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 5
Theories

Peter H. Schmitt and Richard Bubel

5.1 Introduction

For a program verification tool to be really useful it needs to be able to reason about at least the most important data types, both abstract data types and those built into the programming language. In Section 5.2 below the theory of finite sequences of arbitrary objects, allowing in particular arbitrary nestings of sequences in sequences, is presented. This presentation covers axioms for a core theory plus definitional extensions plus consistency considerations.

Section 5.3 contains an axiomatization of Java's String data type. The theory of the mathematical integers has already been dealt with in Subsection 2.4.2. Section 5.4 below explains how the KeY system deals with the Java integer data types.

5.2 Finite Sequences

This section develops and explains the theory T_{seq} of finite sequences. By *Seq* we denote the type of finite sequences. The vocabulary Σ_{seq} of the theory is listed in Figure 5.1. We will start with a simple core theory CoT_{seq} and incrementally enrich it via definitional extensions. Typing of a function symbol f is given as $f : A_1 \times \ldots \times A_n \to R$ with argument types A_i and result type R, typing of a predicate symbol p as $p(A_1 \times \ldots \times A_n)$.

Our notion of a sequence is rather liberal, e.g., $\langle 5,6,7,8 \rangle$ is a sequence, in fact a sequence of integers. But the heterogeneous and nested list $\langle 0, \langle \emptyset, \mathit{seqEmpty}, \mathit{null} \rangle, \mathit{true} \rangle$ is also allowed.

The semantics of the symbols of the core theory will be given in Definition 5.3. We provide here a first informal account of the intended meaning. The function value $\mathit{seqLen}(s)$ is the length of list s. Since for heterogeneous lists there is no way the type of an entry can be recovered from the type of the list, we provide a family of access functions seqGet_A that yields the cast to type A of the i-th entry in list s.

Core Theory
$A{::}seqGet : Seq \times int \to A$ for any type $A \sqsubseteq Any$
$seqGetOutside : Any$
$seqLen : Seq \to int$

Variable Binder
$seqDef : int \times int \times Seq \to Seq$

Definitional Extension
$seqDepth : Seq \to int$
$seqEmpty : Seq$
$seqSingleton : Any \to Seq$
$seqConcat : Seq \times Seq \to Seq$
$seqSub : Seq \times int \times int \to Seq$
$seqReverse : Seq \to Seq$
$seqIndexOf : Seq \times Any \to int$
$seqNPerm(Seq)$
$seqPerm(Seq, Seq)$
$seqSwap : Seq \times int \times int \to Seq$
$seqRemove : Seq \times int \to Seq$
$seqNPermInv : Seq \to Seq$

Fig. 5.1 The vocabulary Σ_{seq} of the theory T_{seq} of finite sequences

(The concrete, ASCII syntax is $A{::}seqGet$, but we stick here with the slightly shorter notation $seqGet_A$.) The constant *seqGetOutside* is an arbitrary element of the top type *Any*. It is, e.g., used as the value of any attempt to access a sequence outside its range. *seqDef* is a variable binder symbol, check Section 2.3.1 for explanation. Its precise semantics is given in Definition 5.2 below. The reader may get a first intuition from the simple example $seqDef\{u\}(1,5,u^2)$ that represents the sequence $\langle 1,4,9,16 \rangle$. We will comment on the symbols in the definitional extension, when we are finished with the core theory following page 152.

lenNonNegative
$\forall Seq\ s; (0 \leq seqLen(s))$

equalityToSeqGetAndSeqLen
$$\forall Seq\ s_1, s_2; (s_1 \doteq s_2 \leftrightarrow seqLen(s_1) \doteq seqLen(s_2) \land \\ \forall int\ i; (0 \leq i < seqLen(s_1) \to seqGetAny(s_1, i) \doteq seqGet_{Any}(s_2, i)))$$

getOfSeqDef
$$\forall int\ i, ri, le; \forall Any\ \bar{x}; (\\ ((0 \leq i \land i < ri - le) \to seqGet_A(seqDef\{u\}(le, ri, t), i) \doteq cast_A(t\{(le+i)/u\})) \land \\ (\neg(0 \leq i \land i < ri - le) \to seqGet_A(seqDef\{u\}(le, ri, t), i) \doteq cast_A(seqGetOutside)))$$

lenOfSeqDef
$$\forall int\ ri, le; ((le < ri \to seqLen(seqDef\{u\}(le, ri, t)) \doteq ri - le) \land \\ (ri \leq le \to seqLen(seqDef\{u\}(le, ri, t)) \doteq 0))$$

Fig. 5.2 Axioms of the core theory CoT_{seq} (in mathematical notation)

The axioms of the core theory CoT_{seq} are shown in Figure 5.2 in mathematical notation together with the names of the corresponding taclets. In getOfSeqDef the quantifier $\forall Any\ \bar{x}$ binds the variables that may occur in term t.

Definition 5.1 below extends the semantics of type domains given in Figure 2.10 on page 45. More precisely, the definition gives the construction to obtain D^{Seq} when all other type domains are fixed.

Definition 5.1 (The type domain D^{Seq}). The type domain D^{Seq} is defined via the following induction:

$$D^{Seq} \quad := \quad \bigcup_{n\geq 0} D^n_{\mathrm{Seq}}$$

where

$$\begin{aligned}
U &= D^{Any} \setminus D^{Seq} \\
D^0_{\mathrm{Seq}} &= \{\langle\rangle\} \\
D^{n+1}_{\mathrm{Seq}} &= \{\langle a_0,\ldots,a_{k-1}\rangle \mid k \in \mathbb{N} \text{ and } a_i \in D^n_{\mathrm{Seq}} \cup U, 0 \leq i < k\} \quad \text{for } n \geq 0
\end{aligned}$$

The type domain for *Seq* being fixed we now may deliver on the forward reference after Definition 2.22 and define precisely the meaning of the variable binder symbol $seqDef\{iv\}(le,ri,e)$ in the JFOL structure $\mathscr{M}$. As already done in Section 2.4.5 we will use the notation $t^{\mathscr{M},\beta}$ for term evaluation instead of $\mathrm{val}_{\mathscr{M},\beta}(t)$. We further will suppress β and write $t^{\mathscr{M}}$ if it is not needed or not relevant.

Definition 5.2.

$$seqDef\{iv\}(le,ri,e)^{\mathscr{M},\beta} = \begin{cases} \langle a_0,\ldots a_{k-1}\rangle & \text{if } (ri-le)^{\mathscr{M},\beta} = k > 0 \text{ and } a_i = e^{\mathscr{M},\beta_i} \\ & \text{with } \beta_i = \beta[le+i/iv] \text{ and all } 0 \leq i < k \\ \langle\rangle & \text{otherwise} \end{cases}$$

Remember, that $\beta[le+i/iv]$ is the variable assignment that coincides with β expect for the argument iv where it takes the value $le+i$.

The core vocabulary of CoT_{seq} is interpreted as follows:

Definition 5.3.

1. $seqGet_A^{\mathscr{M}}(\langle a_0,\ldots,a_{n-1}\rangle, i) = \begin{cases} cast_A^{\mathscr{M}}(a_i) & \text{if } 0 \leq i < n \\ cast_A^{\mathscr{M}}(seqGetOutside^{\mathscr{M}}) & \text{otherwise} \end{cases}$
2. $seqLen^{\mathscr{M}}(\langle a_0,\ldots,a_{n-1}\rangle) = n$
3. $seqGetOutside^{\mathscr{M}} \in D^{Any}$ arbitrary.

To have a name for it we might call a structure $\mathscr{M}$ in the vocabulary Σ_J (see Figure 2.4) plus the core vocabulary of finite sequences a *CoreSeq* structure, if its restriction to the JFOL vocabulary is a JFOL structure as defined in Section 2.4.5 and, in addition $\mathscr{M}$ satisfies Definition 5.3. We observe, that the expansion of a JFOL structure $\mathscr{M}_0$ to a *CoreSeq* structure is uniquely determined once an interpretation $seqGetOutside^{\mathscr{M}_0}$ is chosen.

Theorem 5.4. *The theory CoT_{seq} is consistent.*

Proof. It is easily checked that the axioms in Figure 5.2 are true in all *CoreSeq* structures. The explicit construction guarantees that there is at least one *CoreSeq* structure.

$$
\begin{array}{l}
\forall Seq\ s; (\forall int\ i; ((0 \leq i < seqLen(s) \rightarrow \neg instance_{Seq}(seqGet_{Any}(s,i))) \rightarrow seqDepth(s) \doteq 0) \land \\
\forall Seq\ s; (\forall int\ i; ((0 \leq i < seqLen(s) \land instance_{Seq}(seqGet_{Any}(s,i))) \rightarrow \\
\qquad\qquad\qquad seqDepth(s) > seqDepth(seqGet_{Seq}(s,i))) \land \\
\forall Seq\ s; (\exists int\ i; (0 \leq i < seqLen(s) \land instance_{Seq}(seqGet_{Any}(s,i))) \rightarrow \\
\qquad\quad \exists int\ i; (0 \leq i < seqLen(s) \land instance_{Seq}(seqGet_{Any}(s,i)) \land \\
\qquad\qquad\qquad seqDepth(s) \doteq seqDepth(seqGet_{Seq}(s,i)) + 1)
\end{array}
$$

Fig. 5.3 Definition of *seqDepth*

We observe that $seqDepth(s)$ as defined in Figure 5.3 equals the recursive definition

$$
seqDepth(s) = \max\{seqDepth(seqGet_{Seq}(s,i)) \mid \begin{array}{l} 0 \leq i < seqLen(s) \land \\ instance_{Seq}(seqGet_{Seq}(s,i))\} \end{array}
$$

with the understanding that the maximum of the empty set is 0. Since we have not introduced the maximum operator we had to resort to the formula given above. The function *seqDepth* is foremost of theoretical interest and at the moment of this writing not realized in the KeY system. $seqDepth(s)$ is an integer denoting the *nesting depth* of sequence s. If s has no entries that are themselves sequences then $seqDepth(s) \doteq 0$. For a sequence s_{int} of sequences of integers we would have $seqDepth(s_{\mathrm{int}}) \doteq 0$.

In Figure 5.4 the mathematical formulas defining the remaining noncore vocabulary are accompanied by the names of the corresponding taclets. A few explaining comments will help the reader to grasp their meaning. The subsequence $seqSub(s,i,j)$ from i to j of sequence s includes the i-th entry, but excludes the j-th entry. In the case $\neg(i < j)$ it will be the empty sequence, this is a consequence of the semantics of $seqDef$. The term $seqIndexOf(s,t)$ denotes the least index n such that $seqGet_{Any}(s,n) \doteq t$ if there is one, and is undefined otherwise. See Section 2.3.2 on how undefinedness is handled in our logic. A sequence s satisfies the predicate $seqNPerm(s)$ if it is a permutation of the integers $\{0,\ldots,seqLen(s)-1\}$. The binary predicate $seqPerm(s_1,s_s)$ is true if s_2 is a permutation of s_1. Thus $seqNPerm(\langle 5,4,0,2,3,1\rangle)$ and $seqPerm(\langle a,b,c\rangle,\langle b,a,c\rangle)$ are true.

Careful observation reveals that the interpretation of the vocabulary outside the core vocabulary is uniquely determined by the definitions in Figures 5.3 and 5.4.

We establish the following notation:

Definition 5.5. By T_{seq} we denote the theory given by the core axioms CoT_{seq} plus the definitions from Figures 5.3 and 5.4.

On the semantic side we call a structure $\mathscr{M}$ in the vocabulary Σ_J plus Σ_{Seq} a *Seq* structure if the restriction of $\mathscr{M}$ to Σ_J is a JFOL structure and $\mathscr{M}$ satisfies Definitions 5.3 and 5.4.

defOfEmpty

$seqEmpty \doteq seqDef\{iv\}(0,0,x)$

x is an arbitrary term of type *Any* not containing the variable *iv*.

defOfSeqSingleton

$\forall Any\ x; (seqSingleton(x) \doteq seqDef\{iv\}(0,1,x))$

defOfSeqConcat

$$\begin{aligned}&\forall Seq\ s_1, s_2; (seqConcat(s_1,s_2) \doteq\\ &\qquad seqDef\{iv\}(0, seqLen(s_1) + seqLen(s_2), \text{ if } iv < seqLen(s_1)\\ &\qquad\qquad \text{then } seqGet_{Any}(s_1, iv)\\ &\qquad\qquad \text{else } seqGet_{Any}(s_2, iv - seqLen(s_1)))))\end{aligned}$$

defOfSeqSub

$\forall Seq\ s; \forall int\ i,j; (seqSub(s,i,j) \doteq seqDef\{iv\}(i,j,seqGet_{Any}(s,iv)))$

defOfSeqReverse

$\forall Seq\ s; (seqReverse(s) \doteq seqDef\{iv\}(0, seqLen(s), seqGet_{Any}(s, seqLen(s) - iv - 1)))$

seqIndexOf

$$\begin{aligned}&\forall Seq\ s; \forall Any\ t; \forall int\ n; (0 \leq n < seqLen(s) \land seqGet_{Any}(s,n) \doteq t \land\\ &\qquad \forall int\ m; (0 \leq m < n \rightarrow seqGet_{Any}(s,m) \neq t)\\ &\qquad \rightarrow seqIndexOf(s,t) \doteq n)\end{aligned}$$

seqNPermDefReplace

$$\begin{aligned}&\forall Seq\ s; (seqNPerm(s) \leftrightarrow\\ &\qquad \forall int\ i; (0 \leq i < seqLen(s) \rightarrow \exists int\ j; (0 \leq j < seqLen(s) \land seqGet_{int}(s,j) \doteq i)))\end{aligned}$$

seqPermDef

$$\begin{aligned}&\forall Seq\ s_1, s_2; (seqPerm(s_1,s_2) \leftrightarrow seqLen(s_1) \doteq seqLen(s_2) \land\\ &\qquad \exists Seq\ s; (seqLen(s) \doteq seqLen(s_1) \land seqNPerm(s) \land\\ &\qquad \forall int\ i; (0 \leq i < seqLen(s) \rightarrow\\ &\qquad\quad seqGet_{Any}(s_1,i) \doteq seqGet_{Any}(s_2, seqGet_{int}(s,i)))))\end{aligned}$$

defOfSeqSwap

$$\begin{aligned}&\forall Seq\ s; \forall int\ i,j; (seqSwap(s,i,j) \doteq\\ &\qquad seqDef\{iv\}(0, seqLen(s), \text{ if } \neg(0 \leq i < seqLen(s) \land 0 \leq j < seqLen(s))\\ &\qquad\qquad \text{then } seqGet_{Any}(s,iv)\\ &\qquad\qquad \text{else if } iv \doteq i\\ &\qquad\qquad\quad \text{then } seqGet_{Any}(s,j)\\ &\qquad\qquad\quad \text{else if } iv \doteq j\\ &\qquad\qquad\qquad \text{then } seqGet_{Any}(s,i)\\ &\qquad\qquad\qquad \text{else } seqGet_{Any}(s,iv)))\end{aligned}$$

defOfSeqRemove

$$\begin{aligned}&\forall Seq\ s; \forall int\ i; (seqRemove(s,i) \doteq \text{ if } i < 0 \lor seqLen(s) \leq i\\ &\qquad \text{then } s\\ &\qquad \text{else } seqDef\{iv\}(0, seqLen(s) - 1, \text{ if } iv < i\\ &\qquad\qquad \text{then } seqGet_{Any}(s,iv)\\ &\qquad\qquad \text{else } seqGet_{Any}(s,iv+1)))\end{aligned}$$

defOfSeqNPermInv

$\forall Seq\ s; (seqNPermInv(s) \doteq seqDef\{iv\}(0, seqLen(s), seqIndexOf(s,iv)))$

Fig. 5.4 Definition for noncore vocabulary in mathematical notation

Theorem 5.6. *The theory T_{Seq} is consistent.*

Proof. The consistency of T_{Seq} follows from the consistency of CoT_{Seq} since it is a definitional extension.

A proof of Theorem 5.6 together with a detailed review of the concept of definitional extensions, plus statement and proof of the relative completeness of T_{Seq} can be found in the technical report on first-order logic available from the companion website to this book www.key-project.org/thebook2.

1 seqSelfDefinition
$\forall Seq\ s; (s \doteq seqDef\{u\}(0, seqLen(s), seqGet_{Any}(s,u)))$

2 seqOutsideValue
$\forall Seq\ s; (\forall int\ i; ((i < 0 \lor seqLen(s) \leq i) \to seqGet_{\alpha}(s,i) \doteq (\alpha)seqGetOutside))$

3 castedGetAny
$\forall Seq\ s; \forall int\ i; ((\beta)seqGet_{Any}(s,i) \doteq seqGet_{\beta}(s,i))$

4 getOfSeqSingleton
$\forall Any\ x; \forall int\ i; (seqGet_{\alpha}(seqSingleton(x),i) \doteq \text{if } i \doteq 0 \text{ then } (\alpha)x \text{ else } (\alpha)seqGetOutside)$

5 getOfSeqConcat
$$\begin{aligned}\forall Seq\ s, s2; \forall int\ i; (seqGet_{\alpha}(seqConcat(s,s2),i) \doteq\ & \text{if } i < seqLen(s)\\ & \text{then } seqGet_{\alpha}(s,i)\\ & \text{else } seqGet_{\alpha}(s2, i - seqLen(s)))\end{aligned}$$

6 getOfSeqSub
$$\begin{aligned}\forall Seq\ s; \forall int\ from, to, i; (seqGet_{\alpha}(seqSub(s, from, to), i) \doteq\ & \text{if } 0 \leq i \land i < (to - from)\\ & \text{then } seqGet_{\alpha}(s, i + from)\\ & \text{else } (\alpha)seqGetOutside)\end{aligned}$$

7 getOfSeqReverse
$\forall Seq\ s; \forall int\ from, to, i; (seqGet_{\alpha}(seqReverse(s),i) \doteq seqGet_{\alpha}(s, seqLen(s) - 1 - i))$

8 lenOfSeqEmpty
$seqLen(seqEmpty) \doteq 0$

9 lenOfSeqSingleton
$\forall Any\ x; (seqLen(seqSingleton(x)) \doteq 1)$

10 lenOfSeqConcat
$\forall Seq\ s, s2; (seqLen(seqConcat(s,s2)) \doteq seqLen(s) + seqLen(s2))$

11 lenOfSeqSub
$\forall Seq\ s; \forall in\ from, to; (seqLen(seqSub(s, from, to)) \doteq \text{if } from < to \text{ then } (to - from) \text{ else } 0)$

12 lenOfSeqReverse
$\forall Seq\ s; (seqLen(seqReverse(s)) \doteq seqLen(s))$

13 seqConcatWithSeqEmpty
$\forall Seq\ s; (seqConcat(s, seqEmpty) \doteq s)$

14 seqReverseOfSeqEmpty
$seqReverse(seqEmpty) \doteq seqEmpty$

Fig. 5.5 Some derived rules for finite sequences

Figure 5.5 lists some consequences that can be derived from the definitions in Figure 5.4 and the Core Theory. The entry 1 is a technical lemma that is useful in the derivation of the following lemmas in the list. The entry 2 clarifies the role of the default value *seqGetOutside*; it is the default or error value for any out-of-range access. Rules 2 to 7 are schematic rule. These rules are applicable for any instantiations of the schema variable α by a type. Entry 3 addresses an important issue: on one hand there is the family of function symbols $seqGet_{\alpha}$, on the other hand there are the cast expressions $(\alpha)seqGet_{Any}$. The lemma says that both coincide. The entries 4 to 12 allow to determine the access function and the length of the empty sequence, singleton, concatenation, subsequence and reverse constructors. The last two entries 13 and 14 are examples for a whole set of rules that cover corner cases of the constructors involved.

1 seqNPermRange
$\forall Seq\ s; (seqNPerm(s) \rightarrow$
$\forall int\ i; (0 \leq i \wedge i < seqLen(s) \rightarrow (0 \leq seqGet_{int}(s,i) \wedge seqGet_{int}(s,i) < seqLen(s))))$

2 seqNPermInjective
$\forall Seq\ s;\ (seqNPerm(s) \wedge$
$\forall int\ i,j; (0 \leq i \wedge i < seqLen(s) \wedge 0 \leq j \wedge j < seqLen(s) \wedge seqGet_{int}(s,i) \doteq seqGet_{int}(s,j))$
$\rightarrow i \doteq j)$

3 seqNPermEmpty
$seqNPerm(seqEmpty)$

4 seqNPermSingleton
$\forall int\ i; (seqNPerm(seqSingleton(i)) \leftrightarrow i \doteq 0)$

5 seqNPermComp
$\forall Seq\ s1,s2; (seqNPerm(s1) \wedge seqNPerm(s2) \wedge seqLen(s1) \doteq seqLen(s2) \rightarrow$
$seqNPerm(seqDef\{u\}(0, seqLen(s1), seqGet_{int}(s1, seqGet_{int}(s2,u)))))$

6 seqPermTrans
$\forall Seq\ s1,s2,s3; (seqPerm(s1,s2) \wedge seqPerm(s2,s3) \rightarrow seqPerm(s1,s3))$

7 seqPermRefl
$\forall Seq\ s; (seqPerm(s,s))$

Fig. 5.6 Some derived rules for permutations

Figure 5.6 lists some derived rules for the one-place predicate *seqNPerm* and the two-place predicate *seqPerm* that follow from the definitions in Figure 5.4. Surprisingly, none of the proofs apart from the one for seqNPermRange needs induction. This is mainly due to the presence of the $seqDef\{\}(,,)$ construct. The lemma seqNPermRange itself is a kind of pigeon-hole principle and could only be proved via induction.

Applications of the theory of finite sequences can be found in Section 16.5 and foremost in Chapter 19.

5.3 Strings

Java strings are implemented as objects of their own and they are not identified with arrays of characters. This eases treatment of strings in our program logic as we can reuse all the mechanisms already in place for objects. So, do we need special treatment for them at all? Why not simply use contracts as for other API classes?

To answer the first question: Although strings are normal objects, the Java Language Specification [Gosling et al., 2013] provides some additional infrastructure not available for other kinds of objects. In particular, the existence of string literals like `"Hello␣World"` requires additional thought. A string literal is a reference to a string object whose content coincides with the string literal name within the quotation marks. The problem to be solved is to make string literals 'behave' like integer or Boolean literals. For instance, the expression `"Hello" == "Hello"` should always be true. To solve this issue, Java ensures that all occurrences of the same literal reference the same object. To ensure this behavior Java manages a *pool* of strings in which all strings referenced by string literals (actually, all compile time constants of type `String`) are put. Nevertheless, the taken solution does not hide completely that string literals are different from other literals, e.g., the expression `new String("Hello") == "Hello"` evaluates to false. To represent the pool, we could model it in Java itself. This solution would allow us to be mostly ignorant to strings on the logic level, but introduce a lot of clutter in the reasoning process when string literals are involved. Instead we use an alternative route and model the pool purely on the logic level, which allows us a more streamlined representation and deemphasizes the use of a pool.

We go a step further and introduce a kind of "ghost" field for Java string objects that assigns each string a finite sequence of characters based on the Sequence data type introduce in the previous section. As specifications about strings express properties about their content, e.g., that the content equals or matches a given expression (e.g., a regular expression), providing an abstract data type for strings allows us to separate concerns and to ease writing of specifications.

In this section we describe three parts that constitute our handling of Java strings: The theory T_{cl} of sequences of characters representing the content of a Java string as an extension of the theory of finite sequences, a theory T_{rex} of regular expressions to express and reason conveniently about the content of strings, and finally we conclude with the actual theory $T_{\texttt{java.lang.String}}$ of Java strings, which uses the previous two theories to model Java strings.

5.3.1 Sequences of Characters

Characters are not represented as a distinct type, but as integers which are interpreted as the characters unicode (UTF-16) representation. A finite sequence of characters is used to model the underlying theory to support Java Strings. The length of the sequence is the number of 16-bit unicode values. This value might not coincide with the number of actual characters as some unicode characters need to be represented by two 16-bit numbers. The presentation below and the actual implementation assumes that only unicode characters in the range 0x0000 − 0xFFFF are used.

We extend the theory of finite sequences by the additional functions and predicates shown in Table 5.1. They allow us later to specify the behavior of the `String`'s method concisely. The first four functions return the first (or last) index of a sequence

Table 5.1 The additional functions and predicates of T_{cl} (int^* is used to indicate that a character's UTF-16 unicode representation is expected as argument; the actual type is *int*)

Extensions	
$clIndexOfChar$	$: Seq \times int^* \times int \to int$
$clIndexOfCl$	$: Seq \times int \times Seq \to int$
$clLastIndexOfChar$	$: Seq \times int^* \times int \to int$
$clLastIndexOfCl$	$: Seq \times int \times Seq \to int$
$clReplace$	$: Seq \times int^* \times int^* \to Seq$
$clTranslateInt$	$: int \to Seq$
$clRemoveZeros$	$: Seq \to Seq$
$clHashCode$	$: Seq \to int$
$clStartsWith$	$: Seq \times Seq$
$clEndsWith$	$: Seq \times Seq$
$clContains$	$: Seq \times Seq$

starting from the position given as third argument at which to find the specified character (start of the given character sequence) or -1 if no such character (character sequence) exists (this differs from *seqIndexOf*, which is undefined for elements not occurring in a sequent). Function $clReplace(s, c_1, c_2)$ evaluates to a sequence equal to s except that all occurrences of character c_1 have been replaced by character c_2. Function *clTranslateInt* takes a character sequence specifying a number and translates it into the corresponding integer. It comes paired with the auxiliary function *clRemoveZero* which removes any leading zeros, as in "000123", first, before the result can be handed over to *clTranslateInt* in order to rewrite it into the integer 123.

The predicates *clContains*, *clStartsWith*, and *clEndsWith* evaluate to true if the character sequence given as first argument contains, starts or ends with the character sequence given as second argument, respectively.

The actual axiomatizations are rather technical, but not complicated. Those for *clIndexOfChar* and *clContains* are shown in Figure 5.7. The axiomatization of *clIndexOfChar* makes use of JavaDL's ifEx operator to determine the minimal (first) index of the searched character, if one exists.

For convenience reasons, we write short ”*abc*” instead of the actual term $seqConcat(seqSingleton('a'), seqConcat(seqSingleton('b'), seqSingleton('c')))$. The pretty printer of KeY outputs a character sequence in this way (outside of modalities), and the parser accepts string literals as an alternative syntax for character sequences.

indexOf
$\forall Seq\ l,c; \forall int\ i; \big(clIndexOfChar(l,c,i) \doteq$
$\quad$ ifEx $int\ iv; (i \geq 0 \wedge iv \geq i \wedge iv < seqLen(l) \wedge\ seqGet_{int}(l,iv) \doteq c)$
$\quad$ then(iv) else$(-1)\big)$

containsAxiom
$\forall Seq\ textString,\ searchString; \big($
$\quad clContains(textString, searchString) \leftrightarrow$
$\quad\quad \exists int\ iv; \big(iv \geq 0\ \wedge\ iv + seqLen(searchString) \geq seqLen(textString)$
$\quad\quad\quad \wedge\ seqSub(textString, iv, iv + seqLen(searchString)) \doteq searchString\big)\big)$

Fig. 5.7 Axioms for *clIndexOfChar* and *clContains*

5.3.2 Regular Expressions for Sequences

To be able to conveniently specify methods manipulating strings, the theory T_{rex} allows one to match elements of type *Seq* using regular expression.[1] Pattern expressions (PExp) are represented as terms of type `rex`. Table 5.2 lists the PExp constructors. For instance, the pattern represented by the term $repeatStar(rex(\text{"}ab\text{"}))$ matches a

Table 5.2 Pattern expressions (PExp) with $cl : Seq$ and $pe, pe1, pe2 : rex$.

constructor (of type `rex`)		constructor	
$rex(cl)$	matches exactly cl	$repeatStar(pe)$	pe^*
$opt(pe)$	$pe?$	$repeatPlus(pe)$	pe^+
$alt(pe1, pe2)$	$pe1 + pe2$	$repeat(pe, n)$	pe^n
$regConcat(pe1, pe2)$	$pe1 \cdot pe2$		

finite but arbitrary repetition of the word "*ab*". Match expressions are constructed using the predicate $match(rex, Seq)$. The predicate *match* takes two arguments: a PExp as first argument and the concrete character sequence to be matched against the pattern as second argument. The match expression is true if and only if the provided pattern matches the complete Seq.

Our calculus features a complete axiomatization of the pattern and matching language. Further, there are a number of derived rules to reduce and simplify pattern and match expression terms as far as possible. We give here only a few typical representatives of these axioms and rules.

The first axiom maps the alternative pattern constructor back to a logical disjunction:

altAxiom
$\forall rex\ pe1, pe2; \forall Seq\ cl;$
$\quad match(alt(pe1, pe2), cl) \leftrightarrow (match(pe1, cl) \vee match(pe2, cl))$

[1] Remark: T_{rex} goes actually beyond regular expressions.

The second axiom removes the pattern concatenation by guessing the index where to split the text to be matched into two parts. Each part is then independently matched against the corresponding subpattern:

$$
\begin{array}{l}
\textsf{regConcatAxiom} \\
\quad \forall rex\ pe1, pe2; \forall Seq\ cl; \big(\\
\quad\quad match(regConcat(pe1, pe2), cl) \leftrightarrow \big(\exists int\ i; (i \geq 0 \wedge i \leq seqLen(cl) \\
\quad\quad\quad \wedge\, match(pe1, seqSub(cl, 0, i)) \wedge match(pe2, seqSub(cl, i, seqLen(cl)))))\big)\big)
\end{array}
$$

A typical reduction rule aiming to reduce the complexity of is for instance:

$$
\begin{array}{l}
\textsf{regConcatConcreteStringLeft} \\
\quad \exists rex\ pre; \exists Seq\ s, cl; \big(\\
\quad\quad match(regConcat(rex(s), pe), cl) \\
\quad\quad\quad\quad \leftrightarrow \big(seqLen(s) \leq seqLen(cl) \\
\quad\quad\quad\quad\quad \wedge\ match(rex(s), seqSub(cl, 0, seqLen(s))) \\
\quad\quad\quad\quad\quad \wedge\ match(pe, seqSub(cl, seqLen(s), seqLen(cl))))\big)\big)
\end{array}
$$

5.3.3 Relating Java String Objects to Sequences of Characters

The previous sections introduced the logic representation of character sequences and regular expressions. To achieve our goal to specify and verify programs in presence of Strings, the abstract representation of a string's content and the implementation of Java's `String` class need to be related.

This could be simply achieved by introducing a ghost field and keeping the content on the heap. We choose a similar but slightly different modeling, which simplifies verification down the road by exploiting that once a String instance is created, its content does not change. This is the same situation as for the length field of an array, and we use the same idea. The function $strContent : \mathtt{java.lang.String} \rightarrow Seq$ maps each String instance to a sequence of characters. It is left unspecified initially and upon creation of a new String instance s representing, e.g., the character list sc, the formula $strContent(s) \doteq sc$ is added to the antecedent of the formula. This is well-defined as the content of a `String` instance cannot be changed. The following sequent calculus rule illustrates this mechanism:

$$
\begin{array}{l}
\textsf{stringConcat} \\
\dfrac{\begin{array}{l}\Gamma, strContent(sk) \doteq seqConcat(strContent(s1), strContent(s2)), sk \not\doteq \mathtt{null} \Longrightarrow \\ \qquad \{v := sk\}\ \{\mathtt{heap} := create(\mathtt{heap}, sk)\}\ \langle \pi\omega \rangle \phi, \Delta\end{array}}{\Gamma \Longrightarrow \langle \pi\ v = s1 + s2;\ \omega \rangle \phi, \Delta}
\end{array}
$$

Schema variables $v, s1, s2$ match local program variables of type `java.lang.String` and sk is a fresh Skolem-constant.

5.3.4 String Literals and the String Pool

The Java string pool caches `String` instances using their content as key. On start-up of the virtual machine and after class loading all compile-time constant of type `String` (in particular all string literals) are resolved to an actual String object. New elements can be added to the cache at run-time with method `intern()`, but Java programs cannot remove elements from the cache.

We model the string pool as an injective function

$$\mathtt{strPool} : Seq \to \mathtt{java.lang.String}$$

The assignment rule for string literals in the presence of the string pool can now be defined as follows:

stringAssignment

$$\frac{\begin{array}{l}\Gamma, strContent(\mathtt{strPool}(sLit_{CL})) \doteq sLit_{CL},\\ \mathtt{strPool}(sLit_{CL}) \not\doteq null,\\ select_{\mathrm{boolean}}(\mathtt{heap}, \mathtt{strPool}(strContent(sLit_{CL})), created) \doteq TRUE\\ \Longrightarrow\\ \{v := \mathtt{strPool}(sLit_{CL})\}\ \langle \pi\ \omega\rangle \phi, \Delta\end{array}}{\Gamma \Longrightarrow \langle \pi\ v = sLit;\ \omega\rangle \phi, \Delta}$$

Here *sLit* is a schema variable matching string literals and $sLit_{CL}$ denotes the finite sequence *Seq* representation of the matched string literal *sLit*.

One side remark at this point concerning the concatenation of string literals, i.e., how a program fragment of the kind `v = "a" + "b";` is treated. In this case the expression `"a" + "b"` is a compile time constant, which are as their name suggests evaluated at compile time. Hence, any such expression has already been replaced by the result `"ab"` when reading in the program (in other words, in JavaDL all compile time constants are already replaced by their fully evaluated literal expression).

Finally, we give one of the rules for updating the Java string pool with a new element. Note, this rule is actually specified as a contract of method `intern()` of class `String`:

updatePool

$$\frac{\begin{array}{l}\Gamma, \neg(v \doteq \mathtt{null}), \mathtt{strPool}(strContent(v)) \not\doteq null,\\ select_{\mathrm{boolean}}(\mathtt{heap}, \mathtt{strPool}(strContent(v)), created) \doteq TRUE\\ \Longrightarrow \{r := \mathtt{strPool}(strContent(v))\}\ \langle \pi\ \omega\rangle \phi, \Delta\end{array}}{\Gamma, \neg(v \doteq \mathtt{null}) \Longrightarrow \langle \pi\ r = \mathtt{.intern();}\ \omega\rangle \phi, \Delta}$$

5.3.5 Specification of the Java String API

To obtain a complete calculus for Java strings, additional rules have to be created which translate an integer or the `null` reference to its String representation. The formalization of the necessary translate functions is rather tedious, but otherwise straightforward. The technical details are described in [Geilmann, 2009].

Based on the formalization described in this section, we specified the majority of the methods declared and implemented in the `java.lang.String` class. The Seq ADT functions have been chosen to represent closely the core functionality provided by the String class. The specification of the methods required then merely to consider the border cases of most of the methods. Border cases are typically those cases where the ADT has been left underspecified and that cause an exception in Java.

5.4 Integers

Arithmetic reasoning is required for a number of verification tasks like proving that a certain index is within the array's bounds, that no arithmetic exception occurs and, of course, that a method computes the correct value. Mathematical integers have already been covered in Section 2.4.5, in this section we highlight the axiomatization of integers with respect to their finite integral counterparts used in Java.

In lieu of the whole numbers $\mathbb{Z}$, programming languages usually use finite integral types based on a two-complement representation. For instance, Java's integral types (`byte`, `char`, `short`, `int` and `long`) are represented in 8-bit, 16-bit, 32-bit and 64-bit two-complement representation (with the exception of `char` which is represented as an unsigned 16-bit number).

The finiteness of integral types and the often used modulo arithmetics entail the possibility of underflows and overflows. While sometimes intended, they are also a source of bugs leading to unexpected behavior. As pointed out by Joshua Bloch[2] most binary search algorithms are broken because of an overflow that might happen when computing the middle of the interval `lower...upper` by `(upper+lower)/2` (e.g., for `lower` equal to 100 and `upper` equal to the maximal value of its integral type).

The question arises: How do we model finite integral types within the program logic. One possibility is to define new sorts to model the required fixed-precision numbers together with functions for addition and subtraction, e.g., as a general fixed-width bit-vector theory. Another approach, and that is the one we pursued in JFOL, is to map all Java arithmetic operations to standard arithmetic operations without the need to introduce new additional sorts.

[2] googleresearch.blogspot.se/2006/06/extra-extra-read-all-about-it-nearly.html

5.4.1 Core Integer Theory

The predefined sort `int` is evaluated to the set of whole numbers $\mathbb{Z}$. Table 5.3 shows a selection of the most important interpreted core functions. All of them are interpreted canonically. In case of division, *div* rounding towards the nearer lower number takes place, and *mod* is interpreted as the modulo function.

On top of these core functions, derived functions (shown in Table 5.3 as extensions) are defined. Their domain is still the whole numbers, i.e., no modulo arithmetics are involved, but otherwise they reflect the division and modulo semantics in Java more closely. Namely, function *jdiv* is interpreted as the division on $\mathbb{Z}$ rounding towards zero, while *jmod* is interpreted as the remainder function in opposite to the modulo function.

Finally, functions like *addJint*, *addJlong* etc. are defined in such a way that they reflect the modulo semantics of the Java operations. They are axiomatized solely using the functions shown in Table 5.3

Table 5.3 Core and extension functions for the `int` data type

Core		
add	'+'	addition on $\mathbb{Z}$
sub	'-'	subtraction on $\mathbb{Z}$
div	'/'	division on $\mathbb{Z}$ (Euclidean semantics)
mod	'%'	modulo on $\mathbb{Z}$
mul	'*'	multiplication on $\mathbb{Z}$
Extensions		
jdiv	n/a	division on $\mathbb{Z}$ (rounding towards zero)
jmod	n/a	remainder on $\mathbb{Z}$

Table 5.4 Functions for Java arithmetics

addJint/addJLong	addition with overflow for `int`/`long`
divJint/divJLong	modulo with overflow for `int`/`long`
moduloByte/moduloChar/moduloShort/...	modulo operation mapping numbers into their respective range
...	

On the calculus side the axiom for e.g. *addJlong* is given as rewrite rule

$$addJlong(fst, snd) \rightsquigarrow moduloLong(add(fst, snd))$$

and expresses the meaning of addition with overflow w.r.t. the value range of `long` in terms of the standard arithmetic addition and a modulo operation. The calculus rewrite rule defining function *moduloLong* is

$$\begin{aligned}&moduloLong(i) \rightsquigarrow \\ &\quad add(long\_MIN, mod(add(long\_HALFRANGE, i), long\_RANGE))\end{aligned}$$

Its definition refers only to the standard arithmetic functions for addition and modulo. The only other elements *long_MIN*, *long_HALFRANGE* and *long_RANGE* are constants like the smallest number of type `long` or the cardinality of the set of all numbers in `long`.

In our logic all function symbols are interpreted as total functions. There are several possibilities to deal with terms like $div(x,0)$ like (a) returning a default value, (b) returning a special named error element or (c) using underspecification. Solution (a) might easily hide an existing problem in the specification as it might render it provable but not matching the specifiers intuition, solution (b) would require to extend the definition of all functions defined on the integers to deal with the special error element. For these reasons, we choose underspecification, i.e., the semantics of logic does not fix a specific value for *div* in case of a division by zero. Instead each JFOL structure $\mathscr{M}$ assigns $div^{\mathscr{M}}(d,0)$ a fixed but unknown integer value for each dividend $d \in \mathbb{Z}$. These values may differ between different Kripke structures but are not state-dependent in the sense that $div(1,0)$ is assigned a different value in different states.

Besides the avoidance of a proliferation of special error cases or dealing with partial functions, the use of underspecification in the sketched manner provides additional advantages, e.g., a formula like

- $div(x,0) \doteq div(x+1,0)$ is neither a tautology nor unsatisfiable, but
- $div(x,0) \doteq div(x,0)$ remains a tautology.

5.4.2 Variable Binding Integer Operations

Variable binding operators are used to express sums and products. Our theory of integers supports the general versions $sum\{T\ x\}(\phi(x),t)$, $prod\{T\ x\}(\phi(x),t)$ as well as their bounded variants $bsum\{int\ x\}(start,end,t)$ and $bprod\{int\ x\}(start,end,t)$. These functions are defined as follows:

$$sum\{T\ x\}(\varphi(x),t)^{\mathscr{M}} = \sum_{\phi(x)^{\mathscr{M}}} t^{\mathscr{M}} \qquad prod\{T\ x\}(\varphi(x),t)^{\mathscr{M}} = \prod_{\phi(x)^{\mathscr{M}}} t^{\mathscr{M}}$$

$$bsum\{int\ x\}(start,end,t)^{\mathscr{M}} = \sum_{x=start^{\mathscr{M}}}^{end^{\mathscr{M}}-1} t^{\mathscr{M}} \qquad \text{(if } start > end \text{ , otherwise } 0)$$

$$bprod\{int\ x\}(start,end,t)^{\mathscr{M}} = \prod_{x=start^{\mathscr{M}}}^{end^{\mathscr{M}}-1} t^{\mathscr{M}} \qquad \text{(if } start > end \text{ , otherwise } 1)$$

The logic axiomatization for the bounded sum is given as: For any int-typed term t

$$\begin{array}{l}\forall int\ start,end; (bsum\{int\ x\}(start,end,t) \doteq \\ \quad \text{if}(start < end)\,\text{then}\,bsum\{int\ x\}(start,end-1,t) + (\{\backslash\text{subst}\ x;\ start\}t)\,\text{else}(0))\end{array}$$

The bounded sum and bounded product are inclusive for the first argument *start* and exclusive for the second argument *end*. Besides the axioms there are as usual a large number of lemmas that ease reasoning and increase automation. We show here the Taclet definition for the splitting a bounded sum as it highlights a feature of Taclet language that allows one to specify triggers:

— KeY —

```
bsum_split {
  \schemaVar \term int low, middle, high;
  \schemaVar \variables int x;

  \find(bsum{x;} (low, high, t))
  \varcond ( \notFreeIn(x, low), \notFreeIn(x, middle),
             \notFreeIn(x, high) )
  \replacewith (
     \if(low <= middle & middle <= high)
     \then(bsum{x;}(low, middle, t) + bsum{x;}(middle,high,t))
     \else(bsum{x;}(low, high, t)) )

  \heuristics(comprehension_split, triggered)
  \trigger{middle} bsum{x;}(low, middle, t)
                   \avoid middle <= low, middle >= high;
};
```

— KeY —

The above rule splits the bounded sum expression somewhere between the lower and upper bound. The trigger specification is used by the strategies to determine good candidates for the splitting point. A trigger consists of three parts: (i) the schema variables to be instantiated by the trigger (trigger variables) (here: `middle`); (ii) the pattern to be matched against an existing term (or formula) in the sequent (here: `bsum{x;}(low, middle, t)`) (the match determines the value of the trigger variables) and (iii) an optional part *avoid* that specifies the condition under which a candidate should be rejected. In the above case the trigger searches for positions at which to split the bounded sum into two parts such that the first summand already occurs in the sequent. The optional avoid part prevents superfluous splits outside the bounds (or directly at the start).

5.4.3 Symbolic Execution of Integer Expressions in Programs

In the following section, we explain how integer expressions are translated into logic terms. One obstacle to overcome is that there are several integral types in Java but only the logic type *int*. Given the following sequent

$$\Longrightarrow \langle \texttt{i = i + 1;} \rangle \texttt{i} > 0$$

and assume `i` is a program variable that had been declared to be of program type `long` (the logic type of `i` is *int*). As one of summands is of program type `long` the addition is widened to type `long`, i.e., it results only in an overflow (or underflow) if the normal mathematical addition results in a value outside of the range of type `long`.

The assignment is obviously side-effect free and can be directly moved into an update. Modeling the Java semantics faithfully, the application of the according assignment rule should result in

$$\Longrightarrow \{\mathtt{i} := \mathit{addJLong}(\mathtt{i}, \mathtt{1})\}\langle\rangle \mathtt{i} > \mathtt{0}$$

using the addition with overflow function for `long`. However, performing this step within KeY results instead in the sequent

$$\Longrightarrow \{\mathtt{i} := \mathit{javaAddLong}(\mathtt{i}, \mathtt{1})\}\langle\rangle \mathtt{i} > \mathtt{0}$$

where the Java operator `+` has been translated using the function $\mathit{javaAddLong} : \mathit{int} \times \mathit{int} \to \mathit{int}$ (if `i` would have been declared of program type **`int`** the function $\mathit{javaAddInt} : \mathit{int} \times \mathit{int} \to \mathit{int}$ would have been used).

These functions are intermediate representations which are used to represent the translation result of an integer program operation. The reason that we support three different integer semantics to cater for different usage scenarios. The three semantics are

Integer ignoring overflow semantics: All Java integral types and their operations are interpreted as the normal arithmetic operations without overflow. In this semantics the function *javaAddLong* would be interpreted as the arithmetic addition on $\mathbb{Z}$ (the same holds for *javaAddLong*). On the calculus level the corresponding rule would simply rewrite $\mathit{javaAddLong}(t1,t2)$ to $\mathit{add}(t1,t2)$. This semantics does obviously not model the real program semantics of Java and is hence unsound and incomplete w.r.t. to real-world Java programs. It is nevertheless useful for teaching purposes to avoid the complexities which stem from modulo operations.

Java integer semantics: Java integer semantics is modeled as specified by the JLS, i.e., some operations might cause an overflow. This means *javaAddLong* would be interpreted the same as *addJlong*. In this semantics the axiom rule for *javaAddLong* simply replaces the function symbol by *addJlong*. This semantics is sound and (relatively) complete, but comes with higher demands on the automation and requires in general more user interaction.

Integer prohibiting overflow semantics: This semantics provides a middle ground between the two previous semantics. Intuitively, when using this semantics one has to show that all arithmetic operations are safe in the sense that they do not overflow . In case of an overflow, the result is a fixed, but unspecified value. Verifying a program with this semantics ensures that either no overflow occurs, or in case of an overflow, the value of this operation does not affect the validity of the property under verification (i.e., the property is true for any outcome of the operation). This semantics is sound, but only complete for programs that do not rely on overflows.

The predicate symbols *inByte*, *inChar*, *inShort*, *inInt* and *inLong* which determine if the provided argument is in the value range of the named primitive type are also interpreted dependent on the chose integer semantics. While the ignoring overflow semantics always interprets these predicates as true, the two other semantics evaluate define them to be true if the given argument is within the bounds of the primitive type (for instance, $\mathit{inInt}(x)$ is true in the latter semantics iff $x \geq -2^{31} \wedge x < 2^{31}$ is true).

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.)