Runtime Checking for Program Verification

Karen Zee[1], Viktor Kuncak[2], Michael Taylor[3], and Martin Rinard[1]

[1] MIT Computer Science and Artificial Intelligence Laboratory; Cambridge, USA
[2] École Polytechnique Fédérale de Lausanne; Lausanne, Switzerland
[3] University of California, San Diego; La Jolla, USA
{kkz,rinard}@csail.mit.edu, mbtaylor@ucsd.edu, viktor.kuncak@epfl.ch

Abstract. The process of verifying that a program conforms to its specification is often hampered by errors in both the program and the specification. A runtime checker that can evaluate formal specifications can be useful for quickly identifying such errors. This paper describes our preliminary experience with incorporating run-time checking into the Jahob verification system and discusses some lessons we learned in this process. One of the challenges in building a runtime checker for a program verification system is that the language of invariants and assertions is designed for simplicity of semantics and tractability of proofs, and not for run-time checking. Some of the more challenging constructs include existential and universal quantification, set comprehension, specification variables, and formulas that refer to past program states. In this paper, we describe how we handle these constructs in our runtime checker, and describe directions for future work.

1 Introduction

This paper explores the use of a run-time checker in a program verification system Jahob [29]. Our program verification system can prove that the specified program properties hold in all program executions. The system attempts to prove properties using loop invariant inference algorithms [42], decision procedures [30], and theorem provers [8]. As in many other static analysis systems [3, 14] this process has the property that if a correctness proof is found, then the desired property of the program holds in all executions. However, if a proof is not found, this could be either because the property does not hold (there is an error in specification or code), or because the example triggered a limitation of the static verification system (for example, imprecision of loop invariant inference, or limitation of the theorem proving engines). In contrast, run-time checking [11–13] compiles specifications into executable code and executes the specifications while the program is running. Although run-time checking alone cannot guarantee the absence of errors, it can identify concrete executions when errors do appear. Run-time checking is therefore complementary to static verification. Run-time checking is especially useful when developing the code and specifications, when the specifications and code are likely to contain errors due to developer's errors in formalizing the desired properties.

Combining static and run-time checking. Given the complementary nature of these techniques, recent verification systems for expressive properties such as Spec# [3] and JML tools [14, 33] include both a static verifier and a run-time checker that can operated on same annotated source code. However, these systems use different semantics and apply different restrictions on specifications in these two cases. The reason is that the limitations of these two checking technologies are different: some specification constructs are easy to execute but difficult to check statically (e.g., recursive or looping deterministic code or complex arithmetic computations), whereas others can be checked statically but are difficult or impossible to execute (e.g., quantification or comprehensions over unbounded number of objects, specifications that involve mathematical quantities not representable at run time). In practice, however, most properties we encountered are executable if stated in an appropriate way. Note that the same specification would be written differently depending on whether it is meant to be executed or verified statically: compare for example 1) specifications of linked structures in systems such as Jahob [29], which use treeness annotations, mathematical sets, relations, and transitive closure operators with 2) manually written Java methods in systems for constraint solving over imperative predicates [9], which use deterministic worklist algorithms with loops to check the shape of the data structure.

Executing declarative specifications. The goal to perform both static and run-time checking can serve as the guidance in designing the specification language. We believe that specification languages, even if their goal is to be executable, should remain declarative in the spirit. In this paper we therefore start with Jahob's language, which was designed for static analysis in mind, and explore techniques necessary to verify Jahob's specifications at run-time. To assess some of these techniques we built an interpreter that executes both the abstract syntax trees of the analyzed program and the specifications in program annotations. The primary use of the run-time checker is debugging specifications and the program. In addition to verification, this research can also be viewed as contributing to the long-standing problem of executing expressive declarative languages.

Contributions. This paper outlines the challenges in executing specification language designed for static verification, describes the current state of our run-time checker for Jahob, and presents future directions. Our checker can execute specifications that involve quantifiers, set comprehensions, transitive closure, integer and object expressions, sets, and relations. Unlike the run-time checkers that we know of, it can evaluate certain expressions that denote infinite sets, as well as formulas that refer to old values of fields of an unbounded number of objects. Among the main future directions are the development of techniques for compilation, parallelization, and incremental evaluation of run-time checks, and the use of constraint solvers for modular run-time checking.

```
1  class Node { public /*: claimedby DLL */ Node next, prev; }
2  class DLL {
3     private static Node root;
4     /*: public static specvar content :: "obj set";
5         vardefs "content == {x. (root,x) ∈ {(u,v). next u = v}^* ∧ x ≠ null}";
6         invariant backbone: "tree[next]";
7         invariant rootFirst: "root = null ∨ (∀ n. n..next ≠ root)";
8         invariant noNextOutside: "∀ x y. x ≠ null ∧ y ≠ null ∧ x..next = y
9                     → y : content";
10        invariant prevDef: "∀ x y. prev x = y →
11                    (x ≠ null ∧ (∃ z. next z = x) → next y = x) ∧
12                    (((∀ z. next z ≠ x) ∨ x = null) → y = null)";
13    */
14    public static void addLast(Node n)
15    /*: requires "n ∉ content ∧ n ≠ null"
16        modifies content
17        ensures "content = old content ∪ {n}" */
18    {
19       if (root == null) {
20         root = n;
21         n.next = null; n.prev = null;
22         return;
23       }
24       Node r = root;
25       while (r.next != null) {
26          r = r.next;
27       }
28       r.next = n;
29       n.prev = r;
30    }
31    public static void testDriver()
32    /*: requires "content = {}" */
33    {
34       Node n1 = new Node();
35       addLast(n1);
36       Node n2 = new Node();
37       addLast(n2);
38    }
39 }
```

Fig. 1. Doubly-linked list with one operation specified in Jahob

2 Jahob Verification System

Jahob [29] is a program verification system for a subset of Java. The initial focus of Jahob is data structure verification [8, 29–32, 42, 43] for which a simple memory-safe imperative subset of Java [29, Section 3.1] is sufficient.

Figure 1 shows a fragment of a doubly-linked list implementation in Jahob, with the `addLast` operation that inserts a given node at the end of the list. Developers write Jahob specifications in the source code as special comments that start with the "`:`" sign. Developers can therefore compile and run programs using standard Java interpreters and runtimes. Jahob specifications contain formulas in higher-order logic (HOL), expressed in the syntax of the Isabelle interactive theorem prover [35]. The specifications represent a class field `f` as total function f mapping all objects to values, with the convention that $f\texttt{null} = \texttt{null}$ and also $fx = \texttt{null}$ when $x.f$ is not well-typed in Java. Jahob specifications include declarations and definitions of specification variables (such as `content` in Figure 1), data structure invariants (such as *backbone*, *rootFirst*, *noNextOutside*, and *prevDef*), and procedure contracts consisting of preconditions ("requires" clauses), postconditions ("ensures" clauses) and frame conditions ("modifies" clauses). The contract for `addLast` specifies that the procedure 1) requires its parameter `n` to be outside the list content, 2) modifies the list content, and 3) inserts `n` into the list (and does not insert or delete any other elements). Specification variables such as `content` are abstract fields defined by the programmer for the purpose of specification and may contain a definition given after the `vardefs` keyword, which specifies an abstraction function. The `content` variable has the type of a set of object identities and is given by a set comprehension that first constructs a binary relation between objects and their `next` successors, then computes its transitive closure using the higher-order `*` operator on relations, and finally uses it to find all elements reachable from `root`.

Given the class invariants in Figure 1, Jahob invokes its inference engine Bohne [42, 43], which succeeds in automatically computing a loop invariant, proving that the postcondition of `addLast` holds, and proving that there are no run-time errors during procedure execution. In this case Bohne uses the MONA decision procedure for monadic second-order logic of trees [27], but in other cases it uses resolution-based provers [38], satisfiability-modulo theory provers [4], new decision procedures [30], or combinations of these approaches. In general, a successful verification means that the desired property holds, but a failed verification can also occur either due to an error in program or specification or due to a limitation of Jahob's static analysis techniques.

3 Debugging Annotated Code using Run-Time Checking

For a successful verification of such detailed properties as in the example in Figure 1, the developer must come up with appropriate class invariants. A run-time checker can help in this process. For example, when we were writing this example, we initially wrote the following *prevDef0* version of *prevDef* invariant:

1 **invariant** prevDef0: "∀ x y. prev x = y → (x ≠ null → next y = x)"

This formula is reasonably-looking at first sight. Moreover, modular static verification quickly succeeds in proving that if `addLast` satisfies the invariants initially, it preserves the invariants and establishes the postconditions. Unfortunately, the *prevDef* invariant is false whenever there is a non-null object whose `prev` field points to `null`, so, even if true in the very first initial state, it is not preserved by allocation operations outside the `DLL` class.

Executing our run-time checker on the `testDriver` procedure in Figure 1 immediately detects that the `prevDef0` invariant is violated when the execution enters `addLast`. As another illustration, suppose that we write a correct invariant `prevDef` but we omit in Figure 1 line 29 containing the assignment `n.prev=r`. Running the run-time checker on the same `testDriver` procedure identifies the invariant violation at the exit of `addLast` procedure. Compared to constraint solving techniques that could potentially detect such situation the advantage of run-time checking is that it directly confirms that a specific program fragment violates an invariant, it is applicable to constraints for which no decision procedures exist, and it can handle code execution with any number of loop iterations.

4 The Scope of Our Run-Time Checker

Our run-time checker verifies that program states occurring in a given execution satisfy the desired safety properties. These safety properties refer either to a specific program state (for example, the program point at which the assertion is written), or to a relationship between the current program state and a past state identified by a program point (such as program state at a procedure entry). As a result, if we assumed that each program state itself is finite, such run-time checking problem would reduce to the problem of evaluating a formula in a given finite model. This problem has been studied from the viewpoint of finite model theory [16, 22] where it was related to computational complexity classes, in relational databases [20, 36, 37] where the database takes place of program state, in model checking [7, 25], with techniques based on BDDs, and in constraint solving [23, 41]. While these ideas are relevant for our work, there are several challenges that arise when considering run-time checking based on our specification language: quantification over infinite or large domains, representation of specification variables that denote infinite sets, and computation of values that relate to previous program states. Although we cannot hope to support all these constructs in their most general form, we have identified their uses in the examples we encountered so far and we describe how we support these uses in our run-time checker.

5 Quantifiers and Set Comprehensions

Quantifiers and set comprehensions are a great source of expressive power for a specification language. They are essential for stating that a group of objects in

a data structure satisfies the desired properties, for example, being non-null or initialized. The advantages of using quantifiers as opposed to using imperative constructs such as loops to express the desired properties is that quantifiers enjoy a number of well-understood mathematical properties, which makes them appropriate for manual and automated proofs. On the other hand, quantifiers are one of the main sources of difficulty in run-time checking.

Restriction to first-order quantifiers. Jahob's specifications are written in higher-order logic, which admits quantification over sets of objects. This allows expressing properties like minimum spanning tree or graph isomorphism. Most of our data structure specification examples, however, we do not encounter higher-order quantification (even though there are 120 classes that contain first-order quantification). One of the reasons is that higher-order quantification is difficult to reason about statically, so our examples avoid it. Our run-time checker therefore currently supports only first-order quantifiers. The quantified variables are either of integer or of object type. We note that the run-time checker does support some simple uses of higher-order functions, which it eliminates by beta-reduction.

Bounding integer quantifiers. Integers in Jahob denote unbounded mathematical integers. We encounter quantification over integers, for example, when reasoning about indices of arrays. Such quantifiers and set comprehensions are usually bounded, as in the form $\forall x.0 \leq x \wedge x < n \rightarrow \ldots$ or $\{v.\exists i.\ 0 \leq i \wedge i < n \wedge v = a.[i]\}$. We support such examples by syntactically identifying within the quantifier body the expressions that enforce bounds on integers. We use these bounds to reduce quantifiers to finite iteration over a range of integers.

Bounding object quantifiers. Our interpreter implicitly assumes that object quantifiers range only over allocated objects. While this is a very natural assumption, note that this in fact departs from the static analysis semantics of quantifiers in Jahob as well as systems such as ESC/Java [18] and Spec# [3]. The reason is that object allocation produces fresh objects with respect to all currently allocated objects, and the set of allocated objects changes with each allocation. A typical approach to soundly model allocation is to introduce a set of currently allocated objects, denoted `Object.alloc` in Jahob, and keep the domain of interpretation fixed. A statement such as `x = new Node()` is then represented by

```
1 assume x ∉ Object.alloc;
2 Object.alloc := Object.alloc ∪ {x};
```

The change of the set of allocated objects ensures that allocated objects are fresh, which is a crucial piece of aliasing information necessary to verify code that uses linked structures. With this technique, it is possible to use standard verification condition generation techniques to correctly represent state changes. On the other hand, in this model all objects that will ever be used exist at all program points, even before they are allocated. To execute an arbitrary quantification of objects at run-time, it is necessary to combine run-time evaluation of formula over allocated objects with symbolic techniques that determine the truth value

for all objects that are not allocated. The last step is possible in many cases because objects that are not allocated are all isomorphic: they have no incoming and no outgoing fields.

Propagating variable dependencies in multiple quantified statements. Even bounded domains, however, may be large, and we would like to avoid considering all objects in the heap if at all possible. Consider the following formula:

$$\forall x. \forall y.\ x \in \texttt{Object.alloc} \wedge y \in \texttt{Object.alloc} \wedge \texttt{next}\, x = y \longrightarrow P(x, y)$$

In a naive implementation, the run-time checker would iterate over the set of all allocated objects for both of the universal quantifiers, an $O(n^2)$ operation. But in the above formula, the quantified variable y is introduced for the purposes of naming and can be easily evaluated without enumerating all elements of the heap. The runtime checker handles these cases by doing a simple syntactic check in the body of quantified formula to determine if the bound variable is defined by an equality. If it finds an appropriate definition, the run-time checker evaluates the body of the formula without having to enumerate a large number of objects. For example, when computing a set comprehension over all allocated objects, we could straightforwardly compute the elements of the set by evaluating the body of the formula for each element in the domain. But since this is very inefficient, the runtime checker first searches through the body of the formula to determine if the bound variable is defined by an equality. This is often the case, for example, when the set comprehension is expressed in terms of the reachable objects from some root using reflexive transitive closure. In this case, we can compute the elements of the set without having to enumerate all objects within the domain.

6 Specification Variables

Specification variables are useful for representing the abstract view of the state of a class. The developers can use specification variables to specify the behavior of abstract data types without exposing implementation details. Jahob supports two types of specification variables: derived specification variables and ghost variables. These are sometimes referred to as model fields and ghost fields, respectively, as in JML [33].

Ghost variables. A ghost variable is updated by the developer by assigning it values given by HOL formulas using special specification assignment statements in the code. Our run-time checker treats ghost variables similarly to ordinary Java variables. The difference is that, in addition to standard program types such as booleans, integers, and objects, these variables can also have types of tuples and sets of elements and tuples.

When a ghost variable is updated, the right-hand side of the assignment statement consists of a formula that the runtime checker evaluates to produce the new value of the ghost variable. It then stores the resulting value in the same way as it would for the assignment of a normal program variable. This formula

is a standard Jahob formula and may contain quantifiers, set comprehensions, set operations, and other constructs not typically available in Java assignment statements.

The run-time checker supports certain forms of infinite sets. For example, the checker can evaluation the following code:

```
1 //: private ghost specvar X :: "int set";
2 int y = 0;
3 //: X := {z. z > 0};
4 //: assert y ∉ X;
5 y = y + 1;
6 //: assert y ∈ X;
```

where the ghost variable `X` is assigned the value of an unbounded set. The runtime checker handles such cases by deferring the evaluation of `X` until it reaches the assert statements. It then applies formula simplifications that eliminate the set comprehension. This is a particular case of a more general approach where some elements of theorem proving could be applied at run-time [2].

Derived variables. A derived specification variable (such as `content` in Figure 1) is given by a formula that defines it in terms of the concrete state of the program. When the runtime checker evaluates a formula that refers to a standard specification variable, it evaluates the formula that defines the specification variable in the context of the current program state.

7 The old Construct

In Jahob, an old expression refers to the value of the enclosed expression as evaluated on entry to the current procedure and is very useful to express state changes that procedures perform. One simple but inefficient method of providing the checker access to past program state would be to snapshot the heap before each procedure invocation. Unfortunately, this approach is unlikely to be practical because the memory overhead would be a product of the size of the heap and the depth of the call stack. Instead, our run-time checker obtains access to the pre-state by means of a recovery cache (also known as a recursive cache) [21] that keeps track of the original values of modified heap locations. There are several features of this solution. First, it takes advantage of the fact that we need only know the state of the heap on procedure entry, and not the state of any intermediate heaps between procedure entry and the assertion or invariant to be evaluated. Also, where the state of a variable is unchanged, the old value resides in the heap, so that reads do not incur a performance penalty excepting reads of old values. Finally, one of the ideas underlying this solution is that we expect the amount of memory required to keep track of the initial writes to be small relative to the size of the heap. While there is a trade-off between memory and performance—there is now a performance penalty for each write—the overhead is greatest for initial writes, and less for subsequent writes to the same location.

8 Further Related Work

Run-time assertion checking has a long history [13]. Among the closest systems for run-time checking in the context of static verification system are tools based on the Java Modeling Language (JML) and the Spec# system [3]. The JML compiler, jmlc [12] compiles JML-annotated Java programs into bytecode that also includes instructions for checking JML invariants, pre- and post-conditions, and assertions. Other assertion tools for JML include Jass [5] and jmle [28]. One of the goals in the design of JML was to produce a specification language that was Java-like, to make it easier for software engineers to write JML specifications. It also makes JML specifications easier to execute. Jahob, on the other hand, was designed as a static verification system and uses an expressive logic as its specification language. The advantage of this design is that the semantics of the specifications is clear, and the verification conditions generated by the system can easily be traced back to the relevant portions of the specification, which is very helpful in the proof process. One example of this difference in philosophy appears in the treatment of old expressions. In JML, an old expression may not contain a variable that is bound by a quantifier outside of that expression. This restriction ensures that the body of the old expression can be fully evaluated at the program point to which the old expression refers, but prevents writing certain natural specifications such as $\forall i.0 \leq i \wedge i < a.\texttt{length} \rightarrow a[i] = (\mathsf{old}\ a[i])$.

We are not aware of any techniques used to execute such specifications as in Jahob in the context of programming language run-time checking systems. Techniques for checking constraints on databases [6, 19, 20, 24, 36, 37] contain relevant techniques, but use simpler specification specification languages and are optimized for particular classes of checks.

While run-time assertion checking systems concern themselves with checking properties of the heap, event-based systems [1,34,40] are concerned with checking properties of the trace. Quantification is implicit over all events that adhere to the pattern described by the specification. The matching of an event binds the free variables in the specification to specific objects in the heap. Since explicit quantifiers are generally not available in the specification language of event-based systems, the properties encoded can only refer to a statically-determined number of objects in the heap for each event instance, though the number of event instances matched is unbounded.

9 Conclusions and Future Work

We have described a simple run-time checker for a subset of an expressive higher-order logic assertions in the Jahob verification system. Our run-time checker can execute specifications that involve quantifiers, set comprehensions, transitive closure, integer and object expressions, sets, and relations. It can evaluate certain expressions that denote infinite sets, as well as formulas that refer to old values of fields of an unbounded number of objects. We have found the run-time checker useful for debugging specifications and code. The run-time checker is currently

built as an interpreter and in our examples it exhibits slowdown of several orders of magnitude compared to compiled Java code without run-time checks, and is meant for debugging and analysis purposes as opposed to the instrumentation of large programs. Among the main directions for future work are compilation of run-time checks [10, 15] to enable checking of the assertions that were not proved statically [17], memoization and incremental evaluation of checks [39], and combination with a constraint solver to enable modular run-time checking [26].

References

1. C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *Proc. 20th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 345–364, 2005.
2. K. Arkoudas and M. Rinard. Deductive runtime certification. In *Proceedings of the 2004 Workshop on Runtime Verification (RV 2004)*, Barcelona, Spain, April 2004.
3. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS: Int. Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices*, 2004.
4. C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518, 2004.
5. D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass–Java with assertions. In *RV01*, volume 55 of *ENTCS*, pages 103–117, 2001.
6. P. A. Bernstein and B. T. Blaustein. Fast methods for testing quantified relational calculus assertions. In *Proceedings of the 1982 ACM SIGMOD international conference on Management of data*, pages 39–50. ACM Press, 1982.
7. D. Beyer, A. Noack, and C. Lewerentz. Efficient relational calculation for software analysis. *IEEE Trans. Software Eng.*, 31(2):137–149, 2005.
8. C. Bouillaguet, V. Kuncak, T. Wies, K. Zee, and M. Rinard. Using first-order theorem provers in a data structure verification system. In *VMCAI'07*, November 2007.
9. C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis*, pages 123–133, July 2002.
10. F. Chen, M. d'Amorim, and G. Rosu. Checking and correcting behaviors of java programs at runtime with java-mop. *Electr. Notes Theor. Comput. Sci.*, 144(4):3–20, 2006.
11. F. Chen and G. Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *Object-Oriented Programming, Systems, Languages and Applications(OOPSLA'07)*, 2007.
12. Y. Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Iowa State University, April 2003.
13. L. A. Clarke and D. S. Rosenblum. A historical perspective on runtime assertion checking in software development. *SIGSOFT Softw. Eng. Notes*, 31(3):25–37, 2006.

14. D. R. Cok and J. R. Kiniry. Esc/java2: Uniting ESC/Java and JML. In *CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices*, 2004.
15. B. Demsky, C. Cadar, D. Roy, and M. C. Rinard. Efficient specification-assisted error localization. In *Second International Workshop on Dynamic Analysis*, 2004.
16. H. D. Ebbinghaus and J. Flum. *Finite Model Theory*. Springer-Verlag, 1995.
17. C. Flanagan. Hybrid type checking. In *POPL*, pages 245–256, 2006.
18. C. Flanagan, K. R. M. Leino, M. Lilibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *ACM Conf. Programming Language Design and Implementation (PLDI)*, 2002.
19. T. Griffin, L. Libkin, and H. Trickey. An improved algorithm for incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):508–511, 1997.
20. L. J. Henschen, W. McCune, and S. A. Naqvi. Compiling constraint-checking programs from first-order formulas. In H. Gallaire, J.-M. Nicolas, and J. Minker, editors, *Advances in Data Base Theory, Proceedings of the Workshop on Logical Data Bases, ISBN 0-306-41636-0*, volume 2, pages 145–169, 1984.
21. J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell. A program structure for error detection and recovery. In *Proc. Intl. Sym. on Operating Systems*, volume 16 of *LNCS*, pages 171–187, 1974.
22. N. Immerman. *Descriptive Complexity*. Springer-Verlag, 1998.
23. D. Jackson. *Software Abstractions: Logic, Language, & Analysis*. MIT Press, 2006.
24. H. V. Jagadish and X. Qian. Integrity maintenance in object-oriented databases. In *Proceedings of the 18th Conference on Very Large Databases, Morgan Kaufmann pubs. (Los Altos CA), Vancouver*, 1992.
25. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.
26. S. Khurshid and D. Marinov. TestEra: Specification-based testing of java programs using SAT. *Autom. Softw. Eng.*, 11(4):403–434, 2004.
27. N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA implementation secrets. In *Proc. 5th International Conference on Implementation and Application of Automata*. LNCS, 2000.
28. B. Krause and T. Wahls. jmle: A tool for executing JML specifications via constraint programming. In *FMICS06*, volume 4346 of *LNCS*, pages 293–296, Bonn, Germany, 2007.
29. V. Kuncak. *Modular Data Structure Verification*. PhD thesis, EECS Department, Massachusetts Institute of Technology, February 2007.
30. V. Kuncak, H. H. Nguyen, and M. Rinard. Deciding Boolean Algebra with Presburger Arithmetic. *J. of Automated Reasoning*, 2006. `http://dx.doi.org/10.1007/s10817-006-9042-1`.
31. V. Kuncak and M. Rinard. An overview of the Jahob analysis system: Project goals and current status. In *NSF Next Generation Software Workshop*, 2006.
32. V. Kuncak and M. Rinard. Towards efficient satisfiability checking for boolean algebra with presburger arithmetic. In *Conference on Automateded Deduction (CADE-21)*, 2007.
33. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin. *JML Reference Manual*, February 2007.

34. M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a Program Query Language. In *Proc. 20th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2005.
35. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.
36. R. Paige. Applications of finite differencing to database integrity control and query/transaction optimization. In H. Gallaire, J.-M. Nicolas, and J. Minker, editors, *Advances in Data Base Theory, Proceedings of the Workshop on Logical Data Bases, ISBN 0-306-41636-0*, volume 2, pages 171–209, 1984.
37. X. Qian and G. Wiederhold. Knowledge-based integrity constraint validation. In W. W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi, editors, *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 3–12. Morgan Kaufmann, 1986.
38. S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.
39. A. Shankar and R. Bodik. Ditto: Automatic incrementalization of data structure invariant checks. In *PLDI*, 2007.
40. V. Stolz and E. Bodden. Temporal assertions using AspectJ. 2005.
41. E. Torlak and D. Jackson. Kodkod: A relational model finder. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2007.
42. T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard. Field constraint analysis. In *Proc. Int. Conf. Verification, Model Checking, and Abstract Interpratation*, 2006.
43. T. Wies, V. Kuncak, K. Zee, A. Podelski, and M. Rinard. Verifying complex properties using symbolic shape analysis. In *Workshop on Heap Abstraction and Verification (collocated with ETAPS)*, 2007.