub-type of  $T$ .

Rule (R2) 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 (R3) applies.

## 8 Interactive and Automated Proof Construction

### 8.1 Taclets

Most existing interactive theorem provers are “tactical theorem provers”. The tactics for which these systems are named are programs which act on the proof tree, mostly by many applications of primitive rules, of which there is a small, fixed set. The user constructs the proof by selecting the tactics to run. Writing a new tactic for a certain purpose, e.g. to support a new data type theory requires expert knowledge of the theorem prover.

In the KeY prover, both tactics and primitive rules are replaced by the *taclet* concept.<sup>4</sup> A taclet combines the logical content of a sequent rule with pragmatic information that indicates when and for what it should be used. In contrast to the usual fixed set of primitive rules, taclets can easily be added to the system. They are formulated as simple pattern matching and replacement schemas. For instance, a typical taclet might read as follows:

find (b  $\rightarrow$  c  $\implies$ ) if (b  $\implies$ ) replacewith (c  $\implies$ ) heuristics(simplify)

This means that an implication  $b \rightarrow c$  on the left side of a sequent may be replaced by  $c$ , if the formula  $b$  also appears on the left side of that sequent. Apart from this “logical” content, the keyword **find** indicates that the taclet will be attached to the implication and not to the formula  $b$  for interactive selection, see Sect. 8.3. The clause **heuristics(simplify)** indicates that this rule should be part of the heuristic named **simplify**, meaning that it should be applied automatically whenever possible if that heuristic is activated, see Sect. 8.4.

While taclets can be more complex than the typically minimalistic primitive rules of tactical theorem provers, they do not constitute a tactical programming language. There are no conditional statements, no procedure calls and no loop constructs. This makes taclets easier to understand and easier to formulate than tactics. In conjunction with an appropriate mechanism for application of heuristics, they are nevertheless powerful enough to permit interactive theorem proving in a convenient and efficient way [29].

In principle, nothing prevents one from formulating a taclet that represents an unsound proof step. It is possible, however, to generate a first-order proof obligation from a taclet, at least for taclets not involving DL. If that formula can be proven using a restricted set of “primitive” taclets, then the new taclet is guaranteed to be a correct derived rule. As for the primitive taclets for handling **JAVA** programs in DL, it is possible to show their correctness using the Isabelle formalisation of **JAVA** by Oheimb [56, 57].

<sup>4</sup>Taclets were introduced under the name *schematic theory specific rules (STSR)* by Habermatz [29].

## 8.2 Proof Visualisation

The KeY prover window (see Fig. 8) consists of two panes, the left of which has three tabs. The tab called *Proof* contains a tree representing the current proof state. The nodes of the tree correspond to sequents (goals) at different proof stages. One can click on any node to see the corresponding sequent and the rule that was applied on it in the following proof step (except when the node is a leaf). Leaf nodes in open proof branches are coloured red, whereas leaves of closed branches are coloured green. The tab named *Goals* lists the open proof goals. By clicking on any goal, one can change the active goal that is displayed in the right pane. When an active goal is open, one can work towards its closure by applying proof rules interactively or by activating automated proof search. The KeY prover allows the user to work on several proof obligations simultaneously. The third tab, named *Proof obligations*, keeps track of all currently open proofs and lets the user switch between them.

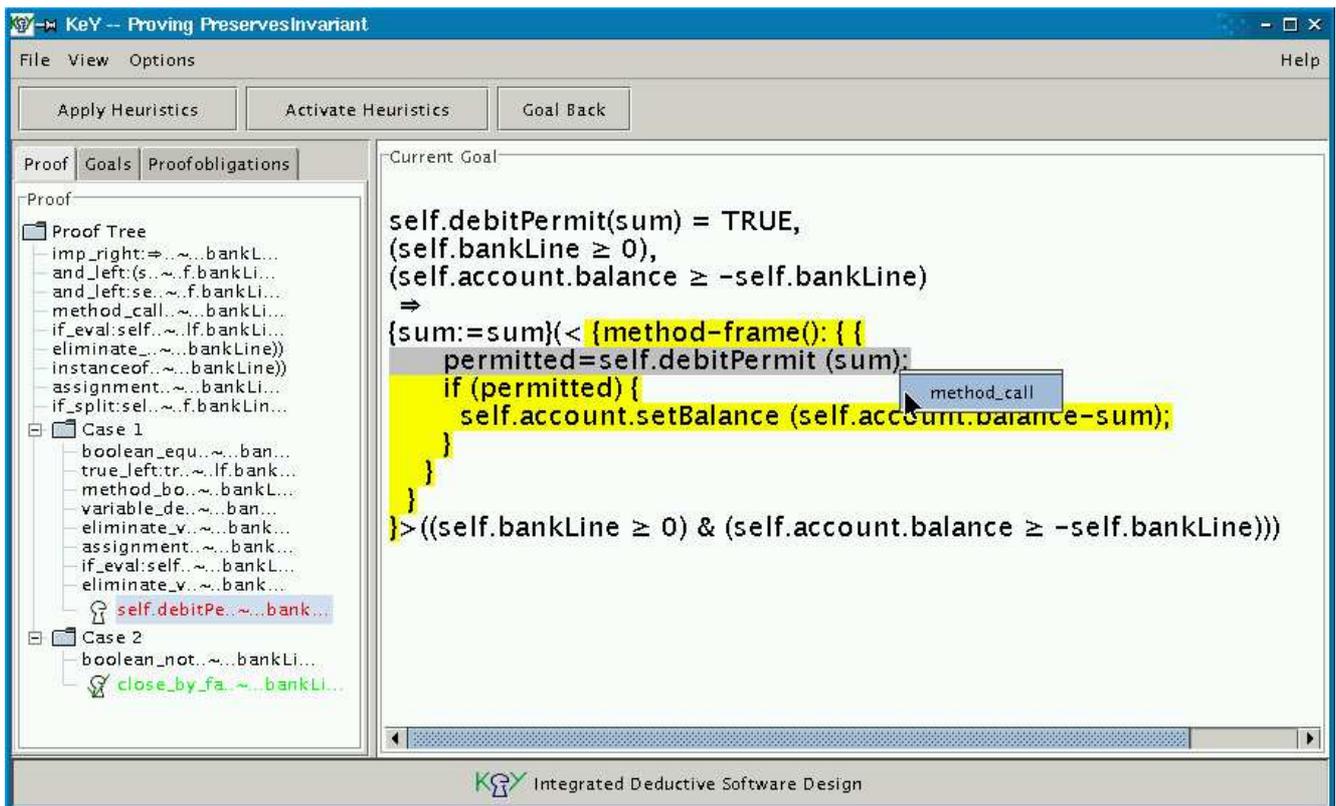


Figure 8. The KeY prover window.

## 8.3 Support for Interactive Proof Construction

Depending on where the mouse pointer is moved, one sub-formula or sub-term of the goal, the *focus term*, is highlighted (for example, in Fig. 8, the diamond operator and the program it contains is the focus term). More precisely, a term is put into focus by pointing at its top-level symbol. Then, pressing the left mouse button displays a list of all proof rules currently applicable to the focus term (in the example, only the method call rule is applicable). One of these can be selected and applied interactively, thus generating a new proof goal.

## 8.4 Automated Proof Search

Automated proof search is performed by applying “heuristics” which can be seen as a collection of rules suited for a certain task. For example, the heuristic `simplify_boolean` contains rules to simplify boolean expressions. The user can activate and de-activate heuristics depending on the state of the proof and goal he or she wants to tackle next. And the automatic application of heuristics can easily be switched on and off during proof construction.

## 9 Implementation Issues

From the user perspective, the KeY tool is an extension of the commercial case tool TogetherCC produced by TogetherSoft. The open API of TogetherCC allows the KeY tool to add items to the CASE tool’s contextual menus for classes, methods, etc. The API also makes it possible to modify the currently open UML model, for instance, during pattern instantiation (see Sect. 5).

### 9.1 Used Technology

The KeY tool is implemented in the JAVA programming language. This choice has several advantages, besides the obvious one of portability. Using the JAVA language makes it easy to link the KeY tool to TogetherCC, which is also written in JAVA. More generally, JAVA is well suited for interaction with other tools, written in JAVA or not. In particular, the imperative nature of the language leads to a comparatively natural native code interface, in contrast to the logic or functional programming languages often preferred for deduction purposes. JAVA was also a good choice for the construction of the graphical user interface, which is an important aspect of the KeY tool. Finally, previous experiments with both interactive [29] and automated [28] theorem proving have shown that the advantages of the language outweigh the additional effort of implementing term data structures, unification, etc.

A simple form of parametric genericity is used in the implementation. For instance, instead of the usual interfaces `Set` and `List`, there are interfaces `ListOfInteger`, `SetOfTerm`, etc. which are semi-automatically generated from templates. Although this approach leads to a certain “code bloat”, it improves readability and type safety.

Apart from TogetherCC, the KeY tool makes use of various third party software. Parsers are generated using ANTLR [4] and JavaCC [36]. The Recoder [48] framework is used to read and analyse JAVA programs. The Dresden OCL parser [23, 21] is used to parse and type-check OCL constraints. Finally, the JUnit framework [37] was used for unit testing during development.

### 9.2 Structure of the implementation

Fig. 9 shows the infrastructure of the KeY tool on the implementation level. The entities shown as cylinders represent external files, while the rectangular ones are programs. The parts rendered with thick lines and bold type are provided by the KeY project.

The mechanism for instantiating KeY patterns and idioms as described in Sect. 5, is an extension of the pattern instantiation provided by TogetherCC. Patterns are represented as JAVA programs which construct the required classes, associations, etc., using the TogetherCC API. For the KeY patterns, the generated entities are annotated with OCL constraints. These are generated from OCL template files [8] belonging to each pattern.

Complying with TogetherSoft’s single source philosophy, OCL constraints are stored as comments in the user’s JAVA files.

For syntax and type checking, as well as transformation to DL, we use the Dresden OCL parser. Type checking requires information about the UML model, which is exported from TogetherCC in XMI format (part of UML standard).

As a consequence of the single source philosophy, the UML model of TogetherCC corresponds one-to-one to the structure of the JAVA implementation.

Alternatively, the model information required by the OCL parser can be extracted without reference to the XMI facility from the JAVA code, using Recoder. Additional information on UML model, for example, associations, is obtained using the TogetherCC API.

When the KeY prover is used to reason about JAVA programs, these are parsed using the Recoder system. Recoder is also used to resolve references, that is, to determine the variable declaration, method declaration, etc., each identifier is referring to.

For the actual proof, different data structures are used, as explained in the following section. Proof obligations are typically generated from OCL constraints. These are translated into DL formulas (see Sect. 6.2) from the data structures provided by the Dresden OCL parser.

### 9.3 Implementation of the theorem prover

The KeY prover permits automated and interactive construction of proofs for DL formulas. The central data structure is the proof tree, which is extended by rule applications.

Each node of the proof tree contains a *sequent*  $\Gamma \vdash \Delta$ , where  $\Gamma$  and  $\Delta$  are sets of formulas. In a typical rule application of the sequent calculus, most of the sequent is not changed. To reduce memory consumption, data structures must be used which allow formula representations to be shared among goals and even among branches of the proof tree. Moreover, rule applications usually affect only a small part of the formulas and programs they act upon, so sharing of *sub*-formulas should also be possible. Sharing implies the use of *non-destructive* or *persistent* tree data structures for terms, formulas and programs. Such data structures are ubiquitous in functional and logic programming, but unusual in OO programming. The shared representation of JAVA programs is achieved using a hierarchy of about 250 classes, which are derived from the Recoder data structures.

All proof rules, including those for the automated simplification of goals, are encoded as *taclets*, see Sect. 8.1. Taclets are described by textual representations which are collected in an external file and processed at prover start-up time. There are no “built-in” rules in the KeY prover. Taclets consist of a *matching* part and an *action* part. For interactive use, the user specifies a formula or term in the sequent, where a taclet should be applied. This requires finding taclets which match a specific position in the sequent. For automated (heuristic) use, taclets which match anywhere in a sequent are automatically selected. Now matching is a potentially expensive operation, and there is a large number of taclets for the many different JAVA constructs and also for various abstract data types. To speed up the process of finding applicable taclets, these are kept in an indexing structure which permits fast access to a subset of potentially matching taclets. Typically the top-most predicate or function symbol is used to find applicable taclets, in the case of DL formulas, the type of the first statement is used.

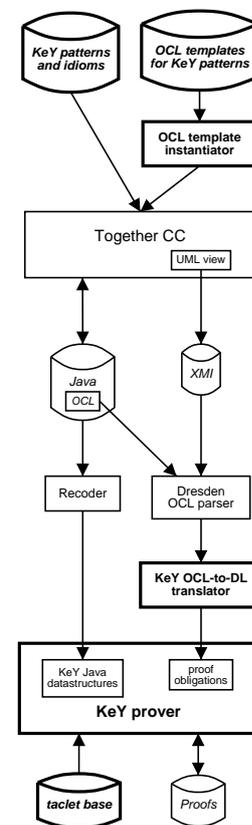


Figure 9. KeY tool Infrastructure.

## 9.4 Stand-alone versions

In general, the KeY tool is designed to be used together with a CASE tool. It was however recognised that parts of the system might be useful independently. It is thus possible to invoke the OCL-to-DL translation of Sect. 6.2 separately, if the UML model is given in XMI format. Furthermore, it is possible to use the KeY prover without a CASE tool. The stand-alone prover parses DL proof obligations from files.

## 10 Case Studies

In this section we discuss some case studies that were used to test the viability of the theoretical approach of KeY and the implementation of the KeY tool. Each case study consists of a (partial) specification and verification of various programs, specifically:

- the *JAVA Collection Framework (JCF)* [50],
- a software for access control (*PAM authentication with iButton*) [42], and
- a software for the computation of speed restrictions used by *Deutsche Bahn AG* (German Railway Company) [18].

The following sections describe each of these case studies in more detail and outline some of the results.

### 10.1 The JAVA Collection Framework

The *JAVA Collection Framework (JCF)* provides an application programming interface (API) plus reference implementation for lists, sets, trees, and related data structures. Many data structures found in JAVA programs are realised via the JCF, hence, it is obvious why they are of main interest for KeY.

In this case study the *JCF* is mainly used to investigate refinement techniques. Special attention is paid to the question of how to reconcile the following two points of view: first, the view of collections as abstract data types (a more functional viewpoint), second, the implementational view, as it is for example realised in the *JCF* reference implementation.

In general, the development process starts with a model on an abstract level. Then the model is stepwise refined to the implementational level, where a one-to-one relationship between UML and JAVA classes exists. The refinement process is visualised in Fig. 10.

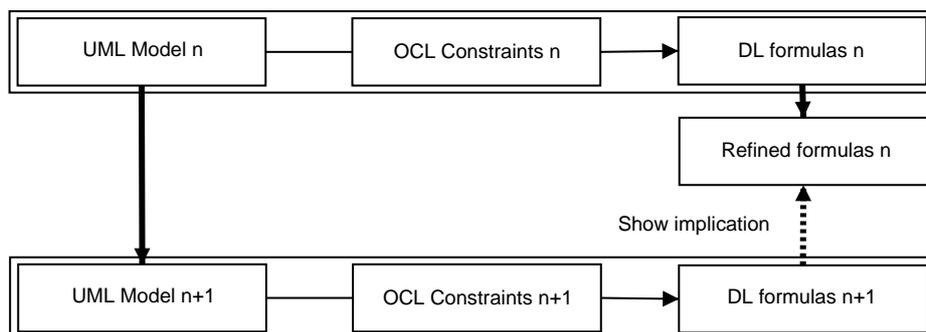


Figure 10. Schema of a typical refinement step.

Usually one wants to assure that (OCL) constraints of refinement level  $n + 1$  satisfy the constraints of level  $n$ . But the constraints of level  $n$  do not pertain to the possibly changed (usually: enlarged) name space of level  $n + 1$ . Therefore, the changes in the model from level  $n$  to level  $n + 1$  plus the constraints of level  $n$  are used to compute these constraints in the name space of level  $n + 1$ .

The approach we took does not work directly on OCL constraints, rather on the formulas resulting from the translation into Dynamic Logic. We explain this by way of example with Fig. 11 that shows the UML model of *List*, similar to the JCF.

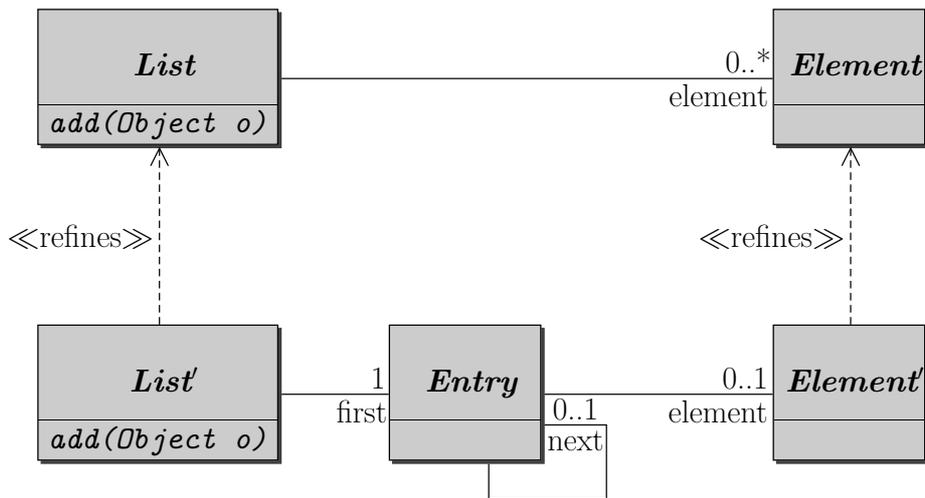


Figure 11. Refinement levels of a list

The upper part of the figure models lists on an abstract level using a simple aggregation from the class *List* to its element class *Element*. The class *List* contains the usual methods for adding, removing, and membership test of elements. The lower part of the figure models lists on a more detailed level, showing implementation details of the inner structure of lists, in particular, how elements are linked together.

The way to connect both levels is to define a refinement relation between the different abstractions. This refinement relation can be denoted graphically using UML dependencies with stereotype `<<refines>>` plus by an additional textual description. Once the developer defined a refinement relation (by drawing the dependencies and giving the mapping), the Dynamic Logic formulas obtained from the OCL constraints on the abstract level can be transformed to the concrete level and serve as proof obligations to show that the refined model is correct.

The technique sketched above was successfully applied to parts of the JCF. A complete account of this case study can be found in [50].

## 10.2 PAM authentication with iButton

This application allows a Linux user to authenticate him- or herself to the system using an *iButton*<sup>5</sup> or a smart card instead of a password. The application consists of two parts:

1. The *Pluggable Authentication Module (PAM)* running on the host system. The module is realised as a PAM library plug-in and, therefore, written in C.

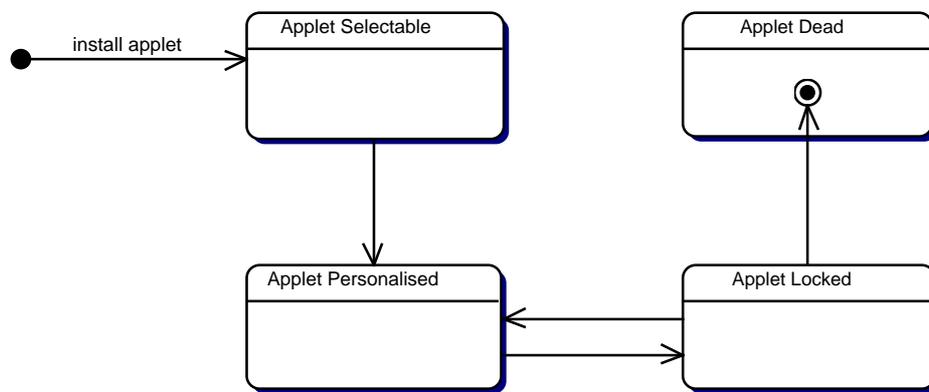
<sup>5</sup>“iButtons” are particular JAVA CARD devices embedded in a button shaped case, see <http://www.ibutton.com/>.

2. The JAVA CARD applet `SafeApplet`, running on the user’s JAVA CARD device.

In contrast to the other case studies, here we consider a genuine JAVA CARD application. The original JAVA CARD application [16] has however been rewritten and cleaned up for this case study. The redesign of the PAM was preceded by an analysis of the system requirements, and guided by questions like:

1. Is the system administrator or the user the owner of the applet PIN code?
2. What are the different deployment states of the applet (how does its life cycle look like)?
3. How can it be ensured that the applet is in a “sound” state when an iButton is ripped out of the reader and how can these “soundness” properties be specified?

The applet life cycle states were captured in a UML state diagrams like the one in Fig. 12. The diagram lists possible commands for each state together with the triggered state transitions in detail. After setting up the state diagrams, the first portions of code (skeleton code) were generated.



**Figure 12. Life cycle states of the `SafeApplet`.**

During the development process, the OCL specifications of parts of the applet were created, modified, and adjusted with the help of the KeY system. The main challenge are the “rip-out” properties. The problem of specifying those boils down to the general problem of expressing atomicity properties.

It turns out that this requires an extension of JAVA CARD DL with certain modal operators. The suggested modal operator is named  $\llbracket \cdot \rrbracket$  (pronounced “throughout”). The formula  $\llbracket p \rrbracket \phi$  means that  $\phi$  holds after *each* atomic step in program  $p$ .

According to the JAVA CARD specification, an atomic step is an update of a variable or a single object field. A sequence of operations can be bundled to a single atomic step called a *transaction* by the programmer. Now, it is important to be able to state that  $\phi$  does not necessarily hold inside a transaction. The main obstacle here is to capture the semantics and properties of the  $\llbracket \cdot \rrbracket$  operator in JAVA CARD DL’s calculus.

The extension of a Dynamic Logic calculus for abstract while-programs with an  $\llbracket \cdot \rrbracket$  operator was done in the paper [14]. The extension of JAVA CARD DL and the design of a calculus is ongoing work [13]. For a full paper on the PAM authentication with iButton case study and the development process used, see [42].

### 10.3 Computation of Restriction Speeds of Trains

On behalf of the *Deutsche Bahn AG* (German Railway Company) many hundred trains are driving across Germany each day. A so called booktable (German: *Buchfahrplan*) informs the engine driver on speed limits to be complied with at each point of the route. Obviously, this is crucial for safe and speedy travel.

Booktables include general information on technical properties of trains, for example, the minimal braking power needed to make the booktable applicable. For each train-route combination there exists a booktable. This makes it possible to respond in a flexible way to the available technology of a given train, for example, tilting technology (German: *Neigetechnik*). This would be prevented by rigid speed instructions as they are in use for the private vehicles on roads. The booktable is available to the engine driver in printed or in electronic form (*Ebula*). The software *Schutzbedarfsfeststellung (SbF)* used to compute the booktables was developed by *DBSystems* and serves as starting point for the present study.

*DBSystems* provided us with the current version of *SbF* and a product description written in natural language. This includes the algorithm used for speed computation. The *SbF* software is written in *Smalltalk (VisualWorks dialect)* and encompasses around 700 classes.

The study concentrated on the part of the system that is responsible for the speed and brake power computation. For this, the number of classes to be specified, verified and cross-translated into JAVA could be reduced to around 80. The translation from *Smalltalk* to JAVA has to deal with certain problems arising from the strict object orientation of *Smalltalk* and other semantic differences between both languages.

As a further consequence of the reimplementaion in JAVA, which is required by KeY, the result of the case study will be a verified program, whose behaviour can be compared with that of the original program. This means that not the correctness of the original program is ensured, but the correctness of the computed booktables via a verified reference implementation.

First, we looked at the available product specification that describes the algorithm for speed computation in a semi-formal way using natural language<sup>6</sup> enriched with UML-like class diagrams. The first task was to formalise this description in UML and OCL. This resulted in a formal specification on the analysis level.

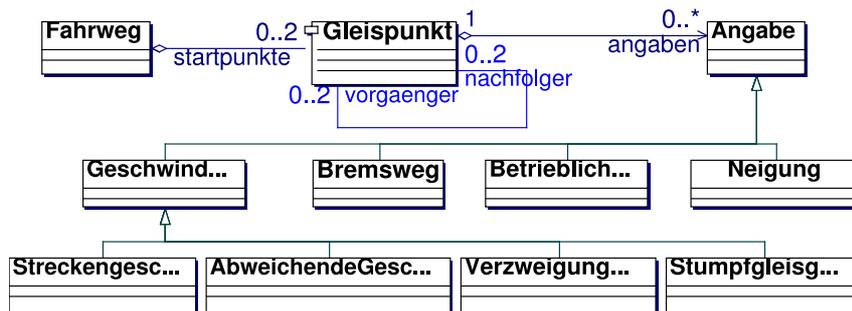


Figure 13. UML model of the SbF system on the analysis level.

The analysis model abstracts away from all implementational details and contains only the necessary constituents of the system. In order to get an impression of the degree of refinement we show in Fig. 14 an (incomplete) part of the infrastructure model as found on the implementation level. The corresponding fragment on the analysis level is shown in Fig. 13.

The relationship between both specifications is currently worked out using the refinement technique described in [50], see Sect. 10.1.

As an intermediate result of writing a formal specification we discovered the presence of ambiguities and incompleteness in the product specification. The verification of *SbF* is under way.

<sup>6</sup>The technical terms in the specification are in German and an exact translation is non-trivial (if at all possible), so we stick to the German expressions, particularly in the figures.

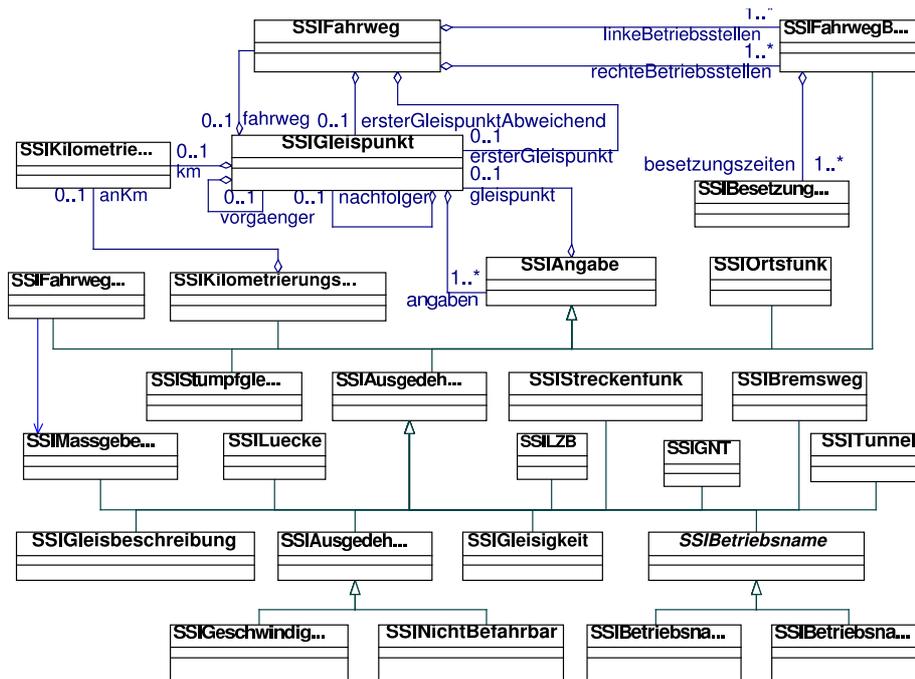


Figure 14. View of the infrastructure on the implementation level.

## 10.4 Summary

The three case studies discussed here differ considerably with respect to the nature of the target programs, but also with respect to their objectives. They show that the KeY approach is flexible enough to cope with varying demands. The case studies, respectively, the intermediate results obtained so far, demonstrate the viability and usability of the KeY approach.

## 11 Current State and Future Work

### 11.1 Current State

The core of the KeY tool is finished and the tool is, with a few minor restrictions, already successfully applicable.<sup>7</sup> The case studies described in Sect. 10 prove this. They also show that it is possible to verify the average JAVA method (consisting of about 10 to 20 lines of code) within a few minutes. Often, the KeY prover can even automatically establish correctness if the code does not contain loops (in this case the user has to provide a loop invariant interactively).

Current implementation activities include the integration of tools that have already been developed as stand-alone prototypes. Here, we mention a tool for generating counter-examples from failed proof attempts and an authoring tool for OCL constraints.

The tool for generating counter-examples from failed proofs assists the user in identifying and solving errors in the implementation. A theoretical framework for counter-example generation based on abstract data types is presented in [1]. The usefulness of this approach has already been shown in a prototype which is based on a model generation theorem prover [26].

<sup>7</sup>A current version of the KeY tool can be downloaded at <http://i12www.ira.uka.de/~key/download.htm>.

A further step to improve the usability of the KeY tool is the integration of an authoring tool for OCL constraints [30]. This tool is already available as a stand-alone version and offers assistance in generating specifications and helps to understand OCL constraints by rendering them automatically in natural language. We believe that the integration of such a tool helps to overcome reservations against formal methods.

Another important aspect we are currently working on is the support of proof re-use. This technique diminishes the amount of work spent with verification after a (minor) change of the specification or implementation. We count this as an essential point since one of the prejudices against program verification is that it is too costly to be ever usable in practice.

Proof re-use makes it feasible to check automatically and periodically whether the implementation still complies with the properties expressed in the OCL constraints. The situation is similar to automatic and periodic runs of unit tests, a proven best practice in software development. Periodical checks prevent specifications from becoming outdated, which is a major (and common) problem when specifications are merely available as informal text.

## 11.2 Future Work

One obvious direction for future work includes support of UML diagram types other than class diagrams, such as state chart or sequence diagrams. This would allow to specify temporal behaviour of programs, which is not possible in class diagrams.

To make software verification scale up to larger programs, it is necessary to have a module concept that makes it possible to independently verify the modules of a program. Modules are well understood for imperative languages and are supported by several languages. Unfortunately, object-oriented languages including JAVA lag somewhat behind. One of our next research efforts will be to look for a module concept for JAVA that is compatible with the requirements of formal verification.

We are aware that formal verification is only one option among many formal methods (and perhaps a rather extreme one). Other approaches, such as abstract interpretation, first order model checking, static analysis, extended static checking, etc., should be integrated into the KeY tool.

Finally, we would like improve the user interface to the theorem prover in a fundamental way: recall that one can view verification in KeY as symbolic program execution. Hence, one may see a single branch in the proof tree as one program execution with symbolic start values. We intend to reformulate verification as much as possible within the well-established paradigm of symbolic source code debugging. We think that a remodelling of formal verification as “abstract debugging” will not only increase acceptance of verification, but will result in a massive improvement of efficiency: such elements of modern debuggers as break points, watches, spy points, inspectors, and navigation aides make eminent sense within the symbolic execution paradigm, too. In contrast to conventional debuggers, in KeY one could even evaluate all expressions (e.g., in guards) symbolically.

## 12 Conclusion

What sets the KeY tool apart from other efforts in formal software specification and verification is the systematic attempt to conceive a formal technique as an extension of established, industrial methods of software development.

Unfortunately, parts of the formal methods community in the past have denounced popular industrial software development methods as unscientific and, hence, unworthy of consideration. Such views are based on a lack of knowledge about industrial software production and the conflation of “scientific” with “formal”. In contrast to this, we believe that formal methods must be tightly integrated with conventional development processes in order to be immediately useful to developers and designers. We see this as a prerequisite to be fulfilled before formal methods can possibly catch on.

The most visible point of the philosophy just sketched is the interface of the KeY tool, which appears to be a state-of-the-art, although conventional CASE tool. Formal specifications can be added anytime during design and

development without having to change the tool or paradigm. Machine assistance in generating formal specification in the form of KeY idioms and patterns help users to get started.

Integration of informal and formal methods is one of the corner stones of the KeY approach. A second one is the consequent choice and design of the formal tools so as to maximise their usability. For example, the program logic JAVADL is transparent with respect to the target language and supports symbolic execution as a proof paradigm. We are convinced that the pragmatics and usability of formal tools are as important as their soundness and theoretical adequacy.

We would like to stress that the integration and scaling-up of our formal methods spawned already a considerable number of theoretically interesting questions (documented in technical papers such as [11, 14, 12]). Vice versa, formalisation of UML and OCL led to clarifications and extensions [7, 5].

Finally, we would like to discuss briefly the most frequently heard counter argument against software verification. It goes like this: “full functional verification will never be a push button technology; but this is a *sine qua non* for formal methods to catch on in industry. Unless you produce something like a model checker as used for verification of hardware and system designs, you will never prevail.”

We agree that formal software verification is extremely unlikely to become fully automated, however, this is a red herring, because the assessment above is based on a (at least) twofold misunderstanding: first, the stumbling block for industrial users when applying formal methods is *not* interactivity. The problem is that current formal approaches are idiosyncratic and require special skills. The usability threshold is simply very high. Widely established methods and tools such as symbolic debuggers, most testing methods, code reviews, etc., are all far from being automated. They are accepted, because they are useful, integral parts of processes, and they can be mastered with reasonable effort. Second, so-called “push button” technologies, notably symbolic model checking, are far from being automated either: system requirements have to be captured formally in a temporal logic, model checkers have to be tweaked to cope with larger problems, and so on.

In summary, the real challenge for formal methods in software development is to make them useful for as many people as possible. This is what the KeY project is about.

## References

- [1] W. Ahrendt. Deductive search for errors in free data type specifications using model generation. In A. Voronkov, editor, *Automated Deduction – CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark*, volume 2392 of *LNCS*. Springer-Verlag, 2002.
- [2] W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermatz, R. Hähnle, W. Menzel, and P. H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. In M. Ojeda-Aciego, I. P. de Guzmán, G. Brewka, and L. M. Pereira, editors, *Proc. 8th European Workshop on Logics in AI (JELIA)*, volume 1919 of *LNCS*, pages 21–36. Springer-Verlag, Oct. 2000.
- [3] W. Ahrendt, T. Baar, B. Beckert, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, and P. H. Schmitt. The KeY system: Integrating object-oriented design and formal methods. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering (FASE), Part of Joint European Conferences on Theory and Practice of Software, ETAPS, Grenoble*, volume 2306 of *LNCS*, pages 327–330. Springer-Verlag, 2002.
- [4] ANTLR homepage. <http://www.antlr.org/>.
- [5] T. Baar. How to ground meta-circular OCL descriptions – a set-theoretic approach –. In T. Clark, A. Evans, and K. Lano, editors, *Proceedings, Fourth Workshop on Rigorous Object-Oriented Methods, London*, 2002.

- [6] T. Baar, B. Beckert, and P. H. Schmitt. An extension of Dynamic Logic for modelling OCL's @pre operator. In *Proceedings, Fourth Andrei Ershov International Conference, Perspectives of System Informatics, Novosibirsk, Russia*, LNCS 2244, pages 47–54. Springer, 2001.
- [7] T. Baar and R. Hähnle. An integrated metamodel for OCL types. In R. France, B. Rumpe, and J. Whittle, editors, *Proc. OOPSLA 2000 Workshop Refactoring the UML: In Search of the Core, Minneapolis/MI, USA*, Oct. 2000.
- [8] T. Baar, R. Hähnle, T. Sattler, and P. H. Schmitt. Entwurfsmustergesteuerte Erzeugung von OCL-Constraints. In K. Mehlhorn and G. Snelting, editors, *Softwaretechnik-Trends*, Informatik Aktuell, pages 389–404. Springer-Verlag, Sept. 2000. (In German).
- [9] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, volume 1783 of LNCS. Springer-Verlag, 2000.
- [10] K. Beck. Embracing change with Extreme Programming. *Computer*, 32:70–77, Oct. 1999.
- [11] B. Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, volume 2041 of LNCS, pages 6–24. Springer-Verlag, 2001.
- [12] B. Beckert, U. Keller, and P. H. Schmitt. Translating the Object Constraint Language into first-order predicate logic. In *Proceedings, VERIFY, Workshop at Federated Logic Conferences (FLoC), Copenhagen, Denmark*, 2002. Available at [www.ira.uka.de/~key/doc/2002/BeckertKellerSchmitt02.ps.gz](http://www.ira.uka.de/~key/doc/2002/BeckertKellerSchmitt02.ps.gz).
- [13] B. Beckert and W. Mostowski. A program logic for handling JAVA CARD's transaction mechanism. 2002. Submitted.
- [14] B. Beckert and S. Schlager. A sequent calculus for first-order dynamic logic with trace modalities. In R. Gorè, A. Leitsch, and T. Nipkow, editors, *Proceedings, International Joint Conference on Automated Reasoning, Siena, Italy*, LNCS 2083, pages 626–641. Springer, 2001.
- [15] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.
- [16] D. Bolten. PAM authentication with an iButton. [http://www-users.rwth-aachen.de/dierk.bolten/pam\\_ibutton.html](http://www-users.rwth-aachen.de/dierk.bolten/pam_ibutton.html).
- [17] 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.
- [18] R. Bubel. Formale Spezifikation und Verifikation sicherheitskritischer Software mit dem Key System: Eine Fallstudie aus dem Eisenbahnwesen. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe, Dec. 2002. (In German, to appear).
- [19] Z. Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Java Series. Addison-Wesley, June 2000.
- [20] S. Cook and J. Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy*. The Object-Oriented Series. Prentice Hall, 1994.

- [21] Dresden-OCL homepage. <http://dresden-ocl.sourceforge.net/>.
- [22] A. Evans, J.-M. Bruel, R. France, K. Lano, and B. Rumpe. Making UML precise. In L. Andrade, A. Moreira, A. Deshpande, and S. Kent, editors, *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?*, 1998.
- [23] F. Finger. Design and implementation of a modular OCL compiler. Diplomarbeit, Technische Universität Dresden, Fakultät für Informatik, Mar. 2000.
- [24] M. Fowler and K. Scott. *UML Distilled. Applying the Standard Object Modeling Language*. Addison-Wesley, 1997.
- [25] R. France. A problem-oriented analysis of basic uml static requirements modeling concepts. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 57–69. ACM Press, 1999.
- [26] H. Fujita and R. Hasegawa. A model generation theorem prover in KL1 using a ramified-stack algorithm. In K. Furukawa, editor, *Proceedings 8th International Conference on Logic Programming, Paris/France*, pages 535–548. MIT Press, 1991.
- [27] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading/MA, 1995.
- [28] M. Giese. Incremental closure of free variable tableaux. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proc. Intl. Joint Conf. on Automated Reasoning IJCAR, Siena, Italy*, volume 2083 of *LNCS*, pages 545–560. Springer-Verlag, 2001.
- [29] E. Habermalz. Interactive theorem proving with schematic theory specific rules. Technical Report 19/00, Fakultät für Informatik, Universität Karlsruhe, 2000. <http://il2www.ira.uka.de/~key/doc/2000/stsr.ps.gz>.
- [30] R. Hähnle, K. Johannisson, and A. Ranta. An authoring tool for informal and formal requirements specifications. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering (FASE), Part of Joint European Conferences on Theory and Practice of Software, ETAPS, Grenoble*, volume 2306 of *LNCS*, pages 233–248. Springer-Verlag, 2002.
- [31] D. Harel. Dynamic logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, volume II: Extensions of Classical Logic, chapter 10, pages 497–604. Reidel, Dordrecht, 1984.
- [32] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
- [33] G. Holzmann. Economics of software verification. In *Proc. Workshop on Program Analysis for Software Tools and Engineering*, Snowbird, Utah, USA, June 2001. ACM.
- [34] D. Hutter, B. Langenstein, C. Sengler, J. H. Siekmann, and W. Stephan. Deduction in the Verification Support Environment (VSE). In M.-C. Gaudel and J. Woodcock, editors, *Proceedings, Formal Methods Europe: Industrial Benefits Advances in Formal Methods*. Springer, 1996.
- [35] I. Jacobson, J. Rumbaugh, and G. Booch. *The Unified Software Development Process*. Object Technology Series. Addison-Wesley, Reading/MA, 1999.
- [36] JavaCC homepage. [http://www.webgain.com/products/java\\_cc/](http://www.webgain.com/products/java_cc/).

- [37] JUnit homepage. <http://junit.sourceforge.net/>.
- [38] U. Keller. Übersetzung von OCL-Constraints in Formeln einer Dynamischen Logik für Java. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe, 2002. (In German).
- [39] D. Kozen and J. Tiuryn. Logics of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 14, pages 789–840. The MIT Press, 1990.
- [40] S. J. Mellor, D. D’Souza, T. Clark, A. Evans, and S. Kent. Infrastructure and Superstructure of the Unified Modeling Language 2.0 (Response to UML2.0 RfP). Technical report, Submission to the OMG, 2001.
- [41] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, second edition, 1997.
- [42] W. Mostowski. Rigorous development of JavaCard applications. In T. Clark, A. Evans, and K. Lano, editors, *Proc. Fourth Workshop on Rigorous Object-Oriented Methods, London, 2002*. URL: <http://www.cs.chalmers.se/~woj/papers/room2002.ps.gz>.
- [43] Object Modeling Group. *Unified Modelling Language Specification, version 1.4*, Sept. 2001.
- [44] Response to the UML OCL RfP, June 2002. OMG document ad/2002-05-09.
- [45] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV ’96*, volume 1102 of *LNCS*, pages 411–414. Springer-Verlag, July/August 1996.
- [46] L. C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *LNCS*. Springer-Verlag, 1994.
- [47] V. R. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proceedings, 18th Annual IEEE Symposium on Foundation of Computer Science, 1977*.
- [48] Recoder homepage. <http://recoder.sourceforge.net/>.
- [49] M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints*, volume 14 of *BISS Monographs*. Logos Verlag, 2002. PhD thesis, Universität Bremen.
- [50] A. Roth. Deduktiver Softwareentwurf am Beispiel des Java Collections Frameworks. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe, June 2002. (In German).
- [51] P. H. Schmitt. A model theoretic semantics of OCL. In B. Beckert, R. France, R. Hähnle, and B. Jacobs, editors, *Proceedings, IJCAR Workshop on Precise Modelling and Deduction for Object-oriented Software Development, Siena, Italy*, pages 43–57. Technical Report DII 07/01, Dipartimento di Ingegneria dell’Informazione, Università degli Studi di Siena, 2001.
- [52] K. Stenzel. Verification of JavaCard Programs. Technical report 2001-5, Institut für Informatik, Universität Augsburg, Germany, 2001. Available at <http://www.Informatik.Uni-Augsburg.DE/swt/fmg/papers/>.
- [53] Sun Microsystems, Inc., Palo Alto/CA. *Java Card 2.0 Language Subset and Virtual Machine Specification*, Oct. 1997.
- [54] Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.2 Platform Specification*, Sept. 2002.
- [55] TogetherSoft homepage. <http://www.togethersoft.com/>.

- [56] D. von Oheimb. Axiomatic semantics for Java<sup>light</sup>. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetsch-Heffter, editors, *Proceedings, Formal Techniques for Java Programs, Workshop at ECOOP'00, Cannes, France, 2000*.
- [57] D. von Oheimb. *Analyzing Java in Isabelle/HOL*. PhD thesis, Institut für Informatik, Technische Universität München, Januar 2001.
- [58] J. Warmer and A. Kleppe. OCL: The constraint language of the UML. *Journal of Object-Oriented Programming*, 12(1):10–13,28, Mar. 1999.