

Wolfgang Ahrendt, Bernhard Beckert,
Richard Bubel, Reiner Hähnle,
Peter H. Schmitt, Mattias Ulbrich
(Editors)

Deductive
Software Verification—
The KeY Book

From Theory to Practice

Springer

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.

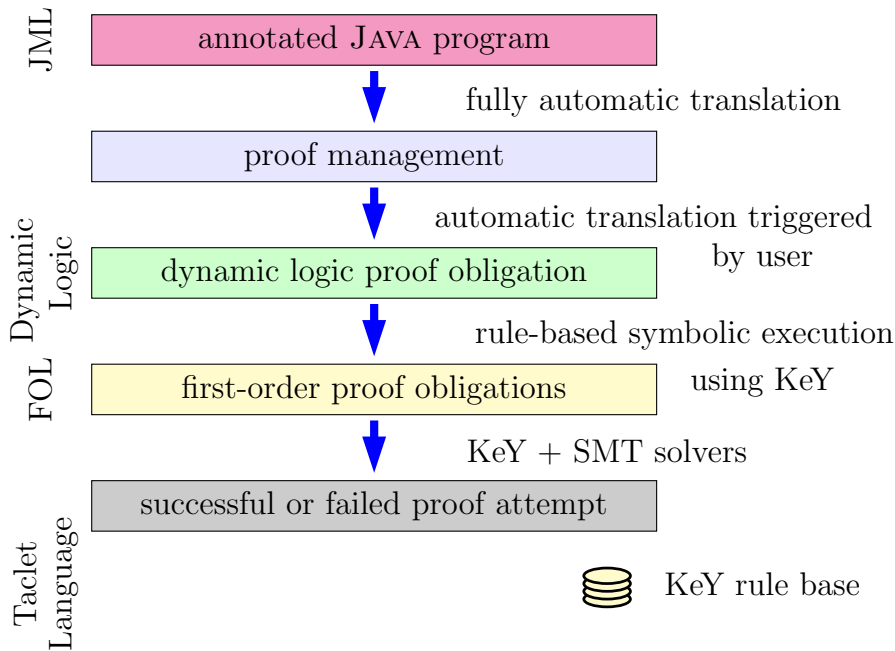


Fig. 16.1 The KeY verification workflow

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 exempted 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:

$$PP(B::select(heap,t,A::\$f)) = PP(t).f$$

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

$$PP(B :: \text{select}(H, t, A :: f)) = PP(t).f@H.$$

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

$$(\text{self.rec@heapAtPre}).y@heapAtPre$$

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 $a[\text{pos2}]@H$ stands for the full version $\text{int} :: \text{select}(H, a, \text{arr}(\text{pos2}))$.

The heap independent function $\text{arr} : \text{Int} \rightarrow \text{Field}$ (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 $A :: \text{select}(h, a, \text{arr}(i))$ 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

$$PP(A :: \text{select}(H, e, \text{arr}(a))) = PP(e)[PP(a)]@H$$

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 loop¹ is encountered, symbolic execution attempts to unwind the loop, using the rule from Section 3.6.4:

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

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 $i < a.\text{length}$, 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, $0 \leq i \ \&\& \ i < \text{Integer}.\text{MAX_VALUE}$), 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 $inv \in \text{DLFml}$ 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.

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

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 $inv \equiv i \leq n$ is an invariant we must instantiate the loop invariant rule with inv as above, $\Gamma \equiv n \geq 0$, $u \equiv i := 0$, $g \equiv i < n$, $p \equiv i = i + 1$; and empty Δ , π , ω . The instantiated (initially valid) premiss becomes

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

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

$$i \leq n, i < n \implies [i = i + 1;](i \leq n)$$

The sequent's succedent becomes after symbolic execution of the assignment and update application $i + 1 \leq n$, which is clearly provable from the antecedent. Therefore, $inv \equiv i \leq n$ 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: $inv \equiv x \doteq x_0 + y_0$. 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 $inv \equiv x + y \doteq x_0 + y_0$, 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 $x + y \doteq x_0 + y_0 \wedge y \leq 0$. It would be sufficient, if we knew that $y \geq 0$. 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, $y \geq 0$ is a loop invariant as well. Using the combined invariant $inv \equiv x + y \doteq x_0 + y_0 \wedge y \geq 0$ 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 $a \neq \text{null} \in \Gamma$ 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 $a \neq \text{null}$ 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 $a \neq \text{null} \in \Gamma$ 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, \dots, 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 $\{i := c\}$, 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:

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

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

Observe that the proof context $\Gamma, \Delta, \{u\}$ 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 `a ≠ 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:

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

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 $0 < 1 < 2 < \dots$. 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.

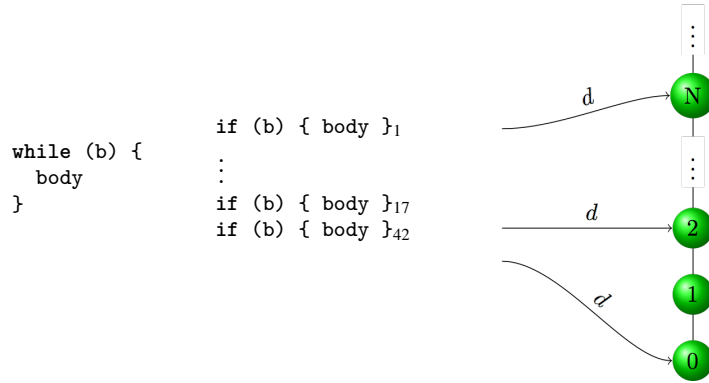


Fig. 16.2 Mapping loop execution into a well-founded order

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 $inv \wedge d \geq 0$.
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*:

$$\frac{\Gamma \Longrightarrow \{u\}inv, \Delta \quad \Gamma \Longrightarrow \{u\}\{v\}(inv \wedge g \doteq TRUE \wedge d \geq 0 \wedge d' \doteq d \rightarrow \langle p \rangle(inv \wedge d \geq 0 \wedge d > d')), \Delta \quad \Gamma \Longrightarrow \{u\}\{v\}(inv \wedge g \doteq FALSE \rightarrow \langle \pi \ \omega \rangle \varphi), \Delta}{\Gamma \Longrightarrow \{u\}\langle \pi \text{ while}(g) p; \ \omega \rangle \varphi, \Delta}$$

where $\{v\}$ is the anonymizing update for the assignable clause **assignable** l_1, \dots, 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 &&
    @     (\forallall 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:

$$\text{small} \geq 0 \ \&\& \ \text{big} \geq \text{small}$$

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

$$_big \neq 0 \implies \text{big} \neq 0 \quad . \quad (16.1)$$

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

$$\left(\text{\forall} \text{forall int } x; x > 0; (_big \% x == 0 \ \&\& \ _small \% x == 0) \right. \\ \left. \iff (\text{big} \% x == 0 \ \&\& \ \text{small} \% x == 0) \right);$$

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) &&
  @ (\forallall 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 $cond \rightarrow inv$ and $\neg cond \rightarrow inv$ 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 ?? 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 $a[0] \dots a[pos-1]$ is sorted in decreasing order. The tail $a[pos] \dots a[a.length-1]$ 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 $a[0] \dots a[a.length-2]$ 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 $a[pos] \dots a[a.length-1]$, 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 $a[pos]$

```

1 public class Sort {
2   public int[] a;
3
4   /*@ public_normal_behavior
5     @ requires a.length > 0 && 0<= start && start < a.length;
6     @ ensures (\forall int i; start<=i && i<a.length;a[\result] >= a[i]);
7     @ ensures start <= \result && \result < a.length;
8     @*/
9   int /*@ strictly_pure @*/ max(int start) {
10    int counter = start;
11    int idx = start;
12    /*@ loop_invariant start<=counter && counter<=a.length &&
13      @ start<=idx && idx<a.length && start<a.length &&
14      @ (\forall int x; x>=start && x<counter; a[idx]>=a[x]);
15      @ assignable \strictly_nothing;
16      @ decreases a.length - counter;
17      @*/
18    while (counter < a.length) {
19      if (a[counter] > a[idx])
20        idx = counter;
21      counter = counter+1;
22    }
23    return idx;
24  }
25
26  /*@ public_normal_behavior
27    @ requires a.length > 0;
28    @ ensures (\forall int i; 0 <= i && i<a.length-1; a[i] >= a[i+1]);
29    @*/
30  void sort() {
31    int pos = 0;
32    int idx = 0;
33    /*@ loop_invariant 0<=pos && pos<=a.length && 0<=idx && idx<a.length
34      @ && (\forall int x; x>=0 && x<pos-1; a[x]>=a[x+1]) &&
35      @ (pos>0 ==>(\forall int y; y>=pos && y<a.length; a[pos-1]>=a[y]));
36      @ assignable a[*];
37      @ decreases a.length - pos;
38      @*/
39    while (pos < a.length-1) {
40      idx = max(pos);
41      int tmp = a[idx];
42      a[idx] = a[pos];
43      a[pos] = tmp;
44      pos = pos+1;
45    }
46  }
47 }

```

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 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 it 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

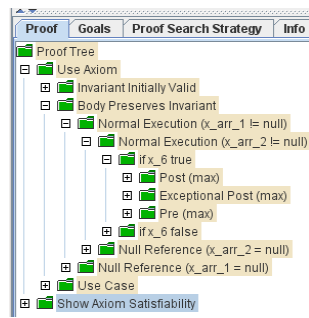


Fig. 16.3 Condensed finished proof tree

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 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 `\seq` is declared. Here, `\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 `\d1_` 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 `\d1_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

```

1 public class SortPerm {
2   public int[] a;
3
4   /*@ model \seq seqa;
5     @ represents seqa = \dl_array2seq(a);
6     @*/
7
8   /*@ public normal_behavior
9     @ requires a.length > 0 && 0<= start && start < a.length;
10    @ ensures (\forall int i; start<=i && i<a.length; a[\result] >= a[i]);
11    @ ensures start <= \result && \result < a.length;
12    @*/
13  int /*@ strictly_pure @*/ max(int start) {
14    int counter = start;
15    int idx = start;
16    /*@ loop_invariant start<=counter && counter<=a.length &&
17      @ start<=idx && idx<a.length && start<a.length &&
18      @ (\forall int x; x>=start && x<counter; a[idx]>=a[x]);
19      @ assignable \strictly_nothing;
20      @ decreases a.length - counter;
21      @*/
22    while (counter<a.length) {
23      if (a[counter] > a[idx])
24        idx=counter;
25      counter=counter+1;
26    }
27    return idx;
28  }
29
30  /*@ public normal_behavior
31    @ requires a.length > 0;
32    @ ensures \dl_seqPerm(seqa,\old(seqa));
33    @*/
34  void sort() {
35    int pos = 0;
36    int idx = 0;
37    /*@ loop_invariant 0<=pos && pos<=a.length && 0<=idx && idx<a.length
38      @ && \dl_seqPerm(seqa,\old(seqa));
39      @ assignable a[*];
40      @ decreases a.length - pos;
41      @*/
42    while (pos < a.length-1) {
43      idx = max(pos);
44      int tmp = a[idx];
45      a[idx] = a[pos];
46      a[pos] = tmp;
47      pos = pos+1;
48    }
49  }
50 }

```

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.

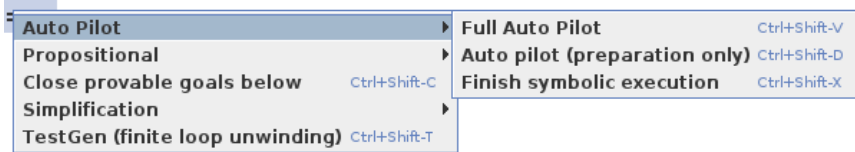


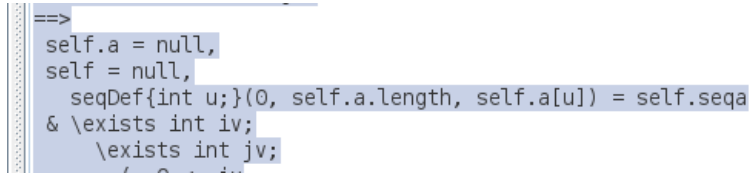
Fig. 16.4 Strategy macros

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;

```

Fig. 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.

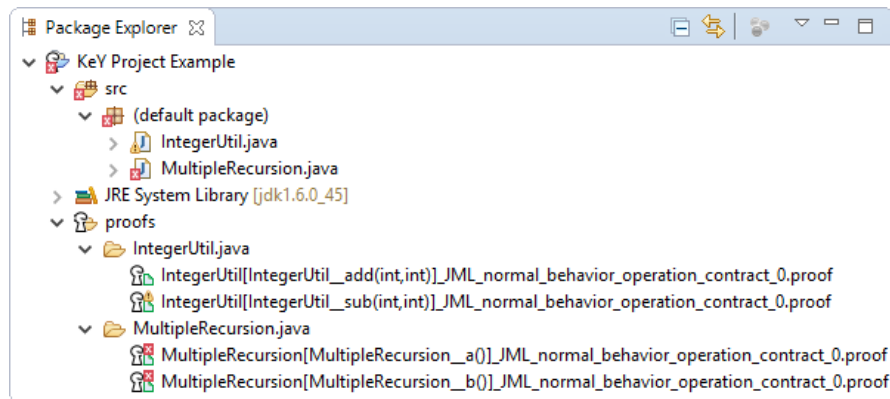


Fig. 16.6 KeY project with example content

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 (i information marker) whereas the proof of method `sub` is still open (a 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 (a) or error marker (b), 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.

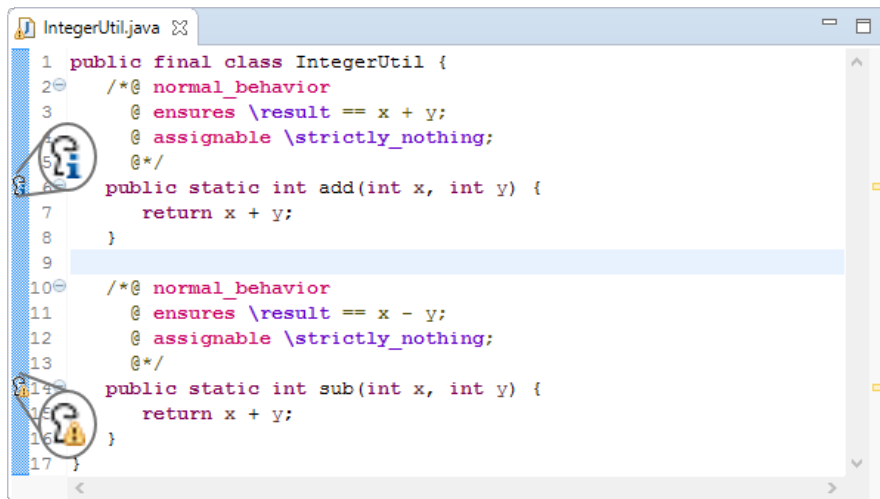


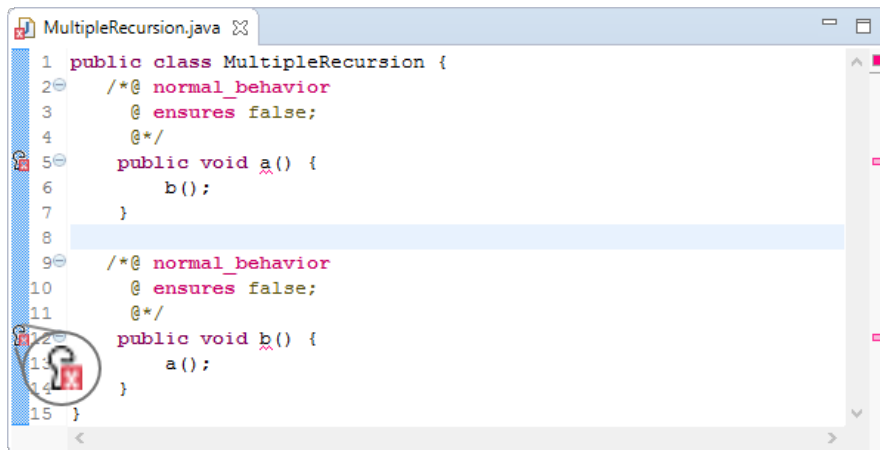
Fig. 16.7 Closed and open proof marker

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 build. 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.



```

1 public class MultipleRecursion {
2     /*@ normal_behavior
3     @ ensures false;
4     @*/
5     public void a() {
6         b();
7     }
8
9     /*@ normal_behavior
10    @ ensures false;
11    @*/
12    public void b() {
13        a();
14    }
15 }

```

Fig. 16.8 Recursive specification marker

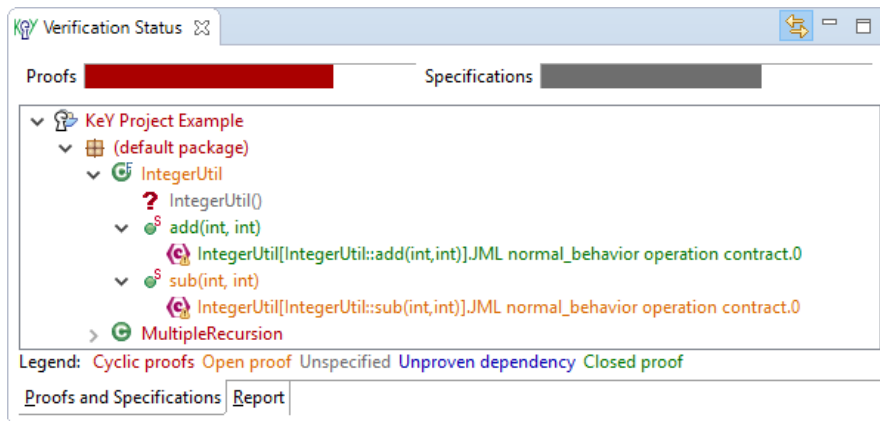


Fig. 16.9 The overall verification status

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.

References

- Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, August 1996. (Cited on page 240.)
- Ben Adida. Helios: Web-based open-audit voting. In Paul C. van Oorschot, editor, *Proceedings of the 17th USENIX Security Symposium, 2008, San Jose, CA, USA*, pages 335–348. USENIX Association, 2008. (Cited on page 606.)
- Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Wolfram Menzel, and Peter H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. In Manuel Ojeda-Aciego, Inma P. de Guzmán, Gerhard Brewka, and Luís Moniz Pereira, editors, *Proceedings of the 8th European Workshop on Logics in Artificial Intelligence (JELIA)*, volume 1919 of *LNCS*, pages 21–36. Springer, October 2000. (Cited on page 13.)
- Wolfgang Ahrendt, Andreas Roth, and Ralf Sasse. Automatic validation of transformation rules for Java verification against a rewriting semantics. In Geoff Sutcliffe and Andrei Voronkov, editors, *Proceedings, 12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Montego Bay, Jamaica*, volume 3835 of *LNCS*, pages 412–426. Springer, December 2005. (Cited on pages 12 and 64.)
- Wolfgang Ahrendt, Richard Bubel, and Reiner Hähnle. Integrated and tool-supported teaching of testing, debugging, and verification. In Jeremy Gibbons and José Nuno Oliveira, editors, *Second International Conference on Teaching Formal Methods, Proceedings*, volume 5846 of *LNCS*, pages 125–143. Springer, 2009a. (Cited on page 7.)
- Wolfgang Ahrendt, Frank S. de Boer, and Immo Grabe. Abstract object creation in dynamic logic. In Ana Cavalcanti and Dennis Dams, editors, *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands. Proceedings*, volume 5850 of *LNCS*, pages 612–627, 2009b. (Cited on page 56.)
- Wolfgang Ahrendt, Wojciech Mostowski, and Gabriele Paganelli. Real-time Java API specifications for high coverage test generation. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems, JTRES '12*, pages 145–154, New York, NY, USA, 2012. ACM. (Cited on pages 6 and 609.)
- Wolfgang Ahrendt, Jesús Mauricio Chimento, Gordon J. Pace, and Gerardo Schneider. A specification language for static and runtime verification of data and control properties. In Nikolaj Bjørner and Frank de Boer, editors, *Formal Methods - 20th International Symposium, Oslo, Norway, Proceedings*, volume 9109 of *LNCS*, pages 108–125. Springer, 2015. (Cited on page 519.)
- Elvira Albert, Miguel Gómez-Zamalloa, and Germán Puebla. Test data generation of bytecode by CLP partial evaluation. In Michael Hanus, editor, *Logic-Based Program Synthesis and Transformation, 18th International Symposium, LOPSTR, Valencia, Spain, Revised Selected Papers*, volume 5438 of *LNCS*, pages 4–23. Springer, 2009. (Cited on page 4.)

- Elvira Albert, Richard Bubel, Samir Genaim, Reiner Hähnle, and Guillermo Román-Díez. Verified resource guarantees for heap manipulating programs. In Juan de Lara and Andrea Zisman, editors, *Fundamental Approaches to Software Engineering - 15th International Conference, FASE 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia. Proceedings*, volume 7212 of *LNCS*. Springer, 2012. (Cited on page 4.)
- Eyad Alkassar, Mark A. Hillebrand, Wolfgang J. Paul, and Elena Petrova. Automated verification of a small hypervisor. In Gary T. Leavens, Peter W. O’Hearn, and Sriram K. Rajamani, editors, *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE, Edinburgh, UK*, volume 6217 of *LNCS*, pages 40–54. Springer, 2010. (Cited on page 9.)
- Afshin Amighi, Stefan Blom, Marieke Huisman, and Marina Zaharieva-Stojanovski. The VerCors project: Setting up basecamp. In Koen Claessen and Nikhil Swamy, editors, *Proceedings of the sixth workshop on Programming Languages meets Program Verification, PLPV 2012, Philadelphia, PA, USA*, pages 71–82. ACM, 2012. (Cited on pages 3, 240, 241 and 377.)
- Afshin Amighi, Stefan Blom, Saeed Darabi, Marieke Huisman, Wojciech Mostowski, and Marina Zaharieva-Stojanovski. Verification of concurrent systems with VerCors. In Marco Bernardo, Ferruccio Damiani, Reiner Hähnle, Einar Broch Johnsen, and Ina Schaefer, editors, *Formal Methods for Executable Software Models - 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2014, Bertinoro, Italy, Advanced Lectures*, volume 8483 of *LNCS*, pages 172–216. Springer, 2014a. (Cited on page 350.)
- Afshin Amighi, Stefan Blom, Marieke Huisman, Wojciech Mostowski, and Marina Zaharieva-Stojanovski. Formal specifications for Java’s synchronisation classes. In Alberto Luch Lafuente and Emilio Tuosto, editors, *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2014, Torino, Italy*, pages 725–733. IEEE Computer Society, 2014b. (Cited on pages 3 and 378.)
- Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 2008. (Cited on pages 421 and 448.)
- Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object-oriented programs. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA*, pages 91–102. ACM, 2006. (Cited on page 454.)
- Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, J. Jenny Li, and Hong Zhu. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013. (Cited on page 448.)
- Sven Apel, Wolfgang Scholz, Christian Lengauer, and Christian Kästner. Detecting dependences and interactions in feature-oriented design. In *IEEE 21st International Symposium on Software Reliability Engineering, ISSRE 2010, San Jose, CA, USA*, pages 161–170. IEEE Computer Society, 2010. (Cited on page 17.)
- Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In Pankaj Jalote, Lionel C. Briand, and André van der Hoek, editors, *36th International Conference on Software Engineering, ICSE ’14, Hyderabad, India, 2014*, pages 1083–1094. ACM, 2014. (Cited on page 450.)
- Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008. (Cited on page 412.)
- Thomas Baar. Metamodels without metacircularities. *L’Objet*, 9(4):95–114, 2003. (Cited on page 2.)
- Thomas Baar, Bernhard Beckert, and Peter H. Schmitt. An extension of dynamic logic for modelling OCL’s @pre operator. In Dines Bjørner, Manfred Broy, and Alexandre V. Zamulin, editors, *Perspectives of System Informatics, 4th International Andrei Ershov Memorial Conference, PSI 2001, Akademgorodok, Novosibirsk, Russia, Revised Papers*, volume 2244 of *LNCS*, pages 47–54. Springer, 2001. (Cited on page 249.)
- Michael Balsler, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Andreas Thums. Formal system development with KIV. In Thomas S. E. Maibaum, editor, *Fundamental Approaches to*

- Software Engineering, Third International Conference, FASE 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany. Proceedings*, volume 1783 of *LNCIS*, pages 363–366. Springer, 2000. (Cited on pages 10, 239 and 353.)
- Anindya Banerjee, Michael Barnett, and David A. Naumann. Boogie meets regions: A verification experience report. In Natarajan Shankar and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments, Second International Conference, VSTTE 2008, Toronto, Canada. Proceedings*, volume 5295 of *LNCIS*, pages 177–191, New York, NY, 2008a. Springer. (Cited on page 350.)
- Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In Jan Vitek, editor, *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, Proceedings*, volume 5142 of *LNCIS*, pages 387–411, New York, NY, 2008b. Springer. (Cited on page 350.)
- Michael Bär. Analyse und Vergleich verifizierbarer Wahlverfahren. Diplomarbeit, Fakultät für Informatik, KIT, 2008. (Cited on page 606.)
- Michael Barnett, Robert DeLine, Manuel Fähndrich, K. Rustin M. Leino, and Wolfgang Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6): 27–56, 2004. (Cited on pages 210, 215 and 348.)
- Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, 2005, Revised Lectures*, volume 4111 of *LNCIS*, pages 364–387. Springer, 2006. (Cited on pages 10 and 216.)
- Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: an overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS), International Workshop, Marseille, France, Revised Selected Papers*, volume 3362 of *LNCIS*, pages 49–69. Springer, 2005a. (Cited on pages 241 and 348.)
- Mike Barnett, David A. Naumann, Wolfram Schulte, and Qi Sun. 99.44% pure: Useful abstractions in specification. In *ECOOP Workshop FTJJP'2004 Formal Techniques for Java-like Programs*, pages 51–60, January 2005b. (Cited on page 210.)
- Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: the Spec# experience. *Communications ACM*, 54(6): 81–91, 2011. (Cited on page 349.)
- Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010. (Cited on pages 12 and 18.)
- Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17), Pacific Grove, CA, USA*, pages 100–114, Washington, USA, 2004. IEEE CS. (Cited on page 454.)
- Gilles Barthe, Mariela Pavlova, and Gerardo Schneider. Precise analysis of memory consumption using program logics. In Bernhard K. Aichernig and Bernhard Beckert, editors, *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005), Koblenz, Germany*, pages 86–95. IEEE Computer Society, 2005. (Cited on page 230.)
- Gilles Barthe, Lilian Burdy, Julien Charles, Benjamin Grégoire, Marieke Huisman, Jean-Louis Lanet, Mariela Pavlova, and Antoine Requet. JACK: A tool for validation of security and behaviour of Java applications. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, Revised Lectures*, volume 4709 of *LNCIS*, pages 152–174, Berlin, 2007. Springer. (Cited on page 240.)
- Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the*

- 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, pages 90–101. ACM, January 2009. (Cited on page 607.)
- Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In Michael Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland. Proceedings*, volume 6664 of *LNCS*, pages 200–214. Springer, 2011. (Cited on page 483.)
- Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, César Kunz, and Mark Marron. From relational verification to SIMD loop synthesis. In Alex Nicolau, Xiaowei Shen, Saman P. Amarasinghe, and Richard W. Vuduc, editors, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '13, Shenzhen, China, 2013*, pages 123–134. ACM, 2013a. (Cited on page 5.)
- Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. Probabilistic relational reasoning for differential privacy. *ACM Transactions on Programming Languages and Systems*, 35(3):9, 2013b. (Cited on page 607.)
- Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*. CEA LIST and INRIA, 2010. Version 1.5. (Cited on page 241.)
- Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. Lessons learned from microkernel verification – specification is the new bottleneck. In Franck Cassez, Ralf Huuck, Gerwin Klein, and Bastian Schlich, editors, *Proceedings Seventh Conference on Systems Software Verification, SSV 2012, Sydney, Australia, volume 102 of EPTCS*, pages 18–32, 2012. (Cited on page 2.)
- Kent Beck. *JUnit Pocket Guide: quick lookup and advice*. O'Reilly, 2004. (Cited on pages 416 and 421.)
- Tobias Beck. Verifizierbar korrekte Implementierung von Bingo Voting. Studienarbeit, Fakultät für Informatik, KIT, March 2010. (Cited on page 606.)
- Bernhard Beckert and Daniel Bruns. Formal semantics of model fields in annotation-based specifications. In Birte Glimm and Antonio Krüger, editors, *KI 2012: Advances in Artificial Intelligence - 35th Annual German Conference on AI, Saarbrücken, Germany. Proceedings*, number 7526 in *LNCS*, pages 13–24. Springer, 2012. (Cited on page 310.)
- Bernhard Beckert and Christoph Gladisch. White-box testing by combining deduction-based specification extraction and black-box testing. In Yuri Gurevich and Bertrand Meyer, editors, *Tests and Proofs, First International Conference, TAP 2007, Zurich, Switzerland. Revised Papers*, volume 4454 of *LNCS*, pages 207–216. Springer, 2007. (Cited on page 416.)
- Bernhard Beckert and Sarah Grebing. Evaluating the usability of interactive verification systems. In Vladimir Klebanov, Bernhard Beckert, Armin Biere, and Geoff Sutcliffe, editors, *Proceedings of the 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems, Manchester, United Kingdom, 2012*, volume 873 of *CEUR Workshop Proceedings*, pages 3–17. CEUR-WS.org, 2012. (Cited on page 8.)
- Bernhard Beckert and Reiner Hähnle. Reasoning and verification. *IEEE Intelligent Systems*, 29(1): 20–29, Jan.–Feb. 2014. (Cited on pages 2, 3 and 18.)
- Bernhard Beckert and Vladimir Klebanov. Must program verification systems and calculi be verified? In *Proceedings, 3rd International Verification Workshop (VERIFY), Workshop at Federated Logic Conferences (FLoC), Seattle, USA*, pages 34–41, 2006. (Cited on page 64.)
- Bernhard Beckert and Wojciech Mostowski. A program logic for handling Java Card's transaction mechanism. In Mauro Pezzé, editor, *Proceedings, Fundamental Approaches to Software Engineering (FASE), Warsaw, Poland*, volume 2621 of *LNCS*, pages 246–260. Springer, 2003. (Cited on pages 354 and 376.)
- Bernhard Beckert and André Platzer. Dynamic logic with non-rigid functions: A basis for object-oriented program verification. In U. Furbach and N. Shankar, editors, *Proceedings, International Joint Conference on Automated Reasoning, Seattle, USA*, volume 4130 of *LNCS*, pages 266–280. Springer, 2006. (Cited on page 65.)
- Bernhard Beckert and Steffen Schlager. Software verification with integrated data type refinement for integer arithmetic. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Integrated*

- Formal Methods, 4th International Conference, IFM 2004, Canterbury, UK. Proceedings*, volume 2999 of LNCS, pages 207–226. Springer, 2004. (Cited on page 51.)
- Bernhard Beckert and Steffen Schlager. Refinement and retrenchment for programming language data types. *Formal Aspects of Computing*, 17(4):423–442, 2005. (Cited on page 51.)
- Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Andreas Roth, Philipp Rümmer, and Steffen Schlager. Taclets: a new paradigm for constructing interactive theorem provers. *Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales, Serie A: Matemáticas*, 98(1):17–53, 2004. Special Issue on Symbolic Computation in Logic and Artificial Intelligence. (Cited on page 11.)
- Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. Number 4334 in LNCS. Springer, 2007. (Cited on pages ix, 16, 230, 240, 272, 306, 376, 384, 527 and 576.)
- Bernhard Beckert, Daniel Bruns, Ralf Küsters, Christoph Scheben, Peter H. Schmitt, and Tomasz Truderung. The KeY approach for the cryptographic verification of Java programs: A case study. Technical Report 2012-8, Department of Informatics, Karlsruhe Institute of Technology, 2012. (Cited on page 594.)
- Bernhard Beckert, Thorsten Bormer, and Markus Wagner. A metric for testing program verification systems. In Margus Veanes and Luca Viganò, editors, *Tests and Proofs. Seventh International Conference, TAP 2013, Budapest, Hungary*, volume 7942 of LNCS, pages 56–75. Springer, 2013. (Cited on page 65.)
- Bernhard Beckert, Daniel Bruns, Vladimir Klebanov, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. Information flow in object-oriented software. In Gopal Gupta and Ricardo Peña, editors, *Logic-Based Program Synthesis and Transformation, 23rd International Symposium, LOPSTR 2013, Madrid, Spain, Revised Selected Papers*, number 8901 in LNCS, pages 19–37. Springer, 2014. (Cited on page 460.)
- Bernhard Beckert, Vladimir Klebanov, and Mattias Ulbrich. Regression verification for Java using a secure information flow calculus. In Rosemary Monahan, editor, *Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs, FTJFP 2015, Prague, Czech Republic*, pages 6:1–6:6. ACM, 2015. (Cited on page 428.)
- Jesper Bengtson, Jonas Braband Jensen, Filip Sieczkowski, and Lars Birkedal. Verifying object-oriented programs with higher-order separation logic in coq. In Marko van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving: Second International Conference, ITP 2011, Berg en Dal, The Netherlands. Proceedings*, pages 22–38. Springer, 2011. (Cited on page 316.)
- Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy*, pages 14–25. ACM, 2004. (Cited on pages 5, 483 and 607.)
- Dirk Beyer. Software verification and verifiable witnesses — (report on SV-COMP 2015). In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK. Proceedings*, volume 9035 of LNCS, pages 401–416. Springer, 2015. (Cited on pages 4 and 18.)
- Joshua Bloch. *Effective Java: Programming Language Guide*. The Java Series. Addison-Wesley, 2nd edition, 2008. (Cited on page 261.)
- Arjan Blom, Gerhard de Koning Gans, Erik Poll, Joeri de Ruiter, and Roel Verdult. Designed to fail: A USB-connected reader for online banking. In Audun Jøsang and Bengt Carlsson, editors, *Secure IT Systems - 17th Nordic Conference, NordSec 2012, Karlskrona, Sweden. Proceedings*, volume 7617 of LNCS, pages 1–16. Springer, 2012. (Cited on page 353.)
- Stefan Blom and Marieke Huisman. The VerCors Tool for verification of concurrent programs. In Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods - 19th International Symposium, Singapore. Proceedings*, volume 8442 of LNCS, pages 127–131. Springer, 2014. (Cited on page 378.)

- Jens-Matthias Bohli, Christian Henrich, Carmen Kempka, Jörn Müller-Quade, and Stefan Röhrich. Enhancing electronic voting machines on the example of Bingo voting. *IEEE Transactions on Information Forensics and Security*, 4(4):745–750, 2009. (Cited on page 606.)
- Greg Bollella and James Gosling. The real-time specification for Java. *IEEE Computer*, pages 47–54, June 2000. (Cited on page 5.)
- Alex Borgida, John Mylopoulos, and Raymond Reiter. “. . . And nothing else changes”: On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, 1995. (Cited on pages 233 and 321.)
- Bernard Botella, Mickaël Delahaye, Stéphane Hong Tuan Ha, Nikolai Kosmatov, Patricia Mouy, Muriel Roger, and Nicky Williams. Automating structural testing of C programs: Experience with PathCrawler. In Dimitris Dranidis, Stephen P. Masticola, and Paul A. Strooper, editors, *Proceedings of the 4th International Workshop on Automation of Software Test, AST 2009, Vancouver, BC, Canada*, pages 70–78. IEEE Computer Society, May 2009. (Cited on page 449.)
- Raymond T. Boute. Calculational semantics: Deriving programming theories from equations by functional predicate calculus. *ACM Transactions on Programming Languages and Systems*, 28(4):747–793, 2006. (Cited on page 574.)
- Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT—A formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices*, 10(6):234–245, June 1975. (Cited on page 383.)
- John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA. Proceedings*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003. (Cited on pages 378 and 379.)
- Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer, 2007. (Cited on page 537.)
- Cees-Bart Breunesse and Erik Poll. Verifying JML specifications with model fields. In *ECOOP workshop on Formal Techniques for Java-like Programs (FTJLP’03)*, Darmstadt, number 408 in Technical Report, ETH Zürich, pages 51–60, July 2003. (Cited on page 350.)
- Cees-Bart Breunesse, Néstor Cataño, Marieke Huisman, and Bart Jacobs. Formal methods for smart cards: an experience report. *Science of Computer Programming*, 55:53–80, 2005. (Cited on page 226.)
- Daniel Bruns. Elektronische Wahlen: Theoretisch möglich, praktisch undemokratisch. *FifF-Kommunikation*, 25(3):33–35, September 2008. (Cited on page 594.)
- Daniel Bruns. Formal semantics for the Java Modeling Language. Diploma thesis, Universität Karlsruhe, 2009. (Cited on pages 195, 215, 243, 245 and 350.)
- Daniel Bruns. Specification of red-black trees: Showcasing dynamic frames, model fields and sequences. In Wolfgang Ahrendt and Richard Bubel, editors, *10th KeY Symposium*, Nijmegen, the Netherlands, 2011. Extended Abstract. (Cited on page 296.)
- Richard Bubel. *Formal Verification of Recursive Predicates*. PhD thesis, Universität Karlsruhe, 2007. (Cited on page 306.)
- Richard Bubel, Andreas Roth, and Philipp Rümmer. Ensuring the correctness of lightweight tactics for Java Card dynamic logic. *Electronic Notes in Theoretical Computer Science*, 199:107–128, 2008. (Cited on pages 138 and 144.)
- Richard Bubel, Reiner Hähnle, and Benjamin Weiß. Abstract interpretation of symbolic execution with explicit state updates. In Frank S. de Boer, Marcello M. Bonsangue, and Eric Madeleine, editors, *Formal Methods for Components and Objects, 7th International Symposium, FMCO 2008, Sophia Antipolis, France, Revised Lectures*, volume 5751 of *LNCS*, pages 247–277. Springer, 2009. (Cited on pages x, 171, 454, 471 and 474.)
- Richard Bubel, Reiner Hähnle, and Ulrich Geilmann. A formalisation of Java strings for program specification and verification. In Gilles Barthe and Gerardo Schneider, editors, *Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay. Proceedings*, volume 7041 of *LNCS*, pages 90–105. Springer, 2011. (Cited on page x.)
- Richard Bubel, Antonio Flores Montoya, and Reiner Hähnle. Analysis of executable software models. In Marco Bernardo, Ferruccio Damiani, Reiner Hähnle, Einar B. Johnsen, and Ina Schaefer, editors, *Executable Software Models: 14th International School on Formal Methods*

- for the Design of Computer, Communication, and Software Systems, Bertinoro, Italy*, volume 8483 of *LNCS*, pages 1–27. Springer, June 2014a. (Cited on page 16.)
- Richard Bubel, Reiner Hähnle, and Maria Pelevina. Fully abstract operation contracts. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation, 6th International Symposium, ISOFA 2014, Corfu, Greece*, volume 8803 of *LNCS*, pages 120–134. Springer, October 2014b. (Cited on page 9.)
- Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. In Thomas Arts and Wan Fokkink, editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03), Proceedings*, volume 80 of *Electronic Notes in Theoretical Computer Science*, pages 73–89. Elsevier, 2003a. (Cited on page 239.)
- Lilian Burdy, Antoine Requet, and Jean-Louis Lanet. Java applet correctness: A developer-oriented approach. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy. Proceedings*, volume 2805 of *LNCS*, pages 422–439. Springer, 2003b. (Cited on page 353.)
- Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), L'Aquila, Italy*, pages 443–446. IEEE Computer Society, 2008. (Cited on page 450.)
- Rod M. Burstall. Program proving as hand simulation with a little induction. In *IFIP Congress '74, Stockholm*, pages 308–312. Elsevier/North-Holland, 1974. (Cited on pages 12 and 383.)
- Cristian Cadar, Daniel Dunbar, and Dawson R Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, San Diego, CA, USA, Proceedings*, pages 209–224. USENIX Association, 2008a. (Cited on page 450.)
- Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2), 2008b. (Cited on page 450.)
- Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In Richard N. Taylor, Harald Gall, and Nenad Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA*, pages 1066–1071. ACM, 2011. (Cited on page 449.)
- Néstor Cataño, Tim Wahls, Camilo Rueda, Víctor Rivera, and Danni Yu. Translating B machines to JML specifications. In Sascha Ossowski and Paola Lecca, editors, *Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy*, pages 1271–1277. New York, NY, USA, 2012. ACM. (Cited on page 240.)
- Néstor Cataño and Marieke Huisman. CHASE: A static checker for JML's assignable clause. In Lenore D. Zuck, Paul C. Attie, Agostino Cortesi, and Supratik Mukhopadhyay, editors, *Verification, Model Checking, and Abstract Interpretation, 4th International Conference, VMCAI 2003, New York, NY, USA. Proceedings*, volume 2575 of *LNCS*, pages 26–40. Springer, 2003. (Cited on page 240.)
- Patrice Chalin. Improving JML: For a safer and more effective language. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy. Proceedings*, volume 2805 of *LNCS*, pages 440–461. Springer, 2003. (Cited on page 231.)
- Patrice Chalin. JML support for primitive arbitrary precision numeric types: Definition and semantics. *Journal of Object Technology*, 3(6):57–79, June 2004. Special issue: ECOOP 2003 Workshop on FTfJP. (Cited on page 232.)
- Patrice Chalin. A sound assertion semantics for the dependable systems evolution verifying compiler. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA*, pages 23–33. IEEE Computer Society, 2007. (Cited on page 286.)
- Patrice Chalin and Frédéric Rioux. Non-null references by default in the Java modeling language. *SIGSOFT Software Engineering Notes*, 31(2), September 2005. (Cited on page 246.)

- Patrice Chalin and Frédéric Rioux. JML runtime assertion checking: Improved error reporting and efficiency using strong validity. In Jorge Cuellar, Tom Maibaum, and Kaisa Sere, editors, *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland. Proceedings*, volume 5014 of *LNCS*, pages 246–261. Springer, 2008. (Cited on page 286.)
- Patrice Chalin, Perry R. James, and Frédéric Rioux. Reducing the use of nullable types through non-null by default and monotonic non-null. *Software, IET*, 2(6):515–531, 2008. (Cited on page 246.)
- Patrice Chalin, Robby, Perry R. James, Jooyong Lee, and George Karabotsos. Towards an industrial grade IVE for Java and next generation research platform for JML. *STTT*, 12(6):429–446, 2010. (Cited on page 239.)
- Crystal Chang Din, Richard Bubel, and Reiner Hähnle. KeY-ABS: A deductive verification tool for the concurrent modelling language ABS. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany. Proceedings*, volume 9195 of *LNCS*, pages 517–526. Springer, 2015. (Cited on page 6.)
- David Chaum, Richard T. Carback, Jeremy Clark, Aleksander Essex, Stefan Popoveniuc, Ronald L. Rivest, Peter Y. A. Ryan, Emily (Emily Huei-Yi) Shen, Alan T. Sherman, and Poorvi L. Vora. Scantegrity II: End-to-end verifiability by voters of optical scan elections through confirmation codes. *IEEE Transactions on Information Forensics and Security*, October 2009. (Cited on page 606.)
- Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley, June 2000. (Cited on pages 353 and 354.)
- Yoonsik Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Department of Computer Science, Iowa State University, Ames, 2003. Technical Report 03-09. (Cited on page 239.)
- Yoonsik Cheon. Automated random testing to detect specification-code inconsistencies. In Dimitris A. Karras, Daming Wei, and Jaroslav Zundulka, editors, *International Conference on Software Engineering Theory and Practice, SETP-07, Orlando, Florida, USA*, pages 112–119. ISRST, 2007. (Cited on page 239.)
- Yoonsik Cheon and Gary T. Leavens. A quick overview of Larch/C++. *Journal of Object-oriented Programming*, 7(6):39–49, 1994. (Cited on page 240.)
- Dave Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In Bjørn N. Freeman-Benson and Craig Chambers, editors, *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98), Vancouver, British Columbia, Canada*, pages 48–64, Vancouver, Canada, October 1998. ACM. (Cited on pages 13 and 348.)
- Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999. (Cited on page 6.)
- Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: Toward a secure voting system. In *2008 IEEE Symposium on Security and Privacy (S&P 2008), Oakland, California, USA*, pages 354–368. IEEE Computer Society, 2008. (Cited on pages 594 and 606.)
- Ellis S. Cohen. Information transmission in computational systems. In Saul Rosen and Peter J. Denning, editors, *Proceedings of the Sixth Symposium on Operating System Principles, SOSP 1977, Purdue University, West Lafayette, Indiana, USA*, pages 133–139. ACM, 1977. (Cited on page 454.)
- Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674 of *LNCS*, pages 23–42, Berlin, August 2009. Springer. (Cited on pages 241 and 349.)
- David R. Cok. Reasoning with specifications containing method calls and model fields. *Journal of Object Technology*, 4(8):77–103, 2005. (Cited on page 350.)

- David R. Cok. Adapting JML to generic types and Java 1.6. In *Seventh International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008)*, number CS-TR-08-07 in Technical Report, pages 27–35, 2008. (Cited on pages 195 and 237.)
- David R. Cok. OpenJML: JML for Java 7 by extending OpenJDK. In Mihaela Bobaru, Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA. Proceedings*, volume 6617 of LNCS, pages 472–479. Springer, Berlin, 2011. (Cited on pages 239 and 426.)
- David R. Cok and Joseph Kiniry. ESC/Java2: Uniting ESC/Java and JML. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Post Conference Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille*, volume 3362 of LNCS, pages 108–128. Springer, 2005. (Cited on pages 195 and 240.)
- David R. Cok and Gary T. Leavens. Extensions of the theory of observational purity and a practical design for JML. In *Seventh International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008)*, number CS-TR-08-07 in Technical Report, pages 43–50, 4000 Central Florida Blvd., Orlando, Florida, 32816-2362, 2008. School of EECS, UCF. (Cited on page 210.)
- Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal of Computing*, 7(1):70–90, 1978. (Cited on page 65.)
- Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '77*, pages 238–252, New York, NY, USA, 1977. ACM. (Cited on pages 167 and 168.)
- Lajos Cseppento and Zoltán Micskei. Evaluating symbolic execution-based test tools. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE Computer Society, April 2015. (Cited on page 449.)
- Marcello D’Agostino, Dov Gabbay, Reiner Hähnle, and Joachim Posegga, editors. *Handbook of Tableau Methods*. Kluwer, Dordrecht, 1999. (Cited on page 11.)
- Ádám Darvas and Rustin Leino. Practical reasoning about invocations and implementations of pure methods. In Matthew B. Dwyer and Antónia Lopes, editors, *Fundamental Approaches to Software Engineering, 10th International Conference, FASE 2007, Held as Part of the Joint European Conferences, on Theory and Practice of Software, ETAPS 2007, Braga, Portugal. Proceedings*, volume 4422 of LNCS, pages 336–351. Springer, 2007. (Cited on page 210.)
- Ádám Darvas and Peter Müller. Reasoning about method calls in interface specifications. *Journal of Object Technology*, 5(5):59–85, 2006. (Cited on page 210.)
- Ádám Darvas and Peter Müller. Formal encoding of JML Level 0 specifications in JIVE. Technical Report 559, ETH Zurich, 2007. (Cited on pages 195 and 243.)
- Ádám Darvas, Reiner Hähnle, and Dave Sands. A theorem proving approach to analysis of secure information flow. In Roberto Gorrieri, editor, *Workshop on Issues in the Theory of Security, WITS. IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS*, 2003. (Cited on pages 5 and 278.)
- Ádám Darvas, Reiner Hähnle, and David Sands. A theorem proving approach to analysis of secure information flow. In Dieter Hutter and Markus Ullmann, editors, *Security in Pervasive Computing, Second International Conference, SPC 2005, Boppard, Germany. Proceedings*, volume 3450 of LNCS, pages 193–209. Springer, 2005. (Cited on pages x, 5, 278 and 454.)
- Ádám Darvas, Farhad Mehta, and Arsenii Rudich. Efficient well-definedness checking. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia. Proceedings*, LNCS, pages 100–115, Berlin, Heidelberg, 2008. Springer. (Cited on page 286.)
- Stijn de Gouw, Frank S. de Boer, and Jurriaan Rot. Proof pearl: The key to correct and stable sorting. *J. Automated Reasoning*, 53(2):129–139, 2014. (Cited on page 609.)
- Stijn De Gouw, Jurriaan Rot, Frank S. De Boer, Richard Bubel, and Reiner Hähnle. OpenJDK’s java.util.collection.sort() is broken: The good, the bad and the worst case. In Daniel Kroening and Corina Pasareanu, editors, *Computer Aided Verification - 27th International Conference*,

- CAV 2015, San Francisco, CA, USA. *Proceedings, Part I*, volume 9206 of *LNCS*, pages 273–289. Springer, July 2015. (Cited on page 9.)
- Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976. (Cited on page 454.)
- Krishna Kishore Dhara and Gary T. Leavens. Weak behavioral subtyping for types with mutable objects. *Electronic Notes in Theoretical Computer Science*, 1:91–113, 1995. This issue contains revised papers presented at the Eleventh Annual Conference on Mathematical Foundations of Programming Semantics, (MFPS XI), Tulane University, New Orleans, 1995. Managing editors: Michael Mislove and Maurice Nivat and Christos Papadimitriou. (Cited on page 219.)
- Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. (Cited on pages 571 and 577.)
- Crystal Chang Din. *Verification Of Asynchronously Communicating Objects*. PhD thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, March 2014. (Cited on page 6.)
- Dino Distefano and Matthew J. Parkinson. jStar: towards practical verification for Java. In Gail E. Harris, editor, *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 213–226, New York, NY, 2008. ACM. (Cited on page 316.)
- Huy Q. Do, Richard Bubel, and Reiner Hähnle. Exploit generation for information flow leaks in object-oriented programs. In Hannes Federath and Dieter Gollmann, editors, *ICT Systems Security and Privacy Protection - 30th IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany. Proceedings*, volume 455 of *LNCS*, pages 401–415. Springer, 2015. (Cited on page 17.)
- Quoc Huy Do, Eduard Kamburjan, and Nathan Wasser. Towards fully automatic logic-based information flow analysis: An electronic-voting case study. In Frank Piessens and Luca Viganò, editors, *Principles of Security and Trust, 5th Intl. Conf., POST, Eindhoven, The Netherlands*, volume 9635 of *LNCS*, pages 97–115. Springer, 2016. (Cited on pages x, 5 and 189.)
- Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983. (Cited on page 595.)
- Felix Dörre and Vladimir Klebanov. Pseudo-random number generator verification: A case study. In Arie Gurfinkel and Sanjit A. Seshia, editors, *Proceedings, Verified Software: Theories, Tools, and Experiments (VSTTE)*, volume 9593 of *LNCS*. Springer, 2015. (Cited on page 455.)
- Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The Coq proof assistant user’s guide. Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8. (Cited on pages 2 and 108.)
- Christian Engel. A translation from JML to JavaDL. Studienarbeit, Fakultät für Informatik, Universität Karlsruhe, February 2005. (Cited on pages 195 and 243.)
- Christian Engel and Reiner Hähnle. Generating unit tests from formal proofs. In Bertrand Meyer and Yuri Gurevich, editors, *Tests and Proofs, First International Conference, TAP 2007, Zurich, Switzerland. Revised Papers*, volume 4454 of *LNCS*. Springer, 2007. (Cited on pages x, 4 and 416.)
- Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007. (Cited on page 240.)
- Michael E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976. (Cited on pages 18 and 413.)
- Azadeh Farzan, Feng Chen, José Meseguer, and Grigore Roşu. Formal analysis of Java programs in JavaFAN. In Rajeev Alur and Doron A. Peled, editors, *Proceedings, 16th International Conference on Computer Aided Verification (CAV)*, volume 3114 of *LNCS*, pages 501–505. Springer, 2004. (Cited on page 64.)
- Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. Automating regression verification. In *29th IEEE/ACM International Conference on Automated Software Engineering (ASE 2014)*, ASE ’14, pages 349–360. ACM, 2014. (Cited on pages 17 and 483.)

- Jean-Christophe Filliâtre and Nicolas Magaud. Certification of sorting algorithms in the system Coq. In *Theorem Proving in Higher Order Logics: Emerging Trends*, Nice, France, 1999. (Cited on page 609.)
- Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spririt of ghost code. In Armin Biere, Swen Jacobs, and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria. Proceedings*, volume 8559 of *LNCS*, pages 1–16. Springer, 2014. (Cited on page 269.)
- John S. Fitzgerald, Peter Gorm Larsen, and Marcel Verhoef. Vienna development method. In Benjamin W. Wah, editor, *Wiley Encyclopedia of Computer Science and Engineering*. John Wiley & Sons, Inc., 2008. (Cited on page 240.)
- Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. Technical Report 2000-003, DEC-SRC, December 2000. (Cited on page 240.)
- Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society. (Cited on pages 194 and 234.)
- M. Foley and C. A. R. Hoare. Proof of a recursive program: Quicksort. *Computer Journal*, 14(4): 391–395, 1971. (Cited on page 609.)
- Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völpl, and André Platzer. Keymaera X: an axiomatic tactical theorem prover for hybrid systems. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany. Proceedings*, volume 9195 of *LNCS*, pages 527–538. Springer, 2015. (Cited on page 6.)
- Stefan J. Galler and Bernhard K. Aichernig. Survey on test data generation tools. *International Journal on Software Tools for Technology Transfer*, 16(6):727–751, 2014. (Cited on page 448.)
- Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Wiley, 1987. (Cited on pages 27 and 35.)
- Flavio D. Garcia, Gerhard Koning Gans, Ruben Muijers, Peter Rossum, Roel Verdult, Ronny Wichers Schreur, and Bart Jacobs. Dismantling MIFARE classic. In Sushil Jajodia and Javier Lopez, editors, *Proceedings of the 13th European Symposium on Research in Computer Security*, volume 5283 of *LNCS*, pages 97–114. Springer, 2008. (Cited on page 353.)
- Tobias Gedell and Reiner Hähnle. Automating verification of loops by parallelization. In Miki Hermann, editor, *Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006, Phnom Penh, Cambodia. Proceedings*, *LNCS*, pages 332–346. Springer, October 2006. (Cited on page 68.)
- Ullrich Geilmann. Formal verification using Java’s String class. Studienarbeit, Chalmers University of Technology and Universität Karlsruhe, November 2009. (Cited on page 161.)
- Robert Geisler, Marcus Klar, and Felix Cornelius. InterACT: An interactive theorem prover for algebraic specifications. In Martin Wirsing and Maurice Nivat, editors, *Algebraic Methodology and Software Technology, 5th International Conference, AMAST '96, Munich, Germany. Proceedings*, volume 1101 of *LNCS*, pages 563–566. Springer, 1996. (Cited on page 108.)
- Steven M. German and Ben Wegbreit. A synthesizer of inductive assertions. *IEEE Transactions on Software Engineering*, SE-1(1):68–75, March 1975. (Cited on page 234.)
- Martin Giese. Taclets and the KeY prover. In David Aspinall and Christoph Lüth, editors, *Proc. User Interfaces for Theorem Provers Workshop, UITP, Rome, 2003*, volume 103 of *Electronic Notes in Theoretical Computer Science*, pages 67–79. Elsevier, 2004. (Cited on page 108.)
- Martin Giese. A calculus for type predicates and type coercion. In Bernhard Beckert, editor, *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX 2005, Koblenz, Germany. Proceedings*, volume 3702 of *LNCS*, pages 123–137. Springer, 2005. (Cited on page 35.)
- Christoph Gladisch. Verification-based test case generation for full feasible branch coverage. In Antonio Cerone and Stefan Gruner, editors, *Proceedings, Sixth IEEE International Conference*

- on *Software Engineering and Formal Methods, SEFM 2008, Cape Town, South Africa*, pages 159–168. IEEE Computer Society, 2008. (Cited on pages x, 434 and 436.)
- Christoph Gladisch and Shmuel Tyszberowicz. Specifying a linked data structure in JML for formal verification and runtime checking. In Leonardo de Moura and Juliano Iyoda, editors, *Formal Methods: Foundations and Applications - 16th Brazilian Symposium, SBMF 2013, Brasilia, Brazil. Proceedings*, volume 8195 of *LNCS*, pages 99–114. Springer, 2013. (Cited on pages 296 and 300.)
- Christoph David Gladisch. *Verification-based software-fault detection*. PhD thesis, Karlsruhe Institute of Technology, 2011. (Cited on pages 416 and 436.)
- Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012. (Cited on page 450.)
- Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931. (Cited on page 65.)
- Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982. (Cited on page 454.)
- Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *LNCS*. Springer, 1979. (Cited on page 10.)
- James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 7 Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2013. (Cited on pages 52, 53, 54, 55, 60, 91, 156, 197, 237, 247, 622 and 623.)
- Jürgen Graf, Martin Hecker, and Martin Mohr. Using JOANA for information flow control in Java programs – A practical guide. In Stefan Wagner and Horst Lichter, editors, *Software Engineering (Workshops)*, volume 215 of *Lecture Notes in Informatics*, pages 123–138. Gesellschaft für Informatik, 2013. (Cited on pages 596 and 605.)
- Daniel Grahl. *Deductive Verification of Concurrent Programs and its Application to Secure Information Flow for Java*. PhD thesis, Karlsruhe Institute of Technology, 29 October 2015. (Cited on pages x, 351, 593 and 596.)
- Jim Gray. Why Do Computers Stop and What Can Be Done About It? Technical Report 85.7, PN87614, Tandem Computers, June 1985. (Cited on page 384.)
- Wolfgang Grieskamp, Nikolai Tillmann, and Wolfram Schulte. XRT — exploring runtime for .NET architecture and applications. *Electronic Notes in Theoretical Computer Science*, 144(3):3–26, 2006. Proceedings of the Workshop on Software Model Checking (SoftMC 2005), Software Model Checking, Edinburgh, UK, 2005. (Cited on page 384.)
- John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer, 1993. (Cited on page 194.)
- Elmar Habermalz. Interactive theorem proving with schematic theory specific rules. Technical Report 19/00, Fakultät für Informatik, Universität Karlsruhe, 2000a. (Cited on page 108.)
- Elmar Habermalz. *Ein dynamisches automatisierbares interaktives Kalkül für schematische theorienspezifische Regeln*. PhD thesis, Universität Karlsruhe, 2000b. (Cited on page 108.)
- Reiner Hähnle. Many-valued logic, partiality, and abstraction in formal specification languages. *Logic Journal of the IPGL*, 13(4):415–433, July 2005. (Cited on page 280.)
- Reiner Hähnle and Richard Bubel. A Hoare-style calculus with explicit state updates. In Zoltán Isteneş, editor, *Proc. Formal Methods in Computer Science Education (FORMED)*, Electronic Notes in Theoretical Computer Science, pages 49–60. Elsevier, 2008. (Cited on pages x, 7, 15 and 572.)
- Reiner Hähnle, Wolfram Menzel, and Peter Schmitt. Integrierter deduktiver Software-Entwurf. *Künstliche Intelligenz*, pages 40–41, December 1998. (Cited on page 1.)
- Reiner Hähnle, Markus Baum, Richard Bubel, and Marcel Rothe. A visual interactive debugger based on symbolic execution. In Jamie Andrews and Elisabetta Di Nitto, editors, *Proc. 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium*, pages 143–146. ACM Press, 2010. (Cited on pages 8, 384 and 412.)
- Reiner Hähnle, Nathan Wasser, and Richard Bubel. Array abstraction with symbolic pivots. In Erika Ábrahám, Marcello Bonsangue, and Broch Einar Johnsen, editors, *Theory and Practice*

- of Formal Methods: Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, pages 104–121. Springer, 2016. (Cited on pages 184 and 187.)
- Christian Hammer. *Information Flow Control for Java – A Comprehensive Approach based on Path Conditions in Dependence Graphs*. PhD thesis, Universität Karlsruhe (TH), July 2009. (Cited on pages 596 and 605.)
- Christian Hammer, Jens Krinke, and Gregor Snelting. Information flow control for Java based on path conditions in dependence graphs. In *IEEE International Symposium on Secure Software Engineering (ISSSE 2006)*, pages 87–96. IEEE, March 2006. (Cited on page 454.)
- David Harel. *First-Order Dynamic Logic*. Springer, 1979. (Cited on page 65.)
- David Harel. Dynamic logic. In D. Gabbay and F. Guentner, editors, *Handbook of Philosophical Logic*, volume II: Extensions of Classical Logic, chapter 10, pages 497–604. Reidel, Dordrecht, 1984. (Cited on page 49.)
- David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, 2000. (Cited on pages 12, 49 and 330.)
- Trevor Harmon and Raymond Klefstad. A survey of worst-case execution time analysis for real-time Java. In *21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings, Long Beach, California, USA*, pages 1–8. IEEE Press, 2007. (Cited on page 582.)
- Maritta Heisel, Wolfgang Reif, and Werner Stephan. Program verification by symbolic execution and induction. In Katharina Morik, editor, *GWAI-87, 11th German Workshop on Artificial Intelligence, Geseke, 1987, Proceedings*, volume 152 of *Informatik Fachberichte*, pages 201–210. Springer, 1987. (Cited on page 12.)
- Martin Hentschel, Richard Bubel, and Reiner Hähnle. Symbolic execution debugger (SED). In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification, 14th International Conference, RV, Toronto, Canada*, volume 8734 of *LNCS*, pages 255–262. Springer, 2014a. (Cited on pages x, 8 and 384.)
- Martin Hentschel, Reiner Hähnle, and Richard Bubel. Visualizing unbounded symbolic execution. In Martina Seidl and Nikolai Tillmann, editors, *Proceedings of Testing and Proofs (TAP) 2014*, *LNCS*, pages 82–98. Springer, July 2014b. (Cited on pages x and 386.)
- Martin Hentschel, Stefan Käsdorf, Reiner Hähnle, and Richard Bubel. An interactive verification tool meets an IDE. In Emil Sekerinski Elvira Albert and Gianluigi Zavattaro, editors, *Proceedings of the 11th International Conference on Integrated Formal Methods*, volume 8739 of *LNCS*, pages 55–70. Springer, 2014c. (Cited on pages x and 566.)
- Martin Hentschel, Reiner Hähnle, and Richard Bubel. Can formal methods improve the efficiency of code reviews? In Erika Ábrahám and Marieke Huisman, editors, *Integrated Formal Methods, 12th International Conference, IFM, Reykjavik, Iceland*, volume 9681 of *LNCS*, pages 3–19. Springer, 2016. (Cited on pages 8 and 18.)
- Mihai Herda. Generating bounded counterexamples for KeY proof obligations. Master thesis, Karlsruhe Institute of Technology, January 2014. (Cited on page 439.)
- C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969. (Cited on pages 7, 208, 234, 349, 571 and 574.)
- C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In Erwin Engeler, editor, *Symposium on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 102–116. Springer, Berlin, Heidelberg, 1971. (Cited on page 299.)
- C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972. (Cited on page 302.)
- C.A.R. Hoare and Jayadev Misra. Verified software: Theories, tools, experiments vision of a grand challenge project. In Bertrand Meyer and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, Revised Selected Papers and Discussions*, volume 4171 of *LNCS*, pages 1–18. Springer, 2005. (Cited on page 289.)
- Gerard J. Holzmann. *The SPIN Model Checker*. Pearson Education, 2003. (Cited on pages 6 and 7.)
- Falk Howar, Dimitra Giannakopoulou, and Zvonimir Rakamaric. Hybrid learning: interface generation through static, dynamic, and symbolic analysis. In Mauro Pezzè and Mark Harman, editors,

- International Symposium on Software Testing and Analysis, ISSA, Lugano, Switzerland*, pages 268–279. ACM, 2013. (Cited on page 18.)
- Engelbert Hubbers and Erik Poll. Reasoning about card tears and transactions in Java Card. In Michel Wermelinger and Tiziana Margaria, editors, *Proc. Fundamental Approaches to Software Engineering (FASE), Barcelona, Spain*, volume 2984 of *LNCS*, pages 114–128. Springer, 2004. (Cited on page 377.)
- Engelbert Hubbers, Wojciech Mostowski, and Erik Poll. Tearing Java Cards. In *Proceedings, e-Smart 2006, Sophia-Antipolis, France*, 2006. (Cited on page 374.)
- Marieke Huisman and Wojciech Mostowski. A symbolic approach to permission accounting for concurrent reasoning. In *14th International Symposium on Parallel and Distributed Computing (ISPD 2015)*, pages 165–174. IEEE Computer Society, 2015. (Cited on pages 378 and 380.)
- Marieke Huisman, Wolfgang Ahrendt, Daniel Bruns, and Martin Hentschel. Formal specification with JML. Technical Report 2014-10, Department of Informatics, Karlsruhe Institute of Technology, 2014. (Cited on page 193.)
- Marieke Huisman, Vladimir Klebanov, and Rosemary Monahan. VerifyThis 2012. *International Journal on Software Tools for Technology Transfer*, 17(6):647–657, 2015. (Cited on page 289.)
- James J. Hunt, Fridtjof B. Siebert, Peter H. Schmitt, and Isabel Tonin. Provably correct loops bounds for realtime Java programs. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 162–169, New York, NY, USA, 2006. ACM. (Cited on page 583.)
- Michael Huth and Mark Dermot Ryan. *Logic in computer science - modelling and reasoning about systems (2. ed.)*. Cambridge University Press, 2004. (Cited on page 572.)
- Malte Isberner, Falk Howar, and Bernhard Steffen. Learning register automata: from languages to program structures. *Machine Learning*, 96(1–2):65–98, 2014. (Cited on page 18.)
- ISO. ISO 26262, road vehicles – functional safety. published by the International Organization for Standardization, 2011. (Cited on page 424.)
- Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions Software Engineering and Methodology*, 11(2):256–290, April 2002. (Cited on page 438.)
- Daniel Jackson. Alloy: A logical modelling language. In Didier Bert, Jonathan P. Bowen, Steve King, and Marina A. Waldén, editors, *ZB 2003: Formal Specification and Development in Z and B, Third International Conference of B and Z Users, Turku, Finland. Proceedings*, volume 2651 of *LNCS*, page 1. Springer, 2003. (Cited on page 240.)
- Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, August 2008. (Cited on pages 2 and 384.)
- Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA*, pages 271–282. ACM, 2011. (Cited on page 241.)
- Bart Jacobs and Erik Poll. A logic for the Java Modeling Language. In Heinrich Hußmann, editor, *Proc. Fundamental Approaches to Software Engineering, 4th International Conference (FASE), Genova, Italy*, volume 2029 of *LNCS*, pages 284–299. Springer, 2001. (Cited on pages 195 and 243.)
- Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of the European Association for Theoretical Computer Science*, 62:222–259, 1997. (Cited on page 252.)
- Bart Jacobs, Hans Meijer, and Erik Poll. VerifiCard: A European project for smart card verification. *Newsletter 5 of the Dutch Association for Theoretical Computer Science (NVTI)*, 2001. (Cited on page 353.)
- Bart Jacobs, Joseph Kiniry, and Martijn Warnier. Java program verification challenges. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, volume 2852 of *LNCS*, pages 202–219. Springer, 2003. (Cited on page 88.)

- Bart Jacobs, Jan Smans, Pieter Philippaerts, and Frank Piessens. The VeriFast program verifier – a tutorial for Java Card developers. Technical report, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, September 2011a. (Cited on page 377.)
- Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA. Proceedings*, volume 6617 of *LNCS*, pages 41–55. Springer, 2011b. (Cited on pages 377 and 378.)
- Bart Jacobs, Jan Smans, and Frank Piessens. Verification of unloadable modules. In Michael Butler and Wolfram Schulte, editors, *17th International Symposium on Formal Methods (FM 2011)*, pages 402–416. Springer, June 2011c. (Cited on page 350.)
- JavaCardRTE. *Java Card 3 Platform Runtime Environment Specification, Classic Edition, Version 3.0.4*, Oracle, September 2012. (Cited on pages 5, 353 and 354.)
- JavaCardVM. *Java Card 3 Platform Virtual Machine Specification, Classic Edition, Version 3.0.4*, Oracle, September 2012. (Cited on page 354.)
- Trevor Jennings. SPARK: the libre language and toolset for high-assurance software engineering. In Greg Gicca and Jeff Boleng, editors, *Proceedings, Annual ACM SIGAda International Conference on Ada, Saint Petersburg, Florida, USA*, pages 9–10. ACM, 2009. (Cited on page 5.)
- Ran Ji. *Sound program transformation based on symbolic execution and deduction*. PhD thesis, Darmstadt University of Technology, Department of Computer Science, 2014. (Cited on pages x, 482, 491 and 492.)
- Ran Ji and Reiner Hähnle. Information flow analysis based on program simplification. Technical Report TUD-CS-2014-0877, Department of Computer Science, 2014. (Cited on page 492.)
- Ran Ji, Reiner Hähnle, and Richard Bubel. Program transformation based on symbolic execution and deduction. In Robert M. Hierons, Mercedes G. Merayo, and Mario Bravetti, editors, *Software Engineering and Formal Methods: 11th International Conference, SEFM 2013, Madrid, Spain*, volume 8137 of *LNCS*, pages 289–304. Springer, 2013. (Cited on pages x and 5.)
- Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Proceedings, 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *LNCS*, pages 142–164. Springer, 2011. (Cited on page 6.)
- Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983. (Cited on page 351.)
- Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, 1993. (Cited on page 475.)
- Kari Kähkönen, Tuomas Launiainen, Olli Saarikivi, Janne Kauttio, Keijo Heljanko, and Ilkka Niemelä. LCT: An open source concolic testing tool for Java programs. In Pierre Ganty and Mark Marron, editors, *Proceedings of the 6th Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'2011)*, pages 75–80, 2011. (Cited on page 450.)
- Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6(2):133–151, 1976. (Cited on page 234.)
- Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada. Proceedings*, volume 4085 of *LNCS*, pages 268–283. Berlin, 2006. Springer. (Cited on pages 13, 320 and 322.)
- Ioannis T. Kassios. The dynamic frames theory. *Formal Aspects Computing*, 23(3):267–288, 2011. (Cited on pages ix, 241, 290, 320 and 322.)
- Shmuel Katz and Zohar Manna. Towards automatic debugging of programs. *ACM SIGPLAN Notices*, 10(6):143–155, 1975. Proceedings of the International Conference on Reliable software, Los Angeles. 1975. (Cited on page 383.)
- Moritz Kiefer, Vladimir Klebanov, and Mattias Ulbrich. Relational program reasoning using compiler IR. In *8th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*, 2016. To appear. (Cited on page 483.)

- James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7): 385–394, July 1976. (Cited on pages 4, 67 and 383.)
- Joseph R. Kiniry, Alan E. Morkan, Dermot Cochran, Fintan Fairmichael, Patrice Chalin, Martijn Oostdijk, and Engelbert Hubbers. The KOA remote voting system: A summary of work to date. In Ugo Montanari, Donald Sannella, and Roberto Bruni, editors, *Proceedings of Trustworthy Global Computing (TGC)*, volume 4661 of *LNCS*, pages 244–262. Springer, 2006. (Cited on page 606.)
- Laurie Kirby and Jeff Paris. Accessible independence results for Peano Arithmetic. *Bulletin of the London Mathematical Society*, 14(4), 1982. (Cited on page 40.)
- Michael Kirsten. Proving well-definedness of JML specifications with KeY. Studienarbeit, KIT, 2013. (Cited on pages 254 and 287.)
- Vladimir Klebanov. Precise quantitative information flow analysis – a symbolic approach. *Theoretical Computer Science*, 538:124–139, 2014. (Cited on page 470.)
- Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T. Leavens, Valentin Wüstholtz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark Hillebrand, Bart Jacobs, K. Rustan M. Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan Tobies, Thomas Tuerk, Mattias Ulbrich, and Benjamin Weiß. The 1st Verified Software Competition: Experience report. In Michael Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland. Proceedings*, volume 6664 of *LNCS*, pages 154–168. Springer, 2011. (Cited on pages 18 and 289.)
- Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an operating system kernel. *Communications of the ACM*, 53(6):107–115, June 2010. (Cited on page 9.)
- Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison–Wesley, third edition, 1998. (Cited on page 558.)
- Dexter Kozen and Jerzy Tiuryn. Logics of programs. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 14, pages 789–840. The MIT Press, 1990. (Cited on page 49.)
- Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 1 edition, 2008. (Cited on page 537.)
- Daniel Kroening and Michael Tautschnig. CBMC - C bounded model checker - (competition contribution). In Erika Ábrahám and Klaus Havelund, editors, *Proceedings, 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 8413 of *LNCS*, pages 389–391. Springer, 2014. (Cited on page 97.)
- Ralf Küsters, Tomasz Truderung, and Andreas Vogt. Verifiability, Privacy, and Coercion-Resistance: New Insights from a Case Study. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, Berkeley, California, USA*, pages 538–553, Oakland, California, USA, 2011. IEEE Computer Society. (Cited on pages 593, 594, 595 and 605.)
- Ralf Küsters, Tomasz Truderung, Bernhard Beckert, Daniel Bruns, Michael Kirsten, and Martin Mohr. A hybrid approach for proving noninterference of Java programs. In Cédric Fournet and Michael Hicks, editors, *28th IEEE Computer Security Foundations Symposium*, pages 305–319. IEEE Computer Society, 2015. (Cited on pages 596 and 605.)
- Leslie Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Proceedings of the IFIP Congress on Information Processing*, pages 657–667, Amsterdam, 1983. North-Holland. (Cited on page 291.)
- Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, 1973. (Cited on page 454.)
- Daniel Larsson and Reiner Hähnle. Symbolic fault injection. In Bernhard Beckert, editor, *Proc. 4th International Verification Workshop (Verify) in connection with CADE-21 Bremen, Germany*, volume 259, pages 85–103. CEUR Workshop Proceedings, July 2007. (Cited on page 17.)
- Gary T. Leavens. *Verifying Object-Oriented Programs that use Subtypes*. PhD thesis, Massachusetts Institute of Technology, December 1988. (Cited on pages 219, 260, 292 and 293.)

- Gary T. Leavens and Yoonsik Cheon. Preliminary design of Larch/C++. In Ursula Martin and Jeannette M. Wing, editors, *Proceedings of the First International Workshop on Larch, 1992*, Workshops in Computing, pages 159–184, New York, NY, 1993. Springer. (Cited on page 194.)
- Gary T. Leavens and Krishna Kishore Dhara. Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 113–135. Cambridge University Press, 2000. (Cited on pages 219 and 293.)
- Gary T. Leavens and David A. Naumann. Behavioral subtyping is equivalent to modular reasoning for object-oriented programs. Technical Report 06-36, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, December 2006. (Cited on pages 219 and 293.)
- Gary T. Leavens and William E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, 1995. (Cited on page 293.)
- Gary T. Leavens and Jeanette M. Wing. Protective interface specifications. *Formal Aspects of Computing*, 10(1):59–75, 1998. (Cited on page 281.)
- Gary T. Leavens, Jean-Raymond Abrial, Don Batory, Michael Butler, Alessandro Coglio, Kathi Fisler, Eric Hehner, Cliff Jones, Dale Miller, Simon Peyton-Jones, Murali Sitaraman, Douglas R. Smith, and Aaron Stump. Roadmap for enhanced languages and methods to aid verification. In Stan Jarzabek, Douglas C. Schmidt, and Todd L. Veldhuizen, editors, *Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA. Proceedings*, pages 221–236, New York, NY, USA, 2006a. ACM. (Cited on page 289.)
- Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006b. (Cited on pages 193 and 253.)
- Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19(2):159–189, 2007. (Cited on page 289.)
- Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, and Werner Dietl. *JML Reference Manual*, May 31, 2013. Draft Revision 2344. (Cited on pages ix, 2, 13, 193, 208, 233, 243, 244, 245, 247, 248, 253, 261, 262, 280, 322, 328, 621 and 628.)
- Dirk Leinenbach. *Compiler Verification in the Context of Pervasive System Verification*. PhD thesis, Saarland University, Saarbrücken, 2008. (Cited on page 473.)
- K. Rustan M. Leino. *Towards Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03. (Cited on page 289.)
- K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In Bjørn N. Freeman-Benson and Craig Chambers, editors, *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98), Vancouver, British Columbia, Canada*, volume 33, pages 144–153. ACM, October 1998. (Cited on pages 320 and 347.)
- K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, 2005. (Cited on page 76.)
- K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, 2010, Revised Selected Papers*, volume 6355 of LNCS, pages 348–370. Springer, 2010. (Cited on pages 2, 7, 10, 241 and 348.)
- K. Rustan M. Leino and Michał Moskal. VACID-0: Verification of ample correctness of invariants of data-structures, Edition 0. In Gary T. Leavens, Peter W. O’Hearn, and Sriram Rajamani, editors, *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE, Edinburgh, UK*, Edinburgh, UK, 2010. (Cited on page 296.)
- K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In Martin Odersky, editor, *European Conference on Object-Oriented Programming*, volume 3086 of LNCS, pages 491–516. Springer, 2004. (Cited on page 215.)

- K. Rustan M. Leino and Peter Müller. A verification methodology for model fields. In Peter Sestoft, editor, *European Symposium on Programming (ESOP)*, volume 3924 of *LNCS*, pages 115–130, New York, NY, March 2006. Springer. (Cited on page 350.)
- K. Rustan M. Leino and Greg Nelson. An extended static checker for Modula-3. In Kai Koskimies, editor, *Compiler Construction, 7th International Conference, CC'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal. Proceedings*, volume 1383 of *LNCS*, pages 302–305. Springer, 1998. (Cited on page 240.)
- K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, September 2002. (Cited on pages 302 and 322.)
- K. Rustan M. Leino, Greg Nelson, and J.B. Saxe. ESC/Java user's manual. Technical Report SRC 2000-002, Compaq System Research Center, 2000. (Cited on pages 195 and 240.)
- K. Rustan M. Leino, Arnd Poetsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, volume 37(5), pages 246–257, New York, NY, June 2002. ACM. (Cited on page 348.)
- K. Rustan M. Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with Chalice. In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 5705 of *LNCS*, pages 195–222. Springer, 2009. (Cited on pages 350 and 378.)
- Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA*, pages 42–54. ACM, 2006. (Cited on page 473.)
- Xavier Leroy. A formally verified compiler back-end. *J. Automated Reasoning*, 43(4):363–446, 2009. (Cited on page 473.)
- Barbara Liskov. Data abstraction and hierarchy. *SIGPLAN Notices*, pages 17–34, May 1988. (Cited on pages 218, 219 and 292.)
- Barbara Liskov and Jeanette M. Wing. Specifications and their use in defining subtypes. In Andreas Paepcke, editor, *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 16–28, Washington DC, USA, 1993. ACM Press. (Cited on pages 217 and 292.)
- Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994. (Cited on pages 218 and 292.)
- Sarah M. Loos, David W. Renshaw, and André Platzer. Formal verification of distributed aircraft controllers. In Calin Belta and Franjo Ivancic, editors, *Proc. 16th Intl. Conference on Hybrid Systems: Computation and Control, HSCC, Philadelphia, PA, USA*, pages 125–130. ACM, 2013. (Cited on page 6.)
- Claude Marché and Nicolas Rousset. Verification of Java Card applets behavior with respect to transactions and card tears. In *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), Pune, India*, pages 137–146. IEEE CS Press, 2006. (Cited on page 377.)
- Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated with JML annotations. *J. Logic and Algebraic Programming*, 58:89–106, 2004. (Cited on pages 195, 239 and 353.)
- John McCarthy. Towards a mathematical science of computation. In Cicely M. Popplewell, editor, *Information Processing 1962, Proceedings of IFIP Congress 62, Munich, Germany*, pages 21–28. North-Holland, 1962. (Cited on page 41.)
- John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. *Mathematical Aspects of Computer Science*, 19:33–41, 1967. Proceedings of Symposia in Applied Mathematics. 1967. (Cited on page 473.)
- José Meseguer and Grigore Rosu. Rewriting logic semantics: From language specifications to formal analysis tools. In D. Basin and M. Rusinowitch, editors, *Automated Reasoning, Second*

- International Joint Conference, IJCAR 2004, Cork, Ireland, Proceedings*, volume 3097 of *LNCS*, pages 1–44. Springer, 2004. (Cited on page 64.)
- Bertrand Meyer. From structured programming to object-oriented design: The road to Eiffel. *Structured Programming*, 1:19–39, 1989. (Cited on page 246.)
- Bertrand Meyer. Applying “design by contract”. *IEEE Computer*, 25(10):40–51, October 1992. (Cited on pages 13, 194, 289 and 291.)
- Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1997. (Cited on pages 194 and 291.)
- Alysson Milanez, Dênnis Sousa, Tiago Massoni, and Rohit Gheyi. JMLOK2: A tool for detecting and categorizing nonconformances. In Uirá Kulesza and Valter Camargo, editors, *Congresso Brasileiro de Software: Teoria e Prática*, pages 69–76, 2014. (Cited on page 239.)
- Robin Milner and Richard Weyhrauch. Proving compiler correctness in a mechanized logic. *Machine Intelligence*, 7:51–72, 1972. Proceedings of the 7th Annual Machine Intelligence Workshop, Edinburgh, 1972. (Cited on page 473.)
- Andrzej Mostowski. On a generalization of quantifiers. *Fundamenta Mathematicæ*, 44(1):12–36, 1957. (Cited on page 248.)
- Wojciech Mostowski. Formalisation and verification of Java Card security properties in dynamic logic. In Maura Cerioli, editor, *Fundamental Approaches to Software Engineering (FASE), Edinburgh, Proceedings*, volume 3442 of *LNCS*, pages 357–371. Springer, April 2005. (Cited on pages 354, 376 and 609.)
- Wojciech Mostowski. Formal reasoning about non-atomic Java Card methods in Dynamic Logic. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *Proceedings, Formal Methods (FM) 2006, Hamilton, Ontario, Canada*, volume 4085 of *LNCS*, pages 444–459. Springer, August 2006. (Cited on pages 354 and 376.)
- Wojciech Mostowski. Fully verified Java Card API reference implementation. In Bernhard Beckert, editor, *Proceedings of 4th International Verification Workshop (VERIFY) in connection with CADE-21, Bremen, Germany, 2007*, 2007. (Cited on pages 3, 6, 354, 376 and 609.)
- Wojciech Mostowski. Dynamic frames based verification method for concurrent Java programs. In Arie Gurfinkel and Sanjit A. Seshia, editors, *Verified Software: Theories, Tools, and Experiments: 7th International Conference, VSTTE, San Francisco, CA, USA, Revised Selected Papers*, volume 9593 of *LNCS*, pages 124–141. Springer, 2015. (Cited on pages 3, 378 and 380.)
- Wojciech Mostowski and Erik Poll. Malicious code on Java Card smartcards: Attacks and countermeasures. In *Smart Card Research and Advanced Application Conference CARDIS 2008*, volume 5189 of *LNCS*, pages 1–16. Springer, September 2008. (Cited on pages 354 and 361.)
- Wojciech Mostowski and Mattias Ulbrich. Dynamic dispatch for method contracts through abstract predicates. In *Proceedings of the 14th International Conference on Modularity, MODULARITY 2015, Fort Collins, CO, USA*, pages 109–116. ACM, 2015. (Cited on pages 311 and 316.)
- Wojciech Mostowski and Mattias Ulbrich. Dynamic dispatch for method contracts through abstract predicates. *Transactions Modularity and Composition*, 1:238–267, 2016. (Cited on page 311.)
- Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer, Berlin, 2002. (Cited on pages 296, 348 and 350.)
- Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular specification of frame properties in JML. *Concurrency and Computation: Practice and Experience*, 15(2):117–154, February 2003. (Cited on pages 233 and 348.)
- Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62(3):253–286, October 2006. (Cited on page 215.)
- Oleg Mürk, Daniel Larsson, and Reiner Hähnle. KeY-C: A tool for verification of C programs. In Frank Pfenning, editor, *Proc. 21st Conference on Automated Deduction (CADE), Bremen, Germany*, volume 4603 of *LNCS*, pages 385–390. Springer, 2007. (Cited on page 16.)
- Andrew C. Myers. JFlow: practical mostly-static information flow control. In Andrew W. Appel and Alex Aiken, editors, *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA*, pages 228–241. New York, NY, USA, 1999. ACM. (Cited on pages 454 and 606.)

- Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, second edition, 2004. (Cited on page 576.)
- Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Verification of information flow and access control policies with dependent types. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, Berkeley, California, USA*, pages 165–179, May 2011. (Cited on page 455.)
- David A. Naumann. Observational purity and encapsulation. *Theoretical Computer Science*, 376(3):205–224, 2007. (Cited on page 210.)
- Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. (Cited on pages 2, 10 and 108.)
- Bashar Nuseibeh. Ariane 5: Who dunnit? *IEEE Software*, 14(3):15–16, May / June 1997. (Cited on page 230.)
- Kirsten Nygaard and Ole-Johan Dahl. The development of the SIMULA languages. In Richard L. Wexelblat, editor, *History of Programming Languages*, ACM monograph series. Academic Press, 1981. (Cited on page 291.)
- Peter W. O’Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France. Proceedings*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001. (Cited on pages 241 and 349.)
- Peter W. O’Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy*, pages 268–280. ACM, January 2004. (Cited on page 241.)
- Peter W. O’Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. *ACM Transactions on Programming Languages and Systems*, 31(3):11:1–11:50, April 2009. (Cited on page 349.)
- Sam Owre, S. Rajan, John M. Rushby, Natarajan Shankar, and Mandayam K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification, 8th International Conference, CAV ’96, New Brunswick, NJ, USA, 1996, Proceedings*, volume 1102 of *LNCS*, pages 411–414. Springer, 1996. (Cited on page 108.)
- Pierre Le Pallec, Ahmad Saif, Olivier Briot, Michael Bensimon, Jérôme Devisme, and Marilyne Eznack. NFC cardlet development guidelines v2.2. Technical report, Association Française du Sans Contact Mobile, 2012. (Cited on pages 354, 355 and 360.)
- Matthew Parkinson. Class invariants: The end of the road? In *International Workshop on Aliasing, Confinement and Ownership (IWACO)*, volume 23. ACM, 2007. position paper. (Cited on page 349.)
- Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. *SIGPLAN Notices*, 40(1): 247–258, January 2005. (Cited on pages 349 and 350.)
- Corina S. Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering*, 20(3):391–425, 2013. (Cited on page 449.)
- Christine Paulin-Mohring. Introduction to the Coq proof-assistant for practical software verification. In Bertrand Meyer and Martin Nordio, editors, *Tools for Practical Software Verification*, volume 7682 of *LNCS*, pages 45–95. Springer, 2012. (Cited on page 10.)
- Guillaume Petiot, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. How test generation helps software specification and deductive verification in Frama-C. In Martina Seidl and Nikolai Tillmann, editors, *Tests and Proofs - 8th International Conference, TAP 2014, Held as Part of STAF 2014, York, UK. Proceedings*, LNCS, pages 204–211. Springer, 2014. (Cited on page 449.)
- André Platzer. An object-oriented dynamic logic with updates. Master’s thesis, Universität Karlsruhe, Fakultät für Informatik, September 2004. (Cited on page 65.)
- André Platzer. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer, 2010. (Cited on page x.)
- André Platzer and Jan-David Quesel. KeYmaera: A hybrid theorem prover for hybrid systems. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning*,

- 4th International Joint Conference, IJCAR, Sydney, Australia*, volume 5195 of *LNCS*, pages 171–178. Springer, 2008. (Cited on pages 6 and 16.)
- Arndt Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs*. PhD thesis, Technical University of Munich, 1997. Habilitation thesis. (Cited on page 215.)
- Nadia Polikarpova, Julian Tschannen, Carlo A. Furia, and Bertrand Meyer. Flexible invariants through semantic collaboration. In Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods – 19th International Symposium, Singapore. Proceedings*, volume 8442 of *LNCS*, pages 514–530. Springer, 2014. (Cited on page 349.)
- Nadia Polikarpova, Julian Tschannen, and Carlo A. Furia. A fully verified container library. In Nikolaj Bjørner and Frank D. de Boer, editors, *FM 2015: Formal Methods - 20th Intl. Symp., Oslo, Norway*, volume 9109 of *LNCS*, pages 414–434. Springer, 2015. (Cited on page 3.)
- Guillaume Pothier, Éric Tanter, and José Piquet. Scalable omniscient debugging. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, Montreal, Quebec, Canada*, pages 535–552. ACM, 2007. (Cited on page 383.)
- Vaughan R. Pratt. Semantical considerations on Floyd-Hoare logic. In *17th Annual IEEE Symposium on Foundation of Computer Science, Houston, TX, USA. Proceedings*, pages 109–121. IEEE Computer Society, 1977. (Cited on pages 12 and 49.)
- Arun D. Raghavan and Gary T. Leavens. Desugaring JML method specifications. Technical Report TR #00-03e, Department of Computer Science, Iowa State University, 2000. Current revision from May 2005. (Cited on pages 206 and 255.)
- Henrique Rebêlo, Gary T. Leavens, Mehdi Bagherzadeh, Hridesh Rajan, Ricardo Lima, Daniel M. Zimmerman, Márcio Cornélio, and Thomas Thüm. Modularizing crosscutting contracts with AspectJML. In Walter Binder, Erik Ernst, Achille Peternier, and Robert Hirschfeld, editors, *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland. Proceedings*, pages 21–24, New York, NY, USA, 2014. ACM. (Cited on page 239.)
- John C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, Foundations of Computing, pages 13–24. The MIT Press, 1994. Reprint of the original 1975 paper. (Cited on page 252.)
- John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. 17th IEEE Symposium on Logic in Computer Science*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society. (Cited on pages 349 and 378.)
- Robby, Edwin Rodríguez, Matthew B. Dwyer, and John Hatcliff. Checking JML specifications using an extensible software model checking framework. *International Journal on Software Tools for Technology Transfer, STTT*, 8(3):280–299, 2006. (Cited on page 239.)
- Stan Rosenberg, Anindya Banerjee, and David A. Naumann. Decision procedures for region logic. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA. Proceedings*, volume 7148 of *LNCS*, pages 379–395, Berlin Heidelberg, 2012. Springer. (Cited on page 350.)
- Andreas Roth. *Specification and Verification of Object-oriented Software Components*. PhD thesis, Universität Karlsruhe, 2006. (Cited on page 296.)
- RTCA. DO-178C, Software considerations in airborne systems and equipment certification. published as RTCA SC-205 and EUROCAE WG-12, 2012. (Cited on page 424.)
- James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, Reading/MA, 2nd edition, 2010. (Cited on page 240.)
- Philipp Rümmer. Proving and disproving in dynamic logic for Java. Licentiate Thesis 2006–26L, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden, 2006. (Cited on pages 576 and 579.)

- Christoph Scheben. *Program-level Specification and Deductive Verification of Security Properties*. PhD thesis, Karlsruhe Institute of Technology, 2014. Karlsruhe, KIT, Diss., 2014. (Cited on pages 454, 455, 456, 457, 458, 460, 463, 467, 593 and 595.)
- Christoph Scheben and Peter H. Schmitt. Verification of information flow properties of Java programs without approximations. In Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov, editors, *Formal Verification of Object-Oriented Software International Conference, Turin, FoVeOOS 2011, Revised Selected Papers*, volume 7421 of *LNCS*, pages 232–249. Springer, 2012. (Cited on pages 455 and 458.)
- Christoph Scheben and Peter H. Schmitt. Efficient self-composition for weakest precondition calculi. In Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods - 19th International Symposium, Singapore. Proceedings*, volume 8442 of *LNCS*, pages 579–594. Springer, 2014. (Cited on pages 455 and 462.)
- Steffen Schlager. Handling of integer arithmetic in the verification of Java programs. Diplomarbeit, University of Karlsruhe, July 10 2002. (Cited on pages 230 and 245.)
- Peter H. Schmitt. A computer-assisted proof of the Bellman-Ford lemma. Technical Report 2011,15, Karlsruhe Institute of Technology, Fakultät für Informatik, 2011. (Cited on page 280.)
- Peter H. Schmitt and Mattias Ulbrich. Axiomatization of typed first-order logic. In Nikolaj Bjørner and Frank de Boer, editors, *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway. Proceedings*, volume 9109 of *LNCS*, pages 470–486. Springer, 2015. (Cited on page 47.)
- Peter H. Schmitt, Mattias Ulbrich, and Benjamin Weiß. Dynamic frames in Java dynamic logic. In Bernhard Beckert and Claude Marché, editors, *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France. Revised Selected Papers*, volume 6528 of *LNCS*, pages 138–152. Springer, 2010. (Cited on page ix.)
- Ulrik P. Schultz, Julia L. Lawall, and Charles Consel. Automatic program specialization for Java. *ACM Transactions on Programming Languages and Systems*, 25:452–499, 2003. (Cited on page 491.)
- Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. An automatic verifier for Java-like programs based on dynamic frames. In José Luiz Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary. Proceedings*, volume 4961 of *LNCS*, pages 261–275, Berlin, April 2008. Springer. (Cited on page 348.)
- Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1):2, 2012. (Cited on pages 350 and 378.)
- Ian Sommerville. *Software Engineering*. Pearson, 10th edition, 2015. (Cited on pages 2 and 18.)
- J. Michael Spivey. *Z Notation - a reference manual (2. ed.)*. Prentice Hall International Series in Computer Science. Prentice Hall, 1992. (Cited on page 240.)
- Kurt Stenzel. *Verification of Java Card Programs*. PhD thesis, Institut für Informatik, Universität Augsburg, Germany, July 2005. (Cited on page 239.)
- Jacques Stern. Why provable security matters? In Eli Biham, editor, *Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland. Proceedings*, volume 2656 of *LNCS*, pages 449–461. Springer, 2003. (Cited on page 607.)
- Christian Sternagel. Proof pearl — A mechanized proof of GHC’s mergesort. *Journal of Automated Reasoning*, pages 357–370, 2013. (Cited on page 609.)
- Alexander J. Summers, Sophia Drossopoulou, and Peter Müller. The need for flexible object invariants. In *International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming, (IWACO) at ECOOP 2008, Paphos, Cyprus*, pages 1–9. ACM, 2009. (Cited on page 350.)
- Robert D. Tennent. *Specifying Software: a Hands-On Introduction*. Cambridge University Press, 2002. (Cited on page 572.)

- Nikolai Tillmann and Jonathan de Halleux. Pex–white box test generation for .net. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs*, volume 4966 of *LNCS*, pages 134–153. Springer, 2008. (Cited on page 450.)
- Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In Michel Wermelinger and Harald Gall, editors, *Proc. 10th European Software Engineering Conference/13th ACM SIGSOFT Intl. Symp. on Foundations of Software Engineering, 2005, Lisbon, Portugal*, pages 253–262. ACM Press, 2005. (Cited on page 4.)
- Kerry Trentelman. Proving correctness of Java Card DL tacelets using Bali. In Bernhard Aichernig and Bernhard Beckert, editors, *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005), Koblenz, Germany*, pages 160–169, 2005. (Cited on page 64.)
- Thomas Tuerk. A formalisation of smallfoot in HOL. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany. Proceedings*, volume 5674 of *LNCS*, pages 469–484. Springer, 2009. (Cited on page 241.)
- Mattias Ulbrich. A dynamic logic for unstructured programs with embedded assertions. In Bernhard Beckert and Claude Marché, editors, *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France. Revised Selected Papers*, volume 6528 of *LNCS*, pages 168–182. Springer, 2011. (Cited on page 473.)
- Mattias Ulbrich. *Dynamic Logic for an Intermediate Language. Verification, Interaction and Refinement*. PhD thesis, Karlsruhe Institut für Technologie, KIT, 2013. (Cited on pages 36 and 473.)
- Bart van Delft and Richard Bubel. Dependency-based information flow analysis with declassification in a program logic. *Computing Research Repository (CoRR)*, 2015. (Cited on page 471.)
- Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In Tiziana Margaria and Wang Yi, editors, *Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Genova, Italy*, volume 2031 of *LNCS*, pages 299–312, 2001. (Cited on page 195.)
- Sergiy A. Vilkomir and Jonathan P. Bowen. Formalization of software testing criteria using the Z notation. In *25th International Computer Software and Applications Conference (COMPSAC 2001), Invigorating Software Development, Chicago, IL, USA*, pages 351–356. IEEE Computer Society, 2001. (Cited on pages 424 and 425.)
- David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001. (Cited on page 64.)
- Simon Wacker. Blockverträge. Studienarbeit, Karlsruhe Institute of Technology, 2012. (Cited on pages 238, 466 and 623.)
- Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series. Addison-Wesley, Reading/MA, 1999. (Cited on pages 1, 13 and 240.)
- Nathan Wasser. Generating specifications for recursive methods by abstracting program states. In Xuandong Li, Zhiming Liu, and Wang Yi, editors, *Dependable Software Engineering: Theories, Tools, and Applications - First International Symposium, SETTA 2015, Nanjing, China. Proceedings*, pages 243–257. Springer, 2015. (Cited on page 189.)
- Benjamin Weiß. Predicate abstraction in a program logic calculus. In Michael Leuschel and Heike Wehrheim, editors, *Integrated Formal Methods, 7th International Conference, IFM 2009, Düsseldorf, Germany. Proceedings*, volume 5423 of *LNCS*, pages 136–150. Springer, 2009. (Cited on page 474.)
- Benjamin Weiß. *Deductive Verification of Object-Oriented Software — Dynamic Frames, Dynamic Logic and Predicate Abstraction*. PhD thesis, Karlsruhe Institute of Technology, Karlsruhe, January 2011. (Cited on pages ix, 241, 243, 251, 290, 306, 307, 319, 322, 335, 336, 338 and 341.)
- Florian Widmann. Crossverification of while loop semantics. Diplomarbeit, Fakultät für Informatik, KIT, 2006. (Cited on page 101.)
- Niklaus Wirth. Modula: a language for modular multiprogramming. *Software Practice and Experience*, 7:3–35, 1977. (Cited on page 291.)

- Peter Y. H. Wong, Elvira Albert, Radu Muschevici, José Proença, Jan Schäfer, and Rudolf Schlatte. The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. *International Journal on Software Tools for Technology Transfer, STTT*, 14(5):567–588, 2012. (Cited on page 6.)
- Jim Woodcock, Susan Stepney, David Cooper, John A. Clark, and Jeremy Jacob. The certification of the mondex electronic purse to ITSEC level E6. *Formal Aspects of Computing*, 20(1):5–19, 2008. (Cited on page 605.)
- Jooyong Yi, Robby, Xianghua Deng, and Abhik Roychoudhury. Past expression: encapsulating pre-states at post-conditions by means of AOP. In *Proceedings of the 12th annual international conference on Aspect-oriented software development, (AOSD), Fukuoka, Japan*, pages 133–144. ACM, 2013. (Cited on page 249.)
- Lei Yu. A formal model of IEEE floating point arithmetic. *Archive of Formal Proofs*, 2013, 2013. (Cited on page 3.)
- Marina Zaharieva-Stojanovski and Marieke Huisman. Verifying class invariants in concurrent programs. In Stefania Gnesi and Arend Rensink, editors, *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France. Proceedings*, volume 8411 of *LNCS*, pages 230–245. Springer, 2014. (Cited on page 216.)
- Karen Zee, Viktor Kuncak, and Martin C. Rinard. Full functional verification of linked data structures. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Programming Language Design and Implementation (PLDI)*, pages 349–361, New York, NY, 2008. ACM. (Cited on page 296.)
- Andreas Zeller. *Why programs fail—A guide to systematic debugging*. Elsevier, 2nd edition, 2006. (Cited on page 412.)
- Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997. (Cited on page 423.)
- Daniel M. Zimmerman and Rinkesh Nagmoti. JMLUnit: The Next Generation. In B. Beckert and C. Marché, editors, *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France. Revised Selected Papers*, volume 6528 of *LNCS*. Springer, 2010. (Cited on page 239.)