1

A New Look at Formal Methods for Software Construction

by

Reiner Hähnle

This chapter sets the stage. We take stock of formal methods for software construction and sketch a path along which formal methods can be brought into mainstream applications. In addition, we provide an overview of the material covered in this book, so that the reader may make optimal use of it.

1.1 What KeY Is

The KeY system[1] was conceived because, after having worked in logic and theorem proving for many years, we became convinced that a different kind of tool than the existing range of editors and theorem provers is necessary to push formal methods further into industrial applications. We know that not everyone agrees that formal methods have a place in the software industry, but recent success stories, such as the SDV project at Microsoft [Ball et al., 2004], are indicators that formal methods can become mainstream provided that they are appropriately packaged and marketed. We think that formal methods are robust and powerful enough for applications, but they need to become (much!) more accessible.

With this in mind, the KeY system was not designed merely as a theorem prover for verification of object-oriented (OO) software, but as a formal methods tool that integrates design, implementation, formal specification and formal verification as seamlessly as possible. The intention is to provide a platform that allows close collaboration of conventional and formal software development methods.

This sounds as if KeY were a silver bullet. So let us be very clear that we do not think that formal specification and verification of complex systems is a task that can be done automatically or by people who are completely unskilled in formal methods. This is as improbable as automatized programming of complex systems. Everyone accepts that specialists are needed to write, say,

[1] `www.key-project.org`

reliable and efficient systems software. If complex software is to be formally specified and verified, it should be clear that some serious work by specialists is called for. But if formal methods specialists are still required for complex tasks, what is gained by KeY then? In a nutshell, the intention is to lower the cost of formal methods to an acceptable level from where it is clear that formal methods actually will save cost in the end. In the following, we map out the basic principles of such a conception of formal methods in software construction.

Easy Things Made Easy

KeY provides interfaces and tools that enable non-specialists in formal methods to use and understand formal artifacts *to a certain extent*. For example, we provide idioms and patterns that can be simply instantiated to create formal specifications. This is comparable to using a visual editor in order to create a Java GUI instead of having to master the Swing framework. Developers can also run standard checks, such as the consistency of existing formal specifications by menu selection from their usual case tool. Such provisions push the boundary beyond which a formal methods specialist is required. It also provides a learning path to formal methods for interested developers.

Integration of Informal and Formal Notation

From the view of the non-expert user, KeY appears not as a stand-alone tool, but as a plugin to a familiar case tool (at the moment Borland Together and the Eclipse IDE are supported). Translation of specifications written in UML's Object Constraint Language (OCL) and the Java Modeling Language (JML) into logic, as well as synthesis of various proof obligations is completely automatic, as is, to a large extent, proof search. In addition, KeY features a syntax-directed editor for OCL that can render OCL expressions in several natural languages while they are being edited. It is even possible to translate OCL expressions automatically into English and German (stylistically perhaps not optimal, but certainly readable). This means that KeY provides a common tool and conceptual base for developers and formal methods specialists. The architecture and interface characteristics of KeY are depicted in Fig. 1.1.

Teaching Formal Methods for Software Construction

We think that it is necessary to change the way formal methods are taught. Many of us used to teach traditional courses in logic, theorem proving, formal languages, formal specification, etc. Ten years ago, in a typical Computer Science programme at a European university you could find a wide variety of such courses with at least logic or formal systems courses being compulsory. While such courses are still taught in theoretical specializations, compulsory logic/-formal specification courses have mostly been scrapped. In the post-Bologna bachelor programmes there will be little room for foundational courses. Even

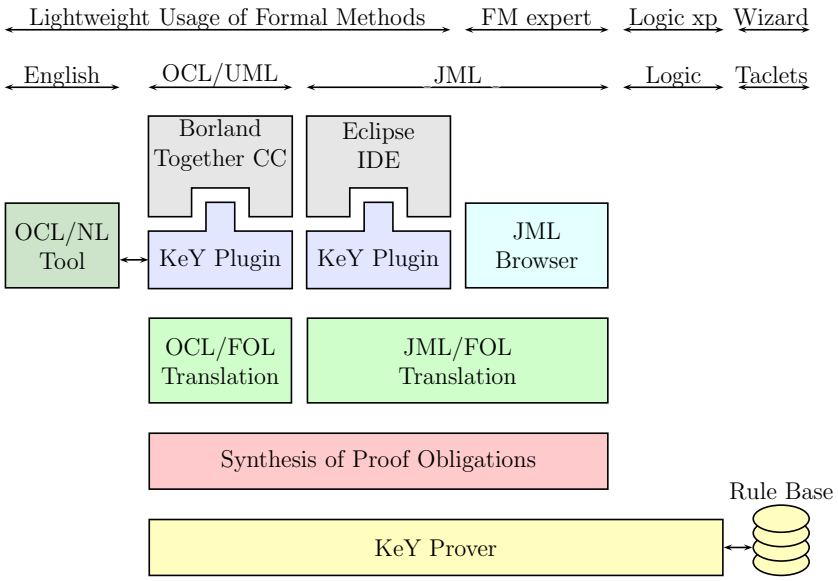

Fig. 1.1. Architecture and interface of the KeY system

if this were not so, we think that it would be time to look at software construction not as an afterthought in formal methods courses, but as the starting point and main driver for the curriculum. The goal of such a *formal methods for software engineering* course is not only to teach formal specification and verification in the context of OO software development, but also exactly those topics in logic, semantics, formal specification, theorem proving that are necessary for a deepened understanding. Such a presentation of this material would necessarily be less systematic and complete than if it were taught in a traditional manner, but we think this is far outweighed by a number of advantages:

- It is notoriously difficult to motivate students (in particular those interested in software development) to theoretical studies, which are often perceived as useless. The tight integration of formal methods into software development provides a strong and direct motivation.
- Many students find it easier to grasp theoretical concepts when these are explained and motivated with natural examples.
- We often encountered students who, even after taking several foundational courses, perceived, for example, logic and programming language semantics as completely different topics and failed to see their close connections. Compartmentalization is increased by presentations based on traditional notations developed in separate fields. It is important to point out sim-

ilarities and identical concepts. Most of all, it is important to relate to concepts from programming languages, because this is what students of computer science or software engineering are most familiar with. To take a trivial example, students often find it easier to grasp universal quantification when the analogies to for-loops are pointed out, including declaration of index variables, scoping, binding, hiding, etc.

- A course that teaches base knowledge in logic, specification, and semantics under the umbrella of high-quality software construction is much easier to integrate into an educational programme than dedicated foundational courses. The latter tend to be optional and are taken only by a small minority of interested students. We see a great danger, in particular with respect to bachelor programmes, that students are completely deprived of foundations. We believe that an attractively packaged course with foundational material tailored to the requirements of software engineering could be a solution.

A course along the lines just sketched is taught by the author of this chapter at Chalmers University since 2004.[2] Many chapters in this book are suitable as background material for (advanced) courses related to logic, specification, and verification (see also the following section).

Towards Formal Verification as a Debugging Tool

Formal verification is unlikely to be a fully automatic procedure in the foreseeable future. This is true even for less demanding tasks than full functional verification of concrete source code: the availability of so-called push button tools notwithstanding, verification remains a highly interactive process. The main problem, of course, is that most of the time the specification or the implementation (or both) are buggy. Hence, proof attempts are doomed to fail. In software verification, it is also often necessary to strengthen induction hypotheses or invariants before they can be proven. In either case, the source of a failed proof must be located and patched. Then the proof must be retried, etc. This means that it must be possible to inspect a partial or stuck proof and make sense of it. This process has strong similarities to debugging. Therefore, it is important to equip the user interface of a prover with similar capabilities than that of a debugger.

While the debugger view has not quite been realized yet for the KeY prover, which also can be used stand-alone without any CASE tool (see Fig. 1.1), the system offers a wide variety of visual aids and controls. These range from highlighting of active parts in proofs and proof nodes, drag'n'drop application of rules, tool tips with explanations of logical rules to execution control with local computations, breakpoints, etc. Automatic reuse of failed proofs and correctness management of open goals and lemmas round off the picture.

[2] The course web site is `http://www.cs.chalmers.se/Cs/Grundutb/Kurser/form`. The LaTeX sources of slides, labs, exercises and exams are available on request.

In order to increase automation, a number of predefined search strategies are available. There is a back end to SMT-LIB syntax[3] for proving near propositional proof goals with external decision procedures. A back end to TPTP syntax[4] is under construction.

Nevertheless, it is in this area, where the book gives only a snapshot of the current capabilities. Ongoing research that is hoped to boost interactive proof construction dramatically, includes proof visualization [Baum, 2006] and automatic search for finite counter examples [Rümmer, 2005].

Industrially Relevant Languages

In our opinion it is essential to support an industrially relevant programming language as the verification target. We have chosen JAVA CARD source code [Sun, b] because of its importance for security-critical applications. We refrained from using a home-spun sublanguage of JAVA, because it is unrealistic to assume that applications are written in it. It would have been simpler to create support for JVM bytecode, but while it is easier to build a verification system for byte code than for source code, it becomes more difficult to verify byte code, because it contains much less information. Besides, neither JAVA CARD nor native code compilers produce JVM bytecode.

The KeY prover and calculus support the full JAVA CARD 2.1 language. This includes all object-oriented features, atomic transactions, JAVA integer types, abrupt termination (local jumps and exceptions) and even a formal specification (in OCL) of the essential parts of the JAVA CARD API. In addition, some JAVA features that are not part of JAVA CARD are supported as well: multi-dimensional arrays, JAVA class initialization semantics, `char` and `String` types. In short, if you have a sequential JAVA program without dynamic class loading and floating point types, then it is (in principle) possible to verify it with KeY.

On the front end, we support the OMG standard Object Constraint Language (OCL) [Warmer and Kleppe, 2003] for specification as well as the Java Modeling Language (JML) [Leavens et al., 2006], which is increasingly used in industrial contexts [Burdy et al., 2005].

The KeY systemis written in JAVA and runs on all usual architectures. The same is true for the Borland Together and Eclipse CASE tools. Everything, with the exception of Borland Together, is freely available, open software.

1.2 About this Book

This book is mainly written for two kinds of readers: first, as explained in the previous section, it can serve as a textbook in an advanced formal-methods course. All, but the most basic, required mathematical notions are contained

[3] `http://combination.cs.uiowa.edu/smtlib/`
[4] `http://www.cs.miami.edu/~tptp/`

and explained. Chapters 2 and 3 are self-contained discussions of first-order and program logics and calculi *tailored to the needs of formal analysis of OO software*. This means, for example, that meta-logical results such as incompleteness are only fleetingly discussed, as far as necessary to explain the limitations of first-order program logics. Among the many calculi available for automated reasoning we concentrate on sequent calculi, because they are most widely used in deductive software verification. On the other hand, we introduce a richly typed first-order logic including essential notions for program analysis such as rigid/flexible terms, none of which is treated in logic textbooks.

We assume that readers of this book are familiar with object-oriented design and software development, including UML class diagrams and the programming language JAVA. As mentioned above, no special mathematical knowledge is required, with the exception of basic set theory and propositional logic. Naturally, we do not deny that a certain mathematical maturity is helpful to obtain a deepened understanding of the material in Part I.

Although the book is about a specific tool, the KeY tool, much of the material can be read independently and is transferable to other contexts. The book becomes more KeY-specific towards the end, more precisely, Parts I and II are fairly independent of KeY while Parts III and IV contain specific solutions and case studies. As a consequence, in a course on software development with formal methods, one might stick to the first two parts plus Chapter 11. In the book we proceed in a bottom-up style to avoid dangling definitions, but in the context of a course the material might well be presented top-down (as it is, in fact, done in the course mentioned above). Chapter 10 is an informal introduction to the main features of the KeY tool and can be recommended as an entry point.

The second kind of reader is any kind of computer professional (developer, researcher, etc.) who is interested in formal methods for software development or in KeY in particular. There is no need to read the book sequentially or in any particular order. The chapters can be read fairly independently (but following some dependencies is unavoidable, if a full grasp on technical details is desired). Those who are familiar with formal techniques would only skim Chapter 2, but will find Chapter 3 still interesting. If you are mainly interested in usage and capabilities, Chapter 10 plus some of the case studies are a good start.

Finally, two things that this book is *not*: it is not a reference manual for the KeY system. Even though many features are explained and discussed, for the sake of readability we did not strive for completeness. Some parts of the manual are available online at the KeY website, for example, a browsable list of all calculus rules. This book is also not simply a collection of papers. All chapters were specifically written for this book and not just culled from a technical report. We aimed at self-containedness, many examples and not too terse explanations. For this we sacrifice some of the technical details. Wherever

technical explanations have been abridged or simplified, we give pointers to the full treatments in papers and theses.

In the remainder of this chapter, we explain some of the problems that must be solved during formal specification and verification of object-oriented software. The idea is to give the reader a better idea of what is covered in various parts of the book. It will also help not to loose sight of the big picture.

1.3 The Case for Formalization

For the following considerations we use a small example. We stress that the size of this example is *not* indicative for the problems that can be modelled with the KeY system. Realistic case studies are discussed in Chapters 14 and 15.

Assume that we are given the following informal specification for developing a simple electronic paycard application:

> "The function of a paycard is to let its owner pay bills or withdraw money from terminals authorized by the card provider.
>
> A paycard contains information about its current balance. The balance must not be negative and must not exceed a given limit. The limit of a paycard cannot be changed, although different cards can have different limits.
>
> Each paycard provides a charge operation that updates the balance according to the amount involved in a transaction."

This specification seems quite precise, but even in a small example like this, unclear issues appear immediately. For example, it is not specified what happens when the charge operation is called with an amount that would exceed the card limit. One of the main advantages of formalization is to exhibit such imprecision.

Imprecise specifications can lead to serious problems that are detected (too) late, when it is expensive to fix them. In the example, without a clear guideline, an implementor might write a charge method that simply does nothing when the amount to be charged exceeds the card limit. Such an implementation gives no feedback when something went wrong during the operation. Probably, the application needs to be redesigned. (In JAVA CARD redesigning an application can be problematic, because additional features, such as a counter for unsuccessful charge operations, might not fit into memory.)

In Fig. 1.2, a very simple UML class diagram with a first design based on the specification above is given. Partly it reflects the above requirements by stipulating instance attributes balance and limit, of which the latter is designated as immutable (indicated by the property {frozen}). But what about the other requirements, for example the legal values of balance being between zero and the value of limit? Of course, one could design a wrapper class that provides a type for the legal values, but this has a number of disadvantages: first, a premature decision on how to implement the balance attribute is taken:

for example, if the implementation language is C++, one would probably use a scalar type instead. Second, wrapper classes for primitive datatypes lead to clumsy and inefficient code. Third, one needs different wrappers for different limit values, which leads to various rather complex implementation options. Fourth, it does not ensure that illegal values of the balance do not occur but, at best, that a runtime error or exception occurs once this happens.

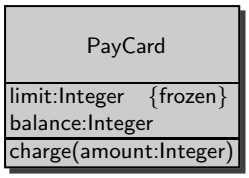

Fig. 1.2. Simple PayCard class diagram

The brief discussion above reflects the fact that purely programming-language-based mechanisms such as type systems cannot ensure all kinds of runtime requirements. The same holds, of course, for design languages (like UML) that are even less expressive than programming languages. In order to specify and guarantee runtime requirements the following ingredients are needed (see also Fig. 1.3):

1. A formal specification language that is expressive enough to capture the requirements a design stipulates on an implementation, for example, that it is an invariant of the PayCard class that the values of the balance attribute are between zero and the value of limit.
2. A framework that allows to formally prove that a given implementation satisfies its requirements. This involves a formalization of an execution model of the programming language in which the verification targets are written, in our case JAVA CARD.

The designers of UML became aware quite early of the need for a formal specification language. It is a part of the UML since version 1.1 and is called *Object Constraint Language* (OCL) [Warmer and Kleppe, 1999b, 2003]. OCL allows to attach invariants to classes of UML diagrams and to specify operation contracts in the form of pre- and postcondition pairs. Being part of the UML, OCL is standardized by the OMG[5]. The consequent visibility of OCL (⇒ Sect. 5.2) and its integration into the world of OO software development motivated us to support it as one of the formal specification languages in the KeY tool. Back to our example, the requirement on the admissible values of balance can be expressed as an OCL invariant:

[5] Object Management Group, `www.omg.org`

OCL (1.1)

```
context PayCard
inv withinLimit : balance >= 0 and balance <= limit
```

OCL

In order to *prove* that a given implementation of the class PayCard (and possibly other classes) respects this invariant, however, a lot more work is necessary. First of all, if we prove something, there must be some underlying notion of what is a valid statement. In formal logic, as well as in the theory of programming languages, validity is defined in terms of a formal semantics: a mapping between expressions of a formal language and a suitable mathematical model of the underlying domain. In our setting, OCL expressions over a given UML class diagram $\mathcal{D}$ are mapped into an algebra that models objects and object diagrams. As a consequence one can precisely say, for example, that a given object diagram satisfies a given Boolean OCL expression.

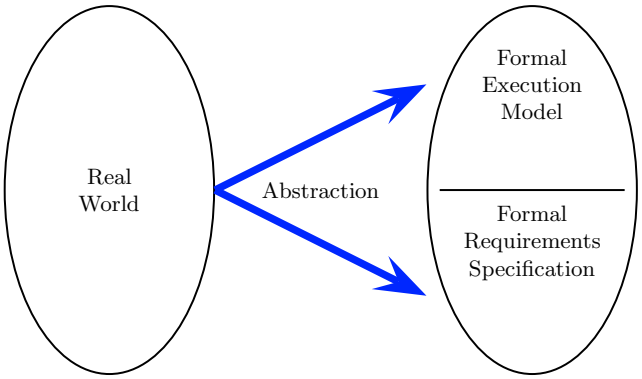

Fig. 1.3. The most important ingredients of formalization

Unfortunately, semantic notions do not lend themselves directly to mechanized proofs, the reason being that semantic domains typically contain infinitely many entities (e.g., all instances of a class). In automated theorem proving, therefore, one reasons over syntactic expressions that represent the semantics adequately. This leads to the notion of a *calculus*, a set of rewrite rules that specify how syntactic expressions are to be manipulated in order to be *derivable*. The idea is to design a calculus that is *sound* with respect to its semantics, that is, all derivable expressions are supposed to be valid. For example, one could conceive of a proof rule that reduces derivability of the invariant `withinLimit` above to derivability of the two invariants `balance >= 0` and `balance <= limit`:

$$\frac{\texttt{context}\ \ \mathcal{C}\ \ \texttt{inv:}\ \mathcal{I}_1 \qquad\qquad \texttt{context}\ \ \mathcal{C}\ \ \texttt{inv:}\ \mathcal{I}_2}{\texttt{context}\ \ \mathcal{C}\ \ \texttt{inv:}\ \mathcal{I}_1 \text{ and } \mathcal{I}_2}$$

Such a syntactic proof rule captures a property of an infinite number of semantic entities: it is valid when $\mathcal{C}$ is replaced by any concrete class name and $\mathcal{I}_1$ and $\mathcal{I}_2$ by any concrete Boolean OCL constraints in any UML diagram.

Although possible, there are good arguments against building such a calculus directly for OCL:

- It is difficult and expensive to develop a theorem prover for a given formal language. OCL is a big language compared to logic languages (such as first-order logic) and, in contrast to them, proof search in OCL is not well understood. Moreover, OCL is frequently revised.
- OCL was not designed with proof support in mind, and like UML it is independent of the implementation language. It does not know about concrete implementations of datatypes such as the integers. Before version 2.0, there was no way to specify initial states of classes. OCL is also not intended to express complex proof obligations that involve several invariants (see below).

As a consequence, we take a "compilation" approach: OCL expressions are translated (⇒ Sect. 5.2) into formulae of first-order logic (FOL). OCL compilation circumvents the difficulties outlined above. It also makes KeY independent from OCL as the sole specification language: recently, JML emerged as a popular specification language used in many formal methods projects dealing with JAVA and JAVA CARD [Burdy et al., 2005]. Replacing the OCL to FOL compiler with a JML front end enables the use of KeY with JML (⇒ Sect. 5.3).

A further major advantage of translating OCL and JML into FOL is that we do not need to define a dedicated formal semantics for these specification languages. Their semantics is implicitly defined by the translation into FOL, the latter having a standard semantics that is widely agreed upon. The translation approach works only if it is natural to represent a specification language by FOL. Admittedly, this is not the case for "vanilla" FOL as encountered in logic textbooks. Object types, undefined expressions, and predefined operators need to be added to the syntax, semantics, and calculus of FOL in order to allow a natural and adequate translation. None of these extensions to FOL is new, but surprisingly no tutorial treatment of this material accessible to non-specialists is available. This justifies Chapter 2 in this book, where we give a self-contained treatment of a FOL tailored to the analysis of object-oriented designs.

1.4 Creating Formal Requirements

Our aim is to develop a formal specification alongside the design. Those requirements that cannot be captured diagrammatically must be formalized

either in OCL or in JML and are connected with the design in the form of class invariants or operation contracts. Technically, formal specifications are written as structured comments in JAVADOC style and precede the declaration of the context element they relate to. For example, constraint (1.1) appears in the file `PayCard.java` as follows:

JAVA
```
/**
 * @invariants
 * balance >= 0 and balance < limit
 */
public class PayCard { ... }
```
JAVA

But before proving that programs fulfill requirements, it is necessary to formalize requirements in the first place. So far, we only know that we can use OCL or JML, but formal requirements specification turns out to be not at all an easy task.

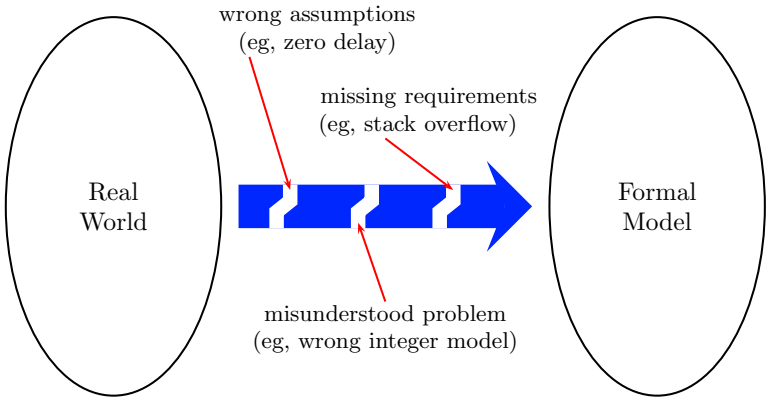

Fig. 1.4. Sources of specification errors

When we formalize any real phenomena, we necessarily have to abstract from the real world (see Fig. 1.3). Many times, this abstraction is the source of errors when specifying requirements. This may involve undue simplifications, missing or misunderstood (and, hence, erroneously modelled) requirements (see Fig. 1.4). Formalization, as we have seen and will see more below, can detect such errors, but it introduces also additional problems: one needs to master a specification language, but this is not enough—just like software, specifications should be well-crafted. Badly written formal specifications are at least as difficult to understand and debug as badly written programs. In

addition, formal specifications must be well structured and it must be possible to render them in natural language, otherwise informal and formal specifications are in constant need of synchronization. As a consequence, besides the two capabilities of expressing formal requirements and proving them, it is important to add a third one:

3. Support authoring of formal requirements by providing libraries with idioms and patterns, editors, integration with CASE tools, automatic translation to natural language.

We do not claim to have solved all this, but we address the problems and give partial solutions.

To take a very simple example, imposing upper and lower bounds on a variable with scalar datatype is a very common and typical class invariant, see (1.1). In the KeY system, we call this a *specification idiom*. The KeY extension of Borland Together offers a facility to use a number of specification idioms without the need to know OCL: by filling in forms. More generally, there is an extensible library of design patterns, each of which comes with a number of generic OCL constraints that capture some of the requirements associated with each pattern [Andersson, 2005]. In the hand of an OCL expert this becomes even a flexible extension mechanism for OCL. In Chapter 6 this is discussed further.

In an extensive case study it was shown that at least 25% of the formal specifications could be obtained from standard idioms and very few application-specific patterns [Bubel and Hähnle, 2005]. Although helpful, this leaves a considerable part of the formalization to be crafted by hand (a similar situation arises in coding, where code generators and templates do not go all the way).

There is some further support in obtaining specifications that can be given. When writing specifications it is useful to keep in mind that there are two important factors driving them:

- Structural properties of the design—for example, the class hierarchy.
- Functional requirements—typically, the state update and returned result that effects from calling a method.

The latter is well-known from the *design-by-contract* methodology [Meyer, 1992], an approximation to full verification. Design-by-contract can be seen as an efficient and elegant alternative (to, for example, dynamic typing) to check requirements at runtime, but it does not prove that the requirements actually hold. Still, the contract metaphor is very useful when specifying the functionality of a method. In our context we often call the pre- and postconditions of a method its contract.

One limitation of contracts is that they emphasize the (method-)local view and do not give a clue as to whether a program achieves its purpose as a whole. Such "global" properties are difficult to define and prove and one needs to

know the implementation of all methods. But already the structure of an OO design (that is, its associations and inheritance relation) gives important information on whether contracts and invariants are sufficient.

The two most important OO techniques for implementation by reuse are inheritance and delegation. They have strong consequences for the properties of an implementation and, therefore, need to be reflected in the formal specification. For inheritance, often Liskov's principle [Liskov and Guttag, 2000] is stipulated: it must be possible to replace an object of a class with an object declared in any of its subclasses without breaking the program. From a specification point of view, this means, for example, that invariants of subclasses must be implied by invariants of super classes.

Delegation is perhaps even more important and mostly preferable to inheritance [Gamma et al., 1995], but the consequences for formal specification are rarely made explicit. Assume, for example, that we want a method checkPIN(PIN:int):boolean in our PayCard class. Typically, this method would be implemented elsewhere, say, in class PIN, but in order to minimize the dependencies between PayCard and PIN (and to allow decoration) one might want to implement a delegator method checkPIN(PIN:int):boolean in PayCard. At this point, it is advisable to copy the contract from the implementing to the delegating method.

In summary, Liskov's principle applied to subclassing and delegation yields an important completion of a formal specification that is driven by structural design properties. In KeY it is even possible to prove that a given specification fulfills various aspects of Liskov's principle, see also the following section.

In the end, it is, of course, unavoidable to author a certain amount of formal specifications in OCL or JML by hand. This means that one has to master one of these languages. For a deeper understanding, it is even advisable to know the principles of the translation of OCL and JML into first-order logic. This, together with a concise introduction to the main syntactic elements of these languages, is the content of Chapter 5.

A major problem with formal specification is that formal and informal specifications tend to drift apart over time, because it is very tedious to keep them in sync. On the other hand, it is not sufficient to maintain merely a formal specification, because it cannot easily be communicated to managers or customers. The KeY tool addresses this problem with a feature for translation of OCL into natural language based on the Grammatical Framework [Ranta, 2004]. Example (1.1) is rendered in (stylistically suboptimal, but readily understandable) English as follows:

> "For the class PayCard the following invariant holds:
> the balance is at least 0 and the balance is less than the limit."

The translation tool, which also features a multi-lingual, syntax-directed editor for OCL, is explained in Chapter 7.

1.5 Proof Obligations

The invariants and contracts present in a design give no immediate clue of what one actually wants to verify. There is a wide range of possible *proof obligations* that can be constructed from invariants, contracts, method implementations, class initializers, and the class hierarchy as building blocks.

In the previous section, we mentioned already Liskov's principle as one possible property that one might wish to ensure for a given design and formal specification. In the KeY system, for such and other *lightweight design validation properties*, corresponding proof obligations expressed in the KeY program logic can be synthesized automatically from context sensitive menus available in the KeY plugins. Other lightweight properties include *invariant consistency*, *precondition disjointness*, *contract consistency*, and *strong operation contracts*. They are all formally defined and explained in detail in Chapter 8. Let us give an example for the last one. Given the following contract for a `charge` method of class PayCard:

OCL

```
context PayCard::charge(amount: Integer)
post :  balance = balance@pre + amount
```

OCL

It is desirable to ensure that after any returned method call, the invariant of its class is restored provided that it held before the call, here (1.1) (see p. 9). This is an essential part of any strategy to ensure that all class invariants hold at all times. We call this property *strong operation contract*.

The KeY prover fails to show this and it is in fact easy to see that the property does not hold, because the sum of `balance` and `amount` may well be greater than `limit`. This gives early feedback on the insufficiency of this particular contract (assuming we want to keep the invariant). Note that we do not need to prove at this time whether the contract is actually respected by `charge`. This can be postponed. In fact, the implementation of `charge` may well be unknown yet.

The contract can be patched either by adding a precondition or by weakening the postcondition. If we go for the former, the result looks like this:

OCL (1.2)

```
context PayCard::charge(amount: Integer)
pre :   balance + amount < limit and amount >=0
post :  balance = balance@pre + amount
```

OCL

Unfortunately, this is not sufficient to establish the strong operation contract property either. The problem is that the attribute `limit` might have been changed by `charge` (there is no problem for the argument `amount` which is immutable according to OCL semantics). One way to proceed is to strengthen

the postcondition with an expression such as `limit=limit@pre`. This becomes tedious in the presence of many attributes. Even worse, `charge` could have destroyed the invariants of *other* classes involving any public attribute in the whole system. The problem to succinctly express what *has not been modified* by an action is known as the *frame problem* in Artificial Intelligence [Shoham, 1986]. To get a handle on it, it is much more efficient to say what *has been modified* and to assume everything else is not. In JML a list of locations that are assignable by a method can be specified. OCL lacks this feature, but can easily be extended. The complete example including an *assignable clause* is:

OCL
```
context      PayCard::charge(amount: Integer)
assignable : balance
pre :        balance + amount < limit and amount >=0
post :       balance = balance@pre + amount
```
OCL

With this information a proof obligation can be synthesized that establishes the strong operation contract property. The key point is that the assignable clause ensures that `limit` is unchanged and this is enough to establish invariant (1.1). Part of the information contained in assignable clauses can sometimes be derived automatically, for example, the {frozen} property in the class diagram Fig. 1.2 suggests to leave out the location `limit` from the locations modifiable by `charge`. Of course, the validity of the assignable clause of a method must be proven for a given implementation.

Even though assignable clauses make proofs much simpler, it is still a problem that *all* class invariants in a system can be potentially affected by *any* method of *any* class. There is much ongoing research to alleviate this problem, for example, program slicing, containment, type-based approaches, etc. Common to all approaches is the idea that suitable statically checkable information about which methods affect which classes can be used to soundly omit most invariants from a proof. Such kind of analyses are indispensable for *modular verification*, because practically all proof obligations make it necessary to include all existing invariants in order to be sound. Modular verification is discussed in Chapter 8.

1.6 Proving Correctness of Programs

Design and coding should be different activities during software construction. In formal verification, this is reflected by the fact that we generate quite different proof obligations for source code verification as compared to design validation in the previous section. The most standard proof obligation is *total correctness* of a method implementation with respect to its contract. Consider the following implementation of the `charge` method:

JAVA
```
public void charge(int amount) {
  if (this.balance + amount >= this.limit) {
    this.unsuccessfulOperations++;
  } else {
    this.balance = this.balance + amount;
  }
}
```
JAVA

Total correctness with respect to contract (1.2) means: if `charge` is called in any state satisfying the precondition, then `charge` terminates normally (that is, without throwing an exception) and in its final state the postcondition holds. The assignable clause is not part of this proof obligation, but creates a proof obligation of its own.

Note that the first branch of the conditional does not compromise correctness, because the precondition ensures that this branch is never taken. This shows that both pre- and postconditions of contracts are essential to specify method correctness.

Correctness cannot be expressed in specification languages such as OCL or JML, because it is necessary to logically relate specification expressions and source code in non-trivial ways in order to produce proof obligations. For example, most notions of a proof obligation for total correctness would encompass not merely the preconditions of the contract, but also the class invariant (in fact, *all* class invariants). Clearly, first-order logic is not sufficient to express correctness either—a dedicated program logic is necessary.

The best-known program logic is Hoare logic [Hoare, 1969]. In KeY we use an extension of Hoare logic called *dynamic logic* [Harel et al., 2000a]. The main difference is that dynamic logic is syntactically closed under all propositional and first-order operators. The advantage is increased expressiveness: one can express not merely program correctness, but also security properties [Mostowski, 2005], correctness of program transformations, or the validity of assignable clauses. Other verification approaches [Paulson, 1994, Boyer, M. Broy and M. Pizka] encode program syntax and semantics in higher-order logic, but this creates considerable overhead, in particular during interactive proving. Dynamic logic, like Hoare logic, works directly on the source code.

The program logic of KeY is called JAVA CARD DL. It has been axiomatized in a sequent calculus and it is relatively complete[6] for any given JAVA CARD program. The actual verification process in KeY can be envisaged as *symbolic execution* of source code. Loops and recursion are handled by induction over

[6] It is well-known that Turing-complete programming languages cannot be completely axiomatized by first-order program logics. As usual, we supply an induction schema to approximate completeness. The axiomatization is relatively complete to Peano arithmetic. The incompleteness phenomenon is irrelevant for programs that occur in practice.

data structures occurring in the verification target. Alternatively, partial correctness of loops can also be shown by a rule that uses invariants. Symbolic execution plus induction as a verification paradigm was originally suggested for informal usage by Burstall [1974]. The idea to use dynamic logic as a basis for mechanization of symbolic execution was first realized in the Karlsruhe Interactive Verifier (KIV) tool [Heisel et al., 1987]. Symbolic execution is extremely suitable for interactive verification, because proof progress corresponds to program execution, which makes it easy to interpret intermediate stages in a proof and failed proof attempts.

JAVA CARD is a complex language and this is reflected in the logic JAVA CARD DL. Therefore, in Chapter 3 we break down JAVA CARD DL into several modular components. The core component (⇒ Sect. 3.6) defines symbolic execution rules for JAVA programs with bounded loops and without method calls. Such programs always terminate and it is possible to execute them symbolically in a fully automatic way. The result is the symbolic state update reached after the program terminates (more precisely, a set of updates, each corresponding to one or more execution branches). Updates are applied to first-order postconditions essentially via syntactic substitution, resulting in pure first-order verification conditions dealt with by the first-order rules (⇒ Chap. 2). Further components of the JAVA CARD DL calculus add independent mechanisms for handling loops (⇒ Sect. 3.7) and method calls (⇒ Sect. 3.8). It is essential for efficiency to simplify resulting state updates eagerly after each symbolic execution step. Update simplification is contained in a sepearate component (⇒ Sect. 3.9).

Full treatment of some JAVA features is so complex that particular chapters have been devoted to them. For example, the `charge` method above is not correct with respect to JAVA integer types, which are finite. How to handle JAVA integer semantics correctly and efficiently is shown in Chapter 12. There is also a dedicated chapter on data structure induction (⇒ Chap. 11).

JAVA CARD has two features that JAVA does not have:

- *Persistent memory* that resides in EEPROM, in addition to standard *transient* memory residing in RAM.
- *Atomic transactions* brace a sequence of statements that are either executed until completion or not executed at all. Transactions are crucial to avoid inconsistent data in the case of interrupted computations caused by card tear-out, power loss, communication failure, etc.

The combination of both features is surprisingly complex to model, because the semantics of transactions treats the two kinds of memory differently. We devote Chapter 9 to the logical modelling of JAVA CARD transactions.

As mentioned above, verification based on the JAVA CARD DL calculus corresponds to symbolic program execution (plus induction). Its rules can, therefore, be seen as an operational semantics for JAVA CARD. As long as neither unbounded loops nor recursion occurs, it is possible to execute programs symbolically almost without search, even though the full calculus contains

several hundred rules. In the case study in Chapter 14 proofs with several ten thousand nodes are automatically constructed in a matter of minutes. At the same time, the KeY prover is very flexible: for example, there is a rule set that axiomatizes Gurevich Abstract State Machines [Nanchen et al., 2003], one for a fragment of C [Gladisch, 2006], one for a object-oriented core language called ODL [Platzer, 2004a], and one for simplification of OCL expressions [Giese and Larsson, 2005]. Work on support for MISRA-C 2004[7] source code is in progress.

It is interesting to look at the reasons how it is possible to create and maintain support for such a variety of imperative languages with relatively modest effort. Many interactive theorem provers are implemented in a meta programming language for rule and proof construction. The advantage is generality: not only software verification but any kind of mathematics can be modelled; in addition, not only rules can be described, but also the way how to prove them. In contrast to this, the KeY rules must adhere to a very specific schema language we call *taclet* [Beckert et al., 2004], which is tailored to the needs of interactive verification. Taclets specify not only the logical content of a rule, but also the context and pragmatics of its application. Since taclets have limited reflection capabilities, the set of primitive rules in JAVA CARD DL that have to be considered as axiomatic is relatively large with over a hundred. Correctness of these rules must be shown with external tools [Ahrendt et al., 2005b, Trentelman, 2005]. But the advantages of taclets are enormous: they can be efficiently compiled not only into the rule engine, but also into the automation heuristics and into the GUI. In addition, it is a matter of hours to master the taclet language. In many cases, new taclets can be verified within KeY by reflection. In Chapter 4, the taclet concept as well as correctness of taclets is discussed. An introduction into proof search and the GUI of the KeY prover is found in Chapter 10.

[7] `http://www.misra-c2.com/`