Chapter 16
Formal Verification with KeY: A Tutorial

Bernhard Beckert, Reiner Hähnle, Martin Hentschel, Peter H. Schmitt

16.1 Introduction

This chapter gives a systematic tutorial introduction on how to perform formal program verification with the KeY system. It illustrates a number of complications and pitfalls, notably programs with loops, and shows how to deal with them. After working through this tutorial, you should be able to formally verify with KeY the correctness of simple Java programs, such as standard sorting algorithms, gcd, etc. This chapter is intended to be read with a computer at hand on which the KeY system is up and running, so that every example can be tried out immediately. The KeY system, specifically its version 2.6 used in this book, is available for download from www.key-project.org. The example input files can be found on the web page for this book, www.key-project.org/thebook2, as well as in the `examples` directory of your KeY system's installation.

In principle, this chapter can be read on its own, but one should be familiar with basic usage of the KeY system and with some fundamental concepts of KeY's program logic. Working through Chapter 15 gives sufficient background. The difference between Chapter 15 and the present chapter is that the former focuses on usage and on interaction with the KeY system by systematically explaining the input and output formats, as well as the possibilities for interaction with the system. It also uses exclusively the KeY GUI (see Figure 1.1) and is concerned with problems formulated in first-order logic or dynamic logic. Figure 15.1 on page 496 displays an overview of the entire verification process.

In the present chapter we mainly look at JML annotated Java programs as inputs and we target the verification process as a whole, as illustrated in Figure 16.1. It shows the whole work flow, including specification annotations written in the Java Modeling Language (JML), the selection of verification tasks, symbolic execution, proving of first-order proof obligations, followed by a possible analysis of a failed proof attempt. In addition, there is a section on how to perform verification using the Eclipse integration of the KeY system.

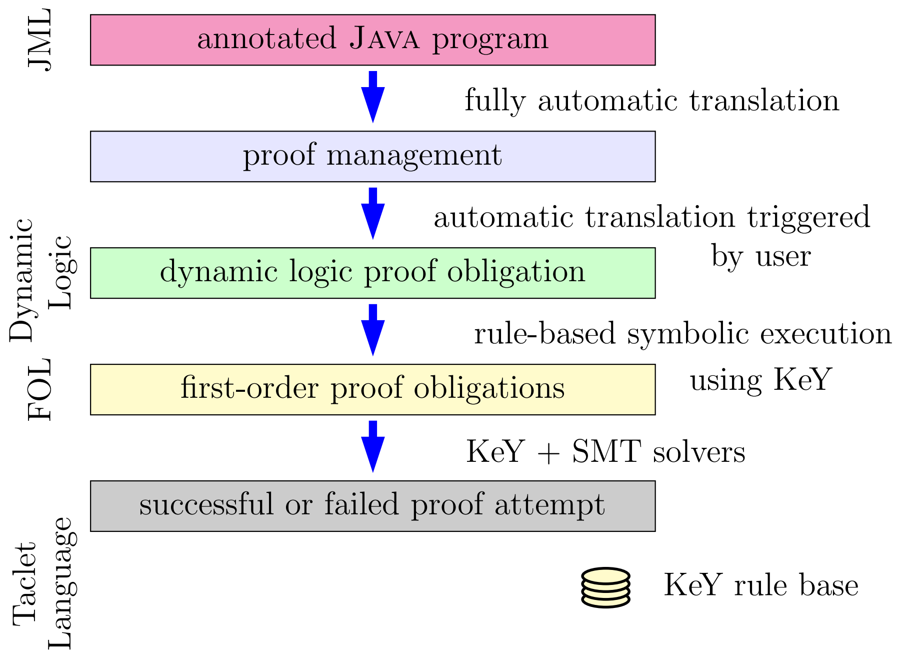

JML
annotated JAVA program
fully automatic translation
proof management
automatic translation triggered by user
Dynamic Logic
dynamic logic proof obligation
rule-based symbolic execution using KeY
FOL
first-order proof obligations
KeY + SMT solvers
successful or failed proof attempt
Taclet Language
KeY rule base

Figure 16.1 The KeY verification workflow

When the input to the KeY system is a `.java` rather than a `.key` file, then it is assumed that the Java code has annotations in JML in the form of structured comments of the source code. Chapter 7 provides a thorough introduction to JML. We give sufficient explanations to understand the examples in this chapter to make it self-contained (after all, JML is marketed as being easily understandable to Java programmers), but we avoid or gloss over the finer points and dark corners of that language.

The organization of the material in this chapter is as follows: In Section 16.2 we illustrate the basic principles of program verification with KeY by way of a simple, loop-free program: JML annotations, loading of problems, selection of proof tasks, configuration of the prover, interactive verification, representation of the symbolic heap. This is followed by Section 16.3 which gives a hands-on introduction into the craft of designing adequate loop invariants and how they are used in KeY. Sections 16.4 and 16.5 walk you through a more complex example (selection sort on arrays). We demonstrate advanced aspects of finding a proof: understanding intermediate states and open subgoals, specifying complex functional properties, working with method contracts, working with model elements in specifications, using strategy macros, guiding the prover when it cannot find a proof itself. Finally, Section 16.6 describes how the Eclipse integration of KeY can be used to automatically manage proofs so that user interaction is only required if a proof is not automatically closable.

16.2 A Program without Loops

```
 1 public class PostInc{
 2     public PostInc rec;
 3     public int x,y;
 4
 5     /*@  public invariant
 6       @      rec.x>=0 && rec.y>=0;
 7       @*/
 8
 9     /*@ public normal_behavior
10       @ requires true;
11       @ ensures rec.x == \old(rec.y) &&
12       @         rec.y == \old(rec.y)+1;
13       @*/
14      public void postinc() {
15              rec.x = rec.y++;
16      }
17 }
```

Listing 16.1 First example: Postincrement

We start with a first simple Java program shown in Listing 16.1. The class `PostInc` has two integer fields `x` and `y` declared in line 3. The only method in the class, `postinc()`, declared in lines 14–15, sets `x` to `y` and increments `y`. To make things a little more interesting these operations are not performed on the **`this`** object, but on the object given by the field `rec` in line 2. The rest of the shown code are JML comments. In lines 5–6 an invariant is declared. An invariant, as the name is supposed to suggest, is—roughly—true in all states. The states during which the variables contained in the invariant are manipulated have, e.g., to be excepted from this requirement. The details of when invariants precisely are required to hold are surprisingly thorny. Detailed explanations are contained in Section 7.4. For now it suffices to understand that invariants may be assumed to hold at every method call and have to be established after every method termination.

Lines 9–13 are filled with a JML method contract. A contract typically consists of two clauses (we will later see more than two): a precondition signaled by the keyword **`requires`** and a postcondition following the keyword **`ensures`**. As with real-life contracts, there are two parties involved in a method contract. The user of a method has to make sure that the precondition is true when the method is called and may depend on the fact that the postcondition is true after termination of the method. The second party, the method provider, has the liability to guarantee that after termination of the method the postcondition is true, provided the precondition was met in the calling state. In the example there is no precondition, more precisely the precondition is the Boolean constant *true* that is true in any state. The postcondition in this case is just the specification of the postincrement operator `_++`. We trust that the reader has figured out that an JML expression of the form `\old(exp)` refers to the value of `exp` before the execution of the method. The postcondition in itself does not make any claims on the termination of the method. It is a partial correctness assertion: if the method terminates, then the postcondition will be true. In the specific example, termination is asserted by the declaration of the specification case in line 9 as a **`normal_behavior`**. When a normal behavior method is called in a state satisfying its precondition, it will terminate and not throw an exception.

To see what KeY does with the annotated program from Figure 16.1, start the KeY system as explained in the beginning of Section 15.2. The file `PostInc.java` is loaded by **File** → **Load** (or selecting in the tool bar) and navigating through the opened file browser. This is the same as loading `.key` files as described in Chapter 15; the result however is different. The **Proof Management** window will pop up. You will notice that not only the file you selected has been loaded but also all other `.java` files in the same directory. So, you could just as well have selected the directory itself. You may now select in the **Proof Management** window a `.java` file, a method and a contract. For this experiment we choose the contract `JML normal_behavior operation contract 0` for method `postinc()` in file `PostInc.java` and press the **Start Proof** button. The verification task formalized in Dynamic Logic will now show up in the **Current Goal** pane. Since we try to discharge it automatically we do not pay attention to it. We rather look at the **Proof Search Strategy** tab in the lower left-hand pane and press the **Defaults** button in its top right corner. You may slide the maximal rule application selector down to 200 if you wish. All that needs to be done is to press the **Start** button (to the left of the **Defaults** button or in the tool bar). A pop-up window will inform you that the proof has been closed and give statistics on the number and nature of rule applications. In the **Proof** tab of the original window the proof can be inspected.

The fact that the invariant `rec.x>=0 && rec.y>=0` can be assumed to be true when the method `postinc()` is called did not contribute to establish the postcondition. But, the automatic proof did include a proof that after termination the invariant is again true. You can convince yourself of this by investigating the proof tree. To do this move the cursor in the **Proof** tab of the lower left-hand pane on the first node, or any other node for that matter, of the proof tree and press the right mouse button. A pop-up menu will appear. Select the **Search** entry. A search pane at the very bottom of the **Proof** tab will show up. Enter `inv` in it. Press the ➡ button to the left of the text field. The first hit, shown in the **Goal** pane, will be `self.<inv>` that corresponds to the assumption of the invariant at the beginning. Another push on the ➡ button yields the next hit of the form `h[self.<inv>]`. Here, `h` is a lengthy expression denoting the heap after termination of the method. This is the formalization of the claim that the invariant is true in the poststate. The green folder symbol shows that this claim could be proved successfully. You can now save the proof by selecting **File** → **Save**. Let us agree that we accept the suggested file name `PostInc.proof`. We remark, that you can also save a partial proof and later load it again to complete it.

There is one more topic that we want to discuss with the `PostInc` example at hand: pretty printing. We start by loading the file `PostInc.proof` that contains the finished proof for the verification task we have just gone through. After loading finishes the tree of the closed proof can be inspected in the **Proof** tab. Click on proof node `13:exc=null;`. The **Inner Node** pane now shows the sequent at this proof node. We first focus on the second line `self.<created> = TRUE`. In the **View** menu the option **Use pretty syntax** is checked in the standard setting. Uncheck it and let us investigate what happened. Line 2 now reads

```
boolean::select(heap,self,java.lang.Object::<created>) = TRUE.
```

Here, `boolean::select` is the ASCII rendering of the function $select_{boolean}$. In general `A::select` renders the functions $select_A$ introduced in Figures 2.4 and 2.11. In the same formula, `<created>` is expanded to `java.lang.Object::<created>`. We thus first observe that pretty printing omits the typing of functions, predicates and fields which in most cases is either fixed in the vocabulary or can be inferred from the context. If nothing helps, you may have to resort to switching pretty printing off. But, the important part is that pretty printing hides the dependence of evaluations on the current heap which is modeled by the attribute `heap`. This parallels the habit that most programmers omit most of the time to explicitly name the `this` object. For a field `f` of type `B` in class `A` and a term t of static type `A`, the following abbreviation will be used for pretty printing:

$$\mathrm{PP}(\texttt{B::select(heap,t,A::\$f)}) = \mathrm{PP}(\texttt{t}).\texttt{f}$$

Note, that in the next line `PostInc::exactInstance(self) = TRUE` remains unchanged by pretty printing since the functions $exactInstance_A$ do not depend on the heap.

When running a Java program all evaluations are done with respect to the current heap. But, in verifying properties of programs we need to talk about evaluations in different heaps. It is frequently the case that we want to compare the value of a field before the program is run with its value in the terminating state. What does pretty printing do in case evaluation is not with respect to the current heap? To see an example we look at the line that contains the end `\>` of the ASCII version of the diamond operator:

```
self.rec.y@heapAtPre = self.rec.x
```

Without pretty printing it looks

—— KeY Output ——

```
int::select(heapAtPre,
            PostInc::select(heapAtPre,
                            self,
                            PostInc::$rec),
            PostInc::$y)
 =
int::select(heap,
            PostInc::select(heap,self,PostInc::$rec),
            PostInc::$x)
```

—— KeY Output ——

The general rule may be stated as

PP(`B::select(H,t,A::f)`) = PP(t).`f@H` .

Applying this literally to the term at the right (upper) side of the above equation we would obtain:

```
(self.rec@heapAtPre).y@heapAtPre
```

There is a second rule that allows the pretty printer to abbreviate `(t0.t1@H).t2@H` by `t0.t1.t2@H`.

Another pretty printing feature, which does not occur in the current example but will pop up in the next section, concerns array access. The pretty printed expression `a[pos2]@H` stands for the full version `int::select(H,a,arr(pos2))`.

The heap independent function $arr : Int \to Field$ (see Figures 2.4 and 2.11) associates with every integer i a field that stands for the access to the i-th entry in an array. Note, that JFOL is again more liberal than Java. We may write `A::select(h,a,arr(i))` even for i greater than the array length, for negative i, or even when a is not of array type.

The general pretty printing rule is

```
PP(A::select(H,e,arr(a))) = PP(e)[PP(a)]@H
```

if the declared type of `e` is `A[]`. Furthermore, `@H` will be omitted if `H` equals `heap`.

16.3 A Brief Primer on Loop Invariants

16.3.1 Introduction

Finding suitable loop invariants is considered to be one of the most difficult tasks in formal program verification and it is arguably the one that is least amenable to automation. For the uninitiated user the ability to come up with loop invariants that permit successful verification of nontrivial programs is bordering on black magic. We show that, on the contrary, the development of loop invariants is a craft that can be learned and applied in a systematic manner.

We highlight the main difficulties during development of loop invariants, such as strengthening, generalization, weakening, introducing special cases, and we discuss heuristics on how these issues can be attacked. Systematic development of loop invariants also involves close interaction with a formal verification tool, because it is otherwise too easy to overlook errors. To this end, we demonstrate a number of typical interaction patterns with the KeY system.

This section has necessarily some amount of overlap with Section 3.7.2, but it is written in a less formal manner and it concentrates on the pragmatics of loop invariant rules rather than on their formal definition. It is assumed that you have acquired a basic understanding of how the KeY prover works, for example, by reading Chapter 15. Even though proving problems involving recursive methods share some problems with proofs about loops, we concentrate here on the latter, because KeY uses somewhat different technical means to deal with recursion. Some information on proving recursive programs is found in Chapter 9.

16.3.2 Why Are Loop Invariants Needed?

Students who start in Computer Science are often puzzled by the question why such a complex and hard to grasp concept as loop invariants is required in the first place. In the context of formal verification, however, their need is easily motivated. In Chapter 3 it is explained how the calculus of JavaDL realizes a symbolic execution engine for Java programs. When, during symbolic execution, a loop[1] is encountered, symbolic execution attempts to unwind the loop, using the rule from Section 3.6.4:

$$\mathsf{loopUnwind}\ \frac{\Longrightarrow \langle \pi\ \mathtt{if}\ \mathtt{(}e\mathtt{)}\ \mathtt{\{}\ p\ \mathtt{while}\ \mathtt{(}e\mathtt{)}\ p\ \mathtt{\}}\ \omega\rangle\phi}{\Longrightarrow \langle \pi\ \mathtt{while}\ \mathtt{(}e\mathtt{)}\ p\ \omega\rangle\phi}$$

If the loop guard is evaluated to true in the current symbolic state, then the loop body p is symbolically executed once and afterwards the program pointer is at the beginning of the loop once again. Otherwise, symbolic execution continues with the code ω following the loop.

Obviously, this works only well, when the number of iterations of the loop is bounded by a small constant. This is not the case in general, however. A loop guard might, for example, look like `i < a.length`, where `a` is an arbitrary array of unknown length.

To reason about unbounded loops or even about loops whose body is executed very often (for example, `0 <= i && i < Integer.MAX_VALUE`), some kind of induction principle is necessary that permits to prove properties of unbounded structures in a finite manner.

16.3.3 What Is A Loop Invariant?

First of all, a loop invariant always relates to some loop that occurs at a specific location in a given program. In the following we assume it is clear which loop is meant when we speak of "the loop."

In the context of KeY, a loop invariant is a formula $inv \in$ DLFml that holds in the program state at the beginning of the loop and in the state immediately after each execution of the loop body. If the loop terminates, this means that the invariant holds also in the state where continuation of the given program after the loop commences. As a consequence, if we manage to prove that a formula inv is a loop invariant, then

[1] To avoid obscuring the essential points with technical complexities, we concentrate in this section on while loops. Moreover, we assume that the loop body does not throw any exceptions and does not contain `break`, `continue`, or `return` statements.

it can be used during symbolic execution of the continuation after the loop. In this way, loop invariants indeed allow us to reason about programs containing unbounded loops.

The considerations in the previous paragraph can be formalized in a first attempt at a loop invariant rule for JavaDL. To simplify things a little, we assume that the program in the loop guard and loop body do not access the heap.

$$\dfrac{\begin{array}{lr} \Gamma \Longrightarrow \{u\}\mathit{inv}, \Delta & \text{(initially valid)} \\ \mathit{inv}, g \doteq \mathit{TRUE} \Longrightarrow [p]\mathit{inv} & \text{(preserved by body)} \\ \mathit{inv}, g \doteq \mathit{FALSE} \Longrightarrow [\pi\ \omega]\varphi & \text{(use case)} \end{array}}{\Gamma \Longrightarrow \{u\}[\pi\, \texttt{while(}g\texttt{)}\,p\texttt{;}\ \omega]\varphi, \Delta}\ \text{loopInvariant1}$$

The first premiss states that *inv* holds in the program state at beginning of the loop, the second premiss states that if *inv* holds in any state that evaluates the loop guard to true—i.e., the loop is entered—then it also holds in the final state after symbolic execution of the loop body, provided that it terminates. Finally, the third premiss permits to use the invariant plus the negated loop guard to prove correctness of the continuation (use case).

Soundness of the loop invariant rule rests on an inductive argument that runs as follows:

Induction Hypothesis: For any $n \geq 1$ the invariant *inv* holds in the state at the beginning of the n-th execution of the loop body.

Induction Base: The invariant *inv* holds in the state at the beginning of the first execution of the loop body, i.e., in the state where symbolic execution of the loop commences. This is exactly what the first premiss says.

Induction Step: If *inv* holds in the state at the beginning of the n-th execution of the loop body, and if the loop is entered at least one more time, then *inv* holds again after execution of the loop body, i.e., in the state at the beginning of the $n+1$-st execution of the loop body.

The problem is that we do not know in which state we are at the beginning of the n-th execution. This problem can be addressed by proving a somewhat more general induction step which does not require that knowledge:

> "For *any* program state, if *inv* holds in it at the beginning of the n-th execution of the loop body, and if the loop is entered at least one more time, then *inv* holds again in the state after execution of the loop body."

The latter clearly implies the *Induction Step* above and it is exactly what is expressed in the second premiss of the invariant rule. Observe that the contexts Γ, Δ, and $\{u\}$ were removed from the sequent to ensure that the induction step is indeed valid in any program state.

Similarly, we don't know in which state we are when the loop terminates, so the context information is erased from the third premiss as well. This means that any information from the context that might be needed in the proof of the continuation must be part of the loop invariant. Obviously, this is not very practical and we will come back to this issue. But now let us look at our first concrete loop invariant.

16.3.4 Goal-Oriented Derivation of Loop Invariants

We start by the observation that *any* loop has the trivially valid invariant true. Indeed, a glance at the invariant rule above shows that its first two premisses are straightforward to prove whenever $inv \equiv$ true. But this trivial invariant is, of course, normally useless to prove correctness of the continuation (i.e., the third premiss). In general, we need to find a nontrivial formula to serve as loop invariant, but which?

Often, it is a good idea to think about what we would like to prove, i.e., to work in a goal-oriented manner. Consider the following formula in `.key` file input syntax:[2]

—— KeY ——
```
n >= 0 & wellFormed(heap) ==>
{i := 0} \[{
  while (i < n) {
    i = i + 1;
  }
}\](i = n)
```
—— KeY ——

Look at the postcondition `i = n` to be proven. What, in addition to the negated guard `i >= n`, is needed to show it? Obviously, the formula `i <= n` is sufficient. Therefore, let us take this formula as a candidate for our loop invariant. To establish that $inv \equiv \mathtt{i} \leq \mathtt{n}$ is an invariant we must instantiate the loop invariant rule with inv as above, $\Gamma \equiv \mathtt{n} \geq 0$, $u \equiv \mathtt{i} := 0$, $g \equiv \mathtt{i} < \mathtt{n}$, $p \equiv$ `i = i + 1;` and empty Δ, π, ω. The instantiated (initially valid) premiss becomes

$$\mathtt{n} \geq 0 \Longrightarrow \{\mathtt{i} := 0\}(\mathtt{i} \leq \mathtt{n})$$

After update application the sequent's succedent becomes $0 \leq \mathtt{n}$, making the sequent obviously provable. Instantiation of the second premiss (preserved by body) and simplification of the guard expression yields:

$$\mathtt{i} \leq \mathtt{n}, \mathtt{i} < \mathtt{n} \Longrightarrow [\mathtt{i\ =\ i\ +\ 1;}](\mathtt{i} \leq \mathtt{n})$$

The sequent's succedent becomes after symbolic execution of the assignment and update application $\mathtt{i} + 1 \leq \mathtt{n}$, which is clearly provable from the antecedent. Therefore, $inv \equiv \mathtt{i} \leq \mathtt{n}$ is indeed a loop invariant that suffices to prove the postcondition at hand.

It is not always the case that a loop is the final statement before the postcondition. In this case, it is necessary to infer the difference between the state after the loop and the final state before the postcondition. For this reason, in the presence of multiple loops it is a good idea

[2] Even for programs that do not access the heap it is necessary to have the well-formedness assumption in order to render the problem provable. This is to exclude initial states that cannot be obtained in the Java runtime environment. We include the well-formedness constraint, because we want to give actually provable examples, but we leave it out from the subsequent reasoning steps for readability. The declaration of program variables, for example "`int i, n;`," is omitted in the following.

- to develop the invariant of the loop that is closest to the end of the program first, and
- to develop the invariant of the outermost loop first, in the case of nested loops.

16.3.5 Generalization

Let us look at a slightly more complex example, where x and y are program variables of type integer and x_0, y_0 are first-order constants of the same type.

—— KeY ——

```
x = x0 & y = y0 & y0 >= 0 & wellFormed(heap) ==>
\[{
  while (y > 0) {
    x = x + 1;
    y = y - 1;
  }
}\] (x = x0 + y0)
```

—— KeY ——

Starting again from the postcondition, we see that the postcondition bears no obvious relation to the guard. Hence, our first attempt at finding a loop invariant is simply to use the postcondition itself: $inv \equiv \mathtt{x} \doteq x_0 + y_0$. This formula, however, is clearly not valid at the beginning of the loop, and neither is it preserved by the loop body.

A closer look at what happens in the loop body reveals that both x and y are modified, but only the former is mentioned in the invariant. It is obvious that the invariant must say something about the relation of x and y to be preserved by the loop body. What could that be? The key observation is that in the loop body first x is increased by one and then y is decreased by one. Therefore, the *sum* of x and y stays invariant. Together with the observation that x initially has the value x_0 and y the value y_0, we arrive at the invariant candidate $inv \equiv \mathtt{x} + \mathtt{y} \doteq x_0 + y_0$, which is indeed initially valid as well as preserved by the loop body.

Is this a good invariant? Not quite: the postcondition is not provable from $\mathtt{x} + \mathtt{y} \doteq x_0 + y_0 \land \mathtt{y} \leq 0$. It would be sufficient, if we knew that $\mathtt{y} \geq 0$. And indeed, we have not made use of the precondition $y_0 \geq 0$ which states that the initial value of y is nonnegative. The loop guard ensures that y is positive when we enter the loop and in the loop body it is decreased only by one, therefore, $\mathtt{y} \geq 0$ is a loop invariant as well. Using the combined invariant $inv \equiv \mathtt{x} + \mathtt{y} \doteq x_0 + y_0 \land \mathtt{y} \geq 0$ it is easy to prove the example. In summary, for this example we made use of some important heuristics:

1. Generalize the postcondition into a relation about the variables modified in the loop body that is preserved.
2. Look for unused information in the proof context that can be used to derive additional invariants.

3. Loop invariants are closed under conjunction: if inv_1 and inv_2 are loop invariants of the same loop, then so is $inv_1 \wedge inv_2$.

16.3.6 Recovering the Context

Recall from Section 16.3.3 that our rule loopInvariant1 throws away the proof context from the second and third premiss to ensure soundness. Let us look at an example that illustrates the problem with this approach. Assume we want to prove something about the following program, where `a` has type `int[]`:

—— Java + JML ——

```
int i = 0;
while(i < a.length) {
  a[i] = 1;
  i++;
}
```

—— Java + JML ——

Whatever property we are going to prove about the loop, we will need the precondition $\texttt{a} \neq \texttt{null} \in \Gamma$ to make sure that the array access does not throw a null pointer exception. As we throw away the context, it will be necessary to add $\texttt{a} \neq \texttt{null}$ to any loop invariant. This may seem not so bad, but now assume that Γ contains a complex class invariant that is needed during the proof of the continuation after the loop. Again, this has to be added to the invariant. Loop invariants tend to become impractically bulky when they are required to include relevant parts of the proof context.

A closer look at the loop body of the program above shows that while the content of the array `a` is updated, the object reference `a` itself is untouched and, therefore, a precondition such as $\texttt{a} \neq \texttt{null} \in \Gamma$ is an implicit invariant of the loop body. What we would like to have is a mechanism that automatically includes all those parts of the context into the invariant whose value is unmodified by the loop body.

As it is undecidable whether the value of a given program location is modified in a loop body, this information must in general be supplied by the user. On the level of JML annotations this is done with the directive "**`assignable`** $l_1, \ldots, l_n$;", where the l_i are program locations or more general expressions of type `\locset`. These may contain a wildcard "`*`" where an index or a field is expected to express that all fields/entries of an object might get updated. For the loop above a correct specification of its assignable locations would be "**`assignable`** `i, a[*];`." KeY accepts that assignable clause, but actually ignores the local variable `i`. Instead its loop invariant rule checks the loop body and guard for any assignments to local variables and adds these implicitly to the assignable clause. Hence, only heap locations need to be specified as part of the assignable clause.

The intended effect of an assignable clause is that any knowledge in the proof context that depends on the value of a location mentioned in that assignable clause is erased in the second and third premiss of the invariant rule. How is this realized at the level of JavaDL? The main idea is to work with suitable updates. For value types, such as `i` in the example above, it is sufficient to add an update of the form $\{\mathtt{i} := c\}$, where c is a fresh constant of the same type as `i`. Such an update assigning fresh values to locations is called *anonymizing update*; details about their structure are explained in Section 9.4.1. A context preserving invariant rule, based on the rule LOOPINVARIANT1 above, therefore, looks as follows:

$$\frac{\begin{array}{ll} \Gamma \Longrightarrow \{u\}inv, \Delta & \text{(initially valid)} \\ \Gamma \Longrightarrow \{u\}\{v\}\big(inv \land g \doteq TRUE \rightarrow [p]inv\big), \Delta & \text{(preserved by body)} \\ \Gamma \Longrightarrow \{u\}\{v\}\big(inv \land g \doteq FALSE \rightarrow [\pi\ \omega]\varphi\big), \Delta & \text{(use case)} \end{array}}{\Gamma \Longrightarrow \{u\}[\pi\ \mathtt{while}(g)p;\ \omega]\varphi, \Delta}$$

where $\{v\}$ is the anonymizing update for the assignable clause **assignable** $l_1, \ldots, l_n$;.

Observe that the proof context Γ, Δ, $\{u\}$ has been reinstated into the second and third premiss. For object types (e.g., `a[*]`) more complex conditions about the heap must be generated. KeY does this automatically for assignable clauses specified in JML and we omit the gory details. The interested reader is referred to Section 8.2.5.

Assignable clauses should be as “tight” as possible, i.e., they should not contain any location that cannot be modified by the loop they refer to. On the other hand, assignable clauses must be sound: they must list all locations that possibly can be modified. In Java care must be taken, because it is possible to modify locations even in the guard expression. An unsound assignable clause renders the invariant rule where it is used unsound as well. For this reason, KeY generates proof obligations for all assignable clauses that ensure their soundness.[3] The exception are local variables where it is possible to compute a sound assignable clause by a simple static analysis and KeY does that automatically, even when no assignable clause is explicitly stated. Otherwise, the default declaration, when no assignable clause is stated, is “**assignable \everything**;.” This should be avoided.

We close this subsection by stating the example from above with JML annotations that are sufficient for KeY to prove it fully automatic:

—— Java + JML ——

```
public int[] a;
/*@ public normal_behavior
  @  ensures (\forall int x; 0<=x && x<a.length; a[x]==1);
  @  diverges true; // termination not proven
  @*/
public void m() {
  int i = 0;
  /*@ loop_invariant
```

[3] This proof obligation is part of the (preserved by body) branch. For ease of presentation it is not included in the rule above.

```
    @  0 <= i && i <= a.length &&
    @  (\forall int x; 0<=x && x<i; a[x]==1);
    @  assignable a[*];
    @*/
  while(i < a.length) {
    a[i] = 1;
    i++;
  }
}
```

Java + JML

Observe that the local variable `i` is not listed in the assignable clause and that the JML default $\mathtt{a} \neq \mathtt{null}$ needs not to be stated in the invariant.

To maximize automation of KeY in the presence of loops, the setting **Invariant** should be chosen in the **Loop Treatment** option of the **Proof Search Strategy** settings (see Chapter 15). This causes the prover to look for `loop_invariant` and `assignable` declarations in the input file and applies the loop invariant rules without user interaction. In addition, it can be useful to set option **Quantifier Treatment** to **No Splits with Progs** (which avoids splitting during symbolic execution) and, if the program contains arithmetic operators `*` or `/`, to set option **Arithmetic Treatment** to **DefOps**.

16.3.7 Proving Termination

Programs with loops may not terminate, but so far we have only looked at partial correctness and at terminating programs. Consider, for example, the sequent:

$$\Longrightarrow [\mathtt{i\ =\ 17;\ while\ (true)}\ \{\}]\,\mathtt{i} \doteq 42$$

Is it provable? It turns out that our formalism so far can correctly handle this example: with the trivial invariant `true` and the declaration `assignable \nothing;` this is proven automatically. Indeed, for the trivial invariant, the (initially valid) and (preserved by body) branches are always closable. The negated guard gives `false` and from that anything is provable, including the stated postcondition. The initialization in front of the loop is completely irrelevant and could have been left out.

On the other hand, to prove *termination* of a loop we need additional machinery. In KeY we use well-founded orders, i.e. partial orders without infinite descending chains. In this chapter we use only the natural numbers in their standard ordering $0 < 1 < 2 < \cdots$. The idea is to define an arithmetic expression d over program variables that is proven to become strictly smaller, but not negative, in each loop iteration. This is called *decreasing term* or *variant*. Since any natural number has only a finite number of predecessors, it follows that a loop with a decreasing term must terminate after a finite number of iterations.

The principle is illustrated in Figure 16.2. Assume that, when we execute the loop body the first time, the decreasing term d is evaluated to $N \geq 0$. In the next iteration it must be evaluated to a value smaller than N, and so on. After a finite number of rounds 0 is reached. As d must be nonnegative, the loop must terminate then.

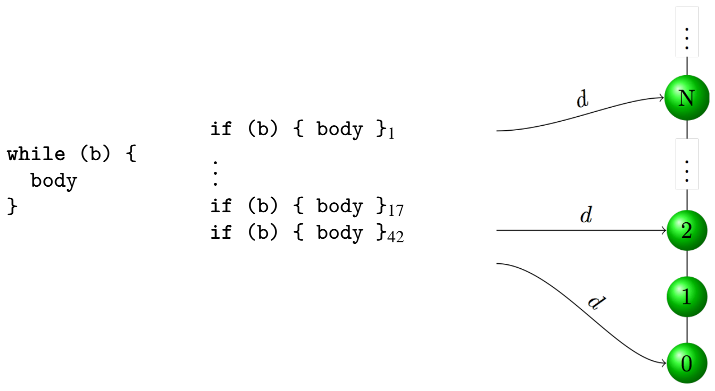


Figure 16.2 Mapping loop execution into a well-founded order

The loop invariant rule for total correctness can now be derived from the version for partial correctness in a straightforward manner, by simply adding the decreasing term with the according proof obligations:

1. We must strengthen the invariant *inv* by stating that the decreasing term d stays nonnegative, resulting in $inv \land d \geq 0$.
2. The postcondition of the (preserved by body) branch must state that the value of d is strictly less than it was at the beginning of the execution of the loop body.

The result is the following *invariant rule with context preservation for termination*:

$$\frac{\begin{array}{c}\Gamma \Longrightarrow \{u\}inv, \Delta \\ \Gamma \Longrightarrow \{u\}\{v\}\big(inv \land g \doteq TRUE \land d \geq 0 \land d' \doteq d \rightarrow \langle p\rangle(inv \land d \geq 0 \land d > d')\big), \Delta \\ \Gamma \Longrightarrow \{u\}\{v\}\big(inv \land g \doteq FALSE \rightarrow \langle \pi\ \omega\rangle\varphi\big), \Delta\end{array}}{\Gamma \Longrightarrow \{u\}\langle \pi\ \texttt{while}(g)p\texttt{;}\ \omega\rangle\varphi, \Delta}$$

where $\{v\}$ is the anonymizing update for the assignable clause **assignable** $l_1, \ldots, l_n$;. Moreover, d' is a fresh integer constant.

At the level of JML, total correctness is achieved by

1. removing the partial correctness directive **diverges true**; from the surrounding contract and
2. adding a directive “**decreasing** d;,” where d is a decreasing term.

This causes KeY to create suitable proof obligations with total correctness modalities and to choose the terminating version of the invariant rule.

To prove that the loop in the example from the previous subsection terminates, it is sufficient to remove the **diverges true**; directive and to add the directive “**decreasing** a.length - i;” to the loop specification.

Sometimes a termination witnessing decreasing term of type integer is very difficult or even impossible to find. JML and KeY support more general **decreases** clauses working, e.g., with pairs or sequences. Details can be found in Section 9.1.4 on the verification of terminating recursive methods.

16.3.8 A More Complex Example

We use a slightly more complex example to illustrate a few more heuristics that can be useful when developing loop invariants. Below is the JML specification and Java implementation of method gcdHelp that computes the greatest common divisor (gcd) of two integers _big and _small under the normalizing assumption that _big is at least as large as _small which in turn is not negative. It can be used to implement a method gcd for arbitrary numbers (not shown here).

Java + JML

```
public class Gcd {
  /*@ public normal_behavior
    @ requires _small >= 0 && _big >= _small;
    @ ensures _big != 0 ==>
    @  (_big % \result == 0 && _small % \result == 0 &&
    @    (\forall int x;
    @             x > 0 && _big % x == 0 && _small % x == 0;
    @             \result % x == 0));
    @ assignable \nothing;
    @*/
  private static int gcdHelp(int _big, int _small) {
    int big = _big; int small = _small;
    while (small != 0) {
      final int t = big % small;
      big = small;
      small = t;
     }
    return big;
  }
}
```

Java + JML

A result is only defined for the nontrivial case when _big is positive. In this case, the returned value must be a common divisor of both _big and _small which is ensured by "`_big % \result == 0 && _small % \result == 0`." In addition, the returned value must be the *greatest* common divisor. This is expressed by the quantified formula which states that any positive `x` that is a common divisor of `_big` and `_small` must also be a divisor of the result and hence not greater.

The code above does not yet specify a loop invariant. We must supply a specification of the loop that allows us to prove the given contract. Obviously, the loop doesn't modify any location that is visible outside, therefore, we use **`assignable \nothing;`**. The decreases term is also straightforward: `small` is initially nonnegative and certainly it decreases whenever the loop is entered, therefore, we use "**`decreasing`** `small;`."

To develop the loop invariant we look first at the requires clause to see what could be preserved. A quick check tells us that the properties of `_big` and `_small` also hold for the variables `big` and `small` that are used in the loop (we introduced these fresh names, because this results in a more readable invariant). Therefore, the first part of our invariant is:

```
small >= 0 && big >= small
```

What else can we say about the boundaries of `big` and `small`? For example, can `big` become zero? Certainly not in the loop body, because it is assigned the old value of `small` which is ensured by the loop guard to be nonzero. However, it is admissible to call the method with `_big` being zero, so `big` > 0 might not initially be valid. Only when `_big` is non zero, we can assume `big` > 0 to be an invariant. Hence, we add the *relative* invariant

```
_big != 0 ==> big != 0  .                    (16.1)
```

But what is the *functional* property that the loop preserves? In the end we need to state something about all common divisors `x` of `_big` and `_small`. Which partial result might have been achieved during execution of the loop? A natural conjecture is to say something about the common divisors of `big` and `small`: in fact these should be *exactly* the common divisors of `_big` and `_small`. Because, if not, we could run in danger to "loose" one of the divisors during execution of the loop body. This property is stated as

```
(\forall int x; x > 0; (_big % x == 0 && _small % x == 0)
                        <==> (big % x == 0 && small % x == 0));
```

We summarize the complete loop specification below. With it, KeY can prove total correctness of `gcdHelp` fully automatically in a few seconds.

—— Java + JML ——————————————————————————

```
int big = _big; int small = _small;
/*@ loop_invariant small >= 0 && big >= small &&
  @ (big == 0 ==> _big == 0) &&
  @ (\forall int x; x > 0; (_big % x == 0 && _small % x == 0)
  @                  <==> (big % x == 0 && small % x == 0));
  @ decreases small;
  @ assignable \nothing;
  @*/
while (small != 0) {
  final int t = big % small;
  big = small;
  small = t;
}
return big; // will be assigned to \result
```
——————————————————————— Java + JML ——

Perhaps the reader wonders why the loop invariant is actually sufficient to achieve the postcondition of the contract, specifically, why is it the case that the returned value, i.e., the final value of big after the loop terminates, is a divisor of both _big and _small? Now, this needs only to be shown when big is positive, because of (16.1). In that case, the third part of the invariant can be instantiated with x/big. Using that small == 0 (the negated loop guard) then completes the argument. This kind of reasoning is easily within the first-order inference capabilities of KeY.

16.3.9 Invariants: Concluding Remarks

The discussion in this section hopefully demonstrated that loop invariants must be systematically developed: they don't come out of thin air or appear magically after staring at a program for long enough. The process of loop invariant discovery is comparable to bug finding: it is a cycle consisting of analysis of the target program, generation of an informed conjecture and then confirmation or refutation of the conjecture. If the latter happens, the reasons for failure must be analyzed and they form the basis of the next attempt.

Good starting points for invariant candidates are the postcondition (what, in addition to the negated loop guard is needed to achieve it?) and the precondition of the problem's contract. Another source is the result of symbolic execution of one loop guard and body. But one such execution yields usually no invariant: it is necessary to relate the state before and after symbolic execution of the loop body to each other in the invariant. A good question to ask is: how can I express the partial result computed by one execution of the loop body? Often, symbolic execution of a few loop iterations can give good hints.

If a loop invariant that suffices to prove the problem at hand seems elusive, don't forget that your program or your specification of it might be buggy. Ask yourself questions such as: does the postcondition really hold in each case? Are assumptions missing from the precondition? Another possibility is that you attempt to use a stronger loop invariant than is required. The *Model Search* feature of the KeY prover (see Section 15.4) can be very useful to generate counter examples that give a hint, in case some proof goal stays open.

For complex loops, it is often the case that several rounds of strengthening and weakening of the invariant candidate is required, before a suitable one is found. In this case, it is a good idea to develop invariants incrementally. This is possible, because invariants are closed under conjunction. Start with simple value bounds and well-formedness assumptions. These may exhibit flaws in the target program or specification already. It is also a good idea to work with *relativized* invariants that can be tested separately. For example, it can be simpler to test $cnd \rightarrow inv$ and $\neg cnd \rightarrow inv$ separately than to work with inv directly.

Remember that there is no single loop invariant that is suitable to prove the problem at hand, but there are typically many reformulations that do the trick. There could be simpler formulations than the first one that comes to mind. In particular, try to avoid quantified formulas as much as possible in invariants, because they are detrimental to a high degree of automation in proof search.

It is recommended to use the KeY prover to confirm or to refute conjectures about invariants, as symbolic execution by hand is slow and error-prone. If a loop occurs within a complex context (for example, nested with/followed by other loops) it can be useful to formulate the invariant as a separate contract and look at just that proof obligation in isolation.

In this section we tried to give some practical hints on systematic development of loop invariants. There is much more to say about this topic. For example, so as not obscure the basic principles we left out the complications arising from heap access or from abrupt termination of loop bodies. More information on how JavaDL handles these issues can be found in Chapter 3. More complex examples of loop invariants can be found in the subsequent section and in Part V of this book.

16.4 A Program with Loops

Listing 16.2 shows the code of a Java class `Sort` implementing the selection sort algorithm. This is a simple, not very effective sorting algorithm, see e.g. [Knuth, 1998, Pages 138—141 of Section 5.2.3]. The integer array to be sorted is stored in the field **`int`**`[] a` of the class `Sort`. At every stage of the algorithm the initial segment `a[0]` ...`a[pos-1]` is sorted in decreasing order. The tail `a[pos]` ...`a[a.length-1]` is still unsorted but every entry `a[i]` in the tail is not greater than `a[pos-1]`. At the beginning `pos=0`. On termination `pos=a.length-1`, which means that `a[0]` ...`a[a.length-2]` is sorted in decreasing order and `a[a.length-1]` is not greater than `a[a.length-2]`. Thus the whole array is indeed sorted.

To proceed from one stage in the algorithm to the next, as long as `pos` is still strictly less than `a.length-1`, an index `idx` is computed such that `a[idx]` is maximal among `a[pos]` ...`a[a.length-1]`, the entries `a[idx]` and `a[pos]` are swapped and `pos` is increased by one.

The main part of this algorithm is implemented in the method `sort()` in lines 26 to 46 of Listing 16.2. The index of a maximal entry in the tail `a[pos]`

```
public class Sort {
  public int[] a;

  /*@ public normal_behavior
    @ requires a.length > 0 && 0<= start && start < a.length;
    @ ensures (\forall int i; start<=i && i<a.length;a[\result] >= a[i]);
    @ ensures start <= \result && \result < a.length;
    @*/
  int /*@ strictly_pure @*/ max(int start) {
    int counter = start;
    int idx = start;
    /*@ loop_invariant  start<=counter && counter<=a.length &&
      @   start<=idx && idx<a.length  && start<a.length &&
      @   (\forall int x; x>=start && x<counter; a[idx]>=a[x]);
      @ assignable \strictly_nothing;
      @ decreases a.length - counter;
      @*/
    while (counter < a.length) {
      if (a[counter] > a[idx])
        idx = counter;
      counter = counter+1;
    }
    return idx;
  }

  /*@ public normal_behavior
    @ requires a.length > 0;
    @ ensures (\forall int i; 0 <= i && i<a.length-1; a[i] >= a[i+1]);
    @*/
  void sort() {
    int pos = 0;
    int idx = 0;
    /*@ loop_invariant 0<=pos && pos<=a.length && 0<=idx && idx<a.length
      @   && (\forall int x; x>=0 && x<pos-1; a[x]>=a[x+1]) &&
      @  (pos>0 ==>(\forall int y; y>=pos && y<a.length; a[pos-1]>=a[y]));
      @ assignable a[*];
      @ decreases a.length - pos;
      @*/
    while (pos < a.length-1) {
      idx = max(pos);
      int tmp = a[idx];
      a[idx] = a[pos];
      a[pos] = tmp;
      pos = pos+1;
    }
  }
}
```

Listing 16.2 Second example: Sorting an array

. . . a[a.length-1] of the array is returned by the method call max(pos). Method max(**int** start) is given in lines 9–24 in Listing 16.2.

The specification of sort() says that this method terminates without an uncaught exception (line 26) and upon termination array a is sorted in decreasing order (line 28). The only precondition, a.length>0, required of method sort() is stated in line 27. Inspection shows that the code would also handle the case a.length=0 correctly. But, the loop invariant would have to be rephrased. As it stands 0<=idx && idx<a.length would not be true at the beginning of the loop. There is no need to also require that a is not the **null** object since JML tacitly takes this as the default.

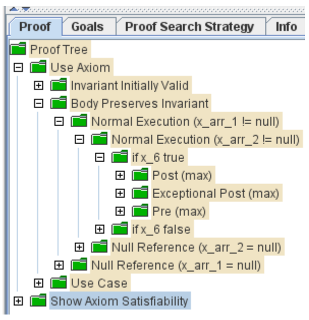
The loop invariant starts in line 33 with the statement that the local variables pos and idx stay within their bounds. The remaining two lines formalize the informal description of the algorithm given a above: The formula in line 34 says that the initial segment a[0] . . . a[pos-1] is sorted in decreasing order while line 35 contains the formalization of the description that every entry a[i] for pos <= i < a.length+ is not greater than a[pos-1]. This is not true for pos=0, so the condition pos>0 ==> has to be prefixed. Line 31 specifies the locations that may at most be changed by the loop body. See Section 16.3.6 for a general introduction of the use of **assignable** clauses. Also pos and idx may be changed in the loop body, but the KeY system can figure this out by itself. Only possible changes to heap locations need to be declared.

To allow the system to prove termination of the loop an integer expression is needed that is never negative and strictly decreases in each loop iteration. The term a.length-pos given in the **decreases** clause in line 37 serves this purpose. See the previous Section 16.3.7 for a gentle introduction to termination proofs.

Let us now turn to the contract for method max. This method is declared to be **strictly_pure** in line 9, which means that is does not change any field of any existing object and also does not create new objects. The precondition, line 5, requires the parameter start to be within the bounds of array a. The conjunctive part a.length>0 is here for the same reason as in the precondition of sort. The postcondition ensures that the returned index, denoted by the JML keyword result, is taken from the tail segment start, . . . a.length-1, line 7, and that a[result] is indeed maximal among a[start], . . . a[a.length-1], line 6.

The loop invariant begins in lines 12 – 13 with a declaration that the method parameter start and the local variables counter and idx stay within their intended ranges. In Section 16.3.9 it was proposed as a guideline for finding invariants to look at the postcondition and the loop guard. This advice works very well in the case at hand. In the end, i.e., when counter=a.length, we want a[idx] to be maximal among a[start],. . . , a[a.length-1]. This suggests as an invariant that a[idx] be maximal among a[start],. . . , a[counter-1]. This is formalized in the formula in line 14. The frame condition in the **assignable** clause in line 15, and the **decreases** clause in line 16 are self explanatory.

The KeY system verifies the contracts for both methods automatically with the settings **Java verif. std.** Make sure that **Max. Rule Applications** is at least 6000. Let us inspect the finished proof. For this open the **Proof** tab in the lower left-hand pane, place the cursor over any node, activate the menu by a right mouse click and select the **Hide Intermediate Proofsteps** entry. After opening some of the green folder symbols



```
Proof Tree
⊟ Use Axiom
    ⊞ Invariant Initially Valid
    ⊟ Body Preserves Invariant
        ⊟ Normal Execution (x_arr_1 != null)
            ⊟ Normal Execution (x_arr_2 != null)
                ⊟ if x_6 true
                    ⊞ Post (max)
                    ⊞ Exceptional Post (max)
                    ⊞ Pre (max)
                ⊞ if x_6 false
            ⊞ Null Reference (x_arr_2 = null)
        ⊞ Null Reference (x_arr_1 = null)
    ⊞ Use Case
⊞ Show Axiom Satisfiability
```

Figure 16.3 Condensed finished proof tree

the proof tree looks as in Figure 16.3 on the left. The proof goals `Use Axiom` at the top and `Show Axiom Satisfiability` at the bottom of the first column refer to the type or class invariant. This JML concepts was already alluded to in Section 15.3. A type invariant is a formula that is stipulated to be true in any *visible* state. E.g., a type invariant is assumed to be true at every method call and must be verified to be true after method termination, or as is the case in the situation under study, the invariant axiom is assumed to be true at the beginning of a while loop and has to be established after its termination. For the class `Sort` the invariant `a!=null` is automatically generated from the JML default. In general, the user may specify any invariant he believes to be useful. To guard against the possibility that the chosen invariant is inconsistent, the proof goal `Show Axiom Satisfiability` is generated and has to be discharged, which is absolutely trivial in the present situation.

The three proof goals on the second vertical line in the screenshot 16.3 are generated when symbolic execution reaches the while loop. As explained in Section 16.3.2, the new goals are *Invariant Initially Valid*, *Body Preserves Invariant* and *Use Case*. The interesting branch is *Body Preserves Invariant* which has been unfolded three times. We skip the next three columns in screenshot 16.3 and turn to the three goals `Post(max)`, `Exceptional Post(max)`, and `Pre(max)`. They are generated when symbolic execution reaches the method call `max`. According to the proof settings the method call to `max` is not symbolically executed, its contract is used instead. This involves verifying that its precondition, `Pre(max)`, is satisfied and continuing in case `max` terminates exceptionally with the proof branch labeled *Exceptional Post(max)* and in case of normal termination, *Post(max)*, with the respective guarantees ensured by the contract in both cases.

There is one more issue that can be demonstrated already with the small example program under investigation. How precise should a postcondition be? There is the notion of a strongest postcondition, but this is not always expressible in first-order logic and may also be undesirably complicated. The postcondition of method `max` in line 6 of Listing 16.2, e.g., is not the strongest possible. One could add that `\result` is the least index of a maximal value among `a[start]` ... `a[a.length-1]`:

```
(\forall int i; start <= i && i < \result; a[i]<\result)
```

But, that would complicate verification without being necessary in the present context. Thus, how precise the postcondition should be may depend in what it is being used for.

16.5 Data Type Properties of Programs

Listing 16.3 contains the same Java code as Listing 16.2. Also the contract for method `max` is the same. The differences lie in the contract for method `sort` in lines 30–33 and in the declaration of *model fields* in lines 4–6. In the postcondition for method `sort` in line 28 in Listing 16.2 an important assurance is missing: that the array `a` after termination is a permutation of the array when the method was called. More precisely, we want to say that there is a permutation σ of the integers $0, \dots, a.length - 1$ such that for all $0 \leq i < a.length$ we have $a_{new}[i] = a_{old}[\sigma(i)]$. To formalize this statement we introduce the abstract data type *Seq* of finite sequences. This data type is described in detail in Section 5.2. For this tutorial it will suffice to think of a finite sequence as mathematical function σ whose domain of definition is a finite initial segment of the positive integers. In general the range of values of σ is quite liberal. Here, we only encounter finite sequences of integers. A permutation is then defined as a finite sequence that is a surjective, and thus also injective, function from its domain onto its domain. The data type *Seq* contains a binary predicate $seqPerm(s_1, s_2)$ with the intended meaning, that sequence s_1 is a permutation of s_2. This is not to be confused with the unary predicate $seqNPerm(s)$ which says that s is a permutation, i.e., that s is a bijective function from $[0, seqLen(a))$ onto $[0, seqLen(a))$, where predictably $seqLen(s)$ is the length of sequence s.

This seems to be the right time to point out a troubling obstacle to our idea to use sequences and permutation to formulate the intended postcondition of method `sort`: sequences and permutations do not occur anywhere in the Java code and Java code is all JML allows us to talk about. As a solution JML offers the declaration of *model fields*. In line 4 of Listing 16.3 a model field of class `SortPerm` named `seqa` of type $\backslash seq$ is declared. Here, $\backslash seq$ simply is the JML name for the data type *Seq*. A model field is a field that is only used for modeling purposes. Written as a special comment, like all JML specifications, it is ignored by the Java compiler. Values to model fields are assigned by the JML **`represents`** clause. In line 5 of Listing 16.3 `seqa` is assigned the sequence that corresponds to the field `a`. The transformation from a state-dependent Java array to a state-independent object of data type *Seq* is effected by the built-in function *array2seq*. The data type *Seq* and also the function *array2seq* are not part of official JML. It belongs to our project specific extension of JML, that we hope will at some time also be adopted in the official version. In the meantime we will use the escape sequence `\dl_` to signal to the JML parser that the following item is not JML syntax and is to be passed unchanged on to the translator from JML into our internal logic *JavaDL*. After these explanations we see that line 32 in Listing 16.3 formalizes the postcondition we want: the sequence corresponding to array `a` after method termination is a permutation of the sequence corresponding to array `a` at method invocation. Since again *seqPerm* is not part of official JML we have to use the escape `\dl_seqPerm`.

Now, that we understand the specification let us see how we can prove it. We start with the taclet base configuration. To this end load any file containing JML annotated Java code and select a contract target. This is necessary since the menu item we are looking for, **Taclet Options**, is not active when no proof is loaded. Clicking on menu

```
public class SortPerm {
  public int[] a;

  /*@ model \seq seqa;
    @ represents seqa = \dl_array2seq(a);
    @*/

  /*@ public normal_behavior
    @ requires a.length > 0 && 0<= start && start < a.length;
    @ ensures (\forall int i;start<=i && i<a.length; a[\result] >= a[i]);
    @ ensures start <= \result && \result < a.length;
    @*/
  int /*@ strictly_pure @*/ max(int start) {
    int counter = start;
    int idx = start;
    /*@ loop_invariant  start<=counter && counter<=a.length &&
      @   start<=idx && idx<a.length  && start<a.length &&
      @   (\forall int x; x>=start && x<counter; a[idx]>=a[x]);
      @ assignable \strictly_nothing;
      @ decreases a.length - counter;
      @*/
    while (counter<a.length) {
      if (a[counter] > a[idx])
        idx=counter;
      counter=counter+1;
    }
    return idx;
  }

  /*@ public normal_behavior
    @ requires a.length > 0;
    @ ensures  \dl_seqPerm(seqa,\old(seqa));
    @*/
  void sort() {
    int pos = 0;
    int idx = 0;
    /*@ loop_invariant 0<=pos && pos<=a.length && 0<=idx && idx<a.length
      @   && \dl_seqPerm(seqa,\old(seqa));
      @ assignable a[*];
      @ decreases a.length - pos;
      @*/
    while (pos < a.length-1) {
      idx = max(pos);
      int tmp = a[idx];
      a[idx] = a[pos];
      a[pos] = tmp;
      pos = pos+1;
    }
  }
}
```

Listing 16.3 Third example: Permutations

item **Options, Taclet Options** after a proof is loaded opens the **Taclet Base Configuration** window. Somewhere in the middle of the list you see `moreSeqRules`. Clicking on it shows the two options **moreSeqRules:off** and **moreSeqRules:on**. By default this option is turned off, but we will need it to reason about permutations. After pushing the **OK** button, the system will inform you that you have to instantiate a new proof for the changes to take effect. Do this, now by loading the file `SortPerm.java`. Since KeY loads all Java files in a directory we have to select in the **Proof Management** window the file `SortPerm` and the method `sort()`. This proof will not close automatically. The prover will need a little help from us. We want to keep interactions to a minimum but at the same time have control over what the prover tries to do. This is where strategy macros come into play.

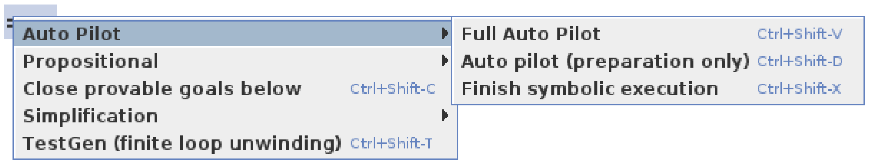
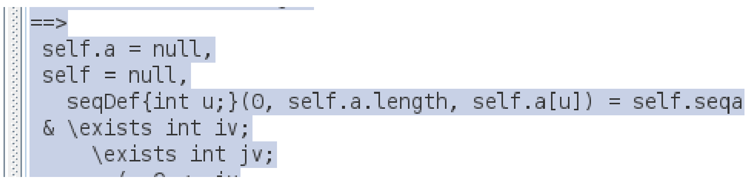

Auto Pilot
Propositional
Close provable goals below Ctrl+Shift-C
Simplification
TestGen (finite loop unwinding) Ctrl+Shift-T
Full Auto Pilot Ctrl+Shift-V
Auto pilot (preparation only) Ctrl+Shift-D
Finish symbolic execution Ctrl+Shift-X

Figure 16.4 Strategy macros

Placing the cursor over the sequent separation arrow ==> a click on the right mouse button will display the list of strategy macros shown in the screenshot 16.4 above. Alternatively, you can press the left mouse button and select the **Strategy macros** menu. For now, we select the **Full Auto Pilot** which does the following:

1. Finish symbolic execution (another macro in itself)
2. Separate proof obligations
3. Expand invariant definitions
4. Close provable goals (another macro in itself)

Alternatively one could click on the left mouse button to obtain a selection of possible next steps. Among them is one named **Apply rules automatically here** which starts the proof search strategy only for the current goal/formula. This differs from the macro **Close provable goals below** in that it runs till the maximal number of proof steps is exhausted and may thus stop in a proof situation that is hard to figure out. So, let us apply the **Full Auto Pilot** macro with the maximal number of rule applications set to 5000. One goal remains. Inspection of the proof tree shows that the open goal claims that the loop invariant is preserved by an execution of the loop body. Above the sequent separator ==> we find the assumption `seqPerm(s1,t1)`, where `s1` denotes the model field `seqa` at the beginning of an arbitrary loop iteration and `t1` stands for `seqa` at the beginning of the loop. Below the sequent separator ==> we find the claim `seqPerm(s2,t2)`. Here `t2` is a different representation for the same sequence as `t1`, in which the represents clause for `seqa` has not yet been applied. Also `s2` denotes the value of `seqa` at the end of the arbitrary loop execution. Thus `s2` is obtained from `s1` by swapping two entries. So we need to prove: if `s1` is a permutation of `t1 (=t2)` and `s2` is a swap of `s1`, then also `s2` is a permutation of `t1 (=t2)`. There is fortunately a taclet that provides exactly this argument. Place the cursor over the occurrence of `seqPerm` below the sequence separator, press the left mouse button and select from the presented suggestions the taclet seqPermFromSwap. Since the system cannot decide when it is useful to apply this taclet, i.e., when `s2` is a swap of `s1`, the heuristics of this taclet forbids automatic application. User interaction is thus needed here.



```
==>
 self.a = null,
 self = null,
   seqDef{int u;}(0, self.a.length, self.a[u]) = self.seqa
 & \exists int iv;
     \exists int jv;
```

Figure 16.5 After `seqPermFromSwap`

After rule application the lower (right) part of the sequent starts as shown in the Snapshot 16.5. Remember that the right side of a sequent is a comma separated disjunction. We may thus assume `self.a != null` and `self != null` and try to prove the remaining conjunction. Place the cursor over the conjunction symbol `&` and select the taclet andRight for the next step. This splits the previous conjunctive goal into two goals, one for each conjunct. Applying the macro **Close provable goals below** we see that KeY can prove both goals on its own. Make sure that **Max. Rule Applications** is at least 10 000.

16.6 KeY and Eclipse

As we have seen in the previous sections the KeY system is powerful enough to close a proof fully automatically in many situations. All that needs to be done is to load the source code, select a contract and start the proof search strategy. After a single contract has been proven successfully, the proof remains in the almost proven state until the correctness of all applied contracts is shown as well. Our goal for this section will be to achieve overall correctness, so we are interested in proving all available proof obligations.

To achieve overall correctness is an arduous path and very likely we will not be able to achieve it the first time. Some proofs might remain open caused by defective method implementations or by too weak or wrong specifications. In such cases we have to modify code or specifications. Previously unclosed proofs can then be retried on the new code version. But also already closed proofs have to be redone since the modification may have violated them.

Tool support for verification in such an ongoing software development process requires the ability to react on source file changes and to store proofs consistently with the sources. This can't be achieved by the KeY system alone simply by the fact that it operates on a specific version of source files. Modifications always have to be done in a different tool which is typically an integrated development environment (IDE) like Eclipse.

In the following subsections we describe the usage of KeY's Eclipse integration [Hentschel et al., 2014c]. The main contribution is an automatic proof management for all proof obligations in the whole project. After each change possibly outdated proofs are determined and automatically redone. User interaction is only required if a proof is not automatically closable.

16.6.1 Installation

The Eclipse integration of KeY and other Eclipse extensions provided by the KeY project can be added to an existing Eclipse installation via an update-site. The supported Eclipse versions and the concrete update-site URLs are available on the KeY website (www.key-project.org). When reading the following sections for the first time, we strongly recommend to have a running Eclipse installation with the verification features at hand, so that they can be tried out immediately. We assume that the reader is familiar with the Java perspective of the Eclipse IDE.

16.6.2 Proof Management with KeY Projects

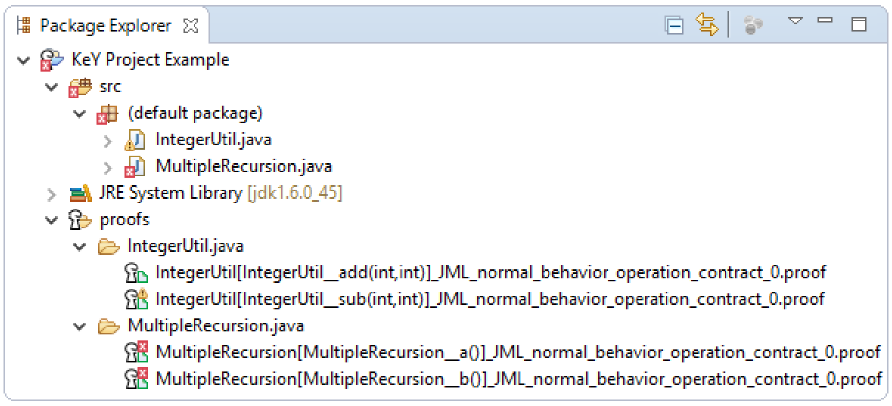
Eclipse is a platform for different purposes including software development in different programming languages. Source files are organized in projects of different kinds associated with *Builders* that are automatically invoked when the project content changes. A *Java Project* is used to develop Java applications and the associated *Java Builder* automatically compiles the contained source code.

KeY's Eclipse integration provides a new project kind named *KeY Project* which is an extended Java project. The additional functionality is that the *KeY Builder* automatically performs relevant proofs whenever source or proof files are modified.

To start we create an example KeY project which is automatically filled with some content. All we have to do is to open the *New Example* wizard, select *KeY/KeY Project Example* and finish the wizard. An empty KeY project can be created with the help of the eponymous *New Project* wizard. Alternatively, it is possible to convert an existing Java project into a KeY project via its context menu.

Performed proofs are automatically maintained in folder *proofs* as shown in Figure 16.6. For each proof obligation a *.proof* file named after it exists.

The advantage of the maintained project structure is the compatibility with version control systems. Thus a KeY Project can be directly shared and source files with proofs are always committed and updated in a consistent way. Even a comparison between different versions is possible.



```
Package Explorer
KeY Project Example
  src
    (default package)
      IntegerUtil.java
      MultipleRecursion.java
  JRE System Library [jdk1.6.0_45]
  proofs
    IntegerUtil.java
      IntegerUtil[IntegerUtil__add(int,int)]_JML_normal_behavior_operation_contract_0.proof
      IntegerUtil[IntegerUtil__sub(int,int)]_JML_normal_behavior_operation_contract_0.proof
    MultipleRecursion.java
      MultipleRecursion[MultipleRecursion__a()]_JML_normal_behavior_operation_contract_0.proof
      MultipleRecursion[MultipleRecursion__b()]_JML_normal_behavior_operation_contract_0.proof
```

Figure 16.6 KeY project with example content

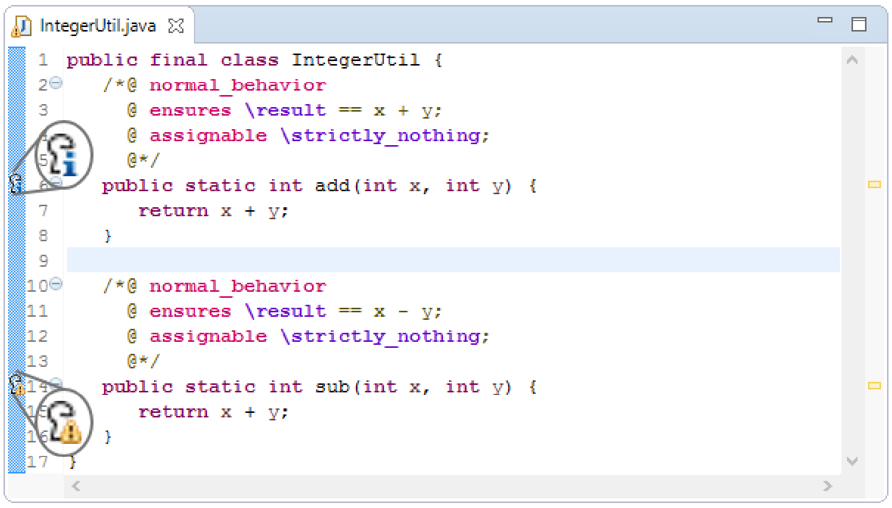
16.6.3 Proof Results via Marker

Each time the KeY Builder completed a proof, the user is immediately informed about the proof result. This is done directly in the source code as close as possible to the proven proof obligation via so called *Marker*. As Figure 16.7 demonstrates, markers are shown as icons next to the line number within the *Java Editor*. In this example the method `add` is successfully proven (information marker) whereas the proof of method `sub` is still open (warning marker).

The presence of warning and error markers can also be seen in view *Package Explorer*. Whenever a Java source file contains a warning ( ) or error marker ( ), an overlay image is added to the file icon. In case that a file contains both, warning and error markers, only the more urgent error icon is shown. In addition, the most urgent marker type is also delegated to parent folders and the project. This can be seen in Figure 16.6 where the default package has an error overlay image, because the error in class `MultipleRecursion` is more urgent than the warning in class `IntegerUtil`.

To find out why the proof of method `sub` was not closed by the proof search strategy all we have to do is to move the mouse over the marker icon. In general two reasons are possible: First, the strategy could be not powerful enough to close it or second, because the implementation or its specifications are defective.

Here, the implementation is obviously defective. We can easily fix it by replacing **`return`** `x + y` with **`return`** `x - y`. When we save the file now, the KeY Builder will be triggered. It performs the proofs again and updates the result marker. It is also possible to inspect and to interactively continue proofs using the *quick fix* functionality (i.e., left click on the marker icon). This opens the proof in the original user interface of KeY as earlier discussed in this chapter. Alternatively, a proof can be inspected using the Symbolic Execution Debugger (see Chapter 11). When an interactively completed proof is saved back to its original location, the KeY Builder will be triggered and in turn update the result marker.



```
public final class IntegerUtil {
   /*@ normal_behavior
     @ ensures \result == x + y;
     @ assignable \strictly_nothing;
     @*/
   public static int add(int x, int y) {
      return x + y;
   }

   /*@ normal_behavior
     @ ensures \result == x - y;
     @ assignable \strictly_nothing;
     @*/
   public static int sub(int x, int y) {
      return x + y;
   }
}
```

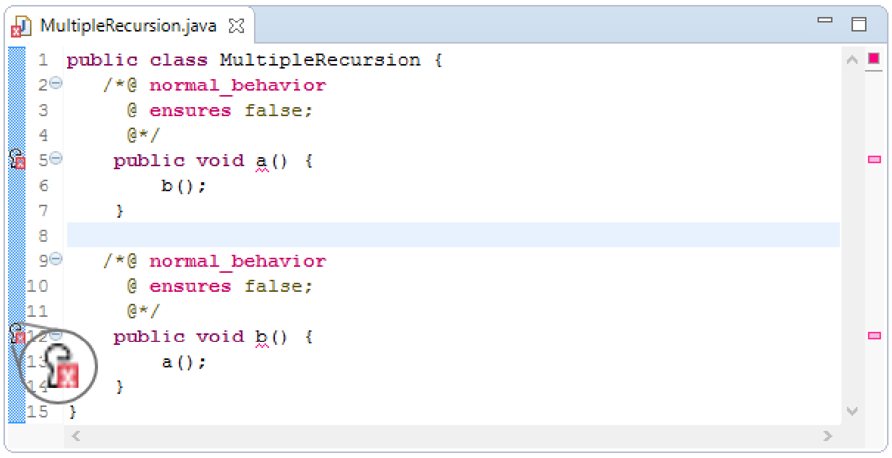
Figure 16.7 Closed and open proof marker

The *Use Operation Contract* rule requires to introduce another marker kind. Whereas in general the applicability of rules depends only on the current sequent, the application of method contracts requires a global correctness management to avoid cyclic contract applications of all proofs shown in the same proof management dialog. Such cycles are problematic because it allows one to prove everything, even false. To avoid cyclic contract applications in the KeY system, the rule which would cause a cycle is not applicable. The drawback is that other proofs and the order in which they are done can influence the current proof result.

This approach does not work for a KeY Project, because we can finish a single proof interactively without caring about other proofs. Consequently, the global correctness check is performed as last step during a built. If a cycle is detected, participating proofs are highlighted with an error marker ( ). An example is shown in Figure 16.8 where methods `a` and `b` successfully prove **`false`**. Both proofs apply the contract of the called method which forms a cycle indicated by the error marker. The tooltip of such marker lists all participating proofs and it is our task to modify at least one of them to break the cycle.

16.6.4 The Overall Verification Status

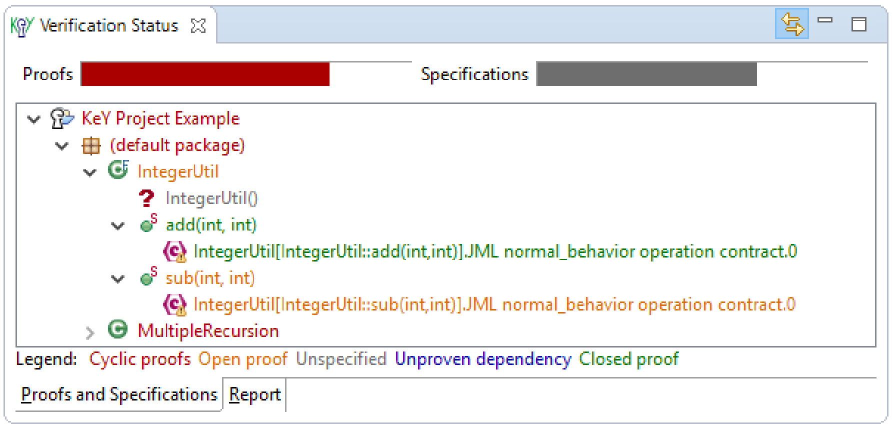
The view *Verification Status* is the best opportunity to inspect the overall verification status. Figure 16.7 shows the status of the example project before fixing the defect in method `sub`. The two progress bars at the top indicate how many proofs are already successfully proven and how many methods are specified. Absolute numbers are shown in the tooltips of the progress bars.



```
public class MultipleRecursion {
   /*@ normal_behavior
     @ ensures false;
     @*/
    public void a() {
        b();
    }

   /*@ normal_behavior
     @ ensures false;
     @*/
    public void b() {
        a();
    }
}
```

Figure 16.8 Recursive specification marker



```
KeY Project Example
  (default package)
    IntegerUtil
      IntegerUtil()
      add(int, int)
        IntegerUtil[IntegerUtil::add(int,int)].JML normal_behavior operation contract.0
      sub(int, int)
        IntegerUtil[IntegerUtil::sub(int,int)].JML normal_behavior operation contract.0
    MultipleRecursion
Legend:  Cyclic proofs  Open proof  Unspecified  Unproven dependency  Closed proof
```

Figure 16.9 The overall verification status

The tree in the middle reflects the code and the specification structure. The color of each item shows its verification result as specified by the legend below the tree. The *most problematic* result is always delegated to ancestors. It is defined as the minimal element in the following ordered list (worst to best): (i) *cyclic proofs* (the usage of specifications forms a cycle; colored red), (ii) *open proof* (colored orange), (iii) *unspecified* (no proof obligation available; colored gray), (iv) *unproven dependency* (proof is closed, but an applied specification is not verified yet; colored blue) and (v) *closed proof* (colored green). Here the only contract of method `add` is successfully proven whereas the one of `sub` is still open. The default constructor of class `IntegerUtil` is unspecified. The most problematic result below class `IntegerUtil` is the open proof and consequently, it is colored with this result. Finally, class `MultipleRecursion` contains proofs forming a cyclic specification use. As this is even more problematic, the result is also delegated to the package and the project which are colored accordingly.

A warning or information icon on a contract indicates that unsound or incomplete Taclet options are used. When we move the mouse over the contract of method `add`, the opened tooltip will list the Taclet options in detail.

Finally, tab *Report* provides a clear HTML report of the verification status including all the information discussed up to now. Additionally, the report lists all assumptions made in the proofs which have to be proven outside of the current KeY project. An example of such assumptions are for instance applied method contracts of API methods for which the correctness is not proven within the current project. Another example are method calls treated by inlining instead of a contract application. In such a case KeY performs a case distinction over all possible method implementations. Consequently, we have to ensure that the overall system in which the code of the current project is used does not influence the case distinction.