Automatic Verification of Strongly Dynamic Software Systems

N. Dor[1,⋆], J. Field[2], D. Gopan[3], T. Lev-Ami[4], A. Loginov[2,⋆⋆], R. Manevich[4], G. Ramalingam[5,⋆⋆⋆], T. Reps[3], N. Rinetzky[4], M. Sagiv[4], R. Wilhelm[6], E. Yahav[2], and G. Yorsh[4]

[1] Panaya Ltd.
nurit@panayainc.com
[2] IBM Research
{jfield,alexey,eyahav}@us.ibm.com
[3] University of Wisconsin
{alexey,gopan,reps}@cs.wisc.edu
[4] Tel Aviv University
{tla,rumster,maon,msagiv,gretay}@tau.ac.il
[5] Microsoft Research
grama@microsoft.com
[6] Universität des Saarlandes
wilhelm@cs.uni-sb.de

Abstract. Strongly dynamic software systems are difficult to verify. By *strongly dynamic*, we mean that the actors in such systems change dynamically, that the resources used by such systems are dynamically allocated and deallocated, and that for both sets, no bounds are statically known. In this position paper, we describe the progress we have made in automated verification of strongly dynamic systems using abstract interpretation with three-valued logical structures. We then enumerate a number of challenges that must be tackled in order for such techniques to be widely adopted.

1 The Problem

We will use the term *strongly dynamic* system to refer to software in which the set of actors in the system changes dynamically, where resources are dynamically allocated and deallocated, and where for both sets no bounds are statically known.

Heap allocation of data structures, which is the principal mechanism for creating structured data in modern languages, is the classical manifestation of a strongly dynamic system. It is well known that manipulating heap-allocated data is error-prone, due primarily to the complexity of potential aliasing relationships among pointer-valued data. However, dynamic resource manipulation

⋆ Work done while the author was at Tel Aviv University.
⋆⋆ Work done while the author was at the University of Wisconsin.
⋆⋆⋆ Work done while the author was at IBM Research.

B. Meyer and J. Woodcock (Eds.): Verified Software, LNCS 4171, pp. 82–92, 2008.

occurs at many levels in modern software; such dynamic resources may include, e.g., persistent data in databases, language-level threads, operating system resources such as files and sockets, and web sessions.

Formally, the state of strongly dynamic systems may be viewed as a evolving universe of entities over which the program operates. Due to its evolving character, such universes are difficult to reason about, both for programmers and for automated reasoning tools. This in turn makes automated verification for such programs both important and challenging.

While automatic memory allocation and garbage collection in modern programming languages has eased the burden of correctly managing the lifetime of heap-allocated memory, reasoning about the *states* of an unbounded number of heap-allocated objects and their interrelationships remains a difficult challenge.

Frequently, strongly dynamic systems are encapsulated in abstract data types that restrict direct access to the underlying evolving universe, and hence allow the programmer to reason only about the data type's interface. In such cases, the principal verification challenge is to ensure that the implementation of the data type correctly realizes the desired abstract properties.

However, in the case of scarce high-level system resources, such as processes, sessions, and buffers, programmers do not have the luxury of automatic resource management or abstract data type encapsulation; instead, they must reason directly about resource state *and* resource lifetime. Verifying the correct usage of such resources is therefore particularly challenging.

Finally, concurrency and distribution drastically complicates the problem of program verification, since both the data *and* control structures of the program operate over unbounded universes.

This position paper sketches the state of art in automatic verification of properties of strongly dynamic systems using abstract interpretation [6] with *three-valued logical structures*.

1.1 Program Properties

Our focus is on verifying that programs satisfy certain specific (but not fixed) safety and liveness properties, such as those illustrated below, as opposed to establishing complete correctness of a program (with respect to its complete specification). The progress made in selective "property verification" in recent years makes us cautiously optimistic about its long-term prospects. It poses several challenging research problems, but promises to play an important and relevant role in industrial software-development practice within a reasonable time frame. We are in general interested in the following safety properties:

Memory Cleanness. In this case, we wish to prove that a program does not perform pointer manipulations that have unpredictable effects. We call these *cleanness* properties, since they are generic to a given programming model, rather than application-specific (although frequently, in order to show cleanness, it is necessary to prove stronger properties as well). For sequential C-like programs, such properties include: (i) absence of null dereferences, (ii) absence of memory

leaks, and (iii) absence of double deallocations. The failure of these properties is frequently exploited by hackers, see, e.g., [26].

The use of garbage collection in modern languages eliminates some of the problems that occur when programmers manage memory allocation and deallocation manually. However, garbage collection does not eliminate the possibility of premature resource depletion due to delayed deallocation.

Similar resource-management problems are possible with other kinds of resources; e.g., database connections, buffers, files, and sockets. In our experience, many serious problems in large applications arise from resources that are freed too late.

Establishing Data-Structure Invariants. Data structures built using pointers can be characterized by invariants describing their "shape" at stable states, i.e., between operations defined by their external interfaces. These invariants are usually not preserved by the execution of individual program statements, and it is challenging to prove that data-structure invariants are reestablished after a sequence of statements executes [15].

Conformance of Library Specifications. In cases where a library's interface is accompanied by a formal specification of key assumptions and guarantees, it is useful to statically verify that a particular client satisfies, or *conforms* to the interface properties. One can then choose to verify that a library's implementation satisfies its interface specification (thus enabling modular reasoning and analysis of full systems), or simply treat the interface specification as presumptively correct (thus limiting the scope of verification to the client). While significant progress has been made in client-component conformance verification (e.g., see [5,8,12,11,2,25,10,7]), doing precise verification that can scale to large and complex programs is challenging.

Concurrency. Concurrent programs introduce a number of challenging verification issues, particularly when the number of concurrent threads may be unbounded. In this context, data and control are strongly related: thread-scheduling information may require an understanding of the structure of the heap (e.g., the structure of the scheduling queue). Also, heap analysis requires information about thread scheduling, because multiple threads may be manipulating the heap simultaneously. In addition to verifying the absence of "generic" concurrency anomalies, such as races and deadlocks, one often wishes to prove application-specific properties of concurrent protocols that are required to hold under arbitrary thread interleavings.

2 What Has Been Achieved So Far

This section summarizes the progress our group has made on property verification using abstract interpretation with three-valued logical structures [30] and the TVLA system [18], a general-purpose abstract-interpretation engine based on three-valued logic.

Abstract interpretation can be used for verification by generating an over-approximation to the set of states that can arise in any valid program execution; the property of interest is established if the over-approximation demonstrates that no undesirable state can be reached. Typically, problems are cast as a set of equations over a semi-lattice of program properties, and solved by means of successive approximation, possibly with extrapolation.

In [30], we showed that first-order logic can be viewed as a parametric framework for defining both the semantics of a program and for expressing a variety of properties to be verified. In this framework, concrete program states are represented by logical structures. Three-valued logic, which adds an "unknown" value to the Boolean values of ordinary two-valued logic, is a natural framework for defining sound, finitary *abstractions* of two-valued structures for the purpose of abstract interpretation.

Memory Cleanness. The first application of TVLA was to show memory cleanness of C programs [9]. The algorithm is rather precise in the sense that it yields very few false alarms but it was only applied to small programs.

Interprocedural Analysis. [27] handles procedures by explicitly representing stacks of activation records as linked lists, allowing rather precise analysis of recursive procedures. However, it does not scale very well. [17] handles procedures by summarizing their behavior. [28] presents a new concrete semantics for programs that manipulate heap-allocated storage which only passes "local" heaps to procedures. A simplified version of this semantics is used in [29] to perform more modular summarization by only representing reachable parts of the heap.

Concurrent Java Programs. [33] presents a general framework for proving safety properties of concurrent Java programs with an unbounded number of objects and threads. In [36] this approach is applied to verify partial correctness of concurrent-queue implementations.

Temporal Properties. [35] proposes a general framework for proving temporal properties of programs by representing program traces as logical structures. A more efficient technique for proving local temporal properties is presented in [31] and applied to compile-time garbage collection in JavaCard programs.

Correctness of Sorting Implementations. In [19], TVLA is applied to analyze programs that sort linked lists. It is shown that the analysis is precise enough to discover that (correct versions) of bubble-sort and insertion-sort procedures do, in fact, produce correctly sorted lists as outputs, and that the invariant "is-sorted" is maintained by list-manipulation operations such as merge. In addition, it is shown that when the analysis is applied to erroneous versions of bubble-sort and insertion-sort procedures, it is able to discover the error. In [20], abstraction refinement is used to *automatically* derive abstractions that are successfully used to prove partial correctness of several sorting algorithms. The derived abstractions are also used to prove that the algorithms possess additional properties, such as stability and anti-stability.

Conformance to API Specifications. [25] shows how to verify that client programs using a library conform to the library's API specifications. In particular, an analysis is provided for verifying the absence of concurrent-modification exceptions in Java programs that use Java collections and iterators. In [34], separation and heterogeneous abstraction are used to scale the verification algorithms and to allow verification of larger programs (several thousands lines of code) that use libraries such as JDBC.

Computing Intersections of Abstractions. [1] considers the problem of computing the intersection (meet) of heap abstractions, namely the greatest lower bound of two sets of 3-valued structures. This problem proves to have many applications in program analysis such as interpreting program conditions, refining abstract configurations, reasoning about procedures [17], and proving temporal properties of heap-manipulating programs, either via greatest-fixed-point approximation over trace semantics or in a staged manner over the collecting semantics. [1] describes a constructive formulation of meet that is based on finding certain relations between abstract heap objects. The enumeration of those relations is reduced to finding constrained matchings over bipartite graphs.

Efficient Heap Abstractions and Representations. [21] addresses the problem of space consumption in first-order state representations by describing and evaluating two new representation techniques for logical structures. One technique uses ordered binary decision diagrams (OBDDs); the other uses a variant of a functional map data structure. The results show that both the OBDD and functional implementations reduce space consumption in TVLA by a factor of 4 to 10 relative to the original TVLA state representation, without compromising analysis time.

[22] presents a new heap abstraction that works by merging shape descriptors according to a partial isomorphism similarity criterion, resulting in a partially disjunctive abstraction. This abstraction provides superior performance compared to the powerset heap abstraction, without any loss of precision, for a suite of TVLA benchmark verification problems.

[23] provides a family of simple abstractions for potentially cyclic linked lists. In particular, it provides a relatively efficient predicate abstraction that allows verification of programs that manipulate potentially cyclic linked lists.

Abstracting Numerical Values. [13] presents a generic solution for combining abstractions of numeric and heap-allocated storage. This solution has been integrated into a version of TVLA. In [14], a new abstraction of numeric values is presented, which like canonical abstraction tracks correlations between aggregates and not just indices. For example, it can identify loops that perform array-kills (i.e., assign values to a an entire array). This approach has been generalized to define a family of abstractions (for relations as well as numeric quantities) that is more precise than pure canonical abstraction and allows the basic idea from [13] to be applied more widely [16].

Assume-Guarantee Reasoning. One of the potential ways to scale up shape analysis is by applying it to smaller pieces of code using specifications. [37] presents

a new algorithm that takes as input a shape descriptor (describing some set of concrete stores X) and a precondition p, and computes the most-precise shape descriptor for the stores in X that satisfy p. This combines abstract interpretation and theorem provers in a novel way. A prototype has been implemented in TVLA, using the SPASS theorem prover.

Safety Properties of Mobile Ambients. The mobile ambient calculus was introduced in [3]. In [24], TVLA was applied to prove safety properties programs in the ambient calculus. The main idea is to code the ambient calculus using two-valued logic, and then use TVLA to obtain a sound over-approximation by reinterpreting the logical formulas in Kleene's three-valued logic.

3 Some Remaining Challenges

We have found the framework of abstract interpretation based on three-valued logic to be remarkably powerful, both in its ability to provide a natural formal framework for reasoning about dynamic resource manipulation, and as a substrate for developing efficient data structures and algorithms for verification of such properties. We believe that the formal and practical strengths of this framework should provide a strong base for further research in verification of strongly dynamic systems in the future. In this section, we outline some of the remaining research challenges in this framework.

Scalability and Precision. For most of the properties discussed in prior sections, automatic verification of a software system of significant size (e.g., web servers, operating systems, or a *compiler*) remains infeasible. The main problem is the scalability of the existing techniques. We believe that we are likely to make steady advances in the scalability of our techniques by (1) exploiting locality in abstractions (e.g., for interprocedural analysis), (2) exploiting compositionality, i.e., exploiting proven properties of small components or ADTs in verification of large systems, (3) dealing with state explosion caused by interleaving of concurrent threads, and (4) developing improved algorithms for manipulating first-order structures.

Determining the properties that are relevant to the verification problem and identifying the objects that need to be reasoned about at any given program point is key to scalable verification using abstract interpretation. This fundamental problem of "choosing the right set of abstractions" appears to be shared by other verification techniques (including deductive approaches) as well. We believe that machine-learning techniques provide one promising *automated* approach to this problem [20]. We are also investigating the use of counterexample-guided abstraction refinement [4] to address this problem in an automated fashion.

Another approach to effective abstraction selection is to induce programmers to annotate programs with information about properties or abstractions relevant to the problem at hand. Currently, programmers have little incentive to add annotations defining properties or abstractions of interest, since the benefit of doing so using current verification technology is low. However, as the power

of verification techniques to perform state-space exploration begins to scale to programs of realistic size, a cycle of positive reinforcement will arise: programmers will be encouraged to annotate their program with properties of utility to verifiers, because by doing so they will receive accurate and precise feedback on critical aspects of program correctness, which will in turn make them more productive programmers. Strong type systems provide a precedent: while programmers were initially skeptical of the benefits of strong typing, there is now little disagreement over its value.

Usability. Even in cases where automated verification is sufficiently scalable, there are a number of usability challenges: (1) For automatic verification to become an accepted part of everyday programming, it must provide useful feedback as quickly as current compilers generate type errors. (2) In cases where verification fails, counterexamples and error explanations must guide programmers quickly to potential sources of errors. (3) Particularly in safety-critical systems, the trusted code base used by a verifier must itself be verified.

Hybrid Verification Techniques. Theorem proving techniques, e.g., [11], have proved extremely useful for verifying properties of programs equipped with user-specified annotations (e.g., procedure pre- and post-conditions, and loop invariants). The power of such techniques derives from their ability to reason precisely about large collections of program states using symbolic techniques. However, such approaches are less successful in the absence of annotations, particularly when induction is required. Some initial steps have been taken at combining theorem proving and abstract interpretation (or model checking), e.g., [25]; further work aimed at exploiting the complementary strengths of these approaches seems desirable.

Combined Static and Dynamic Analysis. Although dynamic analysis (i.e., instrumentation of code execution to detect anomalies at runtime) cannot by itself prove a program correct, the results of dynamic analysis could be used to suggest certain properties, e.g., loop or class invariants, which would then be statically verified as a component of a larger verification problem.

Restricted Specification Formalisms. While it is tempting to allow program properties of interest to be specified using highly expressive formalisms such as first-order logic, by focusing on a more limited set of properties (e.g., *typestate* properties [32]), it is possible that more precise and scalable verification techniques could be developed for this class of properties than would be possible in the more general setting.

Other Issues. Some of the other challenges in making verification tools useful include: (1) The need to deal with missing source code (e.g., proprietary libraries) (2) Analyzing open programs and modeling the environment (3) Verifying distributed applications.

References

1. Arnold, G.: Combining heap analyses by intersecting abstractions. Master's thesis, Tel Aviv University (October 2004)
2. Ashcraft, K., Engler, D.: Using programmer-written compiler extensions to catch security holes. In: Proc. IEEE Symp. on Security and Privacy, Oakland, CA (May 2002)
3. Cardelli, L., Gordon, A.D.: Mobile ambients. In: Nivat, M. (ed.) ETAPS 1998 and FOSSACS 1998. LNCS, vol. 1378, pp. 140–155. Springer, Heidelberg (1998)
4. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proc. Computer Aided Verification, pp. 154–169 (2000)
5. Corbett, J., Dwyer, M., Hatcliff, J., Pasareanu, C., R.,, Laubach, S., Zheng, H.: Bandera: Extracting finite-state models from Java source code. In: Proc. Intl. Conf. on Software Eng, June 2000, pp. 439–448 (2000)
6. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proc. Symp. on Principles of Prog. Languages, pp. 269–282. ACM Press, New York (1979)
7. Das, M., Lerner, S., Seigle, M.: ESP: Path-sensitive program verification in polynomial time. In: Proc. Conf. on Prog. Lang. Design and Impl, pp. 57–68 (June 2002)
8. DeLine, R., Fähndrich, M.: Enforcing high-level protocols in low-level software. In: Proc. Conf. on Prog. Lang. Design and Impl. pp. 59–69 (June 2001)
9. Dor, N., Rodeh, M., Sagiv, M.: Checking cleanness in linked lists. In: Proc. Static Analysis Symp. Springer, Heidelberg (2000)
10. Field, J., Goyal, D., Ramalingam, G., Yahav, E.: Typestate verification: Abstraction techniques and complexity results. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 439–462. Springer, Heidelberg (2003)
11. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. In: Proc. Conf. on Prog. Lang. Design and Impl. Berlin, pp. 234–245 (June 2002)
12. Foster, J.S., Terauchi, T., Aiken, A.: Flow-sensitive type qualifiers. In: Proc. Conf. on Prog. Lang. Design and Impl. Berlin, pp. 1–12 (June 2002)
13. Gopan, D., DiMaio, F., Dor, N., Reps, T., Sagiv, M.: Numeric domains with summarized dimensions. In: Tools and Algs.for the Construct.and Anal.of Syst. pp. 512–529 (2004)
14. Gopan, D., Reps, T., Sagiv, M.: Numeric analysis of array operations. In: Proc. Symp. on Principles of Prog. Languages (2005)
15. Hoare, C.A.R.: Recursive data structures. Int. J. of Comp. and Inf. Sci. 4(2), 105–132 (1975)
16. Jeannet, B., Gopan, D., Reps, T.: A relational abstraction for functions. In: To appear in Proc. 12th Int. Static Analysis Symp. (September 2005) (to appear)
17. Jeannet, B., Loginov, A., Reps, T., Sagiv, M.: A relational approach to interprocedural shape analysis. In: Proc. Static Analysis Symp. Springer, Heidelberg (2004)
18. Lev-Ami, T., Sagiv, M.: TVLA: A system for implementing static analyses. In: Proc. Static Analysis Symp. pp. 280–301 (2000)
19. Lev-Ami, T., Reps, T., Sagiv, M., Wilhelm, R.: Putting static analysis to work for verification: A case study. In: Int.Symp.on Softw.Testing and Analysis, pp. 26–38 (2000)
20. Loginov, A., Reps, T., Sagiv, M.: Learning abstractions for verifying data-structure properties. In: Int.Conf.on Computer Aided Verif. (2005)

21. Manevich, R., Ramalingam, G., Field, J., Goyal, D., Sagiv, M.: Compactly representing first-order structures for static analysis. In: Proc. Static Analysis Symp. pp. 196–212 (2002)
22. Manevich, R., Sagiv, M., Ramalingam, G., Field, J.: Partially disjunctive heap abstraction. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 265–279. Springer, Heidelberg (2004)
23. Manevich, R., Yahav, E., Ramalingam, G., Sagiv, M.: Predicate abstraction and canonical abstraction for singly-linked lists. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, Springer, Heidelberg (2005)
24. Nielson, F., Nielson, H.R., Sagiv, M.: A Kleene Analysis of Mobile Ambients. In: Smolka, G. (ed.) ESOP 2000 and ETAPS 2000. LNCS, vol. 1782, pp. 305–319. Springer, Heidelberg (2000)
25. Ramalingam, G., Warshavsky, A., Field, J., Goyal, D., Sagiv, M.: Deriving specialized program analyses for certifying component-client conformance. In: Proc. Conf. on Prog. Lang. Design and Impl. pp. 83–94 (2002)
26. Reig, F.: Detecting security vulnerabilities in C code with type checking (extended abstract) (2003), `http://www.cs.nott.ac.uk/~fxr/`
27. Rinetskey, N., Sagiv, M.: Interprocedural shape analysis for recursive programs. In: Wilhelm, R. (ed.) CC 2001 and ETAPS 2001. LNCS, vol. 2027, pp. 133–149. Springer, Heidelberg (2001)
28. Rinetzky, N., Bauer, J., Reps, T., Sagiv, M., Wilhelm, R.: A semantics for procedure local heaps and its abstractions. In: Proc. Symp. on Principles of Prog. Languages (2005)
29. Rinetzky, N., Sagiv, M., Yahav, E.: Interprocedural shape analysis for cutpoint-free programs. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, Springer, Heidelberg (2005)
30. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems 24(3), 217–298 (2002)
31. Shaham, R., Yahav, E., Kolodner, E.K., Sagiv, M.: Establishing local temporal heap safety properties with applications to compile-time memory management. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 483–503. Springer, Heidelberg (2003)
32. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE Trans. Software Eng. 12(1), 157–171 (1986)
33. Yahav, E.: Verifying safety properties of concurrent Java programs using 3-valued logic. In: Proc. Symp. on Principles of Prog. Languages, pp. 27–40 (2001)
34. Yahav, E., Ramalingam, G.: Verifying safety properties using separation and heterogeneous abstractions. In: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation, pp. 25–34. ACM Press, New York (2004), doi:10.1145/996841.996846
35. Yahav, E., Reps, T., Sagiv, M., Wilhelm, R.: Verifying temporal heap properties specified via evolution logic. In: Degano, P. (ed.) ESOP 2003 and ETAPS 2003. LNCS, vol. 2618, pp. 204–222. Springer, Heidelberg (2003)
36. Yahav, E., Sagiv, M.: Automatically verifying concurrent queue algorithms. In: Workshop on Software Model Checking (2003)
37. Yorsh, G., Reps, T., Sagiv, M.: Symbolically computing most-precise abstract operations for shape analysis. In: Tools and Algs.for the Construct.and Anal.of Syst. pp. 530–545 (2004)

A Discussion on Thomas Reps's Presentation

Bertrand Meyer

In your list reversal example: the only way I know to teach this kind of algorithm, to understand it properly and (as I have tried to do in my own work) to prove it, is to have a loop invariant that basically says: If you are at a certain point in the list, halfway through the algorithm, then the first part of the list up to that point is the corresponding part in the original list, reversed, and the second part is unchanged from the original. In other words, the mirror image of the first part concatenated with the second part is, in some precise sense, equivalent to the original list.

Does this fundamental property of the algorithm follow from your description? And, if you knew that property, if the programmer had written it in the code, would it help your analysis at all?

Thomas Reps

The answer to the first part is that if we were trying to prove functional correctness of the list-reversal program, you would have to introduce not just the n-relation, but also the n0-relation. You basically want to freeze the initial n-relation on input in the n0-relation, so that you can make a comparison between the two. The property that you would want to show at the end of the program is that, for all pairs of individuals, if they were related by the n0-relation, n0 (v1, v2), you now have n (v2, v1), i.e., that you have reversed all the links.

If you were to look at the descriptors that appear at, say, the head of the loop, for each one of those you would see that in the list pointed to by y you would have that relationship, and in the list pointed to by x you would have that the n-relation would match the n0-relation - those links were not reversed. So what you stated as the property can be found by examining the structures at the head of the loop. But you don't have to state the property explicitly; it just comes out of the abstract interpretation.

Bertrand Meyer

It does not help you, if the programmer tells you.

Thomas Reps

Well, since you do not need it, I have not thought about, whether it would help you. There might be ways of allowing it to help you, but in this case, you do not need it.

Patrick Cousot

I have a semi-technical question. What is nice with TVLA is that you can find a good abstraction by experimentation, in fact: by refining the semantics, introducing relationships, and so on. The inconvenience is that it does not scale up

very, very large. But maybe, there is a way. That would be to make this experiment and then to extract the abstraction function automatically. This is given to people as the specification of the algorithm that they have to write in some efficient way. You see, you can do the same when you have found the proper abstraction; you can reprogram it, using very efficient data structures, and so on. And this could be a way to add efficiency.

Thomas Reps

Well, what we are trying to do is, we are trying to allow the abstractions to follow the hierarchical decomposition of the program, so that if you prove low-level things, then you can get some abstract transformer that you can use at higher level, at higher. Bertrand Jeannet, Mooly, one of my students, and I did some work that was at SAS 2004 a couple of years ago that aims towards this, where you end up using the abstract structures themselves as characterizations of the summary transformers. And once you have that, that allows you to sort of walk up the levels of the hierarchical decomposition of the software. But, there is a problem is with nested data structures, so the problem is not solved.

Greg Nelson

[Question not recorded.]

Thomas Reps

If you have a binary search tree, you would be interested in showing that sortedness properties are maintained, and we can do that.

Let me also mention another thing that doesn't involve insertion and deletion, but it was mentioned yesterday-something about the Deutsch-Schorr-Waite algorithm for traversing a tree without the use of a stack, by temporarily stealing pointer fields of the tree's nodes to serve in place of a stack. So we have actually applied this to Deutsch-Schorr-Waite and shown that the tree that comes in is reestablished at the end. It took 40 hours of TVLA running time on a 3GHz machine. And, of course, TVLA itself is part of the trusted computing base, so I am not sure I trust this answer. But anyway we claim to have been able to handle Deutsch-Schorr-Waite, and not just to say that it is a tree that goes in and a tree that comes out, but to say that the tree that comes out is identical to the tree that went in.