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 9
Modular Specification and Verification

Daniel Grahl, Richard Bubel, Wojciech Mostowski, Peter H. Schmitt, Mattias Ulbrich, and Benjamin Weiß

Software systems can grow large and complex, and various programming disciplines have been developed addressing the problem how programmers can cope with such complex systems. We focus in this book on the paradigm of object-orientation which seems to be the widely adopted mainstream approach.

In parallel to these development in software engineering, formal verification needs complementary techniques for dealing with large software systems and increased complexity. Yet, achieving complete functional verification of a complex piece of software still poses a grand challenge to current research ([Leino, 1995, Hoare and Misra, 2005, Leavens et al., 2006a, 2007, Klebanov et al., 2011, Huisman et al., 2015]). In this chapter we will present which support the KeY system offers in this direction and review the research background it is based on. In most subsections we will come back to concepts already presented in earlier chapters, but now with special emphasis on modularization. We will take extra pain to precisely delineate these dependencies.

It is common wisdom that the keys to scale up a technique for large applications are *modularization* and *abstraction*. In our case, the deductive verification of object-oriented software, the central pillar for modularization and abstraction is the *Design by Contract* principle as pioneered by Meyer [1992]. Once the contract for a method has been separately verified we need not at every call to this method inspect its code again but use its contract instead. In Chapter 7 method contracts have already been introduced as a central concept of the behavioral specification language JML (Java Modeling Language). Syntax and semantics of JML method contracts have been thoroughly explained there. Chapter 8 explained how JML method contracts are translated into proof obligations in JavaDL whose validity entails the correctness of the method w.r.t. the contract. In this chapter, we explain what needs to be considered when *using* a contract instead of the code of a method on the caller side and present logical calculus rules implementing this. A separate subsection is devoted to recursive methods. They are a special case since the contract to be verified is used itself at every recursive call of the method.

Method contracts can only play out their advantages if they do not themselves make use of implementation details. To achieve this it is necessary to have means available that abstract away from the code. To this end JML offers *model fields* and *model methods*, syntax elements that only occur in specifications and are not part of the code. These have already been addressed Section 7.7.1. Here we present in great detail the semantics of these concepts at the level of JavaDL and also show and discuss calculus rules.

Object invariants have already been addressed in Section 7.4.1. Here we present technical details, the representation of invariants as implicit model fields and how they are handled in KeY. In modular specification and verification, knowing which memory locations a method does *not* change is almost as important as knowing the effects of it. How to formalize and utilize this information is known as the *frame problem*. There is a long history of verification techniques that deal with the frame problem. Our approach, see [Weiß, 2011], is inspired by the *dynamic frames* technique from [Kassios, 2011] that aims at providing modular reasoning in the presence of abstractions as they occur in object oriented programs. In Section 7.9 we encountered already the **`assignable`** clause in JML specifications that provides a set of locations that a method might at most assign to. In Section 8.2 we saw how the JML **`assignable`** clause is translated into the `mod` part of a JavaDL loop or method contract. The calculus rules for proving these contracts were already covered in Section 3.7. In this chapter rules will be presented (1) for a more fine grained treatment of anonymization in loop verification and (2) for *using* method contracts.

In a way complementary to the information which locations a method may write to is the information which locations a method may read from. How this information is formulated was already explained (1) on the JML level via **`accessible`** clauses in Section 7.9, (2) as a JavaDL dependency contract in Definition 8.3 and, (3) as a JavaDL proof obligation in Definition 8.5. In this chapter **`accessible`** clauses for model methods are introduced. The previous proof obligation for dependency contracts has to be revised to cover this extension.

Although we use Java and JML as technological basis in this chapter, we expect all mentioned concepts to be adaptable to other object-oriented programming languages and their associated specification languages.

Chapter Overview

We start off this chapter with introducing the basic concepts of modular specification in Section 9.1. This will explain in general method contracts, behavioral subtyping, and lead up to our formalization of modular code correctness. A running example that will be used throughout this chapter will make its first appearance in Section 9.1.2. The special case of unbounded recursion is discussed in Section 9.1.4.

We present model fields as they appear in standard JML and their role in the KeY system in Section 9.2.1, as well as the more advanced concept of model methods in Section 9.2.2. The *frame problem* is the topic of Section 9.3.

Section 9.4 takes us to the second theme in the title of this chapter—verification. Building on the calculus for JavaDL from Section 3.5, we introduce additional rules for modular reasoning. This includes 1. an improved loop invariant rule, that retains most of the execution context and that caters for unbounded recursion depth (Section 9.4.2); 2. a rule for applying functional methods contracts (Section 9.4.3); 3. a rule for dependency contracts, based on the dynamic frame theory (Section 9.4.4); and 4. rules for inserting class invariants into the proof (Section 9.4.5).

In Section 9.5 we will verify the example from Section 9.1.2 putting to work the techniques that will have been introduced by then.

In Section 9.6 we give a quick glimpse of related work and the chapter closes with a summarizing look back on what has been covered in Section 9.7.

9.1 Modular Verification with Contracts

Method contracts are a central pillar of modular program verification. When combined with behavioral subtyping, they provide means for both abstraction and modularization. In this section, we will, after a review of the general background and the presentation of the chapter's running example, discuss the notion of modularity employed in JavaDL and how it can be used for the verification of recursive methods.

9.1.1 Historical and Conceptual Background

The concept of modules in programming languages can be traced back to early examples such as Simula 67 [Nygaard and Dahl, 1981] or Modula [Wirth, 1977]. Single modules (i.e., method implementations or classes containing them) may be added, removed, or changed with only minimal changes to their clients; programs can be reused or evolved in a reliable way. Modular analysis of a module can be based on the module itself in isolation—without a concrete representation of its environment. This allows one to adapt modules to other environments without losing previously established guarantees.

These ideas were put forth with the development of object-oriented programming: "The cornerstone of object-oriented technology is reuse." [Meyer, 1997] In object-oriented programming (OOP), methods (or procedures) consist of declarations and implementations. Declarations are visible to clients while implementations are hidden. One important addition in OOP to the base concept of modularity is that classes (i.e., modules) are meant to define types—and subclasses define subtypes. And, in particular, different classes may implement a method in different ways (*overriding*), including covariant and contravariant type refinement. A client never knows which implementation is actually used. Any call to a (nonprivate) method is subject to *dynamic dispatch*, i.e., the appropriate implementation is chosen at runtime from the context. This concept is also known as *virtual* method invocation.

The concept of a *contract* between software modules was first proposed by Lamport [1983] and later popularized by Meyer [1992] under the trademark *Design by Contract*. It allows one to abstract from those concrete implementations and to approximately predict module behavior statically.[1] The metaphor of a legal contract gives an intuition: A client (method caller) and a provider (method implementer) agree on a contract that states that, under given resources (preconditions), a product with certain properties (postconditions) is provided. This is a separation of duties; the provider can rely on the preconditions, otherwise he or she is free to do anything. Given the preconditions, he or she is only obliged to ensure the postconditions, no matter *how* they are established. On the other hand side, the client is obliged to ensure the preconditions and can only assume a product to the given specifications. In the basic setup, a method contract just consists of such a pair of pre- and postcondition. As it has already been explained in Chapter 7, state of the art specification languages as JML feature contracts with several clauses (of which all can be seen as specialized, functional or nonfunctional pre- or postconditions).

Contracts do not only play an important role in software design, but also in verification. In verifying a method that calls another method, there are two possibilities to deal with that case. Either, the implementation can be inserted or a contract can be used. The former is intriguingly simple; this is what would happen in an actual execution. But it carries three disadvantages:

1. It transgresses the concept of information hiding.
2. The concrete implementation of the callee must be known. This cannot always guaranteed in static verification techniques as in many cases the actual type of objects is not known at verification time. When verifying extensible programs, the implementation code may not even be available at verification time.
3. In the case of recursive implementations (with an unbounded recursion depth), inserting the same implementation again would let the proof run in circles.

This leaves contracts as a good choice to deal with method calls in most cases. In Subsection 16.4 the reader is guided through a tutorial example of using simple contracts.

Behavioral Subtyping

In a completely modular context, the concrete method implementations generally are not known. Nevertheless, a client will assume that all implementations of a common public interface (i.e., a method declaration) behave in a uniform way. This concept is known as *behavioral subtyping*, *Liskov's substitution principle*, or the *Liskov-Leavens-Wing principle* [Liskov, 1988, Leavens, 1988, Liskov and Wing, 1993, 1994].[2,3] It can be formulated as follows: A type T' is a behavioral subtype of a type T if instances of T' can be used in any context where an instance of T is expected by an observer. In other words, behavioral "subtyping prevents surprising

[1] Note that contracts give semantical properties about modules and are in some sense orthogonal to design documents such as class diagrams, that are mostly syntactical.

[2] Liskov and Wing themselves use the term "constraint rule."

[3] Despite first appearing in Leavens' thesis, it has been attributed to Liskov because of her widely influential keynote talk at the OOPSLA conference 1988.

behavior" [Leavens, 1988]. Note that this notion of a 'type' is different to both types in logic (see Section 2.2) and types in Java (i.e., classes and interfaces).

Behavioral subtyping is a semantical property of implementations. Although the concept is tightly associated with design by contract, it cannot be statically enforced by programming languages. It is not uncommon to see—especially in undergraduate exercises—that subclasses in object-oriented programs are misused in a nonbehavioral way. Imagine, for instance, a class `Rectangle` being implemented as a subclass of `Square` because it adds a length to `Square`'s width. This kind of data-centric reuse is a typical pattern for modular programming languages without inheritance. Not all rectangles are squares, so intuitively, this should not define a behavioral subtype. But whether it actually does, depends on the public interface (i.e., the possible observations). If the class signature of `Square` allows one to set the width to a and to observe the area as a^2, then the subclass `Rectangle` is not a behavioral subtype.

For modular reasoning about programs, we may only assume contracts for a dynamically dispatched method that are associated with the receiver's static type, since the precise dynamic type depends on the context. This technique is known as *supertype abstraction* [Leavens and Weihl, 1995]. Behavioral subtyping is essential to sound supertype abstraction.[4] To (partially) enforce it, in the Java Modeling Language, method contracts are inherited to overriding implementations [Leavens and Dhara, 2000]; see also Section 7.4.5. We can provide additional specifications in subclasses, which are conjoined with the inherited specification. This means, whatever the subclass specification states locally, it can only *refine* the superclass specification, effectively. This leads us to a slightly relaxed version of behavioral subtyping: instead of congruence w.r.t. *any* observable behavior, we restrict it to the *specified* behavior.[5] This relaxation renders behavioral subtyping more feasible in practice, as it allows more freedom in implementing unspecified behavior, in particular regarding exceptional cases. Consider, for instance, a class that implements a collection of integers. Is a collection of nonnegative integers a behavioral subtype?—The correct answer is 'maybe;' it depends on whether the operations that add members to the collection are sufficiently abstract to be implemented differently.

This notion of behavioral subtyping w.r.t. specified behavior also enables us to regard interfaces and abstract classes as behavioral supertypes of their implementations. While they do not provide a (complete) implementation themselves, they can be given a specification that is inherited to the implementing classes.

[4] Leavens and Naumann [2006] present a language-independent formalization of behavioral subtyping and prove that it is actually equivalent to supertype abstraction.

[5] Still, it is possible to explicitly specify the observable behavior in its entirety. On the other hand side, specification is slightly more expressive than program constructs. The reason for this is that specification can refer to the entire heap. For instance, the notions of weak purity and strong purity differ in whether objects may be freshly allocated. This difference is not observable programmatically. Yet, JML allows one to declare a method strictly pure or—more generally—to express the number of created objects. While strict purity annotations may simplify modular verification, it cannot be included in a behavioral interface specification since it reveals an implementation detail.

9.1.2 Example: Implementing a List

Consider we want to implement a mutable list of integers in Java. It should support the following operations: (i) adding an element at the front, (ii) removing the first entry, (iii) indicating whether it is empty, (iv) returning its size, (v) retrieving an element at a given position (random access).

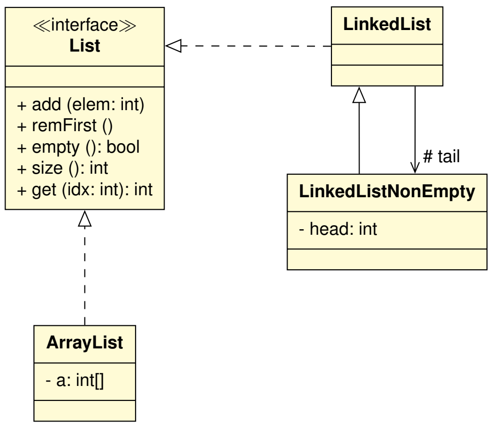
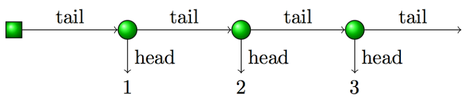


Fig. 9.1 A list interface and its implementations

Figure 9.1 shows a UML class diagram with the interface `List` that provides the intended signature as public methods. There are multiple ways to implement this interface. The figure shows two possibilities attached via dashed triangle-headed arrows: firstly simply as an `ArrayList` and secondly using a variant of the *composite* design pattern by the classes `LinkedList` and `LinkedListNonEmpty`. The annotation on the association from `LinkedList` to `LinkedListNonEmpty` signifies that `LinkedList` contains a protected field of type `LinkedListNonEmpty`.
The list $[1,2,3]$ is represented in this design as follows (with squares for instances of class `LinkedList` and circles for instances of `LinkedListNonEmpty`):



Note how the empty list is represented. This approach is called the *sentinel* pattern and prevents the **`null`** reference to be exposed.

An common alternative pattern for linked lists uses two classes `Nil` and `Cons`, where `Nil` is a singleton representing the empty list and `Cons` is a *sentinel*, that plays the same role as `LinkedListNonEmpty` in our example. This pattern is appropriate to implement *immutable* list objects. The disadvantage is that `Nil` and `Cons` are not (behavioral) subtypes of one or another.

Before looking at an implementation, let us briefly discuss contracts in natural language. The operation ‘removing the first element’ only makes sense when there is at least one element—this would make a precondition. Similarly, ‘retrieving an element at position *n*’ only makes sense if *n* is nonnegative and there are at least *n* elements in the list. Again, implementations are free to do anything if they are called in a context where these preconditions do not hold. Listing 9.1 shows an implementation of class `LinkedList`. Here, we see two different styles of method implementations. In Lines 11ff., method `remFirst()` silently returns directly if it is called on an empty list, i.e., the precondition is violated. Alternatively, we could first check for such violations and then throw a more precise exception explicitly. This style is known as *defensive* implementation, where the implementing code checks for and handles abnormal situations. In lines 24ff., method `get()` is implemented in an *offensive* manner. It does not check for abnormal situations, but optimistically calls `tail.get(idx)` where `tail` may be a `null` reference. In case the precondition is violated, an instance of `NullPointerException` will be thrown. Design by contract itself does not advertise either style, but in practice the latter is usually preferred.

```
 1 public class LinkedList implements List {
 2
 3     protected LinkedListNonEmpty tail;
 4
 5     public void add (int elem) {
 6         LinkedListNonEmpty tmp = new LinkedListNonEmpty(elem);
 7         tmp.tail = this.tail;
 8         this.tail = tmp;
 9     }
10
11     public void remFirst () {
12         if (empty()) return;
13         else tail = tail.tail;
14     }
15
16     public boolean empty () {
17         return tail == null;
18     }
19
20     public int size () {
21         return empty()? 0: tail.size();
22     }
23
24     public int get (int idx) {
25         return tail.get(idx);
26     }
27 }
```

Listing 9.1 An implementation to the `List` interface using a linked data structure

It is instructive to observe that most methods in `LinkedList` delegate to an element of the subclass `LinkedListNonEmpty`. This is possible since every (nonnull) object in `LinkedListNonEmpty` represents a non empty list, while objects in the supertype `LinkedList` represent—*possibly empty* lists. This ensures that we have a behavioral subtype relation here. A nonempty linked list exposes at least the expected behavior of a possibly empty linked list. This allows for a maximum of reuse in class `LinkedListNonEmpty`, which is shown in Listing 9.2; only three methods need to be overridden.

Note that the default constructor of `LinkedList` returns a (nonunique) empty list.

```
1  class LinkedListNonEmpty extends LinkedList {
2
3      private int head;
4
5      LinkedListNonEmpty (int elem) { head = elem; }
6
7      public boolean empty () { return false; }
8
9      public int size () {
10         return 1+(tail==null? 0: tail.size());
11     }
12
13     public int get (int idx) {
14         if (idx == 0) return head;
15         else return tail.get(idx-1);
16     }
17 }
```

Listing 9.2 Nonempty lists is a behavioral subtype to lists

The implementation of `size` in lines 9ff in Figure 9.2 does not work for lists of length greater than $2^{31} - 1$. We will live with this imperfection rather than resort to using `BigInteger`.

The above list example will be used throughout the rest of this chapter. Notable other case studies covering single linked lists can be found in the literature [Zee et al., 2008, Gladisch and Tyszberowicz, 2013]. In [Bruns, 2011] the more general data type of maps is considered. Its specification uses model fields and dynamic frames and its implementation is based on red/black trees. This has been proposed as one of the challenges in [Leino and Moskal, 2010].

9.1.3 Modular Program Correctness

In this section we explain our understanding of modular program correctness in a spirit similar to and inspired by Müller [2002], Roth [2006]. Müller [2002] defines the concept of modular correctness by distinguishing *open programs* and *closed programs*. Open programs are intended to be used in different (not a priori known) contexts, like, for instance, library code. Closed programs are self-contained and not meant to be extended. When analyzing the correctness of a closed program, stronger assumptions can be made than for the analysis open programs; in particular, every object must be an instance of one of the types declared in the program under test which may not be a safe assumption if the program is used in an extending context.

We will define modular correctness by the portability of correctness proofs to program extensions. Before we can look at modular correctness, we need to fix the notion of a program extension. Remember that for the purposes of this book, a Java program is a collection of class and interface declarations.

Definition 9.1 (Program Extension). A Java program p' is called an extension of the Java program p, denoted by $p' \supseteq p$, if

1. p' is obtained from p by adding new class or interface declarations, *and*
2. the declarations obtained from p are in no way modified.

We stress that in passing from p to p' no field, method, `extends` or `implements` declarations of existing classes or interfaces may be added, modified or removed. On the other hand, classes in which are new in p' may implement interfaces or extend classes from p, and methods introduced in p may be overridden in new subclasses in p'.

The soundness of logical inferences in JavaDL may depend on the type system and thus on the investigated program. For example: it is sound to deduce from $\mathit{instance}_B(x) \doteq \mathit{TRUE}$ that $\mathit{instance}_A(x) \doteq \mathit{TRUE}$ if and only if the type hierarchy contains $B \sqsubseteq A$. However, there are also inference rules that are either independent of the program or resilient to a program extension.

Definition 9.2 (Modular soundness). Let p be a Java program.

A logical inference rule is called *modularly sound* for p if it is a sound inference rule for every program extension p' with $p' \supseteq p$.

A proof is called *modularly correct* for p if it has been conducted with only modularly sound inference rules.

Most rules in the JavaDL sequent calculus in KeY are independent of the type hierarchy of the program (rules of propositional logic, rules dealing with numbers, sequences, ...). Some rules depend on the type hierarchy but are modularly sound (like the removal of unnecessary type cast operations). Only very few rules of the calculus are *not* modularly sound. In their core, all of them rely on the principle of enumeration of all subtypes of a type declaration:

Let T be a type declaration in p and furthermore $\{U_1, \ldots, U_k\} = \{U \mid U \sqsubseteq T \text{ and } U \text{ is not abstract.}\}$ denote the set of nonabstract class declarations which extend T (directly or indirectly). The rule typeDist allows replacing an instance predicate $\mathit{instance}_T(x)$ by the disjunction over all possible exact instance predicates $\mathit{exactInstance}_{U_i}(x)$ of the subtypes.

$$\mathit{instance}_T(x) \rightsquigarrow \mathit{exactInstance}_{U_1}(x) \vee \ldots \vee \mathit{exactInstance}_{U_k}(x) \qquad \text{typeDist}$$

In an extension, a new subclass U_{k+1} of T may be added rendering this rule unsound.

The rule methodCall defined in Section 3.6.5.5 is another rule that is not modularly sound based on the same principle of type enumeration. With it, a method call o`.m()` can be replaced by a type distinction over the dynamic type of o during symbolic execution, resulting in method-body statements o`.m()@`U_i enumerating the different overriding implementations of `m()`. Again a new subclass with a new implementation breaks the rule's soundness.

It is evident from Definition 9.2 that a proof which is modularly correct for p can also be conducted in $p' \supseteq p$ without adaptation.

Definition 9.3 (Modular Correctness). Let p be a program and C a set of contracts (functional or dependency) for the declarations in p.

A program p is called *modular correct* if there exists a proof modularly correct for p for every proof obligation for $c \in C$.

Lemma 9.4. *Let $p, p' \supseteq p$ be programs and C a set of contracts for p and $C' \supseteq C$ a set of contracts for p' with $C'\big|_p = C$.*

If p is modularly correct and there exist proofs for all proof obligations in $p' \setminus p$ against C', then p' is correct.

This means that once a library has been proved modularly correct, it can be used in any context, and it suffices to prove the context correct against its contracts and the contracts of the library to obtain a correct composed program.

As a consequence of definition 9.3, modular correctness proofs may only contain method inlining for private or final methods which cannot be overridden. For general method inlining, it is up to the verifying person to decide if they want to conduct a proof for an open or for a closed program.

9.1.4 Verification of Recursive Methods

We turn now to the verification of recursive methods. The simplest version of a recursive method is a method that calls itself, a pattern often found in implementations of divide-and-conquer algorithms. Method `get()` in Listing 9.2, which retrieves the n-th element of a list, is a typical example.

The issue which sets recursive methods apart from normal methods is that when verifying their correctness, we are confronted with a situation that results in a circular proof dependency. Assume we are verifying the correctness of the previously mentioned `get()` method. During the verification we end up at the recursive method invocation `tail.get(idx-1)`. How can we proceed now (and also maintain modular correctness)? Obviously applying the contract for this call of `get` introduces a circular proof dependency, we use a contract whose correctness depends on the contract currently been proven. Closely related, but not identical, is the topic of termination. Next to loops, recursive methods are the other source for nonterminating programs.

One note of caution, in KeY we are oblivious to the method frame stack size. Hence, for instance, a partial correctness proof for a contract requiring a method not to terminate with an exception may succeed even though a `StackOverflowError` would be thrown in real life. In other words, when verifying a Java program, *we are only correct under the assumption that no concrete run of the program causes a* `StackOverflowError` *to be thrown by the virtual machine*.

In case of partial correctness the introduced proof dependency does not pose any problem and we can apply the contract for the method, as already observed by Hoare [1971]. Intuitively, that is sound for the following reason: When symbolically executing the method, we explore all paths that do not lead to a recursive call, i.e., all base cases are covered. Paths with recursive calls can then simply use the contract performing the step cases from $n+1$ recursive invocations to n invocations. The only problem is when the recursion is not well-founded, which would lead to an infinite recursion. But in case of partial correctness, the validity of the contract is then a triviality.

Let us turn to the case of total correctness. Its solution is similar to the treatment of termination in loops: the user has to supply an expression as part of the method contract which we call a *termination witness*. A termination witness is always nonnegative and strictly decreasing with each call. In JML a termination witness is specified using the keyword **`measured_by`**, see also Section 8.2.

Example 9.5. A call `get(idx)` of the `get()` method from Listing 9.2 retrieves the `idx`-th element of the list as follows: if the argument's `idx` value is 0 then the value at the current list element is returned otherwise the method recursively retrieves the `(idx-1)`-th element of the tail of the list. A reasonable choice for the termination witness is the argument itself, namely,

```
@ measured_by idx;
```

which is obviously strictly decreased at each recursive call together with the precondition that the value of `idx` must be nonnegative it follows directly that at each recursive call site the value is strictly decreased and nonnegative.

In the process of verifying a recursive method the value of the termination witness is captured by an equation in the prestate. When we then apply the contract we are just about to prove, we have as part of the precondition to show that the value of the termination witness is nonnegative and less than the captured value of the prestate. This additional preconditions ensures well-foundedness w.r.t. the proof dependencies.

When the user or the system applies a method contract it should be clear if it is a recursive method or not. How is this checked? The detection is trivial for the examples above as we have a direct recursion. But what about mutual recursions or what if an extension adds an (indirect) recursion—how do we maintain correctness in such situations? The solutions is to track the contracts used in a proof. When applying the method contract rule, the system checks if there exists a proof for the contract depending (directly or indirectly) on the proof obligation we are currently verifying. If a dependency is detected, the system allows the application of the contract rule only if the contract in question is equipped with a **`measured_by`** clause and upon contract application we require to show that the value of that expression is strictly less than in the initial state of the current method. Thus, circular reasoning is avoided: Either the dependency is unidirectional or the termination witness guarantees well-foundedness.

We conclude this section with some words on the expressiveness of termination witnesses and loop variants. In the above examples the termination witnesses are always strictly decreasing integer expressions with lower bound 0. The KeY system also allows the declaration of termination witnesses and loop variants of other data types. The binary JavaDL predicate symbol $\prec: \top \times \top$ used in termination proofs is axiomatized as a well-founded (Noetherian) relation. Besides supporting integers, KeY comes with built-in axioms for lexicographic ordering of pairs and finite sequences. One may, e.g., use the declaration `measured_by i, j;` with integer expressions `i` and `j`. This will be interpreted as the lexicographical ordering of pairs (i, j). In this case the definition of $\prec$ requires to show that either i has decreased strictly (while remaining nonnegative) or that i did not change and instead j has been strictly decreased, i.e., $(i_1, j_1) \prec (i_2, j_2)$ iff $i_1 \prec i_2 \vee (i_1 = i_2 \wedge j_1 \prec j_2)$. The relation $\prec$ can be extended to other data types, but it is then the user's responsibility that the axiomatization guarantees that $\prec$ is well-founded.

9.2 Abstract Specification

Specifications in JML are relatively close to the implementation in comparison to other specification mechanisms like OCL or Z that operate on abstract data. But even if specification refer to implementation entities at source code level, abstraction and modularization are indispensable for handling larger programs. Consider, for instance, the interface `List` again. We have not yet stated specifications for its methods. Our goal is to provide an interface specification that is amenable to *modular* specification and verification, i.e., one which is robust to extensions of the implementation. This puts us in a dilemma since such a specification must speak for parts of the software which are not yet there but may be added in an extension. Therefore, it must not expose implementation details.

A common approach is to use *pure methods* in specifications, see [Gladisch and Tyszberowicz, 2013]. Listing 9.3 shows the example of the `List` interface, specified using pure query methods. We can specify the behavior of all methods using the two pure methods `size()` and `get()`. The query `empty()` checks is a list is empty, which is true if its size (returned by the query `size()`) is zero. In the same way, the specification of `add()` uses the pure method `get()`: The observable effect of adding an element is that it can be retrieved again at the last position of the list, and that the elements at all other positions of the list remain unchanged. Using queries in specifications requires that every impure method lists the changes to all relevant queries in its postcondition. The abstract value of a `List` object is thus 'distributed' over the two queries `get()` and `size()`.

Often it is more convenient for the specifier, however, if the abstract object state is not only available through these methods but as an explicit artifact that can be handled effectively.

To this end, JML offers model fields and model methods as specification-only abstract representations of concrete implementation data; they have already been

```
public interface List {

    //@ public invariant size() >= 0;

    /*@ public normal_behavior
      @ ensures size() == \old(size()) + 1;
      @ ensures get(\old(size)) == elem;
      @ ensures (\forall int i; 1 <= i && i < size()-1;
      @                          get(i) == \old(get(i-1)));
      @*/
    public void add (int elem);

    /*@ public normal_behavior
      @ requires !empty();
      @ ensures size() == \old(size()) - 1;
      @ ensures (\forall int i; 0 <= i && i < size();
      @                          get(i) == \old(get(i+1)));
      @*/
    public void remFirst ();

    /*@ public normal_behavior
      @ ensures \result == (size() == 0);
      @*/
    public /*@ pure @*/ boolean empty ();

    /*@ public normal_behavior
      @ requires 0 <= idx && idx < size();
      @*/
    public /*@ pure @*/ int get (int idx);

    public /*@ pure @*/ int size ();
}
```

Listing 9.3 Java interface `List` specified using pure methods

introduced briefly in Section 7.7.1, These mechanisms enable implementation hiding: the requirement specification only refers to model fields while the abstraction relation is part of the (hidden) implementation details. We will discuss model fields in Section 9.2.1 and the more general concept of model methods in Section 9.2.2 below in depth. Both concepts are deliberately close to actual Java (both syntactically and semantically), which makes them more comprehensible for Java programmers.

It is natural to represent the abstract state of an instance of the interface `List` as a finite sequence, and we will use the abstract data type (ADT) **`\seq`** introduced in Section 8.1.3 for this purpose. **`\seq`** is an algebraic data type (a primitive data type in Java terms). It comprises constructors for 1. empty sequence, 2. singleton sequences, 3. sequence concatenation, 4. subsequences, and 5. comprehension, and has random access and length observer functions. See Section 5.2 for details on the corresponding theory in JavaDL.

Example 9.6. Assume that the contents of a list can be abstracted to a `\seq` representation of the list and that the entity `theList` holds this abstract value. Then we can describe the addition of an element (as new first element of the list) as a concatenation of a singleton sequence and the prestate sequence:

—— JML ——
```
/*@ public normal_behavior
  @ ensures theList ==
  @     \seq_concat(\seq_singleton(elem),\old(theList));
  @*/
public void add (int elem);
```
—— JML ——

Analogously, the other modification methods can be specified using sequence operations. The remaining question is: what type of program entity does the identifier `theList` refer to in this example? Or, how does this ADT representation integrate into the specification? One solution is to use model fields, as explained in the following.

9.2.1 Model Fields

Based on the idea of *abstract variables* by Hoare [1972], JML provides model fields [Leino and Nelson, 2002] as a means of abstraction from the concrete program state in a syntactically convenient form (i.e., as fields in a class; see also Section 7.7.1). Together with modeling-only types such as sequences, model fields allow us to give abstract interface specifications.

It is not possible to assign values to model fields, they do not have a proper, modifiable state space of their own, but they *observe* the state space spanned by heap memory by *computing* a value from values on the heap. The relation between a model field and the state, i.e., the abstraction relation, is specified through *represents clauses*. A model field can be defined by a functional relation where the field's name is followed by the assignment operator `=` and an expression compatible with the type of the model field. A more general relational form is also available and uses the keyword `\such_that` followed by a Boolean expression. Model field definitions may refer to Java entities as well as to other specification-only entities like other model fields. In general, the abstraction relation needs not be total, i.e., not every concrete object needs to possess an abstract value. For instance, the model field `x` defined via `represents x = x+1` has an empty abstraction relation; no instance has an abstract value.

We will use the term *concrete field* to denote both Java fields and JML ghost fields (see Section 7.7.2) in order to distinguish them from model fields. Even though model fields syntactically resemble concrete fields, their semantics is closer to methods. The model field declaration corresponds to a method declaration as the public interface and the represents clause corresponds to an implementation. If model fields have publicly observable properties, they need to be specified in class invariants. In contrast, ghost fields behave like ‘real’ Java fields that can only be used within the specification.

9.2.1.1 Example: List Specification with Model Fields

To specify the abstract behavior of the `List` interface, we introduce a model field `theList` as shown in Listing 9.4. It is a member of this interface and can be referred to in specifications like a concrete Java field Model fields declared in classes are instance members by default as for concrete fields. Model fields declared in interfaces are class members by default. This can be overridden by the modifier **`instance`**.

Now all operations declared on lists can be specified in terms of this model field. For instance, the postcondition to method `add()` on line 4 in Listing 9.4 states that an element is added to the front of the sequence, or more precisely that the sequence in the poststate is a concatenation of the singleton sequence containing the element with the sequence in the prestate.

Note that the two queries `get()` and `size()` which were basic operations in the query-based specification in Listing 9.3 are now also specified in terms of the model field. Thanks to the abstract data type, this specification is very concise and intuitive.

In the concrete implementations of the interface, a represents clause specifies the abstraction relation. Listing 9.5 shows an implementation of `ArrayList`, which internally uses an array to store the list elements. The `add` and `remFirst` methods increase/decrease the size every time and copy all elements one place to the right/left. This is not very efficient, but serves its demonstrative purpose here. The model field is represented by the elements of the array (as a sequence):

```
represents theList = (\seq_def int i; 0; size; a[i]);
```

Now that we have a working implementation for `List`, we can verify the methods in `ArrayList` against the abstract contracts given in `List` and linked through the represents clause for `theList`. Given adequate loop invariants for `add/remFirst`, all method implementations can be verified completely automatically with KeY. We turn our attention to the second implementation of the `List` interface through the class `LinkedList` in Listing 9.1. Here we need a *recursive* represents clause i.e., a represents clause that refers to the same model field `theList` again.

A list’s abstract value is defined by the tail’s model field’s value—or the empty sequence if `tail` is a null reference. This leads to the following represents clause in class `LinkedList`

```
represents theList = tail==null? \seq_empty: tail.theList;
```

```
public interface List {
    //@ public instance model \seq theList;

    /*@ public normal_behavior {}
      @ ensures theList == \seq_concat(\seq_singleton(elem),
      @                               \old(theList));
      @*/
    public void add (int elem);

    /*@ public normal_behavior
      @ requires !empty();
      @ ensures theList == \old(theList[1..theList.length]);
      @*/
    public void remFirst ();

    /*@ public normal_behavior
      @ ensures \result == (size() == 0);
      @*/
    public /*@ pure @*/ boolean empty ();

    /*@ public normal_behavior
      @ ensures \result == theList.length;
      @*/
    public /*@ pure @*/ int size ();

    /*@ public normal_behavior
      @ requires 0 <= idx && idx < size();
      @ ensures \result == (int)theList[idx];
      @*/
    public /*@ pure @*/ int get (int idx);
}
```

Listing 9.4 Interface specification using a model field

This is *overridden* by

—— Java + JML ——

```
class LinkedListNonEmpty {
    ⋮
   /*@ private represents theList =
     @      \seq_concat( \seq_singleton(head),
     @                   tail==null? \seq_empty: tail.theList);
     @*/
}
```

—— Java + JML ——

in the subclass `LinkedListNonEmpty`. It is here that the `head` attribute is available.

```
public final class ArrayList implements List {

    private int[] a = new int[0];
    /*@ private represents theList =
      @     (\seq_def int i; 0; a.length; a[i]);
      @*/

    public void add (int elem) {
        int[] tmp = new int[a.length+1];
        /*@ maintaining 0 <= i && i <= a.length;
          @ maintaining (\forall int j; i < j && j <= a.length;
          @                 tmp[j] == \old(a[j-1]));
          @ decreasing i;
          @ assignable tmp[*];
          @*/
        for (int i= a.length; i > 0; i--)
            tmp[i] = a[i-1];
        a = tmp;
        a[0] = elem;
    }

    public void remFirst () {
        int[] tmp = new int[a.length-1];
        /*@ maintaining 0 < i && i <= a.length;
          @ maintaining (\forall int j; 0 < j && j < i;
          @                 tmp[j-1] == \old(a[j]));
          @ decreasing a.length - i;
          @ assignable tmp[*];
          @*/
        for (int i= 1; i < a.length; i++)
            tmp[i-1] = a[i];
        a = tmp;
    }

    public boolean empty () {
        return size() == 0;
    }

    public int size () {
        return a.length;
    }

    public int get (int idx) {
        return a[idx];
    }
}
```

Listing 9.5 `ArrayList` implementation of the `List` interface

9.2.1.2 Semantics of Model Fields

The semantics of model field is given by a set of logical axioms that arise canonically from the represents clauses [Weiß, 2011]. In JavaDL terms, model fields (as well as calls to pure methods) are represented by *observer symbols* (called *location dependent symbols* by Beckert et al. [2007], Bubel [2007]). The intuition is that they 'observe' a set of locations on the heap and compute a value from them. The parameters differ between the different kinds of observer symbols; model fields have the heap and the receiver object as sole arguments[6]. Boolean model fields are translated to observer predicates, model fields of all other types become observer functions.

Definition 9.7 (Observer symbol). An *observer symbol* is either a function symbol $f : Heap^k \times T \times A_1 \times \cdots \times A_n \to A' \in \mathrm{FSym}$ or a predicate symbol $p : Heap^k \times T \times A_1 \times \cdots \times A_n \in \mathrm{PSym}$ where $T \sqsubseteq \mathrm{Object}$ and $k, n \in \mathbb{N}$, $k \geq 1$.

Observer symbols formalize heap-dependent functions, hence, all have in common that they take one or more parameters of type *Heap* .

The fundamental difference between regular Java fields and JML model fields becomes apparent when their translations to JavaDL are compared; which will be done here by an illustrative example:

Example 9.8. Consider the class fragment

```
—— Java + JML ——
class C {
  int f;
  //@ model int mf;
}
—— Java + JML ——
```

in which a regular Java field f and a model field mf are declared. If `c` is a variable of type `C`, then the field references are translated to JavaDL as follows (remember from Section 8.1 that $\lfloor\cdot\rfloor$ is the translation from JML to JavaDL):

$$\lfloor \texttt{c.f} \rfloor = select_{int}(\texttt{heap}, \lfloor \texttt{c} \rfloor, \texttt{C::f})$$
$$\lfloor \texttt{c.mf} \rfloor = \texttt{C::mf}(\texttt{heap}, \lfloor \texttt{c} \rfloor)$$

While the one access `c.f` becomes a heap read access, the other `c.mf` is translated into an application of the according observer symbol. We silently omit the class prefix from the verbose symbol names `C::f` and `C::mf` and use `f` and `mf` in the following if the context is clear.

Do not confuse concrete fields and model fields: To make proof obligations more legible in KeY, the pretty printing mechanisms are the same for both expressions, yielding the seemingly structurally equal terms $c.f$ and $c.mf$ in the logic. If the heap in which the fields are evaluated is not the default heap, but a term h, then the accesses read $c.f@h$ and $c.mf@h$.

[6] Without loss of generality, we only cover *instance* observers here.The static case is similar, only lacking the receiver object parameter.

Observer symbols are the entities that carry the value of model fields, however the value returned by them is not constrained yet. It is the represents clause for the model field which provides this constraint on the side of JML. Two types of represents clauses were introduced in Section 9.2.1, functional and relational. To ease the presentation in this section, we regard a functional clause **`represents`** `mf = ` *def* as a shorthand for the general relational form **`represents`** `mf` **`\such_that`** `mf == ` *def*. The defining represents clauses give rise to JavaDL axioms thus fixing the meaning of the observer symbols.

Definition 9.9 (Represents axiom). Let $m : Heap \times C \to A \in \mathrm{FSym}$ be an observer function[7] symbol representing a model field `m` defined in type T. Let **`represents`** *m* **`\such_that`** *rep* be the definition of m in type $T' \sqsubseteq T$. The *represents axiom* is the formula

$$\forall Heap\ h; \forall T'\ o; \big(exactInstance_{T'}(o) \doteq TRUE \to \\ \{\mathtt{self} := o \,\|\, \mathtt{heap} := h\}(\mathrm{inRange}_A(m(h,o)) \land \lfloor rep \rfloor)\big) \quad (9.1)$$

where $\mathtt{self} \in \mathrm{PVar}$ is the program variable to which **`this`** is translated in JavaDL.

The type restriction $\mathrm{inRange}_A(m(h,o))$ ensures that the model field's value is always valid and not an unallocated object or an integer out of range.

Example 9.10. Consider the class fragment of Example 9.8; now augmented by the JML clause

```
//@ represents mf = f;
```

which binds the value of the model field `mf` directly to that of the Java field f. Whenever $c.f$ changes for an object c of type C, the value $c.mf$ silently changes as well. In the logic, this coupling is fixed in a represents axiom which is equivalent to

$$\forall Heap\ h; \forall C\ c;\ exactInstance_C(c) \doteq TRUE \to mf(h,c) \doteq select_{int}(h,c,\mathtt{f}) \ .$$

This axiom cannot compromise consistency of the logic since it provides a definition for the symbol *mf* in form of a conservative extension.

In general, represents axioms need not be conservative extensions and may introduce inconsistencies. Keep in mind that with an unsatisfiable axiom any statement—even "false"—can be proved valid.

KeY provides some measures to prevent the introduction of most obvious contradictory represent axioms. As a first measure against unsatisfiability, Weiß [2011] proposes to restrict the represent axiom to situations where $\lfloor rep \rfloor$ is satisfiable, i.e., axiom (9.1) is replaced by the conditional formula

[7] If `m` is a Boolean model field, it is represented by a observer predicate symbol $m : Heap \times T \in \mathrm{PSym}$ instead of a function symbol. The definition remains the same.

$$\begin{aligned}\forall Heap\ h; \forall T'\ o; \big(& exactInstance_{T'}(o) \doteq TRUE \land \\ & (\exists R\ r; \{\mathtt{self} := o \,\|\, \mathtt{heap} := h\} \lfloor rep[\mathtt{self.m}/r] \rfloor) \rightarrow \\ & \{\mathtt{self} := o \,\|\, \mathtt{heap} := h\}(\mathrm{inRange}_A(m(h,o)) \land m(h,o) \doteq \lfloor rep \rfloor)\big) \end{aligned} \tag{9.2}$$

in which $rep[\mathtt{self.m}/r]$ denotes the represents clause in which every occurrence of the model field m is replaced by the quantified variable r of type R which is the value type of the model field. This prevents that represents clauses like `represents m \such_that m!=m` lead to immediate inconsistencies. The value of $m(h,o)$ is defined only if the represents clause is satisfiable—and it remains underspecified if rep is not satisfiable.

Not only relational represents clauses can be unsatisfiable, also a functional clause can be unsatisfiable if it employs nonprimitive recursion, like, e.g., in `represents m = m+1`. However, the guard introduced in (9.2) can only guarantee *local satisfiability* for individual clauses. It may still occur that multiple represents clauses contradict each other. In particular, a contradiction may arise from mutually recursive represents clauses. Consider, e.g., a case with two int fields x and y with the two represents clauses `represents x \such_that x > y` and `represents y \such_that y > x`, in which the values of x and y mutually depend on each other. Both are obviously satisfiable on their own, but their conjunction is not. Later in this chapter, we will explain how to deal with recursive represents clauses in order to further mitigate the issue of unsatisfiability.

If inconsistencies are brought into the system through represents clauses, this is not a soundness issue of the calculus. The axioms are part of the specification and thus reflect the intention of the specifier to whom it is up to account for his or her axioms. This is a very liberal view on the matter with one important consequence: When reviewing the specification of a program, it is vital to inspect *all* represents clauses annotated to the program since they may introduce inconsistencies into the specification rendering it entirely worthless. As we will see later in Section 9.2.2.3 a different decision has been taken for the semantics of model methods: Their (possible recursive) computation must always terminate, the functions they define are thus always consistent conservative extensions.

9.2.1.3 A Model Field for Class Invariants

There is one model field which receives special treatment and is considered "built in" by the KeY system: It is called `\inv`, of type `boolean`, and declared in class `Object`. This model field is used to model class invariants (also called object invariants, see also Section 7.4.1 on how to use them in JML). The KeY dialect of JML deliberately deviates from standard JML semantics in this respect because the model field formalization integrates better with the dynamic frames approach taken by KeY (which is explained in the upcoming Section 9.3).

Standard JML implements the *visible state semantics* for class invariants which requires that the invariants of an object o must hold in a state s if s is a poststate of a constructor call on o, if s is a pre- or poststate of a method call on o, or if no call on

o is in progress. This allows invariants to be broken temporarily, as long as a method is in the process of being executed on the object for which the invariant is broken.

The problem with this definition is that the set of methods being executed when entering `m` is not a property of `m` itself, or the current heap state. Rather, it is a property of a particular call to `m`. A modular verification attempt of `m` independently of the rest of the program is not able to know which other methods are already on the call stack when entering `m`. Thus, the only invariants that can safely be assumed to hold in the beginning are the invariants of the receiver object **`this`** which is a very weak, often insufficient assumption. This problem can be mitigated by means of combining visible states with ownership approaches, a few of which are listed in Section 9.6.

In KeY-JML specifications have to state explicitly which object invariants are expected to be satisfied using the (standard JML) keyword **`\invariant_for`**. Only the invariant for the receiver object **`this`** of a method is by default implied by both precondition and postcondition of a method (see Section 8.2.1.2).

Note that, to formulate that the invariant of object `o` holds, two equivalent notations can be used: `o.\inv` is the same as **`\invariant_for`**`(o)` (while only the latter is defined in standard JML).

All calculus rules which are applicable to model field apply to `\inv` as well. However, the definition of invariants works differently and this allows more modular inference rules. Both advanced aspects are explained in Section 9.4.5.

9.2.1.4 Calculus Rules for Model Fields

Like the other kinds of axiom, the axioms generated from represents clauses can also be expressed as rules instead of as formulas. For every class D that declares a represents clause rep for a model field m the following rule $\mathsf{rep}_{D,m}$ is available.

$$\mathsf{rep}_{D,m}\;\frac{\Gamma,\ exactInstance_D(d) \doteq TRUE,\ \{\mathtt{heap} := h \,\|\, \mathtt{self} := d\}rep \ \Longrightarrow \Delta}{\Gamma,\ exactInstance_D(d) \doteq TRUE \ \Longrightarrow \Delta}$$

Even though rule $\mathsf{rep}_{D,m}$ can be applied to any sequents with $exactInstance_D(d) \doteq TRUE$ in the antecedent, the application in KeY is triggered by an occurrence of an application $m(h,d)$ of the observer symbol m on either side of the sequent. Because the rule matches against d being an *exact* instance of class D, there is at most one applicable rule. The rule is an obvious adaptation of the represents axiom (9.3) in Definition 9.9.

For functional clauses of the form **`represents`** `m =` *def*, we can also use the conditional rewriting rule $\mathsf{repSimple}_{D,m}$

$$m(h,d) \rightsquigarrow \{\mathtt{heap} := h \,\|\, \mathtt{self} := d\}def$$
$$\text{if } exactInstance_D(d) \doteq TRUE \Longrightarrow \text{ on the sequent.}$$

The rule $\mathsf{repSimple}_{D,m}$ allows replacing references to a model field by its definition def directly. This is more efficient in practice than adding an equation to the antecedent. Applying $\mathsf{repSimple}_{D,m}$ repeatedly to a recursively defined model field would result in a infinite expansion of the proof sequent. The proof strategy in KeY is designed to apply recursive definitions very sparsely and only up to a certain depth to avoid infinite recursion expansion.

The type restriction from Definition 9.9 is not represented in these rules. Instead, there is a separate rule

$$\frac{\Gamma, \mathrm{inRange}_A(m(h,d)) \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta}\ \mathsf{OnlyCreatedObjectsObserved}$$

that can be applied whenever the observer symbol $\mathtt{m}(h,d)$ of type A appears in the sequent $\Gamma \Longrightarrow \Delta$.

KeY offers a taclet option modelFields : showSatisfiability (see p. 530 for details on taclet options) to control whether local satisfiability is to be checked upon using a represents clause. If this option is activated, the rules rep/repSimple have an additional premiss implementing the existential quantifier from (9.2). Proving local satisfiability usually makes proofs more complicated. Moreover, as mentioned earlier, local satisfiability is only an heuristic measure that cannot always guarantee consistency.

9.2.1.5 Discussion

Model fields are a powerful and often welcome specification instrument. It is however debatable whether general nonfunctional model fields may not create more problems than they solve.

For consistency one would have to prove simultaneous satisfiability of all represents clauses in the system. This is currently not enforced in KeY, it is not modular, and one may doubt whether that will ever be practical. Thus, the responsibility to work with a consistent set of axioms rests on the specifier. A theoretical alternative is presented in [Beckert and Bruns, 2012] that evaluates all model fields simultaneously and checks for *global* satisfiability of represents clauses. It avoids inconsistencies in the logic through underspecification. The practicality of this approach is under investigation. For model methods consistency must always be shown by means of termination witnesses (see Section 9.2.2.3).

In case of generalized relational **`\such_that`** represent clauses, there may be more than one possible value. But, since model fields are represented by functions in the logic, evaluation is deterministic and only depends on the heap and the receiver. to a model field. When the model field is evaluated several times in the same heap, it has the same value. In particular, classical logic equations like $m(heap,o) \doteq m(heap,o)$ are still valid. If the heap changes slightly, the model field value may be different. Dependency contract rules (see Section 9.4.4) can be used to prove that its value stays the same if it does not depend on the changed fields.

Using objects of reference type for abstract object states is problematic since they must point to objects that exist on the concrete heap. This means that represents axioms may postulate the existence of such object, which is another source of potential inconsistency.

From our experience, we recommend to use functional represents clauses only.

9.2.2 Model Methods

In JML expressions, one can not only refer to fields but also to invocations of pure methods. Moreover, JML allows the definition of model methods which are—quite analogous to model fields—method declarations to be used in specifications only. Like model fields, model methods do not reside in locations on the heap but compute a value which *depends* on the values of locations on the heap—they are observer symbols.

As pointed out by Mostowski and Ulbrich [2015, 2016], JML model methods are a generalization of JML model *fields* and go beyond them in several respects:

1. They are parametric, i.e., they can take arguments like Java methods.
2. Method contracts can be specified for model methods like for Java methods.
3. Model methods can be used to abstract from expressions which are evaluated in more than one state (so called *two-state predicates*).
4. Model methods always define conservative extensions. Their definition is given by a constructive method body, and they are required to always terminate (`diverges false`). This implies that their definitions are well-founded and no inconsistencies are introduced.

A JML model method and its contract are stated like all other JML constructs within special JML comments. A model method definition in KeY follows the following general schema (all clauses in `[...]` are optional)

```
class C {
  /*@ model_behavior[8]
    @   [requires pre;]
    @   [ensures post;]
    @   [accessible acc;]
    @   [measured_by mby;]
    @   [two_state] [no_state]
    @ model R m($T_1$ $p_1$, ..., $T_n$ $p_n$) {
    @     return exp;
    @ }
    @*/
}
```

This pattern allows only for those model methods whose body consists of a single return statement, a restriction which simplifies the treatment of a model method in the logical context as it avoids the evaluation of the method body using a JavaDL modality. Later in this section, we will see that the concept of model methods can be generalized to method bodies with real control flow, but the presentation in this section and the implementation in KeY follow the above pattern.

As for pure Java methods and for JML model fields, a model method declaration gives rise to an observer symbol (see Definition 9.7) in JavaDL. For the above declaration, a observer function symbol `C::m` is introduced to represent the model

[8] The keyword `model_behavior` is not strictly required but the specifier is encouraged to use it.

method in JavaDL. Leaving aside the two-state modifier for the moment, its signature is $\mathtt{C::m} : \mathit{Heap} \times C \times T_1 \times \ldots \times T_n \to R$.

Semantically, the evaluation of a model method invocation is coupled to the expression *exp* in the **return** statement by the following definition axiom which refines the general represents axiom (9.1) for observer symbols from Definition 9.9:

$$\begin{aligned}&\forall \mathit{Heap}\ \mathtt{heap}, C\ \mathtt{self}, T_1\ p_1, \ldots, T_n\ p_n;\\ &\qquad\qquad \mathit{exactInstance}_C(\mathtt{self}) \doteq \mathit{TRUE} \land \lfloor \mathit{pre} \rfloor \to\\ &\qquad\qquad\qquad\qquad \mathtt{C::m}(\mathtt{heap}, \mathtt{self}, p_1, \ldots, p_n) \doteq \lfloor \mathit{exp} \rfloor)\end{aligned} \tag{9.3}$$

The formula, as shown, violates the JavaDL restriction that program variables, in this case `heap` and `self`, cannot be quantified. But, to make formulas in this section more readable, we take the liberty to write $\forall \mathit{Heap}\ \mathtt{heap}; \varphi$ as an abbreviation for the formula $\forall \mathit{Heap}\ h; \{\mathtt{heap} := h\}\varphi$.

The function symbol `C::m` is determined by the class (or interface) *C* in which the method *m* has been first declared. All method definitions overriding that initial declaration refer to the same function symbol (and not to a new symbol). Constraining the same function symbol thus realizes the dynamic dispatch of model methods. That is, the function symbol is always the same, while its meaning is implied by the exact type of `self` changes.

For a subclass C' of *C* overriding *m*, another axiom of shape (9.3) is added for `C::m`, with the typing guard changed to $\mathit{exactInstance}_{C'}(c) \doteq \mathit{TRUE}$. If C' chooses not to override *m*, an axiom is added as if the definition with the body of the superclass-method had been repeated, which matches programmers' expectations as it is the same behavior as for Java method declarations. The guards $\mathit{exactInstance}_{C'}(\mathtt{self}) \doteq \mathit{TRUE}$ ensure that the definition only applies if the receiver object `self` is *exactly* of the defining type. These typing guards make sure that (possibly contradicting) definitions of `C::m` constrain different parts of its domain and that definitions are not automatically inherited. Unlike model fields, the definition of model methods may be additionally constrained by a precondition. It is not strictly necessary to restrict the domain in which C::m can be applied, but we decided that it is better to allow a specifier to say when a model method is defined. Also to deal with the well-definedness and well-foundedness (see Section 9.2.2.3), it is important to limit the definition to those situations for which it is well-defined.

Since our model method body (see above) consists only of a single side-effect-free **return** statement, definition (9.3) can make use of its expression directly. If a one-state model method *did* have a nontrivial method body, the above axiom would need to involve a dynamic logic operator and read

$$\begin{aligned}&\forall \mathit{Heap}\ \mathtt{heap}, C\ \mathtt{self}, T_1\ p_1, \ldots, T_n\ p_n;\\ &\qquad \big(\mathit{exactInstance}_C(\mathtt{self}) \doteq \mathit{TRUE} \land \lfloor \mathit{pre} \rfloor \to\\ &\qquad \big[\mathtt{res} = \mathtt{self.m}(p_1, \ldots, p_n);\big](\mathtt{C::m}(\mathtt{heap}, \mathtt{self}, p_1, \ldots, p_n) \doteq \mathtt{res})\big) \quad ,\end{aligned}$$

ensuring that the value of the function symbol is the same as the result value of the method call. This formula points out a crucial advantage of dynamic logic in comparison to other program logics like wp-calculus or Hoare calculus: dynamic logic is closed under all its operators which allows us to state the quantified program formula directly in JavaDL, and not on a meta-level. The dynamic logic thus also allows us to seamlessly extend the presented approach to nonmodel queries.

Besides its method body, a model method may also have a functional contract (in its postcondition). Unlike the body which *defines* the value of the function symbol, the contract *describes a property* of the symbol and is not an axiom, but a theorem. To establish the correctness of the contract theorem, it suffices to prove that the definition makes the postcondition true, i.e., that

$$\begin{aligned}&\forall \mathit{Heap}\;\mathtt{heap}, C\;\mathtt{self}, T_1\;p_1,\dots,T_n\;p_n;\\ &\qquad(\mathit{exactInstance}_C(\mathtt{self}) \doteq \mathit{TRUE} \land \lfloor \mathit{pre} \rfloor \rightarrow\\ &\qquad\qquad\{\mathtt{res} := \mathtt{C}{::}\mathtt{m}(\mathtt{heap},\mathtt{self},p_1,\dots,p_n)\}\lfloor \mathit{post} \rfloor)\;. \qquad (9.4)\end{aligned}$$

follows from axiom (9.3). If (9.4) is shown for every class C' extending C (with a corresponding type guard), the statement is shown for all conceivable instances of C. Therefore, when using the proved contract as additional assumption, it is save to omit the type guard $\mathit{exactInstance}_C(\mathtt{self}) \doteq \mathit{TRUE}$ from (9.4). This approach is still modular, however: The verification of C happens independently of that of its subclasses. At the time of verification, one can even be oblivious to the existence of subtypes.

The properties of model fields cannot be specified in contracts, they need to be captured in class invariants. Model method contracts have one crucial modularization advantage over formalizing properties in invariants: While the former are proved once and for all in a separate proof obligation, the latter need to be reproved whenever the invariant needs to be reestablished.

9.2.2.1 Two-State Model Methods

As has been mentioned before, the expressive power of model methods goes beyond that of Java methods and model fields in that more than one state can be referred to from a method body or contract. The number of accessible heaps is not limited in theory, but in practice three types of model methods have proved useful:

- *No-state model methods* can be used to formalize mathematical statements which are not heap-dependent at all. A query which checks if a sequence (of type `\seq`) is duplicate-free would be an example for a no-state model method. In KeY the JML modifier `no_state` can be used to mark a model method heap-independent.
- *One-state model methods* are like regular Java methods or model fields bound to a single evaluation context. This is the default if no state modifier is annotated to the method

- *Two-state model methods* are evaluated in two evaluation contexts. They are valuable where two-state predicates need to be specified which formalize the relationship between the before- and the after-state of an operation. In KeY-JML, two state. methods are annotated with the modifier **`two_state`**.

No-state model methods are not really observer functions since they do explicitly not dependent on the heap. Two-state model methods, however, are observers that receive *two* heap arguments ($k = 2$ in Definition 9.7). We show how the representation axiom (9.3) needs to be adapted to the two-state case; the other conditions are analogous.

$$\begin{aligned}&\forall \mathit{Heap}\ \mathtt{heap}, \mathit{Heap}\ \mathtt{heap}_2, C\ \mathtt{self}, T_1\ p_1, \ldots, T_n\ p_n;\\ &\qquad\qquad (\mathit{exactInstance}_C(\mathtt{self}) \doteq \mathit{TRUE} \land \lfloor \mathit{pre} \rfloor \rightarrow\\ &\qquad \mathtt{C::m}(\mathtt{heap}, \mathtt{heap}_2, \mathtt{self}, p_1, \ldots, p_n) \doteq \{\mathtt{heap}^{\mathit{pre}} := \mathtt{heap}_2\}\lfloor \mathit{exp} \rfloor)\end{aligned} \tag{9.5}$$

The second heap $\mathtt{heap}_2$ is thus automatically mapped to the prestate heap which is accessed from JML via the **`\old`** operator.

The translation of a reference to a two-state model method in JML to JavaDL remains to be defined. This extends the translation outlined in Section 8.1.2.4 such that we have for a one-state model method *osm* and a two-state model method *tsm* defined in class *C* that

$$\begin{aligned}\lfloor o.\mathtt{osm}(p_1, \ldots, p_n) \rfloor &= \mathtt{C::osm}(\mathtt{heap}, \lfloor o \rfloor, \lfloor p_1 \rfloor, \ldots, \lfloor p_n \rfloor)\\ \lfloor o.\mathtt{tsm}(p_1, \ldots, p_n) \rfloor &= \mathtt{C::tsm}(\mathtt{heap}, \mathtt{heap}^{\mathit{pre}}, \lfloor o \rfloor, \lfloor p_1 \rfloor, \ldots, \lfloor p_n \rfloor)\ .\end{aligned}$$

Note how the heap arguments need not be specified in the JML specification but are added during the translation. The second heap $\mathtt{heap}_2$ is automatically mapped to the prestate heap $\mathtt{heap}^{\mathit{pre}}$.

Example 9.11. Consider the program given in Listing 9.4 once more. To showcase a very simple application scenario for two-state model methods, assume that the specifier wants to capture the difference in the length of the abstraction between before and after an operation into a model method `sizeDiff()`:

JML
```
/*@ model two_state int sizeDiff() {
  @   return theList.length - \old(theList.length);
  @ }
  @*/

/*@ normal_behavior
  @ ensures sizeDiff() == 1;
  @*/
void add(Object o);
```
JML

```
class Cell {
  int val;

  /*@ ensures \result == val; @*/
  int /*@ pure @*/ get() {
    return val;
  }

  /*@ ensures val == v; @*/
  void set(int v) {
    val = v;
  }
}

class Client {
  /*@ ensures c.val == v; @*/
  static void callSet(Cell c,int v){
    c.set(v);
  }
}
```

```
class Recell extends Cell {
  int oval;

  /*@ ensures val == oval; @*/
  void undo() {
    val = oval;
  }

  /*@ ensures oval==\old(val); @*/
  void set(int v) {
    oval = val;
    super.set(v);
  }
}
```

Fig. 9.2 Listings of `Cell`/`Recell` example

Note how `\old` is used to refer canonically to the second heap.

Using such two-state model methods makes obviously only sense when the model method is only invoked (referred to) in places where two states are imminently present: for instance in postconditions of method contracts (but also signals clauses or history constraints)

9.2.2.2 Dynamic Dispatch for Contracts using Model Methods

The possibility to override the implementation of a method defined in a supertype is the essential polymorphism feature of the object orientation paradigm. The mechanism which chooses at runtime the implementation to be taken for a method invocation is called *dynamic dispatch*. Also in the context of design-by-contractand behavioral subtyping, different implementations for the same operation can coexist—if they adhere to a common specification. It is most natural that not only the *implementations* but also the *specifications* vary from subtype to subtype, for instance by adding implementation-dependent aspects. This dynamic dispatch mechanism should, hence, also be available for the formulation of *formal* specifications in an equally flexible way.

Instead of spelling out the definition of this specification element, it should be possible to refer to it *symbolically*. Only when the dynamic type of the object is known, one also knows the actual contract definition.

We motivate and explain our specification approach by means of a small Java example, shown in the listings in Figure 9.2. Another example (modeling the visitor pattern) and a larger case study (modeling symbolic permissions) can be found in [Mostowski and Ulbrich, 2015].

The challenge presented here has originally been proposed by Distefano and Parkinson [2008] and has been dealt with by Bengtson et al. [2011] using a higher-order separation logic. The listings in Figure 9.2 show the program annotated with traditional specification means. `Cell` objects encapsulate integer values which can be set using a method **`set`** and be retrieved using `get`. The class `Recell`, which extends `Cell`, allows an additional one level `undo` operation which restores the cell value to the state before the most recent call to **`set`**. The class `Client` provides a method `callSet` which indirectly calls the **`set`** method of the `Cell` argument it receives. This particular indirection may seem artificial, but indirection is a very natural phenomenon in object orientation, e.g., in a situation where this operation is done only conditionally or after some locks have been acquired or in combination with other operations.

The contract of `callSet` copies the postcondition of `Cell.set` literally. It does not guarantee the stronger postcondition of `Recell.set` if the argument is of type `Recell`. The present contract does not suffice to verify the following test case:

Java
```java
Recell rc = new Recell();
rc.set(4);
Client.callSet(rc, 5);
rc.undo();
assert rc.get() == 4;
```
Java

While this program would not fail its assertion, the proof for that would not succeed as the abstraction of `callSet` by its contract neglects the additional postcondition `oval == \old(val)` introduced in `Recell` and only ensures the weaker postcondition of `Cell`.

This could be amended by introducing case distinctions on the type of the argument in the postcondition of `Cell.set`. This could be achieved by an additional clause `c instanceof Recell ==> ((Recell)c).oval == \old(c.val)`s. However, it has significant limitations regarding the modularity of the specification: (1) Details on the implementation of `Recell` are revealed where it is not necessary and should be kept under the hood and, more severely, (2) the implementation of `Recell` might not yet be known at the time that `Cell` is implemented or specified. Assume `Cell` and `Client` are part of a library and `Recell` is a user-written extension. How can the library account for all potential extensions?

This is precisely where abstract predicates in the form of model methods can be used to solve the issue. In the listings of Figure 9.3, the example has been reformulated using a model method `post_set` (lines 4–8) formalizing the postcondition of the method **`set`** (used in line 15). The model method has a body which defines its value.

```
   class Cell {                              class Recell extends Cell {
2    int val;                                  int oval;

4    /*@ ensures \result ==> get()==x;         /*@ model two_state
       @ model two_state                       boolean post_set(int x) {
6    boolean post_set(int x) {                   return super.post_set(x) &&
        return val == x;                           oval == \old(get());
8    } @*/                                     } @*/

10   /*@ ensures \result == val; @*/           /*@ ensures get()==\old(oval); @*/
     int /*@ pure @*/ get() {                  void undo() {
12     return val;                               val = oval;
     }                                         }
14
     /*@ ensures post_set(v); @*/              void set(int x) {
16   void set(int v) {                           oval = get(); super.set(x);
       val = v;                                }
18   }                                       }
   }
```

Fig. 9.3 Listings of `Cell`/`Recell` example annotated with model methods

In this case, it returns true if and only if its argument `x` is equal to the value stored in field `val`. Looking at class `Cell` alone, no semantic change has been done.

Things change when the class `Recell` is again added to the scenario. In `Recell`, the model method `post_set` is overridden and adds a condition to the result obtained by `Cell.post_set`. By redefining the predicate locally for all instances of class `Recell`, the semantics of the contract `Cell.set` has now also changed, although syntactically it is the same. As the contract refers to the postcondition only *symbolically*, its semantics is left open and can be redefined by an implementing class. Furthermore, `post_set` makes use of its **`two_state`** declaration in class `Recell` as the definition relates values from two execution states, namely `\old(get())` and `oval`. The two states that this definition refers to are the pre- and poststate of the method **`set`**.

The redefinition of `post_set` in `Recell` cannot be arbitrary, however. The model method has got a contract (line 4) saying that whenever its result is true, the condition `val` == `x` needs to hold. All overriding implementations need to obey that contract, but may add to it. This ensures behavioral subtyping.

The above example test case can be proved correct if the model method invocation `c.post_set(v)` is used as postcondition for `Client.callSet` abstracting away from the actual definition of the postcondition.

Model methods can also be used to modularly specify framing conditions (using dynamic frames, which will be discussed in Section 9.3.2 below). Listing 9.6 shows the scenario including frame conditions where the frame has been abstracted by a single state model method `footprint()`.

Note how this method is used to specify the part of the heap on which the cell operates. The actual shape of this set of locations is different in the two classes; this can be addressed by giving the exact definition for `footprint()` in the corresponding classes. To provide global constraints on the footprint we can specify a contract for this model method. In the example we added an upper and a lower bound for the location set.

```
class Cell {
    int val;

    /*@ accessible \nothing;
      @ ensures \subset(\result, this.*) &&
                \subset(\singleton(this.val), \result);
    model \locset footprint() {
        return \singleton(this.val);
    } @*/

    /*@ accessible footprint();
      @ ensures \result ==> get()==x;
    model two_state boolean post_set(int x) {
        return get() == x;
    } @*/

    /*@ accessible footprint();
      @ ensures \result == val; @*/
    int /*@ pure @*/ get() { return val; }

    /*@ ensures post_set(v);
      @ assignable footprint(); @*/
    void set(int v) { val = v; }
}

class Recell extends Cell {
    int oval;

    /*@ model \locset footprint() {
        return \set_union(this.val, this.oval);
    } @*/

    /*@ model two_state
    boolean post_set(int x) {
        return super.post_set(x) &&
               oval == \old(get());
    } @*/

    /*@ ensures get() == \old(oval);
      @ assignable footprint(); @*/
    void undo() { val = oval; }

    void set(int x) {
        oval = get(); super.set(x);
    }
}
```

Listing 9.6 `Cell`/`Recell` annotated with footprint specifications

9.2.2.3 Model Methods and Termination

Showing termination for *programs* is optional; analyzing the partial correctness problem alone can be a challenge already. For the definition of model methods, however, termination is a central point that must *not* be omitted. A model method definition gives rise to a universally quantified axiom claiming that the function has certain properties even if it may be unsatisfiable. Consider for instance the problematic declaration

```
class X { /*@ model int bad() { return this.bad() + 1; } @*/ }
```

for which the model method would be translated into the axiom

$$\forall \mathit{Heap}\ \mathtt{heap}, X\ \mathtt{self}; (\mathit{exactInstance}_X(\mathtt{self}) \rightarrow \\ \mathtt{X::bad}(h, \mathtt{self}) \doteq \mathtt{X::bad}(h, \mathtt{self}) + 1)\ ,$$

which is obviously inconsistent. Consistency can be guaranteed if termination (or *well-foundedness*) of all recursive method references is checked. Here, the **measured_by** clauses are employed to avoid such unsatisfiable recursive definitions. We require that all definitions are primitive recursive. The termination witness *mby* specifies for each method a termination measurement which must be decreased in all referenced (model) method invocations in *exp*. To this end, an additional proof obligation per model method is generated to ensure this. Assuming that the termination witness of a model method referenced in *exp* is mby', it has to be shown that mby' is a strict nonnegative predecessor of *mby*, i.e., $0 \leq \mathit{mby}' < \mathit{mby}$.

In practice, one may also encounter mutually recursive definitions of model methods. In this case simple integer expressions as termination clauses are in general not sufficient. For that reason, we additionally allow tuples of integer expressions with a standard lexicographic order to serve as termination clauses and the above mechanism is modified accordingly to check the lexicographic ordering of the expressions instead, see also the last paragraph of Section 9.1.4 on page 300. Furthermore, to weaken the resulting proof obligations, we use left-to-right evaluation similar to that of well-definedness checking described in Section 8.3.2. Thus, expressions in return statements only need to decrease the termination witness if a prefixing guard is true.

9.3 The Frame Problem

For *modular* static verification, where we assume that the program may be extended, even the goal to check the correctness of individual program parts locally—that is, without considering the program as a whole—puts higher demands both on specifications and on the specification language itself than for approaches working under a closed program assumption. This sets modular verification apart from runtime checking. JML pledged to satisfy the additional demands of modular verification, but the classical static frame annotations fall short of this goal. Weiß [2011] presents a solution with an extension to the framing concept of JML, based on the *dynamic frames* approach by Kassios [2006, 2011]. Dynamic frames is a flexible approach for framing in the presence of dynamic data structures and data abstraction. Compared with alternative solutions, such as data groups [Leino, 1998], the advantages of this approach are its simplicity and generality.

9.3.1 Motivation

In object-oriented programming, data is organized in pointer-based data structures in which references from one object reach out to other objects in the memory, thus combining individual parts of the memory to complex compound data networks. The structure built up by the references is usually not limited by the programming language. In particular, the Java programming language does not pose any restrictions (other than by its type system) on how objects may refer to one another. There are many reasons to employ references between objects: They may point to objects which constitute a separate subpart of a larger structure. References may be used for efficiency reasons like in caches, to point into areas which are shared between various components. One direct consequence of the ability to have arbitrary pointer chains leading from one to another object is that effects of a piece of code cannot be assumed to be local to some object. The code may follow references on the heap and may potentially modify parts of the memory which are seemingly ‘far away’ from the original starting point. If an item is added to a collection that is kept as a heap data structure, for instance, it seems natural to assume that the content of a second, different list, would not be affected by such an action. But there are implementations which deliberately share data between collection instances to save memory. A glitch in such an implementation may indeed result in the modification of more objects than intended and their independence may not always silently be assumed.

It is thus an obligation of formal specification and verification to name the places in memory to which a piece of code has read or write access. An alternative to stating which part of the memory a program may look at or modify is to explicitly state what a program must *not* touch. While this seems like a viable alternative at first glance, it bears many issues concerning modularity: A specification cannot be local since it would have to include that a very distant part of the memory remains unchanged by the code. It may also not be open to extensions of the program because a specification cannot possibly talk about memory entities which are only to be included in an extension of the program.

We will now demonstrate these general concerns by an example. We consider in Listing 9.7 a simple client to our running `List` class example. A client object holds references to two list instances. The `m()` method adds an element to one of them. The question is how to prove the postcondition that states that the other list has not changed in size. We have to add the precondition that `a` and `b` do not *alias*, otherwise the postcondition could never be valid.

```
class Client {
    List a, b;

    //@ requires a != b;
    //@ ensures b.size() == \old(b.size());
    void m() { a.add(23); }
}
```

Listing 9.7 Client code using two instances of the `List` interface (from Listing 9.9)

As we have seen above in Section 9.1.2, a correct implementation of `add()` must satisfy the postcondition that the passed element has been added to the list. This is an impartial description of the method's behavior. For our particular situation here, however, we aim for the property that `a.add()` does *not* do anything harmful to `b`—that, besides the given functional property, "nothing else changes" [Borgida et al., 1995]. Such a property is usually expressed as a set of locations to which the method may write at most, called the *frame* of the method and a set of locations on which the result of a query depends at most, called the *footprint*.

Listing 9.8 shows the client specification with framing. One problem we encounter when trying to specify its frame, is that we need to address the concrete locations on which the method depends and to which it writes. This means that we have to expose the nature of the contained list, i.e., which implementation of the `List` interface is used. Here, we chose the `LinkedList` implementation and fixed it using a class invariant. Its **`accessible`** clause defines the footprint, i.e., the program locations it reads and on which its functionality depends. The **`accessible`** clause defines the locations that might be changed by the method.

```
class Client {
    List a, b;
    //@ invariant a instanceof LinkedList && b instanceof LinkedList;

    //@ requires a != b;
    //@ requires ((LinkedList)a).tail != ((LinkedList)b).tail;
    //@ ensures b.size() == \old(b.size());
    //@ accessible a, ((LinkedList)a).tail;
    //@ assignable ((LinkedList)a).tail;
    void m() { a.add(23); }
}
```

Listing 9.8 Client code using framing (exposing implementation details)

But what if we do want to keep the nature of the used list open? We answer this question in the following section by providing a solution on how to frames elegantly and without exposing (or even fixing) implementation details.

9.3.2 Dynamic Frames

The *dynamic frame* theory [Kassios, 2011] aims at solving the frame problem in the presence of data abstraction. The essence of the dynamic frames approach is to leverage the ubiquitous location sets to first-class citizens of the specification language: specification expressions are enabled to talk about such location sets directly. In particular, this allows us to explicitly specify that two such sets do not overlap, or that a particular concrete location is not part of a particular set. This is an important property for pointer-based programs, which is called the absence of *abstract aliasing* (also known as *deep aliasing*) [Leino and Nelson, 2002, Kassios, 2006]. For example, this property is what is missing in the specification of Listing 9.7. The knowledge that the location sets represented by `a.footprint` and `b.footprint` are disjoint allows us to conclude that the postcondition is actually satisfied.

What is called a *dynamic frame* is an abstract set of locations. A dynamic frame is 'dynamic' in the sense that the set of locations to which it evaluates can change during program execution, just like the value of a model field can change.

Dynamic Frames in JML

Weiß [2011] presented an implementation of the dynamic frames approach in KeY, using an extension of JML that includes high-level specification elements for location set expressions. The type **`\locset`** has already been briefly introduced in Chapter 8, with the underlying JavaDL data type introduced in Section 2.4. Semantically, expressions of type **`\locset`** stand for sets of memory locations. These expressions replace the *store ref* expressions from the JML reference manual [Leavens et al., 2013]as the expressions that are used to write **`assignable`** and **`accessible`** clauses. The primary difference between store ref expressions and **`\locset`** expressions is that **`\locset`** is a proper type. This for example allows us to declare model and ghost fields of this type.

The singleton set consisting of the (Java or ghost) field `f` of the reference expression *o* can be denoted in JML as **`\singleton`**`(`*o*`.f)`, and the singleton set consisting of the *i*-th component of the array reference *a* as **`\singleton`**`(`*a*`[`*i*`])`. The set consisting of a range of array components and the set consisting of all components of an array are written as *a*`[`*i*`..`*j*`]` and *a*`[*]`, and the set of all fields of an object is written as *o*`.*`. The keywords **`\nothing`** and **`\everything`** refer to the empty set and the set of all locations, respectively.More precisely, `\everything` refers to the set of all locations belonging to created objects. In the same spirit, `\nothing` is used to denote the set of locations that belong to freshly allocated objects. Intuitively, they denote the set of 'observably all' locations and 'observably none.' The actual empty set (in the mathematical sense) is denoted by **`\strictly_nothing`** similar to the difference between the **`pure`** and **`strictly_pure`** annotations.

In addition, JML features the following basic set operations on expressions of type **`\locset`**, with the standard mathematical meaning: the set intersection **`\intersect`**, the set difference `\set_minus`, the set union `\set_union`, the subset predicate `\subset`, and the disjointness predicate `\disjoint`.

The notations `o.f` and `a[i]` can be used as short-hands for the singleton sets `\singleton(o.f)` and `\singleton(a[i])`, but only in contexts where understanding them as representing the *value* of `o.f` or `a[i]` is syntactically forbidden. For example, on the top level of a modifies clause, the expression `o.f` is equivalent to `\singleton(o.f)` if `f` is a Java or ghost field of type `int`, but it denotes the value of the field if the `f` is of type `\locset`. As another familiar shorthand, a comma separated list s_1 `, ... ,` s_n can be used to abbreviate the union of the `\locset` expressions s_i where this does not lead to syntactical ambiguity.

Frames and Dependencies

Depend clauses have already been the topic of Subsection 8.3.2 and Definition 8.3 with the emphasis on their representation in JavaDL and the translation from JML. Here, we place dependency contracts in the context of modular verification.

While depends clauses for pure methods are already part of standard JML, we generalize the mechanism of depends clauses to model fields here. A depends clause for a model field is declared as a class member, using the syntax `accessible m: f;` where `m` is a model field (defined for the class containing the depends clause) and where `f` is an expression of type `\locset`. Such a depends clause means that `m` may depend at most on the locations in `f` (in other words, '`f` frames `m`'), provided that the invariants of the `this` object hold in the current state. More formally, `accessible m: f` is true in a state s if any state change (starting in s) that preserves the values of the locations in the evaluation of `f` in s also preserves the value of `m`. This is a contract that all represents clauses for `m` must satisfy (in the current class or interface and in its subclasses), just like a depends clause for a pure method is a contractual obligation on all implementations of the method. As dynamic frames may be model fields themselves, they may also occur on the right hand side of a depends clause. It is a common pattern for a dynamic frame to frame itself: `accessible f: f` means that, if the values of the *locations* in the value of `f` are not changed, then the value of `f` *itself* also remains the same.

We extend the `\fresh` operator so that it can be applied to location sets, in addition to applying it to objects. An expression `\fresh(`f`)`, where f is an expression of type `\locset`, is satisfied in a postcondition if and only if all the locations in the poststate interpretation of f belong to an object that was not yet allocated in the prestate. More formally, it is $\lfloor\texttt{\textbackslash fresh(}f\texttt{)}\rfloor = \mathit{subset}(\lfloor f\rfloor, \mathit{unusedLocs}(\mathtt{heap}^{pre}))$; see Figure 2.11 for the semantics of *unusedLocs*.

The so-called *swinging pivots* operator `\new_elems_fresh` can be applied to a dynamic frame f within a postcondition. The meaning of `\new_elems_fresh(`f`)` is that if there are any locations in the set f in the poststate that have not been there in the prestate, then these must belong to objects that have been freshly allocated in between (in the sense of `\fresh`). It is thus equivalent to the expression `\fresh(\set_minus(`f`,\old(`f`)))`. Intuitively, a swinging pivot indicates a change on the heap that is benign—in the sense that no previous separation properties can be invalidated. In combination with `assignable` and `accessible` clauses, the swinging pivots operator is useful to specify preservation of the absence of abstract aliasing. For example, if for some method execution we know that

1. the dynamic frames f and g do not contain any unallocated locations in the prestate,
2. f and g are disjoint in the prestate,
3. g frames itself in the prestate (`accessible` g`:`g),
4. only the values of the locations in f may be different in the poststate (i.e., `assignable` f), and that
5. the modification respects `\new_elems_fresh(`f`)`,

then we can conclude that f and g are still disjoint in the poststate. The reasoning behind this is as follows: `assignable` f and the disjointness of f and g together imply that the values of the locations in g are not changed. Combined with g being self-framing, this implies that the location set referred to by g itself also remains the same. The set f may change, but `\new_elems_fresh(`f`)` guarantees that if this change adds to f any additional locations, then these locations were previously unallocated. As the set g is unchanged and did not contain any unallocated locations in the prestate, the locations added to f cannot be members of g, and so the sets must still be disjoint. We see a concrete application of this chain of reasoning in Section 9.3.4.

9.3.3 Proof Obligations for Dynamic Frames Specifications

Section 8.3.2 presented a proof obligation which, if proven valid, ensures that the dependency contract of a pure method is correct. The formula presented in Definition 8.5 uses modalities for the evaluation of the method under examination.

In the previous section, we showed how to specify dynamic frames for model fields using `accessible` clauses, and Section 9.2.2 showed that model methods can also be specified with such clauses. But the proof obligation in Definition 8.5 from Chapter 8 cannot be applied in this situation: The rule embeds the method call into a JavaDL modality which is not possible for model fields and model methods.

To this end, we generalize this proof obligation to one which applies to arbitrary observer symbols.

Definition 9.12 (Proof obligation for dependency contracts of observers). Given an observer symbol $obs : Heap \times E \times T_1 \times \ldots \times T_n$ together with its dependency contract $(pre, term, dep)$. The definition of obs is called correct w.r.t. the dependency contract if the following JavaDL formula

$$\begin{aligned} & pre \land freePre \land wellFormed(h) \land \mathtt{mby} \doteq term \\ & \quad \rightarrow \quad obs(\mathtt{heap}, \mathtt{self}, \mathtt{p}_1, \ldots, \mathtt{p}_n) \\ & \qquad \equiv \; obs(anon(\mathtt{heap}, setMinus(allLocs, dep), h), \mathtt{self}, \mathtt{p}_1, \ldots, \mathtt{p}_n) \end{aligned}$$

is valid for the fresh constant $h : Heap$ and parameter variables $\mathtt{p}_1 : T_1, \ldots, \mathtt{p}_n : T_n$. The symbol $\equiv$ is interpreted as $\doteq$ if *obs* is a function symbol and as $\leftrightarrow$ if it is a predicate symbol.

For a model field, there are no arguments to the observer (i.e., $n = 0$), but a model method may possess additional arguments besides the receiver `self`.

9.3.4 Example Specification with Dynamic Frames

A version of the `List` interface from Section 9.1.2 this time specified using dynamic frames, is shown in Listing 9.9. As in Listing 9.3, the specification of the interface is based on the pure methods `get()` and `size()`. It also includes a dynamic frame `footprint`, that abstracts from the concrete memory locations that represent the list in possible subclasses. This dynamic frame is used in the modifies clause of the `add()` method, and in the depends clauses of the pure methods of `List`. In lines 3 and 6 of Listing 9.9, depends clauses are additionally given for the model fields `footprint` and (implicitly) `\inv`: their values, too, may depend at most on the locations in `footprint`.

As none of the methods of `List` are annotated as **`helper`** methods, all contracts contain implicit pre- and postconditions that assert that **`\invariant_for`**`(`**`this`**`)` is true before and after method execution. No other objects have to satisfy their invariants before calling the methods of the interface.

The additional postcondition for `add()` in line 12 demands that, even though the set `footprint` may change, all locations that are added to it must be fresh. This grants an implementation of `add()` the license to discard old data structures in `footprint` and to add fresh ones as needed. The same holds for `remFirst()`, where the footprint is even strictly smaller in the poststate. For the other methods of `List`, there is no need for a postcondition that describes their effect on `footprint`. Roughly, this is because these methods are pure, and thus we expect that they cannot affect `footprint` at all. This expectation is correct, but the precise justification for this is more complex than it may seem at first sight, because pure methods *are* allowed to allocate and initialize new objects, and because without further knowledge, such a state change *might* affect the interpretation of a model field such as `footprint`. Fortunately, the semantics of JML guarantees that dynamic frames like `footprint` never contain any unallocated locations. We know from the depends clause in line 6 that `footprint` frames itself, i.e., that a change to locations that are not in the value of `footprint` cannot affect the value of `footprint`. Thus, any change to previously unallocated locations in a pure method is guaranteed to leave the value of `footprint` untouched.

```
public interface List {
    //@ public model instance \locset footprint;
    //@ public accessible footprint: footprint;

    //@ public instance invariant size() >= 0;
    //@ public instance accessible \inv: footprint;

    /*@ public normal_behavior
      @ ensures size() == \old(size()) + 1 && get(size()-1) == elem;
      @ ensures (\forall int i;0 <= i &&
      @   i < size()-1;get(i) == \old(get(i)));
      @ ensures \new_elems_fresh(footprint);
      @ assignable footprint;
      @*/
    public void add (int elem);

    /*@ public normal_behavior
      @ requires !empty();
      @ ensures size() == \old(size()) - 1;
      @ ensures (\forall int i;0 <= i &&
      @   i < size();get(i) == \old(get(i+1)));
      @ ensures \new_elems_fresh(footprint);
      @ assignable footprint;
      @*/
    public void remFirst ();

    /*@ public normal_behavior
      @ ensures \result == (size() == 0);
      @ accessible footprint;
      @*/
    public /*@ pure @*/ boolean empty ();

    /*@ public normal_behavior
      @ ensures \result == size();
      @ accessible footprint;
      @*/
    public /*@ pure @*/ int size ();

    /*@ public normal_behavior
      @ requires 0 <= idx && idx < size();
      @ ensures \result == get(idx);
      @ accessible footprint;
      @*/
    public /*@ pure @*/ int get (int idx);
}
```

Listing 9.9 Interface `List` specification using pure methods and a dynamic frame `footprint`

9.3.4.1 Specifying the List Client

Listing 9.10 shows the `Client` class from Listing 9.7 with dynamic frame specifications. We have only inserted the additional preconditions in lines 7f.: when entering method `m()` of class `Client`, the invariants of `a` and `b` must hold, and there must not be abstract aliasing between `a.footprint` and `b.footprint`. Given this specification, we are now able to conclude that the postcondition in Line 9 holds, by using only the code and specifications in Listings 9.9 and 9.10.

```
1  class Client {
2      List a, b;
3      static int x;
4
5      /*@ normal_behavior
6        @ requires a != b;
7        @ requires \invariant_for(a) && \invariant_for(b);
8        @ requires \disjoint(a.footprint, b.footprint);
9        @ ensures b.size() == \old(b.size());
10       @ ensures \invariant_for(a) && \invariant_for(b);
11       @*/
12     void m() { a.add(23); }
13 }
```

Listing 9.10 The client from Listing 9.9 specified with dynamic frames

We reach this conclusion as follows. The disjointness of `a.footprint` and `b.footprint` implies that there is no abstract aliasing between `a` and `b` before calling `a.add()`. Thus, the depends clause of `size()` guarantees that changing the locations *in the prestate value* of `a.footprint` would not affect `b.size()`. But calling `a.add()` may have an effect on the model field `a.footprint` itself. But we know that `a.footprint` is only changed in an benign way; this is what the swinging pivots predicate states in line 12 of Listing 9.9. From this we can deduce that both footprints are still disjoint in the poststate. Overall, we can conclude that the postcondition in line 9 holds.

Analogously, the depends clause for the class invariant in `List` guarantees that **`\invariant_for`**`(b)` still holds after the change, as asserted in line 10. Listing 9.10. This property holds independently of the concrete implementations of `List` that may occur as the dynamic type of `List`, as long as all these implementations satisfy the specifications given in the interface.

9.3.4.2 Specifying the `ArrayList` Implementation

A particular implementation of the `List` interface is shown in Listing 9.11, which already appeared earlier. We have now added specifications based on dynamic frames and made the default constructor explicit. The contents of the dynamic frame `footprint` are defined for objects of dynamic type `ArrayList` through the represents clause in line 3. This represents clause satisfies the depends clause for `footprint` in Listing 9.9, because all locations that its right hand side depends on are themselves part of the right hand side. If we would omit `a` in the represents clause, then the depends clause would be violated: the location `this.a` would then not be a member of the value of `this.footprint`, but changing the value of this location would still affect the value of the expression `this.a[*]` and thereby the value of `this.footprint`.

The **`invariant`** declarations of `ArrayList` (the implicit clause `this.a!=null` plus the one inherited from `List`) define the represents clause for the implicit mode field `\inv`. This represents clause satisfies the depends clause for `\inv` specified in Listing 9.9, because it only accesses locations that are part of `footprint` as defined in the applicable represents clause for `footprint`. We do not consider `a.length` to be a location here, because it is unmodifiable.

Line 9 of Listing 9.11 gives a postcondition `\fresh(footprint)` for the constructor of `ArrayList`. This postcondition is satisfied by the implementation of the constructor: in deviation from the JML reference manual [Leavens et al., 2013], the `this` object is considered to be fresh in the postcondition of a constructor,[9] and consequently the location `this.a` is also fresh. By the represents clause of `footprint`, its other members are the locations of the array that is stored in `a`. This array is freshly allocated.

9.4 Calculus Rules for Modular Reasoning

In Chapter 3, an extensive sequent calculus for JavaDL has been introduced, and Section 3.7 gives a brief introduction to the concept of abstraction and presents rules that deal with the two kinds of abstraction relevant for our purposes, loop invariants and method contracts. In this section, we present advanced rules that go beyond those shown in Chapter 3. We begin with invariant rules for loops in Section 9.4.2 and a contract-rule for method calls in Section 9.4.3. These rules incorporate the concept of dynamic frames outlined in Section 9.3.2. Another central rule for frame-aware reasoning are dependency rules which allow deducing if two applications of an observer symbol have the same value by inspection of their footprints. JavaDL dependency contracts have been introduced in Section 8.2.4, according proof obligations in Section 8.3.2. This chapter will present in Section 9.4.4 rules that *use* proved dependency contracts to infer that observer invocations must have the same value. Finally rules will be stated that allow the expansion of model field and method definitions in Section 9.4.5.

[9] This means, the constructor contract is considered a contract for the entire `new` call, that includes object allocation, initialization, and the constructor; see Section 3.6.6.

```
public final class ArrayList implements List {

    //@ private represents footprint = a, a[*];

    private int[] a;

    /*@ public normal_behavior
      @ ensures size() == 0;
      @ ensures \fresh(footprint);
      @*/
    public /*@ pure @*/ ArrayList() {
        a = new int[0];
    }

    public void add (int elem) {
        int[] tmp = new int[a.length+1];
        for (int i= a.length; i > 0; i--)
            tmp[i] = a[i-1];
        a = tmp;
        a[0] = elem;
    }

    public void remFirst () {
        int[] tmp = new int[a.length-1];
        for (int i= 1; i < a.length; i++)
            tmp[i-1] = a[i];
        a = tmp;
    }

    public /*@ strictly_pure @*/ boolean empty () {
        return size() == 0;
    }

    public /*@ strictly_pure @*/ int size () {
        return a.length;
    }

    public /*@ strictly_pure @*/ int get (int idx) {
        return a[idx];
    }
}
```

Listing 9.11 Java class `ArrayList` implementing the `List` interface of Listing 9.9

9.4.1 Anonymizing Updates

When modeling abstraction, it is important that the concrete memory state at a point during execution can be replaced with a fresh unconstrained state. This is needed in particular when dealing with unbounded loops or with method invocations—both of which are abstraction by means of an overapproximation (the contract or the loop invariant).

In order to be able to continue execution with “any value for x satisfying the invariant,” for instance, we need to forget the value of x and assume then the invariant holds. This is done by assigning an unconstrained new value to x in an update. Harel et al. [2000] suggest to incorporate the notation $\mathtt{x} := ?$ into dynamic logic with the semantics $[\mathtt{x} := ?]\varphi \leftrightarrow \forall \mathtt{x}; \varphi$ for such a forgetting assignment. In JavaDL, we employ updates and assign to x a fresh unconstrained Skolem constant x' (of the same type as x) that may hold any value. We call such updates *anonymizing updates*. They are also called *random assignment* or *wildcard assignments* in literature.

Heap anonymization, i.e., anonymization of the program variable heap is particularly interesting in the face of dynamic frames: In the abstraction rules in Chapter 3, we treated heap like any other program variable and anonymized it with a fresh Skolem variable h'. But having dynamic frames at hand, we can do better now and only assign fresh values to those locations inside a frame, leaving all locations outside the frame untouched.

To this end, we use the function $\mathit{anon} : \mathit{Heap} \times \mathit{LocSet} \times \mathit{Heap} \to \mathit{Heap}$ (whose semantics was introduced in Figure 2.11 in Section 2.4.5) which does precisely that. The heap update

$$\{\mathtt{heap} := \mathit{anon}(\mathtt{heap}, \mathit{mod}, h')\}$$

ensures that in its scope, the heap coincides with $h' : \mathit{Heap}$ on all locations in *mod* and all not yet created locations and coincides with heap before the update elsewhere.

9.4.2 An Improved Loop Invariant Rule

Other parts of this book describe how a JavaDL loop specification is obtained: Section 16.3 provides guidelines for the user to find useful loop invariants, Section 7.9.2 explains how loop specifications can be formulated in JML and Section 8.2.5 describes how JavaDL loop specification are obtained from the JML specifications.

Here, we assume that a JavaDL loop specification $(\mathit{inv}, \mathit{mod}, \mathit{term})$ according to Definition 3.23 is given with loop invariant *inv*, modifier set *mod* and termination witness *term*. A first rule for dealing with JavaDL loop specification has already been presented in Section 3.7.2 ignoring the *mod* and *term* parts. Here we will remedy this omission.

The general structure of a loop invariant rule looks like this:

$$\frac{\begin{array}{ll}\text{Invariant Initially Valid} & (\mathsf{INIT})\\ \text{Body Preserves Invariant} & (\mathsf{STEP})\\ \text{Use Case} & (\mathsf{USE})\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\pi\ \mathtt{while}(\mathit{se})\mathit{p};\ \omega]\varphi, \Delta}$$

We assume that the loop condition *se* is a simple expression and that the loop body p always terminates normally. How to deal with the general case, that *se* may not be simple and p may contain return, continue, or break statements if explained in Subsections 3.7.2.3 and 3.7.2.4.

We will comment in detail on the three premisses:

1. In the *base case* INIT, it is to be proved that the invariant holds in the initial state of the loop execution;
2. in the *step case* STEP, it is to be proven that execution of the loop body p in a state which satisfies the loop invariant reestablishes the invariant; if $term \neq$ PARTIAL then it has also to be shown that the termination witness *term* strictly decreases;
3. in the *use case* USE we may assume that the invariant holds after the loop has finished and continue symbolic execution with the remainder program ω.

and end up with the rules loopInvariant and termLoopInvariant in Definitions 9.13 and 9.14.

The base case requires that the invariant is true in the current context spanned by Γ, Δ and $\mathcal{U}$.

$$\Gamma \Longrightarrow \mathcal{U} inv,\ \Delta \qquad \text{(INIT)}$$

The information about the execution context encoded in the update $\mathcal{U}$ and the formula sets Γ and Δ is retrieved by matching the calculus rule against the a sequent it is applied to.

The step and use cases are to be proved in symbolic states where an arbitrary number of loop iterations have already been executed, potentially invalidating all information in the context. The necessary *masking* of the context can be formalized by anonymizing as introduced in the last section. This led to the introduction of the anonymizing update $\mathcal{V}$ in the simple loop invariant rule in Section 3.7.2. For the convenience of the reader we repeat its step case

$$\Gamma \Longrightarrow \mathcal{U}\mathcal{V}\big(inv \wedge se \doteq TRUE \rightarrow [p]inv\big), \Delta \ . \qquad \text{(STEP}_0\text{)}$$

In this condition, $\mathcal{V}$ anonymizes the variable heap and all local variables which are potentially modified in the loop body p. As far as the heap is concerned, this is a very coarse approximation because *all* locations on the heap are assigned a fresh unconstrained value. This implies a burden for the specifying person as he or she must encode into the loop invariant which memory locations the loop does *not* change. Therefore, we will now go one step further and incorporate the modifier set of the loop specification into the rule to limit the anonymization of the heap.

Remember that the loop specification contains a modifier set term $mod : LocSet \cup \{\text{STRICTLYNOTHING}\}$ which models the locations which can be modified by the loop. If $mod \neq \text{STRICTLYNOTHING}$, we replace the coarse anonymizing update $\mathcal{V}$ in (STEP$_0$) by the more precise anonymizing update

$$\mathcal{W} := \{\mathtt{heap} := anon(\mathtt{heap}, mod, h') \,\|\, \mathtt{b}_1 := b'_1 \,\|\, \ldots \,\|\, \mathtt{b}_n := b'_n\} \qquad (9.6)$$

in which $\mathtt{b}_1, \ldots, \mathtt{b}_n$ enumerate the local variables that can be modified by the loop body, and $b'_1, \ldots, b'_n$ are fresh anonymization constants of appropriate type. The heap is partially anonymized with the fresh values taken from a fresh unconstrained heap object $h' : Heap$.

The set of locations affected by the anonymizing update cannot be statically determined, it is the value of the term *mod* which determines the extension of the location set. This is why the frames featuring in this approach are called *dynamic frames*: The location sets may differ between different states.

Let us have a closer look at the case $mod = \emptyset$. Unraveling the semantics definition of *anon* from Figure 2.11, we see that (despite *mod* being empty) all fields of objects not yet created in `heap` are anonymized. This models that in the code block abstracted by the anonymization, new objects may be created. This may put a considerable burden on the verification. In case it is known that the code block does not change anything and does not create new objects, the according **`assignable`** clause of the method contract or loop specification may be set to **`\strictly_nothing`** which will be translated to the special indicator $mod = \textsc{strictlyNothing}$. The corresponding update then is

$$\mathscr{W} := \{\mathtt{b}_1 := b_1' \parallel \ldots \parallel \mathtt{b}_n := b_n'\}$$

which is (9.6) without the assignment to `heap`.

On the other hand, no matter what locations occur in *mod* the semantics definition of *anon* guarantees that no created object may be deleted.

The loop specification $(inv, mod, term)$ guarantees that after an arbitrary number of loop iterations at most the locations in *mod* have changed. We have exploited this fact in the anonymizing update $\mathscr{W}$ just described. On the other hand, we have to prove that after the next loop iteration still at most the location in *mod* may change. To this end, we add the formula *frame* to the postcondition in the step case premiss. We have encountered *frame* already in (8.5) in Section 8.3.1 when the method contract proof obligation was presented. It serves the same purpose there and here: to ensure that at most the locations in M are modified:

$$\begin{aligned} frame(M) := \forall o \forall f;\quad & o.created@\mathtt{heap}^{pre} \doteq FALSE \\ & \vee\, o.f \doteq o.f@\mathtt{heap}^{pre} \\ & \vee\, (o,f) \in M \end{aligned} \tag{9.7}$$

Since the modifies clause *mod* is to be evaluated in the state *before* the loop execution and not in the current state, mod cannot be used directly. Instead, a new program variable $M : LocSet$ is introduced that captures the modifies set in the prestate by means of an update $\mathscr{M} := \{M := mod\}$

At this intermediate point the step case thus reads:

$$\Gamma \Longrightarrow \mathscr{U}\mathscr{M}\mathscr{W}\big(inv \wedge se \doteq TRUE \rightarrow [p](inv \wedge frame(M))\big),\ \Delta \qquad (\mathsf{STEP}_1)$$

The corresponding use case that goes with this step case is the same as the one already introduced with the simple rule in Chapter 3—but with the refined anonymizing update $\mathscr{W}$ that only masks out *mod*.

$$\Gamma \Longrightarrow \mathscr{U}\mathscr{W}\big(inv \wedge se \doteq FALSE \rightarrow [\pi\ \omega]\varphi\big),\ \Delta \qquad (\mathsf{USE}_1)$$

One more addition is needed, concerning what we might call *free invariants* in parallel to the *free preconditions* explained in Section 8.2.4. These are well-formedness statements that we need not verify since they are automatically maintained by the semantics of the Java language. But it is helpful, and in many cases vital, to have them explicitly available at the beginning of each loop iteration. For the local variables $\mathtt{a}_1, \dots, \mathtt{a}_m$ in whose scopes the loop lies we define:

$$
\mathit{locVarInRange} := \bigwedge_{i=1}^{m} \begin{cases} \mathtt{a}_i \doteq \mathtt{null} \vee \mathtt{a}.\mathit{created} \doteq \mathit{TRUE} & \\ & \text{if } \mathtt{a}_i \text{ is of reference type} \\ \mathit{inInt}(\mathtt{a}_i) & \text{if } \mathtt{a}_i \text{ is of type } \mathtt{int} \\ \mathit{inByte}(\mathtt{a}_i) & \text{if } \mathtt{a}_i \text{ is of type } \mathtt{byte} \\ \vdots & \text{likewise for } \mathtt{short}\text{, } \mathtt{long}\text{, } \mathtt{char} \\ \mathit{disjoint}(\mathtt{a}_i, \mathit{unusedLocs}(\mathtt{heap})) & \\ & \text{if } \mathtt{a}_i \text{ is a ghost variable of type } \mathit{LocSet} \\ \mathit{true} & \text{otherwise} \end{cases} \tag{9.8}
$$

This definition of *locVarInRange* parallels (8.3) where the same is assumed about method arguments in a method invocation.

locVarInRange formalizes that all variables $\mathtt{a}_i$ must have reachable values, i.e., they must not refer to noncreated objects, their value must be in range, and they must not hold location sets that contain locations belonging to noncreated objects. We add *locVarInRange* to the INIT premiss to be shown next to the invariant. If this property holds in the initial state of the loop, then the semantics of Java guarantees that it is preserved by arbitrary loop iterations. It may thus be used as an assumption in the second and third premiss.

We have now assembled all we need to formulate the loop invariant rules. We mention once again that the rules are presented under the assumption that the loop condition is a simple expression, and that loop body does not throw exceptions and does not use `return`, `break` and `continue` statements. The rule can be extended to handle these technicalities in the same way as in Section 3.7.2.

Since the sequent context $(\Gamma, \Delta, \mathscr{U})$ is maintained by this invariant rule, we omit it in the following rule schema as explained in Section 3.5.1.

Definition 9.13 (Rule loopInvariant **without Termination).** Let $(\mathit{inv}, \mathit{mod}, \mathit{term})$ be a loop specification (see Definition 3.23) with $\mathit{term} = \text{PARTIAL}$, se a simple expression (see Table 3.2) and p_{norm} a program fragment (see Definition 3.2) that does never throw an exception and does not contain break, continue, return statements.

The rule loopInvariant is defined as

$$
\frac{\begin{array}{l} \Longrightarrow \mathit{inv} \wedge \mathit{wellFormed}(\mathtt{heap}) \wedge \mathit{locVarInRange} \\ \Longrightarrow \mathscr{M}\mathscr{W}\big(\mathit{inv} \wedge \mathit{wellFormed}(h') \wedge \mathit{locVarInRange} \wedge \mathit{se} \doteq \mathit{TRUE} \rightarrow \\ \qquad\qquad\qquad [p_{norm}](\mathit{inv} \wedge \mathit{frame})\big) \\ \Longrightarrow \mathscr{W}\big(\mathit{inv} \wedge \mathit{wellFormed}(h') \wedge \mathit{locVarInRange} \wedge \mathit{se} \doteq \mathit{FALSE} \rightarrow [\pi\ \omega]\varphi\big) \end{array}}{\Longrightarrow [\pi\ \mathtt{while(}\mathit{se}\mathtt{)\ \{\ }p_{norm}\mathtt{\ \}}\ \omega]\varphi}
$$

where:
- *locVarInRange* is defined in (9.8),
- *frame* is defined in (9.7),
- $\mathtt{a}_1, \ldots, \mathtt{a}_m \in \mathrm{ProgVSym}$ are the local program variables in whose scopes the loop lies (except for `heap`),
- $\mathtt{b}_1, \ldots, \mathtt{b}_n \in \mathrm{ProgVSym}$ are the program variables that are potentially modified by the loop body p (except for `heap`),
- h', $b'_1, \ldots, b'_n$ are fresh constant symbols of appropriate type,
- $\mathscr{M} = \{M := mod\}$,
- $\mathscr{W} = \begin{cases} \{\mathtt{b}_1 := b'_1 \parallel \ldots \parallel \mathtt{b}_n := b_n\} & \text{if } mod = \textsc{strictlyNothing} \\ \{\mathtt{heap} := anon(\mathtt{heap}, mod, h') \parallel \mathtt{b}_1 := b'_1 \parallel \ldots \parallel \mathtt{b}_n := b'_n\} & \text{otherwise.} \end{cases}$

For the heap, assuming $wellFormed(\mathtt{heap})$ in the scope of the update $\mathscr{W}$ would amount to assuming $wellFormed(anon(\mathtt{heap}, mod, h))$. Assuming $wellFormed(h')$ is shorter and simpler, in particular because this term does not depend on `heap`.

We have not considered termination so far. Rule loopInvariant is one for the ‘box’ modality. In the corresponding invariant rule for the ‘diamond’ modality, we are required to ensure that the loop terminates. This incorporates two things: (1) every loop iteration terminates, and (2) there is no program execution with infinitely many loop iterations. The first goal can be ensures by using the diamond modality in the step case and the second is established through a well-founded relation and the *term* component of the loop specification. The well-founded relation $\prec : Any \times Any$ has already been introduced in Section 9.1.4 and can be reused here. If every loop iteration makes the variant term smaller, no infinite repetitions are possible.

Definition 9.14 (Rule termLoopInvariant**).** Let $(inv, mod, \mathrm{term})$ be a loop specification (see Definition 3.23) with $term \neq \textsc{partial}$, *se* a simple expression (see Table 3.2) and p_{norm} a program fragment (see Definition 3.2) that does never throw an exception and does not contain break, continue, return statements.

The rule termLoopInvariant is defined as

$$\frac{\begin{array}{l}\Longrightarrow inv \land wellFormed(\mathtt{heap}) \land locVarInRange \\ \Longrightarrow \mathscr{T}\mathscr{M}\mathscr{W}\big(inv \land wellFormed(h') \land locVarInRange \land se \doteq TRUE \rightarrow \\ \qquad\qquad \langle p_{norm}\rangle(inv \land frame \land term \prec term^{\mathrm{pre}})\big) \\ \Longrightarrow \mathscr{W}\big(inv \land wellFormed(h') \land locVarInRange \land se \doteq FALSE \rightarrow \langle\pi\ \omega\rangle\varphi\big)\end{array}}{\Longrightarrow \langle\pi\ \mathtt{while}(se)\ \{\ p_{norm}\ \}\ \omega\rangle\varphi}$$

where:
- all conditions from Definition 6 apply,
- $term^{\mathrm{pre}} : Any$ is a fresh program variable,
- $\mathscr{T} = \{term^{\mathrm{pre}} := term\}$ is the update that stores the value of *term* before the loop body into variable $term^{\mathrm{pre}}$.

One may wonder why in Definitions 9.13 and 9.14 $wellFormed(\mathtt{heap})$ has been added to the proof obligations in the INIT case. After all this can never be violated by a Java program. The explanation is that a user might inadvertently produce a proof state e.g., by the cut rule, where $\mathit{wellFormed}(\mathtt{heap})$ might not hold.

Lemma 9.15 below establishes a formal connection between the two framing formalisms using *frame* and *anon* respectively. This connection is the reason why it is admissible to use *anon* for anonymizing the locations in the modifies clause, while using *frame* in the proof obligation for verifying the correctness of the modifies clause in the second premiss.

Lemma 9.15 (Connection between *frame* and *anon*). *Let $mod \in \mathrm{Trm}_{LocSet}$ and frame be as in* (9.7).

Let furthermore $\mathit{noDeallocs}(h_1, h_2)$ be the formula

$$\begin{aligned} &\mathit{unusedLocs}(h_2) \dot{\subseteq} \mathit{unusedLocs}(h_1) \\ \wedge\, &\mathit{select}_{Any}(h_1, \mathtt{null}, \mathit{created}) \doteq \mathit{select}_{Any}(h_2, \mathtt{null}, \mathit{created}) \quad . \end{aligned}$$

Then the following holds:

$$\begin{aligned} \models \quad & (\mathit{frame}(mod) \wedge \mathit{noDeallocs}(\mathtt{heap}^{pre}, \mathtt{heap})) \\ \leftrightarrow\ & \mathtt{heap} \doteq \mathit{anon}(\mathtt{heap}^{pre}, \{\mathtt{heap} := \mathtt{heap}^{pre}\}mod, \mathtt{heap}) \end{aligned}$$

A proof of Lemma 9.15, an easy comparison of the semantic definition of *anon* and the *frame* formula—though with many case distinctions—can be found in [Weiß, 2011, Appendix A.5]. Roughly speaking, the lemma gives a necessary and sufficient condition for the equation $h_2 = \mathit{anon}(h_1, M, h_2)$. This equation describes a situation where for all locations that h_2 does not overwrite h_2 and h_1 coincide.

The formula $\mathit{noDeallocs}(\mathtt{heap}^{pre}, \mathtt{heap})$ expresses that all objects created in the prestate heap array are still created in the current heap array (and that createdness of *null* does not change). Obviously, this is a very essential property of the Java memory model. The impossibility of deallocating created objects is also built into the semantics definition of *anon*, see Figure 2.11. Lemma 9.15 is a main ingredient in the proof of Theorem 9.17, which establishes that the loop invariant rules are sound.

But why are there two mechanisms for formalizing the framing condition in the first place? One is used where framing needs to be shown, and the other one is used in the use case. The loop invariant rules uses both mechanisms. The reason is that having an explicit function symbol to refer to the updated state allows us to formulate heap anonymization as an update.

The following lemma shows that both notions are semantically equivalent: heap anonymization using *anon* is as good as total anonymization together with assuming the condition *frame*.

Lemma 9.16. *Let noDeallocs be like in Lemma 9.15, frame as defined in* (9.7) *and $\varphi \in \mathrm{DLFml}$ a formula in which M does not occur. Then the following formula is universally valid.*

$$\begin{array}{ll} & \big(\forall \mathit{Heap}\ h; (\mathit{noDeallocs}(\mathtt{heap}, h) \rightarrow \\ & \qquad \{\mathtt{heap}^{pre} := \mathtt{heap} \,\|\, \mathtt{heap} := \mathit{anon}(\mathtt{heap}, \mathit{mod}, h)\}\varphi)\big) \\ \leftrightarrow & \big(\forall \mathit{Heap}\ h; (\mathit{noDeallocs}(\mathtt{heap}, h) \rightarrow \\ & \qquad \{\mathtt{heap}^{pre} := \mathtt{heap} \,\|\, \mathtt{heap} := h \,\|\, M := \mathit{mod}\}(\mathit{frame}(M) \rightarrow \varphi))\big) \end{array}$$

We have formalized this lemma as a JavaDL formula and proved it using the KeY system.

Theorem 9.17. *Rule* termLoopInvariant *is sound.*

A variant of Theorem 9.17 for the 'box' modality is proven by Weiß [2011, Appendix A.6].

9.4.3 A Rule for Method Contracts

In Section 8.3.1 we came across proof obligation formulas whose validity implies the correctness of a method contract. In this section, we will encounter rules which make use of method contracts essentially by abstracting away from the method invocation by assuming its contract's postcondition instead.

These two concepts go hand in glove: The rule useMethodContract shown in the following is sound if the corresponding method contract proof obligation is a valid formula.

A rule that makes use of a functional method contract is defined in Definition 9.18 below. We show the general case of a nonstatic method whose return type is not **`void`**. The rules for `void` or static methods are similar, but lack the assignment to `x` or the references to `self` and *se*, respectively. The presented rule is a refinement of the rule simpleContract presented in Section 3.7.1. It incorporates the issues of framing and termination which had been factored out in Chapter 3.

Like there, the rule makes a few assumptions about the receiver and the arguments of the method call: They are assumed to be simple expressions (see Table 3.2 for a listing of simple expressions), requiring no further symbolic execution. The symbolic execution rules methodCallUnfoldTarget and methodCallUnfoldArguments establishing this property have been presented in Section 3.6.5.4.

Definition 9.18 (Rule useMethodContract**).** Let $R\ \ m(T_1\ \ p_1,\ \ldots,\ T_n\ \ p_n)$ be a method defined in class or interface C, $se \in \mathrm{DLTrm}_{C'}$ a simple expression of type C', $a_1 \in \mathrm{DLTrm}_{T_1}, \ldots, a_n \in \mathrm{DLTrm}_{T_n}$ simple expressions, $\mathtt{x} : R$ a program variable. Let $(\mathit{pre}, \mathit{post}, \mathit{mod}, \mathit{term})$ be a functional method contract for m stated in a class C'' such that $C' \sqsubseteq C'' \sqsubseteq C$ with $\mathit{term} \neq \textsc{partial}$.

The rule useMethodContractTotal is defined as follows:

$$\frac{\begin{array}{l}\Longrightarrow \mathcal{V}(pre \land wellFormed(\texttt{heap}) \land paramsInRange)\\ \Longrightarrow \mathcal{V}(term \prec \texttt{mby})\\ \Longrightarrow \mathcal{V}(\texttt{self} \neq \texttt{null} \land \texttt{self}.created \doteq TRUE)\\ \Longrightarrow \mathcal{V}\mathcal{W}(post \land wellFormed(h) \land reachableRes \land \texttt{exc} \doteq \texttt{null} \rightarrow\\ \qquad\qquad \langle \pi\ \texttt{x=res;}\ \omega\rangle \varphi)\\ \Longrightarrow \mathcal{V}\mathcal{W}(post \land wellFormed(h) \land reachableRes \land \texttt{exc} \neq \texttt{null} \rightarrow\\ \qquad\qquad \langle \pi\ \texttt{throw exc;}\ \omega\rangle \varphi)\end{array}}{\Longrightarrow \langle \pi\ \texttt{x = } se\texttt{.m(}a_1,\ldots,a_m\texttt{);}\ \omega\rangle \varphi}$$

where:

- $paramsInRange \in$ DLFml is defined in (8.3),
- $reachableRes = \text{inRange}_R(\texttt{res}) \land \text{inRange}_{\texttt{Throwable}}(\texttt{exc})$,
- $\mathcal{V} = \{\texttt{self} := se \parallel \texttt{p}_1 := a_1 \parallel \ldots \parallel \texttt{p}_n := a_n\}$,
- $\mathcal{W} = \{\texttt{heap}^{pre} := \texttt{heap} \parallel \texttt{heap} := anon(\texttt{heap}, mod, h) \parallel \texttt{res} := r \parallel \texttt{exc} := e\}$ is an anoymizing update with $h : Heap, r : R, e :$ Throwable $\in$ FSym fresh symbols. If $mod =$ STRICTLYNOTHING, then the heap content is not modified by the method, and the assignment to heap is removed.

The formulas *paramsInRange* and *reachableRes* play the same roles as the formula *locVarInRange* in the loopInvariant rule of Definition 9.13. Similar to that, the update $\mathcal{W}$ anonymizes the locations that may be changed by the call to m by setting them to unknown values with the help of the fresh constant symbol h. It also sets the variables res and exc to unknown values denoted by the fresh constant symbols r and e, respectively. As before, an empty modifies clause still gives rise to anonymization. Specifying the method as strictly pure, however, leads to the update $\mathcal{W}$ leaving the heap untouched. The update $\mathcal{V}$ instantiates the variables used in the contract with the corresponding terms in the method call statement.

In the first premiss, the precondition has to be established. According to our understanding of a contract this is a necessary requirement to use the postcondition as an approximation of the method call. In addition the proof obligations *wellFormed*(heap) and *paramsInRange* have to been dispatched. The reason is the same as in Rule 9.13: to save-guard against inadvertent violation of these conditions by the user, e.g., by the cut rule.

Termination is addressed in the second premiss; the termination witness of the called method must be smaller than the termination witness of the current method context stored in the program variable mby (which is set in the correctness proof obligations, see Definition 8.4). The next premiss requires establishing that the receiver object *se* is created and different from null.

Unlike the rule methodContractPartial presented in Figure 3.7 in Section 3.7.1 handling the case of a null receiver by throwing an exception, this rule strictly requires nonnull receiver and is thus weaker but proves way more efficient in practice. There is a taclet option to control which behavior is taken.

The last two cases effect that method invocation is replaced by using the postcondition. In the fourth premiss, the method call is replaced by an assignment of the result to x, under the assumption that no exception has been thrown ($\texttt{exc} \doteq \texttt{null}$). If the call raises an exception ($\texttt{exc} \neq \texttt{null}$) in the last premiss, the control flow of the program continues with raising this exception (see Section 3.6.7). In both cases, the control flow continues in the context of $\pi \ldots \omega$. Unlike the postcondition *post* the formulas $\mathit{wellFormed}(h)$ and *reachableRes* need not be proved. The semantics of Java guarantees that they are true after termination of any program.

In the KeY implementation, the first two premisses have been combined into one.

Theorem 9.19. *Rule* useMethodContract *is sound, provided that for all subtypes $C' \sqsubseteq C$ of the type C in which method m has been declared, the proof obligation for functional correctness from Definition 8.4 is valid.*

A proof of Theorem 9.19 can be found in [Weiß, 2011, Appendix A.7]. The proof is similar to the proof of Theorem 9.17 in many respects. In particular, it also makes use of Lemma 9.15, which states that the heap is unaffected in all locations outside *mod* if and only if frame condition (9.7) is valid for *mod*.

Using contracts for constructors works essentially the same as in Definition 9.18, except that (1) the first active statement in the conclusion is a constructor invocation of the form `x = new C(`$a'_1, \ldots, a'_m$`);` (2) the propositions about `self` in the first premiss are omitted, (3) in the update $\mathcal{V}$ the subupdate $\mathtt{self} := se$ is replaced with $\mathtt{self} := x$, and (4) the second premiss contains an additional assumption besides $\mathit{post} \land \mathit{wellFormed}(h) \land \mathit{reachableRes}$, namely the formula (9.9) below, which states (i) that the dynamic type of the created object is `C`, (ii) that the object was not created previously, and (iii) that it is created in the current heap.

$$\begin{aligned} &\mathit{exactInstance}_{\mathtt{C}}(\mathtt{x}) \doteq \mathit{TRUE} \\ \land\, &\mathtt{x}.\mathit{created}@\mathtt{heap}^{pre} \doteq \mathit{FALSE} \\ \land\, &\mathtt{x}.\mathit{created} \doteq \mathit{TRUE} \end{aligned} \tag{9.9}$$

When a method contract is attached to a constructor, the subject of this *constructor contract* is the entire object allocation and initialization, see Section 3.6.6. This means, it refers to an allocation statement of the form `new C(...)`. It does not constrain the behavior of nested constructor invocations via `this(...);` nor `super(...);` statements. For this reason, there are no contracts available for calls to `this()` or `super()`.

Using the contract of a recursive method *mr* is in no way different from using the contract of a nonrecursive method. This is, however, not true when in the course of proving the contract of *mr* this contract is used for one of the recursive calls. The KeY system will detect this circularity and only allow it if the contract contains a `measured_by` clause. See Section 9.1.4 for details on dealing with recursion.

Traditionally, the concept of a contract applies to methods (and constructors) only, which represent natural software modules. However, the concept can be used to modularize the target program further by providing a contract to an arbitrary code block within a method body. Rule useBlockContract in Section 13.5.1.3 is an adaptation of the method contract rule.

9.4.4 A Rule for Dependency Contracts

Dually to method contracts which describe the effects of a method, *dependency contracts* describe what *affects* the value of observer expressions. The concept of a JavaDL dependency contract $(pre, term, dep)$ has already been introduced in Definition 8.3, its associated correctness proof obligation in Section 8.3.2. Intuitively, this formula establishes that under assumption of a precondition *pre*, the value of an observer depends at most on the locations in the location set *dep*. Recursive definitions for *dep* are allowed. In this case the termination witness *term* is used to provide well-foundedness of the definition.

In this section, we show how dependency contracts can be used to show that observer terms are equal even if examined in different heap contexts. In contrast to useMethodContract, the rule useDependencyContract is applied on a term or formula in the logic, not on a program modality with a method call as the active statement.

The underlying logical idea behind the dependency contract boils down to the following implication which should give you an intuition of its semantics.

$$\begin{gathered} \mathit{frame}(\overline{\mathit{dep}}) \wedge \{\mathtt{heap} := \mathtt{heap}^{\mathit{pre}}\}\mathit{pre} \wedge \mathit{well} \wedge \mathit{noDeallocs}(\mathtt{heap}^{\mathit{pre}}, \mathtt{heap}) \rightarrow \\ \mathit{obs}(\mathtt{heap}^{\mathit{pre}}, \mathtt{p}_1, \ldots, \mathtt{p}_n) \doteq \mathit{obs}(\mathtt{heap}, \mathtt{p}_1, \ldots, \mathtt{p}_n) \end{gathered} \tag{9.10}$$

The implication states that an observer symbol *obs* yields the same value if evaluated in two heaps $\mathtt{heap}^{\mathit{pre}}$ and `heap` if the two heaps and the arguments $\mathtt{p}_i$ of the observer satisfy the following conditions in the premiss of (9.10):

1. $\mathtt{heap}^{\mathit{pre}}$ and `heap` must coincide on the dependency set *dep*, see (9.11) below,
2. the precondition *pre* of the observer must be satisfied,
3. the two heaps and all arguments must be well-formed, see (9.12) below,
4. there is no deallocation; all objects allocated in $\mathtt{heap}^{\mathit{pre}}$ are still allocated in `heap`.

We have encountered the formula *frame* which captures the equality of the locations in *dep* already in (9.7). We need to formalize here that everything but the locations in *dep* may change, hence we use the complement $\overline{\mathit{dep}}$ of *dep*:

$$\begin{aligned} \mathit{frame}(\overline{\mathit{dep}}) = \forall o \forall f; (\quad & o.\mathit{created}@\mathtt{heap}^{\mathit{pre}} \doteq \mathit{FALSE} \\ & \vee o.f \doteq o.f@\mathtt{heap}^{\mathit{pre}} \\ & \vee \neg(o,f) \in \{\mathtt{heap} := \mathtt{heap}^{\mathit{pre}}\}\mathit{dep}) \end{aligned} \tag{9.11}$$

The well-formedness condition *well* includes the two heaps and all parameters and reuses the predicate *paramsInRange* introduced in (8.3):

$$\mathit{well} = \mathit{wellFormed}(\mathtt{heap}^{\mathit{pre}}) \wedge \mathit{wellFormed}(\mathtt{heap}) \wedge \mathit{paramsInRange} \tag{9.12}$$

The property *noDeallocs* that no objects are ever deleted from the heap has been introduced in Lemma 9.15.

We now introduce the rule useDependencyContract which formalizes the informal semantics explanation outlined in (9.10). It adds an assumption to the sequent that is itself an implication with left-hand side *guard*. The right-hand side relates the value of an observer symbol *obs* for the heaps denoted by the terms $h^{pre}, h^{post} \in \mathrm{Trm}_{Heap}$, where h^{post} results from h^{pre} through a cascade of applications of the function symbols *store*, *create*, and *anon*. Such cascades are the result of symbolic execution of heap manipulating programs with successive update simplification. It is instructive to compare the rule useDependencyContract with the proof obligation of Definition 8.5.

Definition 9.20 (Rule useDependencyContract**).**

$$\frac{\Gamma, guard \rightarrow obs(h^{pre}, o, a_1, \ldots, a_n) \equiv obs(h^{post}, o, a_1, \ldots, a_n) \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta}$$

where:

- $obs \in \mathrm{FSym}$ (or $obs \in \mathrm{PSym}$) is an observer symbol $obs : Heap \times E \times T_1 \times \ldots \times T_n \rightarrow T$ (or $obs : Heap \times E \times T_1 \times \ldots \times T_n$, respectively) with $n \in \mathbb{N}$
- $(pre, dep, term)$ is a dependency contract for *obs*,
- $o \in \mathrm{Trm}_E, a_1 \in \mathrm{Trm}_{T_1}, \ldots, a_n \in \mathrm{Trm}_{T_n}$ are valid arguments for *obs*.
- $h^{post} = f_k(f_{k-1}(\ldots(f_1(h^{pre}, \ldots))))$ with $f_1, \ldots, f_k \in \{store, create, anon\}$
- $\equiv$ stands for $\doteq$ if $obs \in \mathrm{FSym}$ and for $\leftrightarrow$ if $obs \in \mathrm{PSym}$
- *guard* is the formula

$$\begin{aligned}
&\mathscr{P}\{\mathtt{heap}^{pre} := h^{pre} \,\|\, \mathtt{heap} := h^{post}\} frame(\overline{dep}) \\
&\wedge \mathscr{P}\{\mathtt{heap} := h^{pre}\} pre \\
&\wedge wellFormed(h^{pre}) \wedge wellFormed(h^{post}) \\
&\wedge \mathscr{P}\, paramsInRange \wedge o \not\doteq \mathtt{null} \wedge o.created \doteq TRUE
\end{aligned}$$

  in which the update $\mathscr{P} = \{\mathtt{p}_1 := a_1 \,\|\, \ldots \,\|\, \mathtt{p}_n := a_n\}$ assigns the concrete arguments to the formal parameters of *obs*.

Besides the property that only certain locations change, the equality of the observer applications in (9.10) requires the heap evolution does not deallocate previously created objects; as for instance formalized in Lemma 9.15. For the state change from h^{pre} to h^{post}, the absence of deallocations is guaranteed by the fact that the latter is derived from the former by invocations of the function *store*, *create* and *anon*. Their semantics ensure that no object is ever deallocated. This is formalized in the following lemma.

Lemma 9.21 (No deallocations). *Let* $h^{post} \in \mathrm{Trm}_{Heap}$ *with*

$$h^{post} = f_k(f_{k-1}(\ldots(f_1(h^{pre}, \ldots))))$$

for some $f_1, \ldots, f_k \in \{store, create, anon\}$ *with* $1 \leq k$ *and for some* $h^{pre} \in \mathrm{Trm}_{Heap}$. *Then the following holds:*

$$\models noDeallocs(h^{pre}, h^{post})$$

The lemma—is also needed for the proof of Theorem 9.22 below—is the reason why *noDeallocs* can be excluded from the condition *guard* in Definition 9.20.

Theorem 9.22. *Rule* useDependencyContract *is sound, provided that for all subtypes $E' \sqsubseteq E$ of the static receiver type E of obs, the proof obligation for dependency contracts from Definition 8.5 for pure methods and the obligation from Definition 9.12 for general observer symbols, respectively, is valid.*

Proofs for this theorem and for Lemma 9.21 can be found in [Weiß, 2011, Appendix A].

This is plausible since the proof obligation for dependency contracts (Definition 8.5) expresses that the observer *obs* does not depend on locations outside *dep*. If the formula *guard* in (9.12) is valid, i.e., the difference between the heaps h^{pre} and h^{post} lies only in *dep*, then we can conclude that the value of *obs* is the same for both heaps.

Automatic application of the useDependencyContract rule is not as straightforward as for other rules. The rule can be applied to many different combinations of h^{pre} and h^{post} which increases the search space considerably. To avoid a large number of 'unsuccessful' applications where *guard* cannot be proven and where the application thus does not contribute to the proof, a strategy that proves to work well in practice is to apply the rule only lazily (once all other means of advancing the proof have been exhausted), and only for choices of h^{pre} that already occur on the sequent. Best results for an automatic rule application are obtained whenever h^{pre} is a constant and appears in an equation together with h^{post} in the antecedent.

An application of the useDependencyContract rule will be demonstrated in the course of verifying the List example in Section 9.5.

9.4.5 Rules for Class Invariants

As outlined in Section 9.2.1.3, in JavaDL, class invariants are realized by means of a special model field `\inv` whose counterpart in JavaDL is the implicit observer symbol `Object`::*inv* ∈ PSym.

The definition of the class invariant of a type T collects all class invariant declarations in T and the public invariants of T's supertypes, their combination is essentially the represents clauses for the model field `\inv`. If more than one object invariant declaration is relevant, e.g., **`invariant`** `e`$_1$`;` … **`invariant`** `e`$_n$, the collection of the individual invariant declarations stands for a single represents clause

$$\texttt{represents \textbackslash inv = e}_1 \texttt{ \&\& } \ldots \texttt{ \&\& e}_n \texttt{ .} \tag{9.13}$$

This represents clause defines the meaning of the observer `Object`::*inv* : *Heap* × `Object` for objects exactly of type T. The rules for expanding represents clauses (rep$_{D,m}$ from Section 9.2.1.4) and for dependency contracts (useDependencyContract from the last section) can be used in proofs like for any other model field.

One property sets class invariants aside from arbitrary model fields: While for general model fields, the definition may change arbitrarily in subclasses, public class invariants are inherited to subtypes according to the principle of behavioral subtyping. The invariant can only be changed by adding further clauses. The implicit represents clause from (9.13) thus enumerates all clauses in the current class and all clauses inherited from its super types.

The following $\mathsf{classInv}_T^j$ rule allows inferring an individual invariant clause **`invariant`** $\mathtt{e}_j$ present in type T if the invariant $\mathtt{Object}::\mathit{inv}(h,o)$ is known to hold for the object $o \in \mathrm{Trm}_T$.

$$\frac{\mathtt{Object}::\mathit{inv}(h,o),\ \{\mathtt{heap} := h \,\|\, \mathtt{self} := o\}\lfloor \mathtt{e}_j \rfloor \implies}{\mathtt{Object}::\mathit{inv}(h,o) \implies}\ \mathsf{classInv}_T^j$$

Note that this rule[10] can only be applied if $\mathtt{Object}::\mathit{inv}(h,o)$ occurs in the antecedent, i.e., under the assumption that the invariant holds. It can be applied for any object $o \in \mathrm{Trm}_T$ of type T also if it belongs to one of T's subtypes. Unlike the represents axiom $\mathsf{rep}_{D,m}$, it does not require that o is exactly of type T.

Rule $\mathsf{classInv}_T^j$ only adds a *consequence* of the invariant to the sequent, not its definition. The entire invariant can only be soundly added when the dynamic type T of the 'receiver' object o is known. In these cases, the rewrite rule $\mathsf{repSimple}_{D,m}$ for represents clauses can be used to replace an invariant by its definition:

$$\mathtt{Object}::\mathit{inv}(h,o) \rightsquigarrow \bigwedge_{j=1}^{n} \lfloor \mathtt{e}_j \rfloor$$

$$\text{if } \mathit{exactInstance}_T(o) \doteq \mathit{TRUE} \text{ occurs in the antecedent}$$

In many cases, in particular when conducting modular proofs, the definition of the invariant cannot be fully expanded because its actual definition is unknown to the current context. When modularly reasoning that an invariant still holds after a modification of the heap, dependency contracts can be valuable. When both the formulas $\mathtt{Object}::\mathit{inv}(h,o)$ and $\mathtt{Object}::\mathit{inv}(h',o)$ appear in the sequent, rule $\mathsf{useDependencyContract}$ can be applied to reduce one to the other.

Example 9.23. Let us turn back to the `List` interface outline in Listing 9.9 whose sole class invariant in line 5 states that the size of a list is nonnegative. Assume we have a program variable `al` of type `ArrayList` (Listing 9.11), and we know that the invariant for the list `al` is satisfied, i.e., that $\mathtt{Object}::\mathit{inv}(\mathtt{heap},\mathtt{al})$ is true.

Then the rule $\mathsf{classInv}^1_{\mathtt{List}}$ allows us to deduce that $\mathtt{List}::\mathtt{size}(\mathtt{heap},\mathtt{al}) \geq 0$ since the invariant clause is inherited from `List` to `ArrayList`.

However, to establish that the invariant for `al` holds, it does not suffice to show this property. The array list class has an additional (implicit) invariant $\neg\mathtt{self.a} \doteq \mathtt{null}$ which also needs to be proved. If $\mathit{exactInstance}_{\mathtt{ArrayList}}(\mathtt{al}) \doteq \mathit{TRUE}$ is known, then these two properties make up the definition of the invariant.

When reasoning modularly, on the other hand, there might exist a further subtype of `ArrayList` (which is not declared **`final`**) which has an invariant definition which

[10] The actual rule name used in the KeY prover fits the template $\mathsf{Partial\_inv\_axiom\_for\_JML\_class\_invariant\_nr\_}j\mathsf{\_in\_}T$.

differs from the one in `ArrayList`. This prohibits the calculus from replacing the invariant symbol by the collection of known invariant clauses—there might be more, yet unknown, clauses.

There are few situations in which a specifier wants to constrain implementation details to the enclosing class. In such cases, a class invariant declaration intentionally should not be inherited by the refining subclasses. To distinguish between class invariant declarations subject to inheritance and local declarations, the former can be declared as **`public invariant`** and the latter as **`private invariant`**. By default, class invariants are private.

9.5 Verifying the List Example

This section is a continuation of Section 9.3.4. We assume a list implementation according to the structure shown in the class diagram in Figure 9.1. The program that we consider contains the interface `List` from Listing 9.9 annotated with dynamic frames and a class `Client`. In this modular proof scenario, we do not consider specific implementations of the `List` interface, such as `LinkedList` or `ArrayList`, that were presented above. All reasoning can be based on the interface specification alone. As an example for the verification of JML specifications with dynamic frames, we consider a proof for the method `m` of the `Client` class:

```
/*@ normal_behavior
  @ requires \invariant_for(list);
  @ requires \disjoint(list.footprint, ((Client)null).*);
  @ requires 0 < list.size();
  @*/
static void m(List list) {
    x++;
    list.get(0);
}
```

The JML method contract is translated to a JavaDL method contract where the precondition *pre*, the postcondition *post* and the modifies clause *mod* are:

$$\begin{aligned} \mathit{pre} &= \mathtt{list}.\mathit{inv} \wedge \mathit{disjoint}\big(\mathtt{list.footprint}, \mathit{allFields}((\mathtt{Client})\mathtt{null})\big) \\ &\quad \wedge 0 < \mathtt{list.size()} \wedge \mathtt{list} \not\doteq \mathtt{null} \\ \mathit{post} &= \mathtt{exc} \doteq \mathtt{null} \\ \mathit{mod} &= \mathit{allLocs} \setminus \mathit{unusedLocs}(\mathtt{heap}) \end{aligned}$$

To ease the presentation of the more bulky formulas of the concrete example, we employ a few self-explanatory abbreviations in this subsection and write $\in, \cap, \setminus, \ldots$ instead of *elementOf*, *union*, *setMinus*, ...

The corresponding proof obligation from Definition 8.4 is:

$$\begin{aligned}
&\texttt{list}.\mathit{inv} \land \mathit{disjoint}\big(\texttt{list.footprint}, \mathit{allFields}((\texttt{Client})\texttt{null})\big)\\
&\land 0 < \texttt{list.size()} \land \texttt{list} \not\doteq \texttt{null}\\
&\land \mathit{wellFormed}(\texttt{heap}) \land (\texttt{list} \doteq \texttt{null} \lor \texttt{list}.\mathit{created} \doteq \mathit{TRUE})\\
&\to \{\texttt{heap}^{pre} := \texttt{heap}\}\langle\texttt{exc = null;}\\
&\qquad\qquad\qquad\qquad \texttt{try \{ self.m(list); \}}\\
&\qquad\qquad\qquad\qquad \texttt{catch(Exception e) \{ exc = e; \}}\rangle\\
&\quad\big(\texttt{exc} \doteq \texttt{null}\\
&\quad\land \forall \mathit{Object}\, o; \forall \mathit{Field}\, f;\\
&\qquad ((o,f) \in \{\texttt{heap} := \texttt{heap}^{pre}\}(\mathit{allLocs} \setminus \mathit{unusedLocs}(\texttt{heap}))\\
&\qquad\qquad \cup\ \mathit{unusedLocs}(\texttt{heap}^{pre})\\
&\qquad \lor \mathit{select}_{Any}(\texttt{heap}, o, f) \doteq \mathit{select}_{Any}(\texttt{heap}^{pre}, o, f))))\big)
\end{aligned} \tag{9.14}$$

Note that the method does not return a value, and that thus the assignment of the returned value to the program variable `res` is omitted. The following invariant axiom rule of Section 9.4.5 is visible when proving the validity of formula (9.14)

- The object invariant declaration '`public invariant 0 <= size()`' in `List` gives rise to an inv axiom for *inv* on objects of type `List`, as discussed in Section 9.4.5 and in particular in Example 9.23:

$$\frac{\Gamma,\ \mathit{inv}(h, \mathit{list}),\ \{\texttt{heap} := h \,\|\, \texttt{self} := \mathit{list}\}\big(0 \le \texttt{self.size()}\big) \implies \Delta}{\Gamma,\ \mathit{inv}(h, \mathit{list}) \implies \Delta}$$

  where *list* is a placeholder for a term of type `List` (or of a subtype). The axiom is visible in the context of `Client` because of the `public` visibility of the underlying invariant declaration.

The structure of a proof for the proof obligation is shown in Figure 9.4. Starting from the root sequent '$\implies \mathit{formula}(9.14)$,' the first steps are simplifying the sequent and applying nonsplitting first-order rules (indicated as 'FOL' in the figure), which leads to the following sequent:

$$\begin{aligned}
&\left.\begin{aligned}
&\texttt{list}.\mathit{inv},\\
&\mathit{allFields}(\texttt{null}) \cap \texttt{list.footprint} \doteq \mathit{empty},\\
&0 < \texttt{list.size()},\\
&\mathit{wellFormed}(\texttt{heap}),\\
&\texttt{list}.\mathit{created} \doteq \mathit{TRUE},\\
&\texttt{self}.\mathit{created} \doteq \mathit{TRUE}
\end{aligned}\right\}\Gamma\\
&\implies\\
&\texttt{list} \doteq \texttt{null},\\
&\langle\texttt{exc = null;}\\
&\ \texttt{try \{ Client.m(list); \}}\\
&\ \texttt{catch(Exception e) \{ exc = e; \}}\rangle(\texttt{exc} \doteq \texttt{null})
\end{aligned}$$

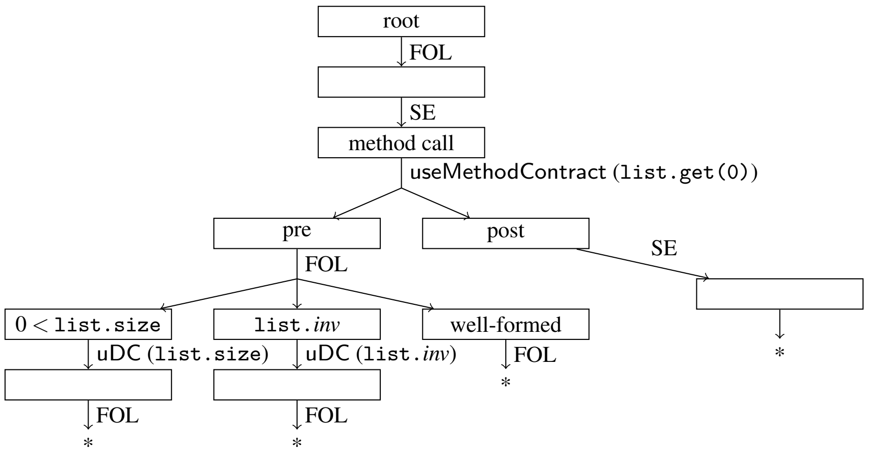


Fig. 9.4 Structure of proof for the method contract of method `m` in class `Client`

The formula $\mathit{disjoint}(\mathtt{list.footprint}, \mathit{allFields}(\mathtt{self}))$ has been reduced to the formula $\mathit{allFields}(\mathtt{null}) \cap \mathtt{list.footprint} \doteq \mathit{empty}$.The negated occurrence of the formula $\mathtt{list} \doteq \mathtt{null}$ in the antecedent has been replaced by the nonnegated occurrence in the succedent via the notLeft rule. The formula *frame* below the modality has vanished entirely, because it holds trivially due to the modifies clause being `everything`. Subsequently, the update $\mathtt{heap}^{pre} := \mathtt{heap}$ has been eliminated using the dropUpdate$_2$ rule of Table 3.1, because $\mathtt{heap}^{pre}$ no longer occurred in its scope.

Next, we start symbolic execution of the program inside the diamond modality, indicated as 'SE' in Figure 9.4. As one of the first steps of symbolic execution, the body of the method `m` being verified is inlined as described in Section 3.6.5. Eventually, symbolic execution reaches the method call '`list.get(0)`' inside `m()`. This call is dispatched using its **`normal_behavior`** JML contract by applying the useMethodContract rule of Section 9.4.3. The application of useMethodContract splits the proof into four branches. We consider here only the two branches for 1. proving the precondition ('pre' branch) valid and 2. continuing after normal termination using the method's postcondition ('post' branch). The other branches close trivially.

- After applying the update w to the formula below it, the 'pre' branch is:

$$\begin{array}{l}
\Gamma \Longrightarrow \\
\mathtt{list} \doteq \mathtt{null}, \\
\{\mathtt{exc} := \mathtt{null} \parallel \mathtt{heap} := \mathit{store}(\mathtt{heap}, \mathtt{null}, \mathtt{Client::\$x}, \mathtt{Client.x} + 1)\} \\
\quad \big(0 \leq 0 \land 0 < \mathtt{list.size()} \land \mathtt{list}.\mathit{inv} \land \mathit{wellFormed}(\mathtt{heap}) \\
\quad\;\; \land \mathtt{list} \not\doteq \mathtt{null} \land \mathtt{list}.\mathit{created} \doteq \mathit{TRUE}\big)
\end{array}$$

where Γ is the same antecedent as before. Closing the 'pre' branch requires showing that the six conjuncts below update hold. The first conjunct $0 \leq 0$ holds trivially. For the other conjuncts, we consider a further split of the proof tree into three subbranches, where the first one corresponds to '$0 < \texttt{list.size()}$,' the second one to '$\texttt{list}.\mathit{inv}$,' and the third one to '$\mathit{wellFormed}(\texttt{heap}) \land \texttt{list} \neq \texttt{null} \land \texttt{list}.\mathit{created} \doteq \mathit{TRUE}$):'

– "$0 < \texttt{list.size()}$." This branch is:

$$\begin{aligned}&\Gamma \Longrightarrow\\ &\texttt{list} \doteq \texttt{null},\\ &0 < \texttt{size}\big(\mathit{store}(\texttt{heap}, \texttt{null}, \texttt{Client::\$x}, \texttt{Client.x} + 1)\}, \texttt{list}\big)\end{aligned}$$

The sequent now contains both the term $\texttt{size}(\texttt{heap}, \texttt{list})$ (inside Γ) and the term $\texttt{size}\big(\mathit{store}(\texttt{heap}, \texttt{self}, \texttt{x}, \texttt{self.x} + 1), \texttt{list}\big)$. This triggers an application of the useDependencyContract rule of Section 9.4.4 (indicated as uDC in Figure 9.4), where we choose $h^{pre} = \texttt{heap}$ and $h^{post} = \mathit{store}(\texttt{heap}, \texttt{null}, \texttt{Client::\$x}, \texttt{Client.x} + 1)\}$. The rule uses the dependency contract for `size` generated out of the JML depends clause '`accessible footprint`' in line 9 of Listing 9.9. It adds the formula $\mathit{guard} \rightarrow \mathit{equal}$ to antecedent, where the subformula *guard* (after some simplification) is:

$$\begin{aligned}&\mathit{wellFormed}(\texttt{heap})\\ &\land \mathit{wellFormed}\big(\mathit{store}(\texttt{heap}, \texttt{null}, \texttt{Client::\$x}, \texttt{Client.x} + 1)\big)\\ &\land \texttt{list}.\mathit{inv} \land \texttt{list} \neq \texttt{null} \land \texttt{list}.\mathit{created} \doteq \mathit{TRUE}\\ &\land \big(\mathit{allFields}(\texttt{null}) \cap \texttt{list.footprint} \doteq \mathit{empty}\big)\end{aligned}$$

All conjuncts of *guard* follow directly from the rest of the sequent. The formula *equal* is:

$$\begin{aligned}&\texttt{size}(\texttt{heap}, \texttt{list}) \doteq\\ &\qquad \texttt{size}\big(\mathit{store}(\texttt{heap}, \texttt{null}, \texttt{Client::\$x}, \texttt{Client.x} + 1), \texttt{list}\big)\end{aligned}$$

Because Γ demands that $0 < \texttt{size}(\texttt{heap}, \texttt{list})$ and the succedent contains $0 < \texttt{size}\big(\mathit{store}(\texttt{heap}, \texttt{null}, \texttt{Client::\$x}, \texttt{Client.x} + 1), \texttt{list}\big)$, the information given by *equal* is enough to close this branch of the proof.

– "$\texttt{list}.\mathit{inv}$." The branch is:

$$\begin{aligned}&\Gamma \Longrightarrow\\ &\texttt{list} \doteq \texttt{null},\\ &\mathit{inv}\big(\mathit{store}(\texttt{heap}, \texttt{null}, \texttt{Client::\$x}, \texttt{Client.x} + 1), \texttt{list}\big)\end{aligned}$$

The sequent now contains the formulas $inv(\texttt{heap},\texttt{list})$ (inside Γ) and $inv\big(store(\texttt{heap},\texttt{null},\texttt{Client::\$x},\texttt{Client.x}+1),\texttt{list}\big)$. The proof continues as on the "$0 < \texttt{list.size()}$" branch above, except that we apply the useDependencyContract rule for *inv* instead of for `size()`.

– '$wellFormed(\texttt{heap}) \land \texttt{list} \not\doteq \texttt{null} \land \texttt{list}.created \doteq TRUE$.' This branch is easy to close, using propositional reasoning only.

• After some simplification, the "post" branch is:

$$
\begin{array}{l}
\Gamma \implies \\
\texttt{list} \doteq \texttt{null}, \\
\{\texttt{exc} := \texttt{null} \,\|\, \texttt{heap} := store(\texttt{heap},\texttt{null},\texttt{Client::\$x},\texttt{Client.x}+1)\} \\
\quad \{\texttt{heap} := anon(\texttt{heap},,h) \,\|\, \texttt{exc}' := e\} \\
\quad\quad \big(\texttt{exc}' \doteq \texttt{null} \land (\texttt{exc}' \doteq \texttt{null} \to \texttt{list}.inv) \\
\quad\quad\; \land (instance_{\texttt{Exception}}(\texttt{exc}') \to (false \land \texttt{list}.inv)) \\
\quad\quad\; \land wellFormed(h) \\
\quad\quad\; \to \langle \texttt{try \{ method-frame(source=m(List)@Client):\{\} \}} \\
\quad\quad\quad\; \texttt{catch(Exception e) \{ exc = e; \}}\rangle(\texttt{exc} \doteq \texttt{null})\big)
\end{array}
$$

where $\texttt{exc}'\!:\!\texttt{Exception} \in \text{ProgVSym}$ is the variable used in the applied contract for `get`, and where the constant symbol $e\!:\!\texttt{Exception} \in \text{FSym}$ are fresh. The remaining program is basically a **`try-catch`** with an empty **`try`** body. Symbolic execution finishes without entering the **`catch`** block, and hence, `exc`is still `null` afterwards, which allows us to close the branch.

This concludes the example proof for the method contract $mct_{\texttt{m}}$. The proof shows that the implementation of method `m` in `Client` satisfies the contract $mct_{\texttt{m}}$, provided that all implementations of `get` in subclasses of `List` satisfy the **`normal_behavior`** method contract for `get`, and provided that all implementations of `size()` and *inv* in subclasses of `List` satisfy the respective dependency contracts.

9.6 Related Methodologies for Modular Verification

Data Groups

KeY's dialect of JML uses dynamic frames whereas standard JML supports *data groups*. Data groups enable the specification of modifies and depends clauses while leaving a certain amount of freedom to implementations about the actual locations that are modified or read. Inclusion of a location into a data group can either be *static* (using data group inclusions [Leino, 1998] via **`in`**) or *dynamic* (via **`maps`** ... **`\into`** clauses). Static inclusion of a field adds the locations of the field of all instances to the data group. This makes membership checking easy, but is little suited for dynamic structures. Dynamic inclusion allows a data group of an object to contain locations of other objects and is suitable for dynamic data structures.

Due to dynamic inclusion, the usage of JML's data groups is unsuitable for modular verification as it cannot be known locally whether a location belongs to a data group or not. This may depend on the subclasses. Solutions by imposing global restrictions on the usage of data groups in programs have been proposed by Leino et al. [2002], but are not part of the standard.

Techniques Related to Dynamic Frames

KeY-JML has been inspired by and is very closely related to the dynamic frames based version of the Spec# specification language [Barnett et al., 2005a] that has been proposed by Smans, Jacobs, Piessens, and Schulte [2008]. The main difference is that their language operates on pure functions (and does not support model fields). The advantage is uniformity, but pure method bodies are not allowed to contain specification-only features like quantifiers.

As an extension of their language, Smans et al. propose an implicit framing field `footprint` which is used as default value in modifies and depends clauses. This approach could be adopted in KeY and JML as well.

Another relative of dynamic frames in JML is the programming and verification language *Dafny* by Leino [2010]. In Dafny specifications, dynamic frame footprints usually occur as ghost fields of type 'set of objects'. Frame specifications in Dafny are thus coarser (all locations of an object are considered), but reasoning is simpler than with arbitrary location sets. Much like with the model field `\inv` in KeY-JML, Dafny specifications encode invariants by introducing a Boolean pure function `Valid`.

Ownership

Müller et al. [2003] describe a version of JML that features abstraction dependencies in place of data groups. *Ownership types* [Clarke et al., 1998], more precisely, the *universe types* of Müller [2002], can be used to make dependency specifications modular. Roughly, the idea of *ownership* is to structure the domain of objects hierarchically into a tree of disjoint *contexts*. An ownership *type system* guarantees statically that, at run-time, every object is only ever referenced from within its context or from its owner object. Ownership can thus prevent unwanted aliasing and abstract aliasing.

A widely used ownership based approach to object invariants is the *Spec# methodology* of Barnett et al. [2004], also known as the *Boogie methodology*. Here, objects are furnished with a ghost field `st` representing their state concerning the invariants. The value of `st` is either 'valid' or 'invalid'. If an object `o` is valid, all objects owned by `o` are valid, and the invariants of `o` is guaranteed to hold. If it is invalid, its owner must have been invalidated, too. Invariants may refer only to locations of **`this`** and of owned objects, and object fields can only be modified when the object has been put in the 'invalid' state.

The methodology also addresses the frame problem: Code cannot compromise the invariant of valid objects. Even if classes may be unknown at verification time, objects are guaranteed to be valid unless they are in the process of being worked on.

A more recent development is the ownership-related invariant protocol *semantic collaboration* by [Polikarpova et al., 2014]. It is a generalization which weakens the hierarchical principle of ownership and allows for more liberal structures. This is achieved by introducing new relationships `subjects` and `observers`: the objects in `subjects` may be used in invariants even if they are not strictly below in the ownership hierarchy. Conversely, the subjects must require that all its `observers` are invalidated when modified. Semantic collaboration can be used to specify and verify design patterns like the observer or visitor pattern which are difficult to treat with ownership alone.

The authors of Spec# report that the Spec# methodology proved too restrictive for some programs they encountered [Barnett et al., 2011]. On the other hand, the *VCC* project turned back to an ownership based approach, after reportedly encountering limiting performance problems with an approach based on dynamic frames [Cohen et al., 2009].

An advantage of ownership based specification and verification techniques over the very liberal technique of dynamic frames is that the framework clearly fixes which invariants can be expected to hold and need to be established. This results in clearer and shorter specifications. Dynamic frames, on the other hand, are not restricted to strictly hierarchical structures but their liberal framework allows for any kind of interaction and interdependencies between objects and their invariants. While this relieves a burden as far as the layout of data structures is concerned, it requires the specifier to write more extensive specifications.

Separation Logic

Separation logic [Reynolds, 2002, O'Hearn et al., 2001, 2009] is a nonclassical extension to Hoare logic. Similar to the dynamic frames approach, it allows explicit reasoning about the heap, which makes it suitable for reasoning about pointer programs and about concurrent programs. Separation properties are however not formulated explicitly using location sets. They are rather blended with functional specifications, using special 'separating' logical connectives. Instead of modifies clauses and depends clauses, framing information is inferred from a method's precondition: only locations mentioned by the precondition may be read or written by the method. This leads to specifications that tend to be shorter, but perhaps less intuitive, than dynamic frames specifications.

Abstraction in separation logic is achieved by abstraction predicates [Parkinson and Bierman, 2005] which serve a similar purpose as object invariants with model fields. Parkinson [2007] makes the case that class invariants may be obsolete as a fundamental concept in specifying object-oriented programs, pointing out the restrictions of the existing modular global invariant protocols and arguing that a concept like abstract predicates can provide a more flexible foundation for expressing consistency properties of object structures. A defense of invariants as an independent concept controlled by a global invariant protocol has been put forward by Summers et al. [2009].

The VerCors system by Amighi et al. [2014a] features a high-level specification language inspired by JML. It uses separating conjunctions and implications, a built-in permission predicate, and abstract specification predicates [Parkinson and Bierman, 2005] (which are similar to Boolean model methods). Programs and specifications are translated to the Chalice tool [Leino et al., 2009] for verification.

Implicit Dynamic Frames

Implicit dynamic frames [Smans et al., 2012] is an approach inspired both by dynamic frames and by separation logic. Instead of using location sets explicitly, the technique centers around a concept of *permissions*: a method may read or write a location only if it has acquired the permission to do so, and these permissions are passed around between method calls by mentioning them in pre- and postconditions. The C and Java verifier VeriFast [Jacobs et al., 2011c] is based on implicit dynamic frames.

Region Logic

Specifications in region logic [Banerjee et al., 2008b] are closely related to dynamic frames specifications, more so than specifications in the implicit dynamic frames approach. There, modifies and depends clauses are expressed with the help of *regions*, that are expressions that evaluate to sets of object references. Region logic is an extension of Hoare logic for reasoning about such specifications [Banerjee et al., 2008a, Rosenberg et al., 2012].

Model Fields

Although model fields are an important element of specifications in JML, there is not yet a common understanding of their semantics. There are several proposed semantics implicitly given through their implementation in actual verification and runtime checking tools. These are sometimes restricted to 'functional' represents clauses [Müller, 2002, Cok, 2005], to model fields of a primitive type, or by restricting the syntax of represents clauses [Breunesse and Poll, 2003, Leino and Müller, 2006]. A detailed discussion can be found in [Bruns, 2009, Sect. 3.1.5].

9.7 Conclusion

This chapter presented a framework for composing and verifying modular design-by-contract specifications. One of its core features is the introduction of a type `\locset`, elevating sets of memory locations to first-class citizens of the language, thus allowing the specification of memory dependency constraints using *dynamic frames*. A feature of this framework is the flexibility in writing specifications without assumptions on the heap structure: Almost any memory dependency pattern can be formulated using dynamic frames, and it allows for a remarkably simple and uniform treatment of model fields and methods, pure methods, and class invariants. Specifications can not only determine the dependencies of methods but also of model fields. The absence of abstract aliasing can be specified explicitly in contracts and invariants, using operators such as `\disjoint` and `\new_elems_fresh`. The downside of this simplicity is that specifications may get more verbose, and that their verification may be computationally more expensive.

Furthermore, modularity is also achieved by means of *abstraction*. The framework has a variety of means for abstraction in specifications which can be used to formulate and verify specifications modularly; modular correctness proofs are still valid if the program is extended.

To achieve these modularity goals in the verification system addressed in this book, the chapter presented advanced calculus rules.

Outlook

KeY's contributions to specification and verification of concurrent programs have not reached a state to warrant inclusion in this book. One of the most promising lines of attack is the use of permissions as outlined in Section 10.7.2. There is ongoing research also with respect to modularity. Grahl [2015] describes a modular approach to the verification of concurrent programs based on the *rely/guarantee* technique from [Jones, 1983]. Grahl extends the specification concepts by dynamic frames. The classical rely/guarantee approach is not entirely modular since it considers programs that are closed under parallel composition. This issue is solved by Grahl [2015] through the addition of frame annotations.

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 Sutcliff 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 Lluch 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 Balser, 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 Internationsl 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 *LNCS*, 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 *LNCS*, 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 *LNCS*, 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 *LNCS*, 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 *LNCS*, 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 FTfJP'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 *LNCS*, 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, FTfJP 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 (FTfJP'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, ISoLA 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 Zendulka, 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 Programing*, 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.utils.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 Völp, 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 Muijrers, 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 Herrmann, 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 theoriespezifische 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 Instenes, 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. Guenthner, 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, ISSTA, 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 (ISPDC 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, *Proceedigngs, Annual ACM SIGAda International Conference on Ada, Saint Petersburg, Florida, USA*, pages 9–10. ACM, 2009. (Cited on page 5.)

Ran Ji. *Sound programm 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, *Proceedigns, 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 Wüstholz, 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 Poetzsch-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, Jérome 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 Piquer. 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 taclets 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.)