Verifying Object-Oriented Programs with KeY: A Tutorial

Wolfgang Ahrendt[1], Bernhard Beckert[2], Reiner Hähnle[1], Philipp Rümmer[1], and Peter H. Schmitt[3]

[1] Department of Computer Science and Engineering,
Chalmers University of Technology and Göteborg University
ahrendt|reiner|philipp@chalmers.se
[2] Department of Computer Science,
University of Koblenz-Landau
beckert@uni-koblenz.de
[3] Department of Theoretical Computer Science,
University of Karlsruhe
pschmitt@ira.uka.de

Abstract. This paper is a tutorial on performing formal specification and semi-automatic verification of Java programs with the formal software development tool KeY. This tutorial aims to fill the gap between elementary introductions using toy examples and state-of-art case studies by going through a self-contained, yet non-trivial, example. It is hoped that this contributes to explain the problems encountered in verification of imperative, object-oriented programs to a readership outside the limited community of active researchers.

1 Introduction

The KeY system is the main software product of the KeY project, a joint effort between the University of Karlsruhe, Chalmers University of Technology in Göteborg, and the University of Koblenz. The KeY system is a formal software development tool that aims to integrate design, implementation, formal specification, and formal verification of object-oriented software as seamlessly as possible.

This paper is a tutorial on performing formal specification and semi-automatic verification of Java programs with KeY. There is already a tutorial introduction to the KeY prover that is set at the beginner's level and presupposes no knowledge of specification languages, program logic, or theorem proving [3, Chapt. 10]. At the other end of the spectrum are descriptions of rather advanced case studies [3, Chapt. 14 and 15] that are far from being self-contained. The present tutorial intends to fill the gap between first steps using toy examples and state-of-art case studies by going through a self-contained, yet non-trivial, example. We found few precisely documented and explained, yet realistic, case studies even for other verification systems. Therefore, we believe that this tutorial is of interest in its own right, not only for those who want to know about KeY.

We hope that it can contribute to explain the problems encountered in verification of imperative, object-oriented programs to a readership outside the limited community of active researchers.

We assume that the reader is familiar with the Java programming language, with first-order logic and has some experience in formal specification and verification of software, presumably using different approaches than KeY. Specifications in the Java Modeling Language (JML) [16] and expressions in KeY's program logic Java Card DL [3, Chapt. 3] are explained as far as needed.

In this tutorial we demonstrate in detail how to specify and verify a Java application that uses most object-oriented and imperative features of the Java language. The presentation is such that the reader can trace and understand almost all aspects. To this end, we provided the complete source code and specifications at `www.key-project.org/fmco06`. We strongly encourage reading this paper next to a computer with a running KeY system. Recently, version 1.0 of the KeY system has been released in connection with the KeY book [3]. The KeY tool is available under GPL and can be freely downloaded from `www.key-project.org`. Information on how to install the KeY tool can also be found on that web site. The experiments in this tutorial were performed with KeY version 1.1 which has improved support for proof automation. It is available at the location mentioned above together with the case study.

The tutorial is organised as follows: in Section 2 we provide some background on the architecture and technologies employed in the KeY system. In Section 3 we describe the case study that is used throughout the remaining paper. It is impossible to discuss all verification tasks arising from the case study. Therefore, in Section 4, we walk through a typical proof obligation (inserting an element into a datastructure) in detail including the source code, the formal specification of a functional property in JML, and, finally, the verification proof. In Section 5 we repeat this process with a more difficult proof obligation. This time around, we abstract away from most features learned in the previous section in favour of discussing some advanced topics, in particular complex specifications written in Java Card DL, handling of complex loops, and proof modularisation with method contracts. We conclude with a brief discussion.

2 The KeY Approach

The KeY Program Verification System. KeY supports several languages for specifying properties of object-oriented models. Many people working with UML and MDA have familiarity with the specification language OCL (Object Constraint Language), as part of UML 2.0. KeY can also translate OCL expressions to natural language (English and German). Another specification language supported by KeY, which enjoys popularity among Java developers and which we use in this paper, is the Java Modeling Language (JML). Optional plugins of KeY into the popular Eclipse IDE and the Borland Together CASE tool suite are available with the intention to lower initial adoption cost for users with no or little training in formal methods.

The target language for verification in KeY is Java Card 2.2.1. KeY is the only publicly available verification tool that supports the full Java Card standard including the persistent/transient memory model and atomic transactions. Rich specifications of the Java Card API are available both in OCL and JML. Java 1.4 programs that respect the limitations of Java Card (no floats, no concurrency, no dynamic class loading) can be verified as well.

The Eclipse and Together KeY plugins allow to select Java classes or methods that are annotated with formal specifications and both plugins offer to prove a number of correctness judgements such as behavioural subtyping, partial and total correctness, invariant preservation, or frame properties. In addition to the JML/OCL-based interfaces one may supply proof obligations directly on the level of Java Card DL. For this, a stand-alone version of the KeY prover not relying on Eclipse or Together is available.

The program logic Java Card DL is axiomatised in a *sequent calculus*. Those calculus rules that axiomatise program formulas define a symbolic execution engine for Java Card and so directly reflect the operational semantics. The calculus is written in a small domain-specific so-called *taclet* language that was designed for concise description of rules. Taclets specify not merely the logical content of a rule, but also the context and pragmatics of its application. They can be efficiently compiled not only into the rule engine, but also into the automation heuristics and into the GUI. Depending on the configuration, the axiomatisation of Java Card in the KeY prover uses 1000–1300 taclets.

The KeY system is not merely a verification condition generator (VCG), but a theorem prover for program logic that combines a variety of automated reasoning techniques. The KeY prover is distinguished from most other deductive verification systems in that symbolic execution of programs, first-order reasoning, arithmetic simplification, external decision procedures, and symbolic state simplification are interleaved.

At the core of the KeY system is the deductive verification component, which also can be used as a stand-alone prover. It employs a free-variable sequent calculus for first-order Dynamic Logic for Java. The calculus is proof-confluent, i.e., no backtracking is necessary during proof search.

While we constantly strive to increase the degree of automation, user interaction remains inexpendable in deductive program verification. The main design goal of the KeY prover is thus a seamless integration of automated and interactive proving. Efficiency must be measured in terms of user plus prover, not just prover alone. Therefore, a good user interface for presentation of proof states and rule application, a high level of automation, extensibility of the rule base, and a calculus without backtracking are all important features.

Syntax and Semantics of the KeY Logic. The foundation of the KeY logic is a typed first-order predicate logic with subtyping. This foundation is extended with parameterised modal operators $\langle p \rangle$ and $[p]$, where p can be any sequence of legal Java Card statements. The resulting multi-modal program logic is called Java Card Dynamic Logic or, for short, Java Card DL [3, Chapt. 3].

As is typical for Dynamic Logic, Java Card DL integrates programs and formulas within a single language. The modal operators refer to the final state of program p and can be placed in front of any formula. The formula $\langle p \rangle \phi$ expresses that the program p terminates in a state in which ϕ holds, while $[p]\phi$ does not demand termination and expresses that *if* p terminates, then ϕ holds in the final state. For example, "when started in a state where `x` is zero, `x++;` terminates in a state where `x` is one" can be expressed as $\texttt{x} \doteq 0 \texttt{ -> } \langle \texttt{x++;} \rangle (\texttt{x} \doteq 1)$. The states used to interpret formulas are first-order structures sharing a common universe.

The type system of the KeY logic is designed to match the Java type system but can be used for other purposes as well. The logic includes *type casts* (changing the static type of a term) and *type predicates* (checking the dynamic type of a term) in order to reason about inheritance and polymorphism in Java programs [3, Chapt. 2]. The type hierarchy contains the types such as *boolean*, the root reference type Object, and the type Null, which is a subtype of all reference types. It contains a set of user-defined types, which are usually used to represent the interfaces and classes of a given Java Card program. Finally, it contains several integer types, including both the range-limited types of Java and the infinite integer type $\mathbb{Z}$.

Besides built-in symbols (such as type-cast functions, equality, and operations on integers), user-defined functions and predicates can be added to the signature. They can be either *rigid* or *non-rigid*. Intuitively, rigid symbols have the same meaning in all program states (e.g., the addition on integers), whereas the meaning of non-rigid symbols may differ from state to state.

Moreover, there is another kind of modal operators called *updates*. They can be seen as a language for describing program transitions. There are simple function updates corresponding to assignments in an imperative programming language, which in turn can be composed sequentially and used to form parallel or quantified updates. Updates play a central role in KeY: the verification calculus transforms Java Card programs into updates. KeY contains a powerful and efficient mechanism for simplifying updates and applying them to formulas.

Rule Formalisation and Application. The KeY system has an automated-proof-search mode and an interactive mode. The user can easily switch modes during the construction of a proof.

For interactive rule application, the KeY prover has an easy to use graphical user interface that is built around the idea of direct manipulation. To apply a rule, the user first selects a *focus of application* by highlighting a (sub-)formula or a (sub-)term in the goal sequent. The prover then offers a choice of rules applicable at this focus. This choice remains manageable even for very large rule bases. Rule schema variable instantiations are mostly inferred by matching. A simpler way to apply rules and give instantiations is by drag and drop. If the user drags an equation onto a term the system will try to rewrite the term with the equation. If the user drags a term onto a quantifier the system will try to instantiate the quantifier with this term.

The interaction style is closely related to the way rules are formalised in the KeY prover. There are no hard-coded rules; all rules are defined in the "taclet

language" instead. Besides the conventional declarative semantics, taclets have a clear operational semantics, as the following example shows—a "modus ponens" rule in textbook notation (left) and as a taclet (right):

$$\frac{\phi, \psi, \Gamma \Longrightarrow \Delta}{\phi, \phi \rightarrow \psi, \Gamma \Longrightarrow \Delta}$$

```
\find (p -> q ==>)     // implication in antecedent
\assumes (p ==>)       // side condition
\replacewith(q ==>)    // action on focus
\heuristics(simplify)  // strategy information
```

The `find` clause specifies the potential application focus. The taclet will be offered to the user on selecting a matching focus and if a formula mentioned in the `assumes` clause is present in the sequent. The action clauses `replacewith` and `add` allow modifying (or deleting) the formula in focus, as well as adding additional formulas (not present here). The `heuristics` clause records information for the parameterised automated proof search strategy.

The taclet language is quickly mastered and makes the rule base easy to maintain and extend. Taclets can be proven correct against a set of base taclets [4]. A full account of the taclet language is given in [3, Chapt. 4 and Appendix B.3.3].

Applications. Among the major achievements using KeY in the field of program verification so far are the treatment of the Demoney case study, an electronic purse application provided by Trusted Logic S.A., and the verification of a Java implementation of the Schorr-Waite graph marking algorithm. This algorithm, originally developed for garbage collectors, has recently become a popular benchmark for program verification tools. Chapters 14 and 15 of the KeY book [3] are devoted to a detailed description of these case studies. A case study [14] performed within the HIJA project has verified the lateral module of the flight management system, a part of the on-board control software from Thales Avionics.

Lately we have applied the KeY system also on topics in security analysis [7], and in the area of model-based test case generation [2, 10] where, in particular, the prover is used to compute path conditions and to identify infeasible paths.

The flexibility of KeY w.r.t. the used logic and calculus manifests itself in the fact that the prover has been chosen as a reasoning engine for a variety of other purposes. These include the mechanisation of a logic for Abstract State Machines [17] and the implementation of a calculus for simplifying OCL constraints [13].

KeY is also very useful for teaching logic, deduction, and formal methods. Its graphical user interface makes KeY easy to use for students. They can step through proofs with different degrees of automation (using the full verification calculus or just the first-order core rules). The authors have been successfully teaching courses for several years using the KeY system. An overview and course material is available at `www.key-project.org/teaching`.

Related Tools. There exist a number of other verification systems for object-oriented programs. The KIV[4] tool [1] is closest to ours in that it is also interactive

[4] `www.informatik.uni-augsburg.de/lehrstuehle/swt/se/kiv/`

and also based on Dynamic Logic. Most other systems are based on a verification condition generator (VCG) architecture and separate the translation of programs into logic from the actual proof process. A very popular tool of this kind is ESC/Java2[5] (Extended Static Checker for Java2) [12], which uses the Simplify theorem prover [8] and attempts to find run-time errors in JML-annotated Java programs. ESC/Java2 compromises on completeness and even soundness for the sake of ease of use and scalability. Further systems are JACK [5], Krakatoa [11], LOOP [18], which can also generate verification conditions in higher order logic that may then be proved using interactive theorem provers like PVS, Coq, Isabel, etc. Like KeY, JACK, Krakatoa, and LOOP support JML specifications. With JACK, we moreover share the focus on smart card applications.

3 Verification Case Study: A Calendar Using Interval Trees

In this tutorial, we use a small Java calendar application to illustrate how specifications are written and programs are verified with the KeY system. The application provides typical functionality like creating new calendars, adding or removing appointments, notification services that inform about changes to a particular appointment or a calendar, and views for displaying a time period (like a particular day or month) or for more advanced lookup capabilities.

The class structure of the calendar application is shown in Fig. 1 and 2. It consists of two main packages: a datastructure layer `intervals`, that provides classes for working with (multisets of) intervals, and a domain layer `calendar`, that defines the actual logic of a calendar. Intervals (interface `Interval`) are the basic entities that our calendars are built upon. In an abstract sense, each entry or appointment in a calendar is primarily an interval spanned by its start and its end point in time. A calendar is a multiset of such intervals. For reasons of simplicity, we represent discrete points of time as integers, similarly to the time representation in Unix (the actual unit and offset are irrelevant here). Further, we use the *observer design pattern* (package `observerPattern`) for being able to observe all modifications that occur in a calendar entry.

Interval Datastructures. The most important lookup functionality that our calendar provides, is the ability to retrieve all entries that overlap a certain query time interval (i.e., have a point of time in common with the query interval). Such queries are used, for instance, when displaying all appointments for a particular day. We consequently store intervals in an *interval tree* datastructure [6] (class `IntervalTree` in Fig. 2), which allows to retrieve overlapping entries with logarithmic complexity in the size of the calendar. An interval tree is a binary tree, in which each node (class `IntervalTreeNode`) stores (a) the multiset of intervals that include a certain point (the `cutPoint`) and (b) pointers to the subtrees that handle the intervals strictly smaller (association `left`) resp. strictly bigger

[5] `http://secure.ucd.ie/products/opensource/ESCJava2/`

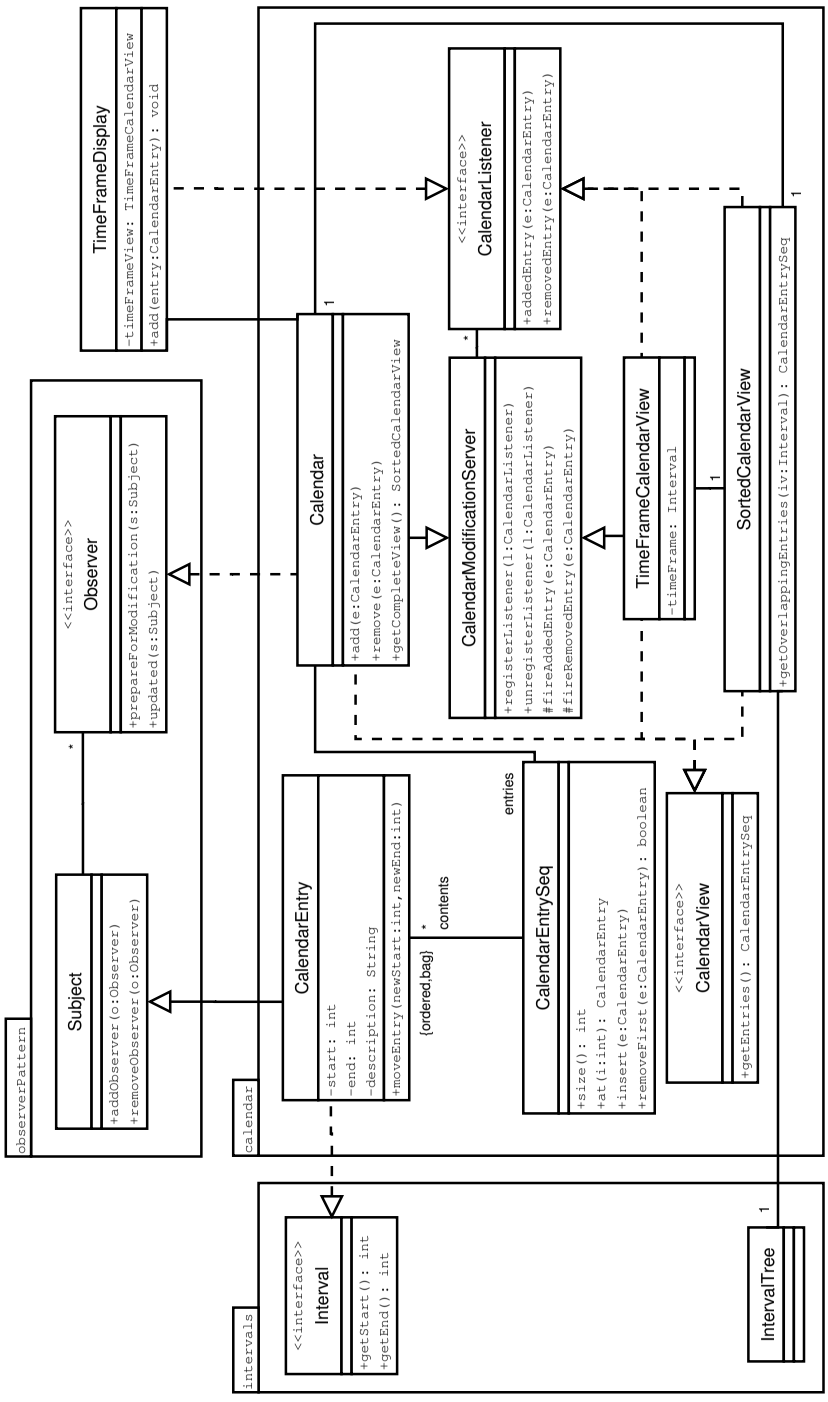

Fig. 1. The packages `observerPattern` and `calendar` of the calendar case study

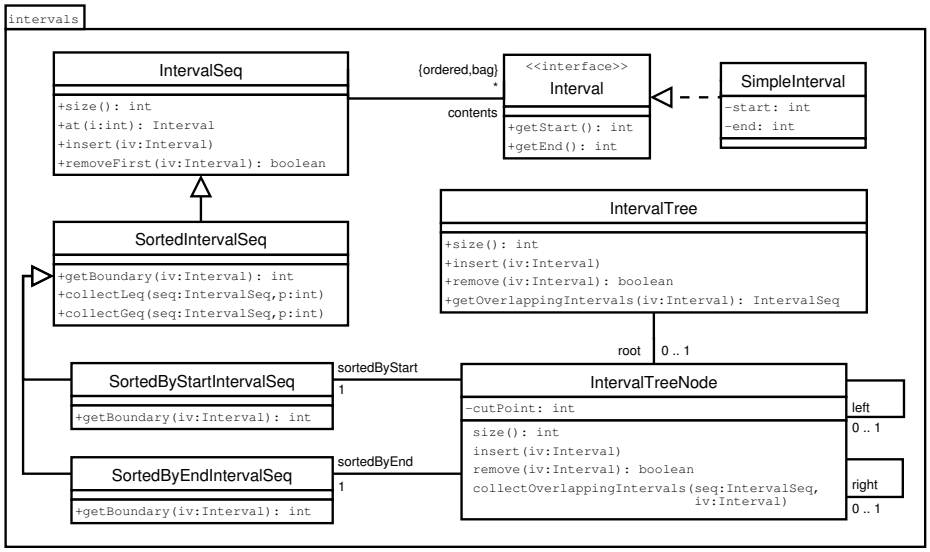

Fig. 2. The package `intervals` of the calendar case study

(association `right`) than the cut point. The intervals belonging to a particular node have to be stored both sorted by the start and by the end point, which is further discussed in Sect. 4.

Package Calendar. The two primary classes that implement a calendar are `CalendarEntry` for single appointments (an implementation of the interface `Interval`) and `Calendar` for whole calendars. The basic `Calendar` provides the interface `CalendarView` for accessing all entries that are part of the calendar in an unspecified order. A more advanced lookup interface, `SortedCalendarView`, can be accessed through the method `getCompleteView` of `Calendar`. It allows to retrieve all entries that overlap with a given interval. This interface is realised using the interval trees from package `intervals`.

A further view on calendars is `TimeFrameCalendarView`, which pre-selects all appointments within a given period of time, and which is based on the class `SortedCalendarView`. Both `Calendar` and `TimeFrameCalendarView` also provide a notification service (`CalendarModificationServer`) that informs about newly added and removed entries. We illustrate the usage of this service (and of `TimeFrameCalendarView`) in class `TimeFrameDisplay`, which is further discussed and verified in Sect. 5.

4 First Walk-through: Verifying Insertion into Interval Sequences

In this section, we zoom into a small part of the scenario described above, namely the `insert()` method belonging to the class `IntervalSeq` and its subclasses. In

the context of that method, we demonstrate the different basic stages of formal software development with the KeY system. We discuss the *formal specification* of the `insert()` method, the generation of corresponding *proof obligations* in the used program logic, and the *formal verification* with the KeY prover. Along with demonstrating the basic work-flow, we introduce the used formalisms on the way, when they appear, but just to the extent which allows to follow the example. These formalisms are: the specification language JML (Java Modeling Language) [16], the program logic Java Card DL, and the corresponding calculus.[6]

As described in Section 3, the basic data structure of the case study scenario is a tree, the nodes of which are instances of the class `IntervalTreeNode`. Each such node contains one integer number (representing a point in time, the "cut point" of the node), and two interval sequences, both containing the same intervals (all of which contain the cut point of the node). The difference between the two sequences is that the contained intervals are sorted differently, once by their start, and once by their end.

Correspondingly, the two sequences contained in each node are instances of the classes `SortedByStartIntervalSeq` and `SortedByEndIntervalSeq`, respectively. Both are subclasses of `SortedIntervalSeq`, which in turn is a subclass of `IntervalSeq`. One of the basic methods provided by (instances of) these classes is `insert(Interval iv)`. In this section, we discuss, as an example, the implementation and specification of that method, as well as the verification of corresponding proof obligations.

The specification of the `insert()` method of the class `SortedIntervalSeq` also involves the superclass, `IntervalSeq`, because parts of the specification are *inherited* from there. Later, in the verification we will also be concerned with the two subclasses of `SortedIntervalSeq`, which provide different implementations of a method called by `insert()`, namely `getBoundary()`.

4.1 Formal Specification and Implementation

Within the Class `IntervalSeq`. This class is the topmost one in this small hierarchy, instances of which represent a sequence of intervals. Internally, the sequence is realised via an array `contents` of type `Interval[]`. This array can be longer than the actual `size` of the interval sequence. Thereby, we avoid having to allocate a new array at each and every increase of the sequence's `size`. Instead, `size` points to the index up to which we consider `contents` be filled with "real" intervals; only if `size` exceeds `contents.length`, a new array is allocated, into which the old one is copied. This case distinction is encapsulated in the method `incSize()`, to be called by `insert()`.

Java (1.1)

```
protected void incSize() {
```

[6] All these are described in more detail in the KeY book [3]: JML in Section 5.3, Java Card DL and the calculus in Chapter 3.

```
  ++size;
  if ( size > contents.length ) {
    final Interval[] oldAr = contents;
    contents = new Interval[contents.length * 2];
    int i = 0;
    while (i < oldAr.length) {contents[i] = oldAr[i]; ++i;}
  }
}
```

Java

We turn to the actual `insert()` method now. The class `IntervalSeq` is ignorant of sorting, so all we require from `insert(iv)` is that `iv` is indeed inserted, *wherever*, in the sequence. To the very least, this means that, in a post state, `iv` is stored at *any* of the indices of `contents`. Using mathematical standard notation, we can write this as

$$\exists i.\ 0 \leq i \wedge i < \texttt{size} \wedge \texttt{contents}[i] = \texttt{iv}$$

Note that, already in this mathematical notation, we are mixing in elements from the programming language level, namely the instance field names, and the array access operator "`[ ]`". Now, the specification language JML takes this several steps further, using Java(like) syntax wherever possible: `<=` for $\leq$, `&&` for $\wedge$, `==` for $=$, `!=` for $\neq$, and so on. Special keywords are provided for concepts not covered by Java, like `\exists` for $\exists$. Altogether, the above formula is expressed in JML as:

```
\exists int i; 0 <= i && i < size; contents[i] == iv
```

As we can see, quantified formulas in JML have three parts, separated by "`;`". The first declares the type of the quantified variable, the second is intended to further restrict the range of the variable, while the third states the "main" property, intuitively speaking. Logically, however, the second and the third part of a JML "`\exists`"-formula are connected via "and" ($\wedge$).

The above formula is a *postcondition*, as it constrains the admissible states after execution of `insert()`. A sensible *precondition* would be that the interval to be inserted is defined, i.e., not `null`.

Even if we will later expand on the postcondition, we show already how the formulas we have so far can syntactically be glued together, in order to form a JML specification of `insert()`. In general, JML specifications are written into Java source code files, in form of Java comments starting with the symbol "`@`". In case of pre/post-conditions, these comments precede the method they specify. In our example, `IntervalSeq.java` would contain the following lines:

Java + JML (1.2)

```
/*@ public normal_behavior
  @  requires iv != null;
  @  ensures (\exists int i; 0 <= i && i < size;
  @               contents[i] == iv);
```

```
  @*/
public void insert(Interval iv) {
    ...
```

Java + JML

This is an example for a *method contract* in JML. For the purpose of our example, this contract is, however, still very weak. It does, for instance, not specify how the values of `size` before and after execution of `insert()` relate to each other. For such a purpose, JML offers the "`\old`" construct, which is used in a postcondition to refer back to the pre-state. With that, we can state `size == \old(size) + 1`. Further, the contract does not yet tell whether all (or, in fact, any) of the intervals previously contained in `contents` remain therein, not to speak of the indices under which they appear. What we need to say is (a) that, up to the index `i` where `iv` is inserted, the elements of `contents` are left untouched, and (b) that all other elements are shifted by one index. Both can be expressed using the universal quantifier in JML, "`\forall`", which is quite analogous to the "`\exists`" operator. Using that, (b) would translate to:

```
\forall int k; i < k && k < size;
               contents[k] == \old(contents[k-1])
```

Note that, in case of "`\forall`", the second "`;`" logically is an implication, not a conjunction as was the case for "`\exists`". In the above formula, `i` refers to the index of insertion, which we have existentially quantified over earlier, meaning we get a nested quantification here.

Together with an appropriate `assignable` clause to be explained below, we now arrive at the following JML specification of `insert()`:

Java + JML (1.3)

```
/*@ public normal_behavior
  @  requires iv != null;
  @  ensures size == \old(size) + 1;
  @  ensures (\exists int i; 0 <= i && i < size;
  @               contents[i] == iv
  @               && (\forall int j; 0 <= j && j < i;
  @                      contents[j] == \old(contents[j]))
  @               && (\forall int k; i < k && k < size;
  @                      contents[k] == \old(contents[k-1])));
  @  assignable contents, contents[*], size;
  @*/
public void insert(Interval iv) {
    ...
```

Java + JML

The `assignable` clause, in this example, says that the `insert()` is allowed to change the value of `contents`, the value of the element locations of `contents`,

and of `size`, *but nothing else*. The purpose of the `assignable` clauses is not so much the verification of the method `insert` (in this case), but rather to keep feasible the verification of other methods calling `insert()`.

Within the Abstract Class `SortedIntervalSeq`. This class `extends` the class `IntervalSeq`, augmenting it with the notion of *sortedness*. In particular, this class' implementation of `insert()` must respect the sorting. To specify this requirement in JML, one could be tempted to add sortedness to both, the pre- and the postcondition of `insert()`. However, such invariant properties should rather be placed in JML *class invariants*, which like method contracts are added as comments to the source code.

The following lines are put *anywhere* within the class `SortedIntervalSeq`:

JML (1.4)

```
/*@ public invariant
  @   (\forall int i; 0 <= i && i < size - 1;
  @       getBoundary(contents[i]) <= getBoundary(contents[i+1]));
  @*/
```

JML

The actual sorting criterion, `getBoundary()`, is left to subclasses of this class, by making it an `abstract` method.

Java + JML (1.5)

```
protected /*@ pure @*/ abstract int getBoundary(Interval iv);
```

Java + JML

The phrase "`/*@ pure @*/`" is another piece of JML specification, stating that all implementations of this method terminate (on all inputs), and are free of side effects. Without that, we would not be allowed to use `getBoundary()` in the invariant above, nor in any other JML formula.

Finally, we give the `SortedIntervalSeq` implementation of `insert()` (overriding some non-sorted implementation from `IntervalSeq`):

Java (1.6)

```
public void insert(Interval iv) {
    int i = size;
    incSize ();
    final int ivBoundary = getBoundary( iv );
    while ( i > 0 && ivBoundary < getBoundary( contents[i-1] ) ) {
        contents[i] = contents[i - 1];
        --i;
    }
    contents[i] = iv;
}
```

Java

Within the `SortedByStart...` and `SortedByEnd...` Classes. These two classes extend `SortedIntervalSeq` by defining the sorting criteria to be the "start" resp. "end" of the interval. Within `SortedByStartIntervalSeq`, we have:

Java + JML (1.7)

```
protected /*@ pure @*/ int getBoundary(Interval iv) {
    return iv.getStart ();
}
```

Java + JML

and within `SortedByEndIntervalSeq`, we have

Java + JML (1.8)

```
protected /*@ pure @*/ int getBoundary(Interval iv) {
    return iv.getEnd ();
}
```

Java + JML

4.2 Dynamic Logic and Proof Obligations

After having completed the specification as described in the previous section we start `bin/runProver` (in your KeY installation directory) as a first step towards verification. The graphical user interface of the KeY prover will pop up. To load files with Java source code and JML specifications, we select File → Load ... (or in the tool bar). For the purposes of this introduction we navigate to where the `calender-sources`[7] are stored locally, select that very directory (not any of the sub-directories), and push the open button. After an instant the JML specification browser will appear on the screen. In the left-most of its three window panes, the Classes pane, we expand the folder corresponding to the package `intervals` and select the class `IntervalSeq`. The Methods pane now shows all methods of class `IntervalSeq`. We select void insert(Interval iv). Now also the Proof Obligations pane shows some entries. We select first the specification case normal_behavior and push Load Proof Obligation. This brings us back to the KeY prover interface.

Now, the Tasks pane records the tasks we have loaded (currently one) and the main window Current Goal shows the proof obligation. It looks quite daunting and we use the rest of this section to explain what you see there. The construction of the actual proof is covered in the next section. Ignoring the leading `==>`, the proof obligation is of the form shown in Fig. 3.

Java Card DL. Fig. 3 shows a formula of Dynamic Logic (DL), more precisely Java Card DL, see Section 2. The reader might recognise typical features of first-order logic: the propositional connectives (e.g., `->` and `&`), predi-

[7] The sources can be downloaded from `www.key-project.org/fmco06`.

KeY

```
1     inReachableState
2  -> \forall intervals.Interval iv_lv;
3      {iv:=iv_lv}
4       (  iv.<created> = TRUE & !iv = null
5          | iv = null
6       -> \forall intervals.IntervalSeq self_IntervalSeq_lv;
7           {self_IntervalSeq:=self_IntervalSeq_lv}
8            ( \forall jint k;
9                _old20(k) = self_IntervalSeq.contents[k - 1]
10           -> \forall jint j; _old19(j) = self_IntervalSeq.contents[j]
11           ->   !self_IntervalSeq = null
12              & self_IntervalSeq.<created> = TRUE
13              & !iv = null
14              & ...
15              & !self_IntervalSeq.contents = null
16              & self_IntervalSeq.contents.length >=
17                self_IntervalSeq.size
18              & self_IntervalSeq.contents.length >= 1
19              & \forall jint i;
20                 ( 0 <= i & i < self_IntervalSeq.size
21                 -> !self_IntervalSeq.contents[i] = null )
22              & ...
23              & self_IntervalSeq.size >= 0
24           -> {_old17:=self_IntervalSeq.size || _old18_iv:=iv}
25               \<{self_IntervalSeq.insert(iv)@intervals.IntervalSeq;}\>
26                (   self_IntervalSeq.size = _old17 + 1
27                  & \exists jint i; ... )
28           )
29       )
```

KeY

Fig. 3. Proof obligation for the `insert` method in class `IntervalSeq`

cates (e.g., `inReachableState`, a predicate of arity 0), equality, constant symbols (e.g., `self_IntervalSeq`), unary function symbols (e.g., `size`), and quantifiers (e.g., `\exists jint i;`). The function symbol `size` is the logical counterpart of the attribute of the same name. Note also that Java Card DL uses dot-notation for function application, for example, `self_IntervalSeq.size` instead of `size(self_IntervalSeq)` on line 17. On line 2, the quantification `\forall intervals.Interval iv_lv;` introduces the quantified variable `iv_lv` of type `Interval` (for disambiguation the package name `intervals` is prefixed). What makes Java Card DL a proper extension of first-order logic are modal operators. In the above example the diamond operator

```
\<{self_IntervalSeq.insert(iv)@intervals.IntervalSeq;}\>
```

occurs on line 25 (note that in KeY the modal operators `<>` and `[]` are written with leading backslashes). In general, if *prog* is any sequence of legal Java Card statements and F is a Java Card DL formula, then \<*prog*\>F is a Java Card DL formula too. As already explained in Section 2, the formula \<*prog*\>F is true in a state s_1 if there is a state s_2 such that `prog` terminates in s_2 when started in s_1 and `F` is true in s_2. The box operator `\[...\]` has the same semantics except that it does not require termination.

In theoretical treatments of Dynamic Logic there is only one kind of variable. In Java Card DL we find it more convenient to separate logical variables (e.g., `iv_lv` in the above example), from program variables (e.g. `iv`). Program variables are considered as (non-rigid) constant symbols in Java Card DL and may thus not be quantified over. Logical variables on the other hand are not allowed to occur within modal operators, because they cannot occur as part of Java programs.

State Updates. We are certainly not able to touch on all central points of Java Card DL in this quick introduction, but there is one item we cannot drop, namely *updates*. Let us look at line 3 in Fig. 3. Here `{iv:=iv_lv}` is an example of an update. More precisely, it is an example of a special kind of update, called *function update*. The left-hand side of a function update is typically a program variable, as `iv` in this example, or an array or field access. The right-hand side can be an arbitrary Java Card DL term, which of course must be compatible with the type of the left-hand expression. Constructs like `{i:=j++}` where the right-hand side would have side-effects are *not* allowed in updates. If `{`*lhs*`:=`*rhs*`}` is a function update and F is a formula, then `{`*lhs*`:=`*rhs*`}`F is a Java Card DL formula. The formula `{`*lhs*`:=`*rhs*`}`F is true in state s_1 if F is true in state s_2 where s_2 is obtained from s_1 by *performing* the update. For example, the state s_2 obtained from s_1 by performing the update `{iv:=iv_lv}` (only) differs in the value of `iv`, which is in s_2 the value that `iv_lv` has in s_1. Java Card DL furthermore allows combinations of updates, e.g., by sequential or parallel composition. An example of such parallel updates, composed via "`||`", appears in line 24 of Fig. 3.

One difference between updates and Java assignment statements is that logical variables such as `iv_lv` may occur on the right hand side of updates. In Java Card DL it is not possible to quantify over program variables. This is made up for by the possibility of quantifying over logical variables in updates (as in lines 2 and 3 in Fig. 3). Another role of updates is to store "old" values from the pre-state, that then can be referred to in the post-state (as in lines 24 and 26). Finally, the most important role of updates is that of *delayed substitutions*. During symbolic execution (performed by the prover using the Java Card DL calculus) the effects of a program are removed from the modality \<...\> and turned into updates, where they are *simplified* and *parallelised*. Only when the modality has been eliminated, updates are substituted into the post-state. For a more thorough discussion, we refer to the KeY book [3, Chapt. 3].

Kripke Semantics. A state $s \in S$ contains all information necessary to describe the complete snapshot of a computation: the existing instances of all types, the

values of instance fields and local program variables etc. Modal logic expressions are not evaluated relative to one state model but relative to a collection of those, called a *Kripke Structure*. There are *rigid* symbols that evaluate to one constant meaning in all states of a *Kripke Structure*. The type `jint` (see e.g., line 8 in Fig. 3) in all states evaluates to the (infinite) set of integers, also addition `+` on `jint` are always evaluated as the usual mathematical addition. Logical variables also count among the rigid symbols, no program may change their value. On the other hand there are *non-rigid* symbols like `self_IntervalSeq`, `iv`, `contents`, or `at(i)`.

Proof Obligations. We have to add more details on Java Card DL as we go along but we are now well prepared to talk about proof obligations. We are still looking at Fig. 3 containing the proof obligation in the Current Goal pane that was generated by selecting the normal_behaviour specification case. Line 11 contains the `requires` clause from the JML method specification for `insert`, while the conjunction of all JML invariants for class `IntervalSeq` appears in lines 15 to 21. If you also need invariants of superclasses or classes that occur as the type of a field in `IntervalSeq` you would tick the *use all applicable invariants* box in the JML browser before generating the proof obligation. Since in the JML semantics normal_behaviour includes the termination requirement, the diamond modality is used. Starting with line 26, the first line within the scope of the modal operator, follows the conjunction of the `ensures` clauses in the JML method specification. It is implicit in the JML specification that the above implication should be true for all values of the method parameter `iv` and all instances of class `IntervalSeq` as calling object. As we have already observed this is accomplished in Java Card DL by universal quantification of the logical variables `iv_lv` and `self_IntervalSeq_lv` and *binding* these values to the program variables via the updates `{iv := iv_lv}` and `{self_IntervalSeq := self_IntervalSeq_lv}`. Looking again at Fig. 3, we notice in lines 4 and 5 additional restrictions on the implicitly, via `iv_lv`, universally quantified parameter `iv`. To understand what we see here, it is necessary to explain how Java Card DL handles object creation. Instead of adding a new element to the target state s_2 of a statement `prog` that contains a call to the `new` method we adopt what is called the *constant domain assumption* in modal logic theory. According to this assumption all states share the same objects for all occurring types. In addition there is an implicit field `<created>` of the class `java.lang.Object` (to emphasise that this is not a normal field, it is set within angled brackets). Initially we have `o.<created> = FALSE` for all objects `o`. If a `new` method call is performed we look for the next object `o` to be created and change the value of `o.<created>` from `FALSE` to `TRUE`, which now is nothing more than any other function update.

Pre-Values of Arrays. It is quite easy to track the preconditions occurring in the Java Card DL proof obligation to their JML origin as a `requires` or `invariant` clause. But, there are also parts in the Java Card DL precondition that do not apparently correspond to a JML clause, for example in Fig. 3 line 10:

```
\forall jint j; _old19(j) = self_IntervalSeq.contents[j]
```

This is triggered by the use of the `\old` construct in the following part of the JML `ensures` clause:

JML
```
@    && (\forall int j; 0 <= j && j < i;
@                        contents[j] == \old(contents[j]))
```
JML

KeY handles this by introducing a new function, here `_old19`, and requiring it to coincide for all arguments with `self_IntervalSeq.contents`. Since `_old19` is not affected by the program it can be used after the diamond operator to refer to the old values of `self_IntervalSeq.contents` as in:

```
\forall jint j; (0 <= j & j < i  ->
                 self_IntervalSeq.contents[j] = _old19(j))
```

Imagine that you were to write a run-time checker for JML. That will give you the idea for how you would implement the `\old` construct. Corresponding to the JML term `\old(size)` the definition `_old17:=self_IntervalSeq.size` is introduced. This time not as an equality, but as already mentioned above as an update. The advantage is that the update will be automatically applied to prove the postcondition `size = \old(size) + 1`. The application of the definition of `_old19(j)` on the other hand requires user interaction or special heuristics to pick the needed instantiations of `j`.

Proper Java States. It still remains to comment on the precondition that we skipped on first reading, `inReachableState`. In KeY a method contract is proved by showing that the method terminates in a state satisfying the postcondition when started in any state s_1 satisfying the preconditions and the invariants. This may also include states s_1 that cannot be reached from the `main` method. But, usually the preconditions and invariants narrow down this possibility and in the end it does not hurt much to prove a bit more than is needed. But, there is another problem here: the implicit fields. A state with object `o` and field `a` such that `o.<created> = TRUE`, `o.a != null`, and `o.a.<created> = FALSE` is not possible in Java, but could be produced via updates. It is the precondition `inReachableState` that excludes this kind of anomalies.

Capturing JML Specifications in Java Card DL. Let us go back to the JML specification browser and select `Assignable PO` for the `insert` method. When proving, e.g., the normal_behaviour clause of a method contract, we also take advantage of the JML `assignable` clause. The current proof obligation now checks if the `assignable` clauses are indeed correct. In the case at hand there does not seem to be much to do, since all fields of class `IntervalSeq` (there are just two) may change. To be precise, a call to the `insert` method only assigns to these fields for the calling object, otherwise they should remain unchanged. This is what this proof obligation states.

Now, let us select the last proof obligation in the JML specification browser, which is named `class specification`. Its purpose is to make sure that, for

any state s_1 that satisfies all invariants of the `IntervalSeq` class and the preconditions of `insert(iv)`, the invariants are again true in the end state s_2 of this method. Note that here the modal box operator is used. Termination of the method was already part of its method contract, so we need not prove it again here. The proof obligation requires the invariants to also hold when the methods terminates via an exception. This is the reason why `insert(iv)` is enclosed in a `try-catch` block. Also the `inReachableState` predicate is among the invariants to be proved.

4.3 Verification

In this section, we demonstrate how the KeY prover is used to verify a proof obligation resulting from our example. It is important to note, however, that a systematic introduction into the usage of the prover is beyond the scope of this paper. Such an introduction can be found in Chapter 10 of the KeY book [3]. On the other hand, the examples in that chapter are of toy size as compared to the more realistic proof obligations we consider in this paper.

This section is meant to be read with the KeY prover up and running, to perform the described steps with the system right away. The exposition aims at giving an *impression* only, on how verification of more realistic examples is performed, while we cannot explain in detail *why* we are doing what we are doing. Again, please refer to [3, Chapt. 10] instead.

We will now verify that the implementation of the method `insert()` in class `SortedIntervalSeq` (not in `IntervalSeq`) respects the contract that it inherits from `IntervalSeq`. Before starting the proof, we remind ourselves of the code we are going to verify: the implementation of `insert()` was given in listing (1.6) in Sect. 4.1, and it calls the inherited method `incSize()`, see listing (1.1). Both these methods contain one `while` loop, which we advise the reader to look at, as we have to recognise them at some point during the verification.

We first let KeY generate the corresponding proof obligation, by following the same steps as described at the beginning of Sect. 4.2 (from File → Load ... onwards), but with the difference that this time we select, in the Classes pane, `SortedIntervalSeq` instead of `IntervalSeq`. (Apart from that, it is again the method `void insert(Interval iv)` that we select in the Methods pane, and the specification case normal_behavior that we select in the Proof Obligations pane. Before clicking on the Load Proof Obligation button, we make sure that the two check-boxes at the bottom of the specification browser are *not* checked.)

Afterwards, the Current Goal pane contains a proof obligation that is very similar to the one discussed in Sect. 4.2, just that now the (translated) class invariant of `SortedIntervalSeq`, see listing (1.4), serves as an additional assumption.

This now is a good time to comment on the the leading "==>" symbol in the Current Goal pane. As described in Sect. 2, the KeY prover builds proofs based on a *sequent calculus*. Sequents are of the form $\phi_1, \ldots, \phi_n \Longrightarrow \phi'_1, \ldots, \phi'_m$, where $\phi_1, \ldots, \phi_n$ and $\phi'_1, \ldots, \phi'_m$ are two (possibly empty) comma-separated lists of formulas, separated by the sequent arrow $\Longrightarrow$ (that is written as "==>" in the

KeY system). The intuitive meaning of a sequent is: if we assume all formulas $\phi_1, \ldots, \phi_n$ to hold, then *at least one* of the formulas $\phi'_1, \ldots, \phi'_m$ holds. We refer to "$\phi_1, \ldots, \phi_n$" and "$\phi'_1, \ldots, \phi'_m$" as the "left-hand side" (or "antecedent") and "right-hand side" (or "succedent") of the sequent, respectively.

The particular sequent we see now in the Current Goal pane has only one formula on the right-hand side, and no formulas on the left-hand side, which is the typical shape for generated proof obligations, prior to application of any calculus rule. It is the purpose of the sequent calculus to, step by step, take such formulas apart, while collecting assumptions on the left-hand side, and alternatives on the right-hand side, until the sheer shape of a sequent makes it trivially true. Meanwhile, certain rules make the proof branch.

We prove this goal with the highest possible degree of automation. However, we first apply one rule interactively, just to show how that is done. In general, interactive rule application is supported by the system offering only those rules which are applicable to the highlighted formula, resp. term (or, more precisely, to its top-level operator). If we now click on the leading "->" of the right-hand side formula, a context menu for rule selection appears. It offers several rules applicable to "->", among them imp_right, which in textbook notation looks like this:

$$\text{imp\_right}\ \frac{\Gamma, \phi \Longrightarrow \psi, \Delta}{\Gamma \Longrightarrow \phi \texttt{->} \psi, \Delta}$$

A tool-tip shows the corresponding taclet. Clicking on imp_right will apply the rule in our proof, and the Current Goal pane displays the new goal. Moreover, the Proof tab in the lower left corner displays the structure of the (unfinished) proof. The nodes are labelled either by the name of the rule which was applied to that node, or by "OPEN GOAL" in case of a goal. (In case of several goals, the one currently in focus is highlighted in blue.) We can see that impRight has been applied *interactively* (indicated by a hand symbol), and that afterwards, Update Simplification has been applied automatically. Updates are simplified automatically after each interactive step, because there is usually no reason to work with formulas that contain unsimplified updates.

The proof we are constructing will be several thousand steps big, so we better switch to automated proof construction now. For that, we select the Proof Search Strategy tab in the lower left corner, and configure the proof strategy as follows:

- Max. rule applications: 5000 (or just any big number)
- Java DL (the strategy for proving in JavaDL)
- Loop treatment: None (we want symbolic execution to stop in front of loops)
- Method treatment: Expand (methods are inlined during symbolic execution)
- Query treatment: Expand (queries in specifications are inlined)
- Arithmetic treatment: Basic (simple automatic handling of linear arithmetic)
- Quantifier treatment: Non-Splitting Instantiation
  (use heuristics for automatic quantifier handling, but do not perform instantiations that might cause proof splitting)

We run the strategy by clicking the ▶ button (either in the Proof Search Strategy tab or in the tool bar). The strategy will stop after about 1000 rule applications once the symbolic execution arrives at loops in the program (due to Loop treatment: None). We open the Goals tab, where we can see that there are currently five goals left to be proven.

We can view each of these goals in the Current Goal pane, by selecting one after the other in the Goals tab. In four of the five goals, the modality (preceded by a parallel update) starts with:

KeY (1.9)

```
\<{method-frame(...): {
      while ( i>0 && ivBoundary<getBoundary(contents[i-1]) ) {
        ...
```

KeY

In those four proof branches, symbolic execution is just about to "enter" the while loop in the method `insert()`. In the remaining fifth branch, the same holds for the while loop in the method `incSize()`. For the loop in `insert()`, we get four cases due to the two existing implementations of the interface `Interval` and the two concrete subclasses of the abstract class `SortedIntervalSeq`. All four cases can be handled in the same way and by performing the same interactions, to be described in the following.

In each case, we first have to process the while loop at the beginning of the modality. It is well known that loops cannot be handled in a similarly automated fashion as most other constructs.

Loop Invariants. Generally, for programs containing loops we have to choose a suitable *loop invariant*[8] (a formula) in order to prove that the loop has the desired effect and a *loop variant* (an integer term) for proving that the loop terminates. We also have to specify the *assignable memory locations* that can be altered during execution of the loop. All this information can be entered as part of an interactive proof step in KeY. However, the prover also supports the JML feature of annotating loops with invariants, variants, and assignable locations.

Invariants typically express that the loop counter is in a valid range, and give a closed description of the effect of the first n iterations. For the loop in the method `insert()`, it is necessary to state in the invariant that:

- the loop counter `i` never leaves the interval $[0, \texttt{size})$,
- the interval is not inserted too far left in the array, and
- the original contents of the sequence are properly shifted to the right.

The last component of the invariant is very similar to the post-condition, see listing (1.3), of the whole `insert()` method, which expresses that the argument of the method is properly inserted in the sequent (at an arbitrary place). In JML, we could capture this part of the loop invariant as:

[8] As an alternative to using invariants, KeY offers induction, see [3, Chapt. 11].

JML

```
(\forall int k; 0 <= k && k < i;
         contents[k] == \old(contents[k])) &&
(\forall int k; i < k && k < size;
         contents[k] == \old(contents[k-1]))
```

JML

In this constraint, `i` is the loop variable. It is stated that all array components that have already been visited by the loop are shifted to the right, whereas a prefix of the array remains unchanged.

Due to a certain shortcoming in the current JML support within KeY, however, it is for the time being not possible to use the `\old` operator in JML loop invariants. (It will be possible in future versions of KeY.) We can work around this problem by introducing a *ghost field* (a specification-only variable) that stores the old contents of the array. This is done by adding the following lines to the class `IntervalSeq`:

Java + JML

```
//@ public ghost Interval[] oldContents;

/*@ public normal_behavior
  @  ...
  @  requires contents != oldContents && oldContents != null
  @           && oldContents.length == contents.length
  @           && (\forall int i; 0 <= i && i < size;
  @                              contents[i] == oldContents[i]);
  @*/
public void insert(Interval iv) {
    ...
```

Java + JML

We can then write the complete loop invariant as shown below. Stating the termination of the loop is simple, because the variable `i` is always non-negative and decreased in each iteration. Further, we specify that the only modifiable memory locations are the loop counter and the elements of the array `contents`.

Java + JML

```
/*@ loop_invariant 0 <= i && i < size &&
  @    (i < size - 1 ==>
  @            ivBoundary < getBoundary( contents[i] )) &&
  @    (\forall int k; 0 <= k && k < i;
  @            contents[k] == oldContents[k]) &&
  @    (\forall int k; i < k && k < size;
  @            contents[k] == oldContents[k-1]);
  @ decreases i;
  @ assignable contents[*], i;
  @*/
```

```
while ( i > 0 && ivBoundary < getBoundary( contents[i-1] ) ) {
     contents[i] = contents[i - 1];
     --i;
}
```

Java + JML

Verification Using Invariants. We continue our proof on one of the four similar goals, all of which containing the modality of the form (1.9), such that processing the loop in `insert()` is the next step. Because all these four goals can be handled in the same way, we can pick an arbitrary one of them, by selecting it in the Goals tab. Before proceeding, we switch to the Proof tab, to better see the effect of the upcoming proof step.

We apply an invariant rule which automatically extracts the JML annotation of our loop from the source code. For that, we click on any of the ":=" symbols in the parallel update preceding the modality \<...\>, and select while_invariant_with_variant_dec from the rules offered. A Choose Taclet Instantiation window pops up, where we just press the Apply button.

Afterwards, the Proof tab tells us (possibly after scrolling down a bit) that the application of this invariant rule has resulted in four proof branches:

- Invariant Initially Valid: It has to be shown that the chosen invariant holds when entering the loop.
- Body Preserves Invariant: Under the assumption that the invariant and the loop condition hold, after one loop iteration the invariant still has to be true.
- Termination: Under the assumption that the invariant and the loop condition hold, the chosen variant has to be decreased by the loop body, but has to stay non-negative.
- Use Case: The remaining program has to be verified now using the fact that after the loop terminates, the invariant is true and the loop condition is false.

The four cases can be proven as follows. Generally, for a complex proof like this, it is best to handle the proof goals one by one and to start the automatic application of rules only locally for a particular branch. This is done by clicking on a sequent arrow ==> and choosing Apply rules automatically here, or by shift-clicking on a sequent arrow, or by right-clicking on a node in the proof tree display and selecting Apply Strategy from the context menu. (Clicking on ▶, in contrast, will apply rules to *all* remaining proof goals, which is too coarse-grained if different search strategy settings have to be used for different parts of the proof.)

Also, please note that a proof branch beginning with a green folder symbol is closed. Therefore, this symbol is a success criteria in each of the following four cases. Moreover, branches in the Proof tab can be expanded/collapsed by clicking on ⊞/⊟. To keep a better overview, we advise the reader to collapse the branches of the following four cases once they are closed.

Invariant Initially Valid. The proof obligation can easily be handled automatically by KeY and requires about 100 rule applications.

Body Preserves Invariant. This is the goal that requires the biggest (sub-)proof with about 11000 rule applications. In the Proof Search Strategy pane, choose a maximum number of rule applications of 20000 and run the prover in auto-mode on the goal as described above. This will, after a while, close the "Body Preserves Invariant" branch.

Termination. Proceed as for the case Body Preserves Invariant (possibly after expanding the branch and selecting its OPEN GOAL). This case will be closed after about 6000 steps.

Use Case. Proceed similarly as for the case Body Preserves Invariant. At first, calling the automated strategy will perform about 5000 rule applications. In contrast to the other three cases, for this last branch it is also necessary to manually provide witnesses for certain existentially quantified formulas (in the succedent) that can neither be found by KeY, nor by the external prover Simplify [9], automatically. These formulas correspond to the post-condition of the method `insert()`, where a point has to be "guessed" at which a further element has been added to the sequence. The form of the formulas is:

`\exists jint i; \forall jint j; \forall jint k;` F

Fortunately, for this problem, it is easy to read off the witness `i` that allows to prove the formulas: the body F always contains equations of the form `i =` t, where t is the desired witness. To actually perform the instantiation of the formula, drag the term t to the quantifier `\exists jint i` and choose the rule ex_right_hide in the appearing menu.[9] After this instantiation step, call the automated strategy, locally, and in those cases where this does not close that very branch, apply Simplify, *only locally.* A branch local call of Simplify is performed by clicking on the sequent arrow ==>, and selecting Decision Procedure Simplify in the context menu. In this way, handle all branches with formulas of the above form, until the "Use Case" has a green folder at its beginning, meaning this case is closed.

5 Second Walk-through: Specifying and Verifying Timeframe Displays

In this section, we practice specification and verification a second time, now with higher speed, and coarser granularity. The example is the method `add()` of the class `TimeFrameDisplay`.

5.1 Formal Specification and Implementation

[9] In case the option Options → DnD Direction Sensitive is enabled, KeY will perform the instantiation without showing a menu.

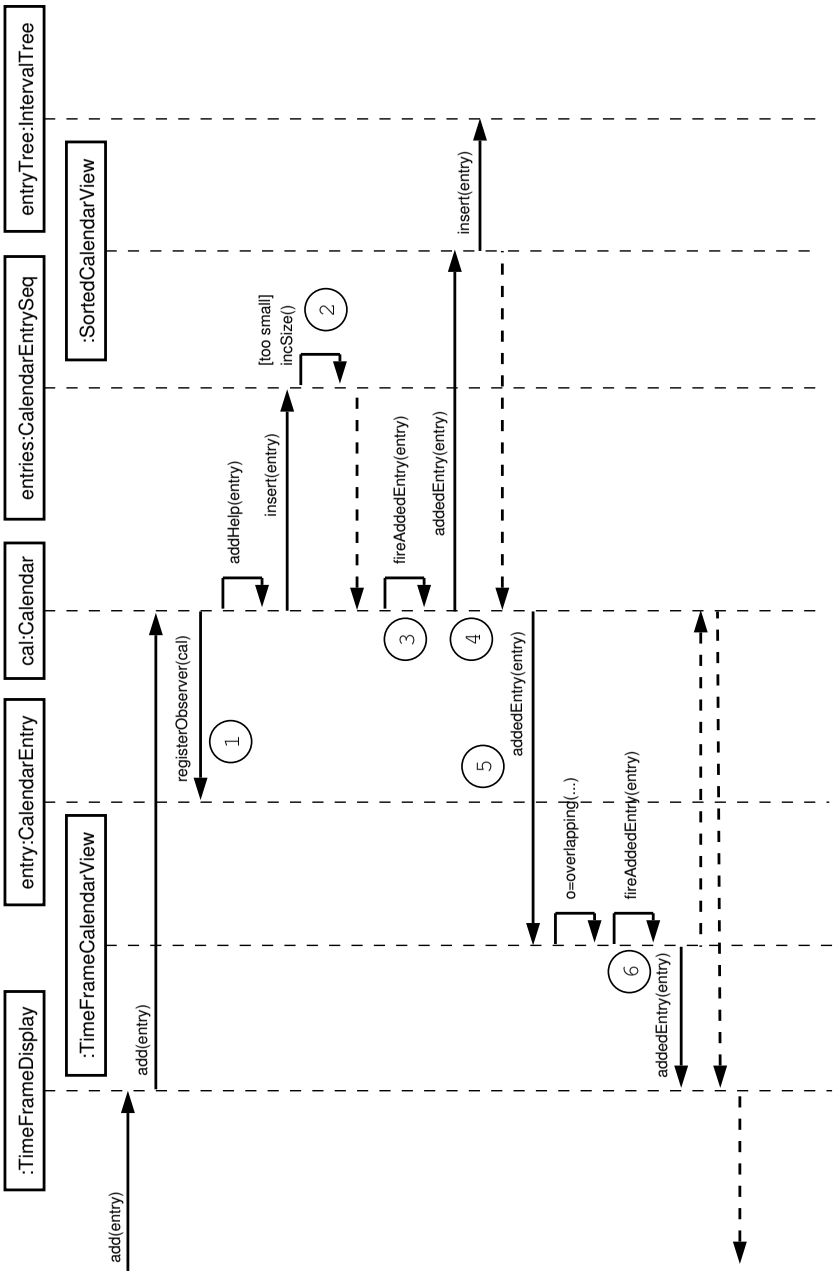

Fig. 4. UML sequence diagram showing the effect of calling `TimeFrameDisplay::add`

Within the Class `TimeFrameDisplay`. The class `TimeFrameDisplay` is a concrete application of the calendar view `TimeFrameCalendarView`, and could be (the skeleton of) a dialogue displaying a certain time period in a calendar. On the following pages, we demonstrate how we can give a more behavioural specification for some aspects of such a dialogue. The investigated method is `TimeFrameDisplay::add`, which simply delegates the addition of a new entry to the underlying `Calendar` object:

Java + JML (1.10)

```
public class TimeFrameDisplay implements CalendarListener {
  ...
  /*@ public normal_behavior
    @  requires entry != null;
    @  requires overlapping ( timeFrame, entry );
    @  ensures lastEntryAdded == entry;
    @*/
  public void add(CalendarEntry entry) {
    cal.add ( entry );
  }
  ...
  private CalendarEntry lastEntryAdded = null;
  public void addedEntry(CalendarEntry e) {
    lastEntryAdded = e;
  }
```

Java + JML

In this context, we would like to specify that calling `add` actually results in a new calendar entry being displayed on the screen. In order to simulate this effect, we introduce an attribute `lastEntryAdded` that is assigned in the method `addedEntry`. The post-condition of method `add`, `lastEntryAdded == entry`, consequently states that calling `add` eventually raises the signal `addedEntry` with the right argument (see Fig. 4 for an illustration).

5.2 Proof Obligations and Verification

This time, we demonstrate the use of a hand-written Java Card DL proof obligation instead of importing a JML specification into KeY. Formulating a problem directly in DL is more flexible and gives us full control over which assumptions we want to make, but it is also more low-level, more intricate, and requires more knowledge about the logic and the prover. Figure 5 shows the main parts of the file `timeFrameDisplayAdd.key` containing the proof obligation. A full account on the syntax used in KeY input files is given in [3, Appendix B]. As before, we can load `timeFrameDisplayAdd.key` by selecting File → Load ... (or in the tool bar) and choosing the file in the appearing dialogue.

The KeY input file in Fig. 5 starts with the path to the Java sources under investigation, and with a part that declares a number of program variables

KeY

```
\javaSource "calendar-sources/";
\programVariables {
  calendar.CalendarEntry entry; calendar.CalendarEntry old_entry;
  TimeFrameDisplay self;                                                }
\problem {
    inReachableState
  & self != null & self.<created> = TRUE
  & entry != null & entry.<created> = TRUE
  & TimeFrameDisplay.overlapping(self.timeFrame, entry) = TRUE

  & self.cal!=null & self.timeFrame!=null & self.timeFrameView!=null
  & self.timeFrameView.listenersNum = 1 & self.cal.listenersNum = 2
  & self.timeFrameView.listeners[0] = self
  & self.cal.listeners[0] = self.cal.completeView
  & self.cal.listeners[1] = self.timeFrameView
  & self.timeFrameView.timeFrame = self.timeFrame

  & \forall calendar.CalendarModificationServer serv;
      (   serv != null & serv.<created> = TRUE
       ->    serv.listeners != null
           & 0 <= serv.listenersNum & 1 <= serv.listeners.length
           & serv.listenersNum <= serv.listeners.length)
  & \forall observerPattern.Subject subj;
      (   subj != null & subj.<created> = TRUE
       ->   subj.observers != null
          & 0 <= subj.observersNum & 1 <= subj.observers.length
          & subj.observersNum <= subj.observers.length)
  & \forall calendar.CalendarEntrySeq entry;
      (   entry != null & entry.<created> = TRUE
       ->   entry.contents != null
          & 0 <= entry.size & 1 <= entry.contents.length
          & entry.size <= entry.contents.length)
  & \forall calendar.Calendar cal;
      (   cal != null & cal.<created> = TRUE
       -> cal.entries != null & cal.completeView != null)
  & \forall calendar.TimeFrameCalendarView view;
      (   view != null & view.<created> = TRUE
       ->   view.completeView != null & view.timeFrame != null
          & view.cal != null)
  & \forall calendar.SortedCalendarView view;
      (view != null & view.<created> = TRUE -> view.entryTree != null)

  -> {old_entry := entry} \<{ self.add(entry)@TimeFrameDisplay; }\>
                                    self.lastEntryAdded = old_entry   }
```

KeY

Fig. 5. The hand-written proof obligation for Sect. 5

(lines 2–4) used in the specification. The main part of the file describes one particular scenario that we want to simulate:

- In lines 7–8, we assume that `self` and `entry` refer to proper objects of classes `TimeFrameDisplay` resp. `CalendarEntry`. The calendar entry is also supposed to overlap with the attribute `self.timeFrame` (line 9), which is the pre-condition of the method `TimeFrameDisplay::add`.
- The `TimeFrameDisplay` object `self` has been properly set up and connected to a `Calendar` and to a `TimeFrameCalendarView` (lines 11, 16). The freshly created `TimeFrameCalendarView` has exactly one listener attached, namely the object `self` (lines 12, 13). Likewise, the calendar `self.cal` does not have any listeners registered apart from its `SortedCalendarView` and the `TimeFrameCalendarView` (lines 12, 14, 15).
- In order to perform the verification, we need to assume a number of invariants. Lines 18–32 contain three very similar class invariants for the classes `CalendarModificationServer`, `Subject`, and `CalendarEntrySeq`, mostly expressing that the arrays for storing listeners and calendar entries are sufficiently large. In lines 33–41, we state somewhat simpler invariants for `Calendar`, `TimeFrameCalendarView`, and `SortedCalendarView` that ensure that attributes are non-null.

In this setting, we want to show that an invocation of the method `self.add` with parameter `entry` has the effect of raising a signal `addedEntry`. This property is stated in lines 43–44 using a diamond modal operator.

Loop Handling. Apart from sequential code that can simply be executed symbolically, there are three loops in the system that require our attention in this setting. The loops in the methods `Subject::registerObserver` (① in Fig. 4) and `CalendarEntrySeq::incSize` (② in Fig. 4) are similar in shape and are necessary for handling the dynamically growing arrays of entries and listeners:

Java + JML

```
public void registerObserver(Observer obs) {
  ++observersNum;
  if ( observersNum > observers.length ) {
    final Observer[] oldAr = observers;
    observers = new Observer[observers.length * 2];
    int i = 0;
    while ( i < oldAr.length ) {observers[i] = oldAr[i]; ++i;}
  }
  observers[observersNum - 1] = obs;
}
...
protected void incSize() {
  ++size;
  if ( size > contents.length ) {
    final CalendarEntry[] oldAr = contents;
```

```
    contents = new CalendarEntry[contents.length * 2];
    int i = 0;
    while ( i < oldAr.length ) {contents[i] = oldAr[i]; ++i;}
  }
}
```

Java + JML

We can handle both loops in the same way (and with the same or similar invariants) as in Sect. 4.3. As before, it is enough to annotate the loops with JML invariants and variants, which can be read and extracted by KeY during the verification.

The third occurrence of a loop is in the class `CalendarModificationServer` in package `calendar` (③ and ⑥ in Fig. 4):

KeY

```
protected void fireAddedEntry(CalendarEntry entry) {
  int i = 0;
  while ( i != listenersNum ) {
    listeners[i].addedEntry ( entry ); ++i;}
}
```

KeY

This loop is executed after adding a new entry to the calendar and is responsible for informing all attached listeners about the new entry. In our particular scenario, there are exactly two listeners (the objects `self.cal.completeView` and `self.timeFrameView`), and therefore we can handle this loop by unwinding it twice.

The Actual Verification, Step by Step. After loading the problem file shown in Fig. 5, we select proof search options as in Sect. 4.3:

- Loop treatment: None
- Method treatment: Expand
- Query treatment: None
  (we do not inline queries immediately, because we want to keep the expression `TimeFrameDisplay.overlapping(self.timeFrame,entry)` that occurs in Fig.5 for later)
- Arithmetic treatment: Basic
- Quantifier treatment: Non-splitting instantiation

Running the prover with these options and about 1000 rule applications gets us to the point where we have to handle the loops of the verification problem. There are three goals left, corresponding to the points ①, ② and ③ in Fig. 4, one for each of the loops that are described in the previous paragraph. This is due to the fact that the loops in the methods `incSize` and `registerObserver` are only executed if it is necessary to increase the size of the arrays involved. Consequently, the proof constructed so far contains two case distinctions and

three possible cases. As the loops in `incSize` and `registerObserver` can be eliminated using invariants (exactly as in the previous section), we concentrate on the third loop in method `fireAddedEntry` that is met at point ③ in Fig. 4.

In order to unwind the loop of `fireAddedEntry` once, click on the program block containing the method body and choose the rule unwind_while. This duplicated the loop body and guards it with a conditional statement. That is, the loop "`while(`*b*`){`*prog*`}`" is replaced by "`if(`*b*`){`*prog*`;while(`*b*`){`*prog*`}}`".

After unwinding the loop, we have to deal with the first object listening for changes in the calendar, which is a `SortedCalendarView`. To continue, select Method treatment: None and run the prover in automode. The prover will stop at the invocation `SortedCalendarView::addedEntry` (④ in Fig. 4), which we can unfold using the rule method_body_expand. After that, continue in automode.

Method Contracts. The method `SortedCalendarView::addedEntry` inserts the new `CalendarEntry` into an interval tree to enable subsequent efficient lookups. Consequently, the next point where the prover stops is an invocation of the method `IntervalTree::insert`. The exact behaviour of this insertion is not important for the present verification problem, however, so we get rid of it using a *method contract* that only specifies which parts of the program state could possibly be affected by the insertion operation. Such a contract can be written based on Dynamic Logic and is shown in Fig. 6 (it is contained in the file `timeFrameDisplayAdd.key`). We specify that the pre-condition of the method `IntervalTree::insert` is `ivt != null & iv != null`, that arbitrary things can hold after execution of the method (the post-condition is `true`), but that only certain attributes of classes in the `intervals` package can be modified (the attributes listed behind the keyword `\modifies`).

In order to apply the method contract, we click on the program and select the item Use Method Contract in the context menu. In the appearing dialogue, we have to select the right contract `intervalTreeInsert`. This leads to two new proof goals, in one of which it has to be shown that the pre-condition of the contract holds, and one where the post-condition is assumed and the remaining program has to be handled. By continuing in automode, the first goal can easily be closed, and in the second goal the prover will again stop at point ③ in front of the loop of method `fireAddedEntry` (the second iteration of the loop).

Coming Back to `TimeFrameDisplay`. The next and last callback that needs to be handled is the invocation of `TimeFrameCalendarView::addedEntry` at point ⑤. This method checks whether the calendar entry at hand overlaps with the time period `TimeFrameCalendarView::timeFrame`, and in this case it will forward the entry to the `TimeFrameDisplay`:

Java + JML

```
public void addedEntry(CalendarEntry e) {
  if ( overlapping ( timeFrame, e ) ) fireAddedEntry ( e );
}
```

Java + JML

KeY

```
\contracts {
  intervalTreeInsert {
    \programVariables {
      intervals.IntervalTree ivt; intervals.Interval iv;
    }
    ivt != null & iv != null
    -> \<{ ivt.insert(iv)@intervals.IntervalTree; }\>
    true
    \modifies { ivt.root,
      \for intervals.IntervalTreeNode n; n.cutPoint,
      \for intervals.IntervalTreeNode n; n.left,
      \for intervals.IntervalTreeNode n; n.right,
      \for intervals.IntervalTreeNode n; n.sortedByStart,
      \for intervals.IntervalTreeNode n; n.sortedByEnd,
      \for intervals.IntervalTreeNode n; n.sortedByStart.size,
      \for intervals.IntervalTreeNode n; n.sortedByStart.contents,
      \for (intervals.IntervalTreeNode n; int i)
        n.sortedByStart.contents[i],
      \for intervals.IntervalTreeNode n; n.sortedByEnd.size,
      \for intervals.IntervalTreeNode n; n.sortedByEnd.contents,
      \for (intervals.IntervalTreeNode n; int i)
        n.sortedByEnd.contents[i] }
  };
}
```

KeY

Fig. 6. Java Card DL contract for the method `CalendarEntry::insert`

The property `overlapping(timeFrame, e)` is given as a pre-condition of the method `TimeFrameDisplay::add` and now occurs as an assumption in the antecedent of the goal:

```
TimeFrameDisplay.overlapping(self.timeFrame, entry) = TRUE
```

We can simply continue with symbolic execution on the proof branch. Because we want the prover to take all available information into account and not to stop in front of loops and methods anymore, select Loop treatment: Expand, Method treatment: Expand, and Query treatment: Expand. Choose a maximum number of rule applications of about 5000. Then, click on the sequent arrow ==> and select Apply rules automatically here. This eventually closes the goal.

6 Conclusion

In this paper we walked step-by-step through two main verification tasks of a non-trivial case study using the KeY prover. Many of the problems encountered

here—for example, the frame problem, what to include into invariants, how to modularise proofs—are discussed elsewhere in the research literature, however, typical research papers cannot provide the level of detail that would one enable to actually trace the details. We do not claim that all problems encountered are yet optimally solved in the KeY system, after all, several are the target of active research [15]. What we intended to show is that realistic Java programs actually can be specified and verified in a modern verification system and, moreover, all crucial aspects can be explained within the bounds of a paper while the verification process is to a very large degree automatic. After studying this tutorial, the ambitious reader can complete the remaining verification tasks in the case study.

As for "future work," our ambition is to be able to write this tutorial without technical explanations on how the verification is done while covering at least as many verification tasks. We would like to treat modularisation and invariant selection neatly on the level of JML, and the selection of proof obligations in the GUI. It should not be necessary anymore to mention Java Card DL in any detail. From failed proof attempts, counter examples should be generated and animated without the necessity to inspect Java Card DL proof trees. There is still some way to go to mature formal software verification into a technology usable in the mainstream of software development.

Acknowledgements

We would like to thank Richard Bubel for many discussions on various topics of the paper, and for his enormous contribution to the constant improvement of the KeY system.

References

1. 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*, number 1783 in LNCS. Springer, 2000.
2. B. Beckert and C. Gladisch. White-box testing by combining deduction-based specification extraction and black-box testing. In Y. Gurevich, editor, *Proceedings, Testing and Proofs, Zürich, Switzerland*, LNCS. Springer, 2007.
3. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.
4. B. Beckert and V. Klebanov. Must program verification systems and calculi be verified? In *Proceedings, 3rd International Verification Workshop (VERIFY), Workshop at Federated Logic Conferences (FLoC), Seattle, USA*, 2006.
5. L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: A developer-oriented approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods: International Symposium of Formal Methods Europe*, volume 2805 of *Lecture Notes in Computer Science*, pages 422–439. Springer-Verlag, 2003.
6. T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

7. A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, *Proc. 2nd Int. Conf. on Security in Pervasive Computing*, LNCS 3450, pages 193–209. Springer, 2005.
8. D. Detlefs, G. Nelson, and J. Saxe. Simplify: A Theorem Prover for Program Checking. Technical Report HPL-2003-148, HP Labs, July 2003.
9. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.
10. C. Engel and R. Hähnle. Generating unit tests from formal proofs. In Y. Gurevich, editor, *Proceedings, Testing and Proofs, Zürich, Switzerland*, LNCS. Springer, 2007.
11. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, Berlin, Germany, July 2007. Springer.
12. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin*, pages 234–245. ACM Press, 2002.
13. M. Giese and D. Larsson. Simplifying transformations of OCL constraints. In L. Briand and C. Williams, editors, *Proceedings, Model Driven Engineering Languages and Systems (MoDELS), Montego Bay, Jamaica*, LNCS 3713. Springer, 2005.
14. J. J. Hunt, E. Jenn, S. Leriche, P. Schmitt, I. Tonin, and C. Wonnemann. A case study of specification and verification using JML in an avionics application. In M. Rochard-Foy and A. Wellings, editors, *Proc. of the 4th Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*. ACM Press, 2006.
15. G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. Technical Report 06-14a, Department of Computer Science, Iowa State University, Aug. 2006. To appear in *Formal Aspects of Computing*.
16. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin. *JML Reference Manual*, Aug. 2006. Draft revision 1.197.
17. S. Nanchen, H. Schmid, P. Schmitt, and R. F. Stärk. The ASMKeY prover. Technical Report 436, Department of Computer Science, ETH Zürich, 2004.
18. J. van den Berg and B. Jacobs. The loop compiler for java and jml. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Software (TACAS)*, volume 2031 of *LNCS*, pages 299–312. Springer, 2001.