An Overview of the Jahob Analysis System

Project Goals and Current Status

Viktor Kuncak and Martin Rinard
MIT Computer Science and Artificial Intelligence Lab, Cambridge, USA
{vkuncak,rinard}@csail.mit.edu

Abstract

We present an overview of the Jahob system for modular analysis of data structure properties. Jahob uses a subset of Java as the implementation language and annotations with formulas in a subset of Isabelle as the specification language. It uses monadic second-order logic over trees to reason about reachability in linked data structures, the Isabelle theorem prover and Nelson-Oppen style theorem provers to reason about high-level properties and arrays, and a new technique to combine reasoning about constraints on uninterpreted function symbols with other decision procedures. It also incorporates new decision procedures for reasoning about sets with cardinality constraints. The system can infer loop invariants using new symbolic shape analysis. Initial results in the use of our system are promising; we are continuing to develop and evaluate it.

1 Introduction

Complex software systems currently play a crucial role in the management and operation of our society. Moreover, this role will only increase in importance as software becomes even more pervasively deployed across the activities, infrastructure, and devices of our society. Given this central role, software reliability is a critical and increasingly important issue.

The goal of the Jahob project is to increase software reliability by statically verifying that certain classes of errors can never occur. The Jahob system analyzes annotated programs written in a subset of Java. A basic idea behind Jahob is to model the state that the program manipulates (its data structures) as abstract sets of objects and relations between these objects. The program uses these sets and relations to state key data structure consistency constraints that must hold between the data structures. Each method also uses these sets and relations to state its specification, which consists of a precondition and a postcondition. Given the invariants and specifications, the Jahob verifier statically analyzes the program to ensure that 1) it preserves important data structure consistency properties, and 2) each method conforms to its specification. Method specifications connect the actions of the program to the data structures that it manipulates, enabling the verification of properties that relate actions and state.

There are several challenges associated with this effort. First, there must be a verified connection between the concrete data structures that the program manipulates and the sets and relations that the Jahob analyzer operates with. To establish this connection, Jahob program encapsulate their data structures in modules, with each module containing an abstraction function that maps the encapsulated concrete data structure to the corresponding sets and relations.

A second challenge is that the Jahob analyzer is designed to verify extremely precise, detailed properties that are significantly beyond the reach of traditional analyses. Moreover, the range of potential properties to verify is extremely large, making it implausible that any single analysis will be able to verify all properties of interest. The Jahob system is therefore structured to incorporate multiple specialized analyses, each of which is tailored to analyze a targeted class of properties. Together, these analyses are capable of verifying important properties of unprecedented sophistication and importance.

2 Example

In this section we use a simple list example to demonstrate how the Jahob system can verify data structure consistency properties. Figure 1 presents the specification for a standard `List` class. The running program uses a standard linked list data structure to implement instances of this class (we present and discuss the implementation below). Clients, however, should not be concerned with the details of any particular implementation. The specification of the `List` class therefore serves as an interface that abstracts away the particular implementation details of the class, leav-

```
class List
{
  /*: public static specvar content :: objset; */

  public List() /*:
    modifies content
    ensures "content = {}"
  */
  public void add(Object o) /*:
    requires "o ~: content & o ~= null"
    modifies content
    ensures "content = old content Un {o}"
  */
  public boolean empty() /*:
    ensures "result = (content = {})"
  */
  public Object getOne() /*:
    requires "content ~= {}"
    ensures "result : content"
  */
  public void remove (Object o) /*:
    requires "o : content"
    modifies content
    ensures "content = old content - {o}"
  */
}
```

Figure 1. List Specification

ing behind only those aspects of the class upon which clients rely.

In this case, the `List` specification uses the abstract specification variable `content` to hold the set of objects present in the list. This set does not exist when the program runs — it is simply an abstraction that the Jahob program uses to express the specification and that the Jahob verifier uses as it verifies the program.

As Figure 1 shows, Jahob is structured as an annotation language for Java. Jahob annotations appear as comments to the standard Java compiler. It is possible to distinguish Jahob annotations from standard comments by the fact that Jahob annotations all start with either `//:` or `/*:` — in other words, they have a ":" after the initial comment token. For example, the first comment in Figure 1 declares the specification variable `content` (which, as mentioned above, abstracts the contents of the list).

2.1 Method Interfaces

After the declaration of the `specvar` specification variable, the `List` specification contains a sequence of method interface declarations. Each declaration may contain a `requires` clause, which states the precondition of the method; a `modifies` clause, which states the sets and relations that the method may modify; and an `ensures` clause, which states the properties that the method guarantees will hold when it returns, assuming that the precondition held when it was invoked.

```
class Client {
  List a, b;
  /*:
    public ghost specvar init :: bool;

    invariant
    "init -->
       a ~= null & b ~= null &
       a..List.content Int b..List.content = {}";
  */
  public Client() /*:
    modifies "List.content"
    ensures "init"
  */
  {
      a = new List();
      b = new List();
      Object x = new Object(); a.add(x);
      Object y = new Object(); a.add(y);
      //: init := "True";
  }
  public static void move() /*:
    requires "init"
    modifies "List.content"
    ensures "a..List.content = {}"
  */
  {
      while (!a.empty()) {
          Object o = a.getOne();
          a.remove(o);
          b.add(o);
      }
  }
}
```

Figure 2. List Client

The `List` constructor `List()`, for example, modifies the `content` specification variable — specifically, it ensures that the `content` specification variable is empty when it constructs the `List`.

Note that the program may invoke the `List` constructor multiple times to construct many different lists. According to the semantics of Jahob, each instantiation has its own specification variable `content`. It is therefore possible to write specifications that relate different instances of the `specvar` variable that come from different instantiations of the `List` module. One could, for example, state that one instantiation contains a set of objects that is a subset of another, or that two lists contain disjoint objects.

We next consider the interface for the `add(o)` method, which adds the object `o` to the list. Here the `requires` clause states that `o` must not already be in the list (`o ~: content`) and that `o` must not be `null` (`o~=null`). As this example illustrates, developers can use boolean combinations of clauses in the `requires` and `ensures` clauses.

The `ensures` clause of the `add` method uses the `Un` (union) operator to state that the effect of the `add` method is to add the object o to the set of objects content already in the list. The remaining methods (empty, getOne, and remove) similarly use requires, modifies, and ensures clauses to specify their interfaces.

2.2 List Client

We next show how a client can instantiate the List class to obtain multiple List instances, specify invariants involving these instances, manipulate the lists, and use the Jahob system to verify that the program correctly respects the invariants. The Client class in Figure 2 creates two lists (a and b), adds some objects to these lists, then moves all of the elements from a into b.

The key invariant in this example is that the sets of elements in the two lists are disjoint and remain disjoint throughout all of the manipulations of the client. There is, however, a technical detail that somewhat complicates the expression of this invariant. Specifically, before the client is instantiated, the lists do not exist. It therefore does not make sense to express the invariant directly as holding whenever the program executes. Instead, the Client uses the boolean init specification variable to state that the invariant holds whenever the Client exists. The invariant in Figure 2 also states that, once the Client has been initialized, that a and b are not null.

2.3 List Implementation

We next discuss the implementation of the List class. There are two key considerations: 1) implementing the List methods in Java, and 2) establishing the connection between the List's Java data structures and the abstract specification variables used to specify the List interface. Figure 3 presents the state of the List.

The implementation uses the variable first to refer to the first Node in the list. Each Node object has a field next that contains a reference to the next object in the list and a field data that contains a reference to the object in the list. The private specification variable nodes is the set of all Nodes that is reachable by following next references starting from the first variable.

The Jahob specification uses an abstraction function to define the contents of the nodes set. This abstraction function consists of a set comprehension that states that nodes is the set of all objects n in the reflexive, transitive closure of the next relation on Node objects starting with first. The specification can then use the node set to define the content set as the set of all objects which objects in the node set reference. This definition uses the existential quantifier EX in its set comprehension.

Note that these abstraction functions directly reference implementation entities (first, next) to define the sets nodes and contents in terms of the state that the implementation uses to represent the list. The abstraction functions therefore establish a formal connection between the concrete implementation state and the abstract specification state. This connection allows the the Jahob verifier to start with facts that have been established by reasoning about the abstract state and conclude facts that are valid about the concrete state of the program as it is running.

In our example, the Jahob verification of the disjointness of the two List contents sets in the client in Figure 2, in combination with the abstraction function, enables the Jahob verifier to conclude that the concrete lists are disjoint as well. Of course, this verification also depends on the verification that the list methods correctly implement their interfaces. For this verification to succeed, the concrete data structures must satisfy several additional invariants. Figure 3 presents these properties — specifically, the list must be acyclic with no sharing of sublists, no node in the list refers to the first node, and the data references are not shared.

```
public List() { }

public void add(Object o) {
    Node n = new Node();
    n.data = o;
    n.next = first;
    first = n;
}

public boolean empty() {
    return (first==null);
}

public Object getOne() {
    return first.data;
}

public void remove (Object o) {
    if (first!=null) {
        if (first.data==o) {
            first = first.next;
        } else {
            Node prev = first;
            Node current = first.next;
            boolean go = true;
            while (go && (current!=null)) {
                if (current.data==o) {
                    prev.next = current.next;
                    go = false;
                }
                current = current.next;
            }
        }
    }
}
```

Figure 4. List Implementation Methods

```
class List
{
    private Node first;
    /*:
      // representation nodes:
      specvar nodes :: objset;
      private vardefs "nodes == { n. n ~= null & rtrancl_pt (% x y. x..Node.next = y) first n}";

      // list content:
      public specvar content :: objset;
      private vardefs "content == {x. EX n. x = n..Node.data & n : nodes}";

      // next is acyclic and unshared:
      invariant "tree [List.first, Node.next]";

      // 'first' is the beginning of the list:
      invariant "first = null |
                 (first : Object.alloc &
                  (ALL n. n..Node.next ~= first &
                          (n ~= this --> n..List.first ~= first)))";

      // no sharing of data:
      invariant "ALL n1 n2. n1 : nodes & n2 : nodes & n1..Node.data = n2..Node.data --> n1=n2";
    */
}
class Node {
  public /*: claimedby List */ Object data;
  public /*: claimedby List */ Node next;
}
```

Figure 3. List Implementation State and Invariants

Figure 4 presents the implementation of the `List` methods. These methods provide a standard list implementation. They manipulate only the concrete data structures that make up the list. The Jahob verifier must check that, given the definition of the abstract `content` set in Figure 3 and the method interfaces in Figure 1 (which provide the interfaces in terms of the abstract `content` set), that the method implementations in Figure 4 correctly implement the abstract method interfaces in Figure 1.

2.4 Verification

A key verification challenge is that there are an enormous number of possible data structures, many of which may require specialized verification strategies. It is therefore difficult to imagine that any single verification algorithm could successfully verify all data structure implementations. In our example, the verification of the `List` implementation involves detailed reasoning about the references in the implementation. Other programs may use array-based data structures such as hash tables that produce very different verification conditions. The Jahob framework is therefore set up as a verification condition generator that can invoke any one of a number of decision procedures to discharge the proof obligations provided by the verification condition generator. By populating Jahob with a variety of decision procedures, each of which may be specialized to the verification conditions that arise in the analysis of different data structures or clients, Jahob can effectively deploy very specialized, even unscalable, techniques to verify the full range of data structure implementations and clients.

One issue that arises in the generation of the verification conditions is loop invariants. The current verification condition generator is able to exploit the availability of explicitly-provided loop invariants for complex code. It is also able to leverage loop invariant inference engines, including speculative engines that may generate incorrect loop invariants. Any incorrect loop invariants would be detected and rejected during the verification condition analysis.

In our example, the verification condition generator analyzes each method in turn. It appropriately augments the `requires` and `ensures` clauses with the specified invariants to ensure that the methods preserve them. The verification conditions for the data structure implementation could be verified, for example, by a combination of field constraint analysis [80] and the MONA decision procedure [40]. Loop invariants could be provided explicitly or inferred by symbolic shape analysis [80, 65, 79].

The verification conditions for the client could be discharged by a decision procedure specialized for reasoning about membership changes in abstract sets of objects. It is also possible in many cases to use off-the-shelf automated theorem provers [78] to discharge these kinds of verification conditions.

3 Status

We have implemented the Jahob framework, populated it with interfaces to the Isabelle interactive theorem prover [63], the SMT-LIB interface [67] to Nelson-Oppen style [62] theorem provers, the MONA decision procedure [40], and a decision procedure for Boolean Algebra with Presburger Arithmetic [43] based on reduction to the Omega decision procedure [66] for Presburger arithmetic. We are using a simple goal decomposition technique to prove different conjuncts in the goal using different decision procedures. In addition, we are using field constraint analysis [80] to combine reasoning about uninterpreted function symbols with reasoning using other decision procedures.

We have verified implementations and uses of global data structures. By providing intermediate assertions we have verified implementations of operations on association lists. We have also annotated and partially verified high-level properties in an implementation of a turn-based strategy game. We have also implemented a mechanisms for reasoning about data structure representation in the presence of dynamic data structure instantiation, combining the ideas from the Hob project [47] with approaches from systems such as Spec# [6]. We are currently evaluating the practicality of our approach.

4 Related Work

Key features of Jahob system are modular reasoning with expressive procedure contracts and support for data abstraction, and automated support for reasoning about linked data structure implementation and usage. Jahob therefore builds on program verification research to provide a framework for modular analysis, and builds on new analyses for data structure implementation and data structure use to provide a higher degree of automation than verification frameworks based on general-purpose reasoning.

Verification systems with modular reasoning. Systems based on verification-condition generation and theorem proving include the program verifier [39], the interactive program verifier [17], the Stanford Pascal Verifier [74, 60], the Gypsy environment [28], Larch [30], ESC/Modula-3 [16], ESC/Java [22], ESC/Java2 [12], Boogie [6], Krakatoa [55], KeY [3], as well as more general frameworks such as ACL2 [38, 59], and STeP [8], and PVS [64]. Traditionally, these systems are based on verification condition generation combined with theorem provers. They typically require loop invariants, and additionally either require either interaction with the theorem prover or lemmas specific for the program being verified. Specification frameworks include Z [81], VDM [36], B [2], RAISE [13]. Many of these frameworks recognize the importance of data abstraction [36], which is an important component of Jahob. Some of these frameworks provide no automation for performing formal proofs, and some provide support in terms of verification condition generators and interactive theorem provers [1]. Jahob, on the other hand, aims at providing automated proofs that data structures conform to their abstraction; previous approaches have been less ambitious either in terms of automation [36] or in terms of using lighter-weight substitute of specification variables [51].

Recently, verification systems have incorporated techniques for inferring loop invariants [23, 21, 11, 50]. Like more specialized analyses [75, 82, 19, 70, 24], such techniques for loop invariant inference are effective for analyzing simple array data structures and basic memory safety properties, but have so far been limited in the range of properties that they can prove about linked data structures. These systems are compatible with our methodology of combining specialized analyses based on abstract interpretation to increase the automation in the context of a verification framework; one of the properties that makes Jahob different is the ability to utilize recently developed precise data structure analyses such as shape analysis.

Shape analysis. Shape analyses are precise analyses for linked data structures. They were originally used for compiler optimizations [37, 27, 26], but subsequently evolved into more precise analyses that have been successfully used to analyze invariants of data structures that are of interest for verification [42, 25, 41, 49, 58, 71]. Most shape analyses that synthesize loop invariants are based on precomputed transfer functions and a fixed set of properties to be tracked; recent approaches enable automation of such computation using decision procedures [86, 84, 85, 65, 80] or finite differencing [69].

Recently there has been a resurgence of decision procedures and analyses for linked list data structures [4, 18, 54, 7, 68], where the emphasis is on predictability (decision procedures for well-defined classes of properties of linked lists), efficiency (membership in NP), the ability to interoperate with other reasoning procedures, and modularity.

Shape analyses are among the most sophisticated analyses for structural properties of programs; they have also been applied to verify properties such as sorting, by abstracting the ordering relation [53, 58]. Analyses and decision procedures have also been constructed that combine reasoning about reachability and reasoning about quantitative properties such as length of lists and height and balancing of trees [32, 31, 45, 57, 9]. Size constraints can be imposed on set abstractions of data structures, yielding logics that can reason about numbers of data structure elements and support quantifiers [43].

New logics were recently proposed for reasoning about reachability, such as the logic of reachable shapes [83]. Existing logics, such as guarded fixpoint logic [29] and description logics with reachability [10] are attractive because of their expressive power, but so far no decision procedures for these logics have been implemented. Automated theorem provers such as Vampire [78] can be used to reason about properties of linked data structures, but axiomatizing reachability in first-order logic is non-trivial in practice [61, 52] and not possible in general.

Software Model Checking. Recent trends indicate the convergence of shape analysis with predicate abstraction [5, 33], with a spectrum of increasingly complex domains ranging from propositional combinations of predicates [5], through quantified propositional combinations [23], indexed predicates [44], to symbolic shape analysis [80, 65, 79]. The field remains an active area of research, with different approaches demonstrating different precision/efficiency/automation tradeoffs.

Typestate systems. Because many precise analysis approaches are difficult to scale, it is important to be able to combine them with more scalable analyses. Jahob uses expressive procedure interfaces to achieve such a combination, which means that scalable analyses must be able to communicate using procedure interfaces. Typestate analyses have emerged as data-flow analyses that take into account user-supplied interfaces [73, 14, 15]. In the Hob project [48, 87, 46] we have demonstrated that a combination of typestate analysis with shape analysis is feasible when interfaces use abstract sets to abstract global data structures. One of the goals of Jahob is to demonstrate that such an approach is feasible for a more general class of procedure interfaces that involve not only sets, but also relations.

Bug finding tools for complex properties. Given that many verification attempts demonstrate bugs in specifications or code, it is useful to supplement verification tools with bug fining tools. Finite model checkers such as the Alloy Analyzer [34] can be used to find bugs in code that manipulates linked data structures [35, 76]. Explicit state model checking and testing approaches can also be effective for this purpose [20, 56, 72, 77]. Although somewhat orthogonal to verification, bug finding can be combined with verification in productive ways, and we may consider such combinations in the future.

5 Conclusion

Software reliability is an increasingly important concern for our society. The automatic verification of program properties promises to address this concern by eliminating potential sources of software errors. The Jahob project focuses on data structure consistency properties and connections between the actions of the program and the effect that these actions have on the state. The combination of a general verification condition generator and an architecture that supports the integration of multiple specialized analyses is designed to enable the verification of properties of unprecedented precision.

Acknowledgements. We thank Thomas Wies, Karen Zee, Peter Schmitt, and Hai Huu Nguyen for contributions to the Jahob project.

References

[1] The B-toolkit. http://www.b-core.com/ONLINEDOC/BToolkit.html, January 2006.

[2] J.-R. Abrial, M. K. O. Lee, D. Neilson, P. N. Scharbach, and I. Sørensen. The B-method. In *Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development-Volume 2*, pages 398–405. Springer-Verlag, 1991.

[3] W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Haehnle, W. Menzel, and P. H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. In *Proceedings, 8th European Workshop on Logics in AI (JELIA), Malaga, Spain*, 2000.

[4] I. Balaban, A. Pnueli, and L. Zuck. Shape analysis by predicate abstraction. In *VMCAI'05*, 2005.

[5] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. ACM PLDI*, 2001.

[6] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS 2004: International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices*, March 2004.

[7] J. Bingham and Z. Rakamarić. A logic and decision procedure for predicate abstraction of heap-manipulating programs. Technical Report TR-2005-19, UBC Department of Computer Science, September 2005.

[8] N. Bjørner, A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H. B. Sipma, and T. E. Uribe. STeP: Deductive-algorithmic verification of reactive and real-time systems. In *8th CAV*, volume 1102, pages 415–418, 1996.

[9] M. Bozga and R. Iosif. Quantitative verification of programs with lists. In *Proc. NATO Workshop on Verification of Infinite-state Systems with Applications to Security (VISSAS 2005)*, 2005.

[10] D. Calvanese, G. De Giacomo, and M. Lenzerini. Reasoning in expressive description logics with fixpoints

based on automata on infinite trees. In *Proc. of the 16th Int. Joint Conf. on Artificial Intelligence (IJCAI'99)*, pages 84–89, 1999.

[11] B.-Y. E. Chang and K. R. M. Leino. Abstract interpretation with alien expressions and heap structures. In *VMCAI'05*, January 2005.

[12] D. R. Cok and J. R. Kiniry. Esc/java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an internet voting tally system. In *CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices*, 2004.

[13] B. Dandanell. Rigorous development using RAISE. In *Proceedings of the conference on Software for citical systems*, pages 29–43. ACM Press, 1991.

[14] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proc. ACM PLDI*, 2002.

[15] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. ACM PLDI*, 2001.

[16] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report 159, COMPAQ Systems Research Center, 1998.

[17] L. P. Deutsch. *An Interactive Program Verifier*. PhD thesis, University of California Berkeley, 1973.

[18] D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS'06*, 2006.

[19] N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In *Proc. 7th International Static Analysis Symposium*, 2000.

[20] D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In *VMCAI*, 2004.

[21] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for esc/java. In *FME '01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, pages 500–517, London, UK, 2001. Springer-Verlag.

[22] 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.

[23] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Proc. 29th ACM POPL*, 2002.

[24] D. Foulger and S. King. Using the spark toolset for showing the absence of run-time errors in safety-critical software. In *Ada-Europe 2001*, pages 229–240, 2001.

[25] P. Fradet and D. L. Métayer. Shape types. In *Proc. 24th ACM POPL*, 1997.

[26] R. Ghiya and L. Hendren. Is it a tree, a DAG, or a cyclic graph? In *Proc. 23rd ACM POPL*, 1996.

[27] R. Ghiya and L. J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. In *Proc. 8th Workshop on Languages and Compilers for Parallel Computing*, 1995.

[28] D. I. Good, R. M. Cohen, and J. Keeton-Williams. Principles of proving concurrent programs in gypsy. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 42–52, New York, NY, USA, 1979. ACM Press.

[29] E. Grädel. Decision procedures for guarded logics. In *Automated Deduction - CADE16. Proceedings of 16th International Conference on Automated Deduction, Trento, 1999*, volume 1632 of *LNCS*. Springer-Verlag, 1999.

[30] J. Guttag and J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[31] P. Habermehl, R. Iosif, and T. Vojnar. Automata-based verification of programs with tree updates. In *TACAS'06*, 2006.

[32] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *ACM SIGPLAN Symp. on Principles of Programming Languages (POPL)*, 2005.

[33] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *31st POPL*, 2004.

[34] D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering / European Software Engineering Conference (FSE/ESEC '01)*, 2001.

[35] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *International Symposium on Software Testing and Analysis, Portland, OR*, 2000.

[36] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International (UK) Ltd., 1986.

[37] N. D. Jones and S. S. Muchnik. *Program Flow Analysis: Theory and Applications*, chapter Chapter 4: Flow Analysis and Optimization of LISP-like Structures. Prentice Hall, 1981.

[38] M. Kaufmann, P. Manolios, and J. S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, 2000.

[39] J. C. King. *A Program Verifier*. PhD thesis, CMU, 1970.

[40] N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA implementation secrets. In *Proc. 5th International Conference on Implementation and Application of Automata*. LNCS, 2000.

[41] N. Klarlund and M. I. Schwartzbach. Graph types. In *Proc. 20th ACM POPL*, Charleston, SC, 1993.

[42] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Annual ACM Symp. on Principles of Programming Languages (POPL)*, 2002.

[43] V. Kuncak, H. H. Nguyen, and M. Rinard. An algorithm for deciding BAPA: Boolean Algebra with Presburger Arithmetic. In *20th International Conference on Automated Deduction, CADE-20*, Tallinn, Estonia, July 2005.

[44] S. K. Lahiri and R. E. Bryant. Indexed predicate discovery for unbounded system verification. In *CAV'04*, 2004.

[45] S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *POPL'06*, 2006.

[46] P. Lam, V. Kuncak, and M. Rinard. On our experience with modular pluggable analyses. Technical Report 965, MIT CSAIL, September 2004.

[47] P. Lam, V. Kuncak, and M. Rinard. Cross-cutting techniques in program specification and analysis. In *4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, 2005.

[48] P. Lam, V. Kuncak, and M. Rinard. Generalized typestate checking for data structure consistency. In *6th International Conference on Verification, Model Checking and Abstract Interpretation*, 2005.

[49] O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *ESOP*, 2005.

[50] K. R. M. Leino and F. Logozzo. Loop invariants on demand. In *Proceedings of the the 3rd Asian Symposium on Programming Languages and Systems (APLAS'05)*, volume 3780 of *LNCS*, 2005.

[51] K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *Proc. ACM PLDI*, 2002.

[52] T. Lev-Ami, N. Immerman, T. Reps, M. Sagiv, S. Srivastava, and G. Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. In *CADE-20*, 2005.

[53] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *International Symposium on Software Testing and Analysis*, 2000.

[54] R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In R. Cousot, editor, *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI 2005*, volume 3148 of *Lecture Notes in Computer Science*, pages 181–198. Springer, Jan. 2005. Available at http://www.cs.tau.ac.il/∼rumster/vmcai05.pdf.

[55] C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 2003.

[56] D. Marinov. *Automatic Testing of Software with Structurally Complex Inputs*. PhD thesis, MIT, 2005.

[57] S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In *CAV*, pages 476–490, 2005.

[58] A. Møller and M. I. Schwartzbach. The Pointer Assertion Logic Engine. In *Programming Language Design and Implementation*, 2001.

[59] J. S. Moore. A mechanized program verifier. In *IFIP Working Conference on the Program Verifier Challenge*, 2005.

[60] G. Nelson. Techniques for program verification. Technical report, XEROX Palo Alto Research Center, 1981.

[61] G. Nelson. Verifying reachability invariants of linked structures. In *POPL*, 1983.

[62] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM TOPLAS*, 1(2):245–257, 1979.

[63] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[64] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th CADE*, volume 607 of *LNAI*, pages 748–752, jun 1992.

[65] A. Podelski and T. Wies. Boolean heaps. In *Proc. Int. Static Analysis Symposium*, 2005.

[66] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13. ACM Press, 1991.

[67] S. Ranise and C. Tinelli. The smt-lib standard: Version 1.1, 12 April 2005.

[68] S. Ranise and C. G. Zarba. A decidable logic for pointer programs manipulating linked lists, 2005. http://cs.unm.edu/∼zarba/papers/pointers.ps.

[69] T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. In *Proc. 12th ESOP*, 2003.

[70] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 182–195. ACM Press, 2000.

[71] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.

[72] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *ESEC/SIGSOFT FSE*, pages 263–272, 2005.

[73] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE TSE*, January 1986.

[74] N. Suzuki. *Automatic Verification of Programs with Complex Data Structure*. PhD thesis, Stanford University, 1976. reprinted in Outstanding Dissertations in the Computer Science. Garland Publ., Inc., 1979.

[75] N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 132–143, New York, NY, USA, 1977. ACM Press.

[76] M. Taghdiri. Inferring specifications to detect errors in code. In *ASE'04*, 2004.

[77] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Int. Conf. on Automated Software Engineering, 2000*, 2000.

[78] A. Voronkov. The anatomy of Vampire (implementing bottom-up procedures with code trees). *Journal of Automated Reasoning*, 15(2):237–265, 1995.

[79] T. Wies. Symbolic shape analysis. Master's thesis, Universität des Saarlandes, Saarbrücken, Germany, September 2004.

[80] 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.

[81] J. Woodcock and J. Davies. *Using Z*. Prentice-Hall, 1996.

[82] Z. Xu, B. Miller, and T. Reps. Safety checking of machine code. In *Proc. ACM PLDI*, 2000.

[83] G. Yorsh, A. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani. A logic of reachable patterns in linked data-structures. In *Proc. Foundations of Software Science and Computation Structures (FOSSACS 2006)*, 2006.

[84] G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *10th TACAS*, 2004.

[85] G. Yorsh, T. Reps, M. Sagiv, and R. Wilhelm. Logical characterizations of heap abstractions. *TOCL*, 2005. (to appear).

[86] G. Yorsh, A. Skidanov, T. Reps, and M. Sagiv. Automatic assume/guarantee reasoning for heap-manupilating programs. In *1st AIOOL Workshop*, 2005.

[87] K. Zee, P. Lam, V. Kuncak, and M. Rinard. Combining theorem proving with static analysis for data structure consistency. In *International Workshop on Software Verification and Validation (SVV 2004)*, Seattle, November 2004.