A Case Study in Automated, Modular, and Full Functional Verification

Jason Kirschenbaum
The Ohio State University
Columbus, OH 43210, USA
kirschen@cse.ohio-state.edu

Heather Harton
Clemson University
Clemson, S.C. 29634, USA
hkeown@clemson.edu

Murali Sitaraman
Clemson University
Clemson, S.C. 29634, USA
murali@cs.clemson.edu

ABSTRACT

Mechanical and full verification of behavior of object-based programs is a central software engineering problem. Any successful solution to the problem should strike a delicate compromise between amenability to automation and several software engineering factors, such as the form and ease of specifications, demands on software developers to provide invariants and hints, development and use of relevant mathematical theories, and language and software design. The object of this paper is to illustrate the issues that need to be addressed for full behavioral verification through our experiments towards push-button verification of an imperative object-based code in a modular fashion. In the process, the case study indirectly characterizes the requirements of a language for developing verifiable software.

1. INTRODUCTION

Automated verification of component-based software is a difficult and challenging problem, and it needs to be tackled for the verification grand challenge [8, 21] to succeed. The present paper illustrates the issues in automating verification using object-based sorting as a case study.

This case study differs from previous efforts in automated verification of properties in that the focus is on full behavioral verification. The verification approach used here is clean [10], because it uses both a language and specifications that do not involve modeling references or aliasing. It also differs from the work on automated verification of sorting algorithms (e.g. [1]) in several respects. The verification approach is modular; it is based only on the specification of components that are reused and allows verification of one component at a time. The sorting specification and the verified code are both generic. The proof of correctness is total. Finally, the mathematical notions come from a user-defined library illustrating an open system for verification of non-trivial software. Overall, a key contribution of the case study is in indirectly characterizing the requirements of an an integrated language for mathematical specification and implementation as a first step towards automating verification.

By design, the chosen example software is simple and is in the same spirit as the set of benchmarks for automatic verification that we have proposed in a recent paper [19]. However, while the example is simple, the issues raised by this code must be dealt with by any verification approach. Even with a simple example, our attempt to achieve push-button verification of the software quickly ran into several difficulties, demonstrating a variety of issues that need to be addressed for automated verification to become practical. The case study is a "real" example in the sense that it was chosen for verification from the collection of components available at the OSU RSRG software components library [2]. Furthermore, the code used in the case study predates the present verification effort. Though the component had been engineered with an eye toward verification, automating this task has proved challenging.

The first of the challenges in our case study, likely typical of other such efforts, concerns human errors in writing assertions. That humans will have to write mathematical assertions, such as to specify the behavior of software they intend to verify, is an inescapable requirement of any attempt to produce correct software [18]. There was a subtle error in the specification from the use of an output value of a variable instead of its input value in an assertion. Even after that error was fixed, one of the sub-problems in establishing an automated proof had to do with a weak invariant for a loop; while valid and apparently adequate, the invariant was not sufficiently strong to complete the proof. What is noteworthy here is that these errors existed in a piece of software that has been studied (and used) by ourselves, other instructors, and literally thousands of students. This observation also underscores why "social proofs" [14] are necessarily subject to errors implying that complete automation, the focus of this workshop, is absolutely essential.

The second set of challenges concern difficulties in achieving automation. Our proof process is based on [7] and the verification conditions (VC) resulting from our VC generator are fed to the Isabelle proof assistant [15]. Even after human errors in writing specification and invariants were eliminated, automation using the Isabelle proof assistant as the back-end prover remained elusive. The problem was traced to insufficient theory development and the inability to instantiate a universal quantifier suitably without any human help. Our attempt to tackle these challenges has led to a novel way of using suitable definitions and theory devel-

opment to minimize and avoid, if possible, the explicit need for universal quantifiers. It is too early to tell whether the proposed approach will generalize or whether it would simply raise problems elsewhere that are not apparent in our current setting.

It is important that we note that our choice of Isabelle, which is not a totally automated prover, is motivated by several factors. While dedicated first-order theorem provers might be able to automate more given the right axioms and background facts, we need a higher order prover. Isabelle also allows us to call external theorem provers (e.g. [17, 20]). Furthermore, we want our verification to be performed using verified theories; Isabelle has one of the largest libraries of proven mathematical theorems. This is especially important in full verification where behaviors of objects may be specified using sophisticated mathematical models. The second factor is that Isabelle has support for both automated and user-guided proof methods. This is especially useful to experimental researchers, because that support helps us find where a proof of a VC gets stuck, find a correct proof, and use that information to modify the automated proof methods (thereby increasing the types of VCs that are provable automatically). Finally, we note that we know of no provers that can automatically discharge the VCs from our case study.

In attempting to solve the subproblems during automation, we also generalized our mathematical definitions and software specifications in ways that we hadn't initially conceived. The resulting software is clearly more reusable. This last point may indicate that software designed for automated verification might be, in general, better engineered than other software and that automation has benefits, not just for correctness, but also for the overall quality of software.

The methods and tools used for the subject of this paper have been set up such that they can be repeated elsewhere with other scenarios and case studies. We include only partial code and proofs in this paper. Complete details may be found at: `www.cs.clemson.edu/~resolve/benchmarks`.

2. CASE STUDY IN DETAIL

```
Begin String_Theory

Definition String( Str(G : Set) : Set,
                   empty_string : Str(G),
                   ext( a : Str(G), x : G): B =

...

Inductive Definition on b : Str(G : Set)
                 of (a : Str(G)) o (b : Str(G)) is
...

Lemma ConcatenationAssociative
 a o (b o c) = (a o b) o c

Definition IsPermutation where
 IsPermutation a b = ...

Lemma PermutationCommutative
 IsPermutation (x o y) (y o x)

...
End String_Theory
```

Figure 1: Example of String Theory

A specification and implementation of a sort operation that extends a Queue abstract data type is given below in the RESOLVE [16, 4, 3, 11] notation. Here, queues are conceptualized mathematically as strings of entries, constrained to be within a given Max_Length. The specification of the sort operation is parameterized both by the type of the entries the sorted queue contains and by the ordering relation used for sorting. A sample of the theory is given in Figure 1. The proofs of the lemmas and theorems in the theory are given in a separate proof module which is not shown.

Operation Sort_Q has no preconditions. It ensures that the Queue parameter Q after the operation must be nondecreasing (with respect to the client supplied LEQV relation) and must be a permutation of the incoming Q (denoted by #Q).

In the figure, the local definition IsNondecreasing is based on the the caller-supplied LEQV relation. It uses "o" to denote string concatenation. The definition of IsPermutation is in String_Theory a mathematical unit imported by the specification Queue_Template neither of which is shown.

```
Enhancement Sort_Capability(def
                              LEQV(x,y: Entry): B)
            for Queue_Template;
       requires Is_Total_Preordering(LEQV);

    Definition IsNondecreasing(a: Str(Entry)): B=
      (for all b,c: Str(Entry),
          for all x,y: Entry,
            if a = b o <x> o <y> o c
              then LEQV(x,y));

    Operation Sort_Q(updates Q: Queue);
        ensures IsNondecreasing (Q)
                  and IsPermutation(#Q,Q);
end Sort_Capability;
```

2.1 Experimentation Part 1: Human Errors

The lessons learned in the first phase of this case study are likely typical of any such effort. They involved human errors in writing specifications and loop invariants. The implementation of the sort operation chosen for this verification experiment is a selection sort. The Sort_Q code (procedure) uses a local operation Remove_Min, only the specification of which is given. Its code also makes use of a programming operation to compare two entries; the operation has the behavior of the relation LEQV used in the specification. To use the sorting capability, a client would have to supply a compatible mathematical relation and an operation for ordering.

The procedure (code) for sorting defines and uses a local operation Remove_Min to find and remove a minimum element of a queue according to the ordering as shown in in Figure 2. The loop in the Sort_Q procedure is annotated by a software engineer with a list of variables that are modified by the loop, an invariant (expressed as a **maintaining** assertion), and a progress metric (**decreasing** expression) to establish total correctness.

Following the loop, :=: denotes the swap operator which can be used to exchange the values of two objects cleanly, without introducing aliasing [6].

We used our VC generator to generate verification conditions that represent the correctness of both the Remove_Min and Sort_Q procedures and attempted to prove them using Isabelle. The generated verification conditions correspond to the postcondition of the operation that is verified, precondi-

```
    Definition IsASmallest(a: Str(Entry);
                           x: Entry): B=
        (for all b,c: Str(Entry),
          for all y: Entry,
            if a = b o <y> o c then LEQV(x,y));

    Operation Remove_Min(updates Q: Queue;
                         replaces min: Entry);
       requires |Q| /= 0;
       ensures IsPermutation(Q o <min>, #Q)
               and IsASmallest(Q, min);
-- code omitted

     Procedure Sort_Q(updates Q: Queue);
      Var sorted:Queue;
      Var min:Entry;
      While (Length(Q) /= 0)
       changing Q, sorted, min;
       maintaining IsPermutation(Q o sorted, #Q)
                   and IsNondecreasing(sorted);
       decreasing |Q|;
      do
         Remove_Min(Q, min);
         Enqueue(min, sorted);
      end;
      Q :=: sorted;

     end Sort_Q;
```

Figure 2: Original Sort_Q Implementation

```
    Operation Remove_Min(updates Q: Queue;
                         replaces min: Entry);
       requires |Q| /= 0;
       ensures IsPermutation(Q o <min>, #Q)
               and IsASmallest(#Q, min);
-- code omitted

     Procedure Sort_Q(updates Q: Queue);
      Var sorted:Queue;
      Var min:Entry;
      While (Length(Q) /= 0)
       changing Q, sorted, min;
       maintaining IsPermutation(Q o sorted, #Q)
          and IsNondecreasing(sorted)
          and
           (for all y: Entry,
              if IsSubstring (<y>, Q) then
                  IsNondecreasing(sorted o <y>));
       decreasing |Q|;
       do
         Remove_Min(Q, min);
         Enqueue(min, sorted);
       end;
       Q :=: sorted;

     end Sort_Q;
```

Figure 3: Corrected Sort_Q Implementation

tions of operations that are called, and the correctness of the programmer-supplied loop invariant and progress metric.

Some of the verification conditions could not be proved. One them was the VC that represents the postcondition of Remove_Min. This VC could not be proved because of an incorrect specification of Remove_Min, which should have been:

```
ensures ... and IsASmallest(#Q, min);
```

Once the specification error was fixed, there was still a problem in proving the Sort_Q procedure. This was quickly traced to the loop invariant being too weak. While the two conjuncts in the loop invariant seem to match up properly with the two conjuncts in the postcondition of Sort_Q, the invariant IsNondecreasing(sorted) itself could not be established after min was enqueued. In particular, the invariant that every element in sorted is related to every element in Q was missing. This information is captured in the corrected code as shown in Figure 3.

2.2 Experimentation Part 2: Automation Difficulties

Once we corrected the annotations and generated new verification conditions, it was easy to check manually that each VC could be proved. In automating the proof with Isabelle, though a majority of the VCs could be proved, proofs of some VCs required human assistance.

The use of Isabelle as a tool to prove the VCs automatically requires the use of needed theories and lemmas from the RESOLVE mathematical theory library. Our approach is to write each RESOLVE mathematical theory (in this instance, string theory) as a new Isabelle theory, complete with defined functions and needed algebraic lemmas and theorems. We also leverage the already developed List theory in Isabelle for the internal representation of Strings (according to the recommended practice for Isabelle [15]); however, all VCs are proved with lemmas from String theory. Since we are focused on the ease and provability of VCs, in the present paper, we do not discuss the proofs of the lemmas themselves.

Once the basic theory is imported into Isabelle, lemmas about strings must be identified and tagged. The tags indicate to Isabelle how the lemma should be used in an automatic proof. Isabelle can use lemmas as simplification rules, introduction rules, destruction rules, and elimination rules. The simplification rules rewrite a term as another term. The introduction rules replace a goal of a lemma to be proved with the assumptions of an already proved lemma. The destruction rules replace a set of assumptions in a lemma to be proved with the goal of an already proved lemma. Essentially, how the lemmas are tagged determines how Isabelle will attempt to prove a VC automatically.

The VCs in this case are provable automatically, once the appropriate lemmas are tagged for use in the proofs, except for two cases discussed in some detail here. The first case is the proof of the loop invariant at the end of the while loop in the Remove_Min procedure as shown in Fig. 4. Here, the main issue is the development of the lemmas and theorems for the IsPermutation predicate. Once the IsPermutation predicate is expanded and the quantifiers instantiated appropriately, Isabelle can then prove the VC. A more well developed theory, with lemmas that allow for more powerful manipulations of the commutativity of concatenation within IsPermutation would help mitigate this issue.

The second case is shown in Figure 5. This lemma is the VC resulting from the obligation to prove the loop invariant at the end of the while loop in the Sort_Q procedure. In Figure 5, we use lemma ND4; the lemma is displayed in Figure 6. Lemma ND4 allows us to look at only the last entry in a string to determine whether adding another element would preserve the IsNondecreasing property. The main problem with this lemma is that Isabelle cannot find the proof automatically without being told that ND4 should be used and that is should be used in a particular manner, as an introduction rule. Without the knowledge that the usage of the lemma ND4 is always good, Isabelle stops exploring before a successful proof can be found. Once a person adds a manual declaration to apply the correct lemma (as shown

```
lemma 4:
"[|
 (min_int <= 0);
 (0 < max_int) ;
 (Max_Length > 0);
 is_initial min;
 (length Q <= Max_Length);
 length Q ~= 0;
 Q = (<min2> o Q3);
 IsPermutation ((temp1 o Q2) o <min1>) Q;
 IsASmallest (temp1 o <min1>) min1;
 (length Q2 ~= 0);
 Q2 = (<x1> o Q1);
 x1 <= min1
|]
==>
 IsPermutation (((temp1 o <min1>)
                 o Q1) o <x1>) Q "
apply auto
apply (unfold IsPermutation_def)
apply (auto)
apply (drule_tac x=x in spec)
apply auto
done
```

Figure 4: First Case of Problem VCs

by the "apply (rule ND4)" command in Figure 5), Isabelle's reasoner can find the proof.

```
lemma 10:
"[|
 (min_int <= 0);
 (0 < max_int);
 (Max_Length > 0);
 (length Q <= Max_Length);
 (IsPermutation (Q2 o sorted1) Q);
 (IsNondecreasing sorted1);
 ALL y. Is_SubString <y> Q2
        --> (IsNondecreasing (sorted1 o <y>));
 (length Q2 ~= 0);
 (IsPermutation (Q1 o <min1>) Q2);
 (IsASmallest Q2 min1)
|]
==>
 ALL y. Is_SubString <y> Q1 -->
    (IsNondecreasing ((sorted1 o <min1>) o <y>))"
apply auto
apply (rule ND4)
apply auto
done
```

Figure 5: Second Case of Problem VCs

The two issues raised in this subsection seem to be critical to the verification problem; namely the theory is not well developed enough (relative to the capabilities of the tool used) to permit the proof of VCs or the theory is well developed enough but the tool cannot construct the proof without hints such as the lemmas necessary or even when each lemma should be applied in a proof.

2.3 Experimentation Part 3: Impact of Theory Development on Automation

One of the difficulties in automation had to do with universal instantiation. Though this problem has received attention, our next line of thinking was to avoid creating this problem in the first place for automated provers. Our solution approach is to eliminate universal quantification in our annotations through new definitions, whenever possible. For our case study, this approach has led to a new definition, as well as development of new theorems involving the new definition. If we are successful in automating this totally, then it

```
lemma ND4:
"
[|
 IsNondecreasing (a o <y>);
 y <= b
|]
==>
  IsNondecreasing ((a o <y>) o <b>)
"
```

Figure 6: Lemma ND4

would also suggest a more general attempt to eliminate universal quantification through suitable mathematical theory development. Our verification system already allows for the use of ghost variables and manipulation of ghost variables to eliminate existential quantifiers in assertions (following King's principles [9] for the idea). This finding led us to the specification of Remove_Min operation and the invariant for the Sort_Q procedure as shown in Figures 7 and 8.

```
   Definition IsPreceding ...

   Operation Remove_Min(updates Q: Queue;
                        replaces min: Entry);
      requires |Q| /= 0;
      ensures IsPermutation(Q o <min>, #Q)
              and IsPreceding(<min>, #Q);
-- code omitted

    Procedure Sort_Q(updates Q: Queue);
     Var sorted:Queue;
     Var min:Entry;
     While (Length(Q) /= 0)
      changing Q, sorted, min;
      maintaining IsPermutation(Q o sorted, #Q)
            and IsNondecreasing(sorted)
            and IsPreceding(<min>, Q);
      decreasing |Q|;
      do
       Remove_Min(Q, min);
       Enqueue(min, sorted);
      end;
      Q :=: sorted;

    end Sort_Q;
```

Figure 7: Modified Theory Development for Specification

The new predicates are created and additional lemmas are proved that show how to the new predicates relate to others. The goal for this theory development is to eventually create enough new functions/predicates and relevant facts about those functions/predicates to remove the universal quantifiers from the specifications as we gain more experience with the proofs of the VCs. We envision large theory developments that are created *a priori* to be used in the specifications of programmatic procedures. These theory developments would be mostly fixed, e.g. most programming specifications would fall within such theories. Of course, when new theory development is needed, the newly created theories/functions/predicates would be added to the theory repository.

In this example, notice that the definitions of IsNondecreasing and IsPreceding are not usable elsewhere because they are specialized based on the user-supplied relation for ordering LEQV. A consequence of this deficiency is that any theorems that are proved for these definitions will also be specialized. Since the underlying principles are more general, it has led us to the following more gen-

```
Enhancement Sort_Capability(def LEQV(x,y: Entry): B)
            for Queue_Template;
      requires Is_Total_Preordering(LEQV);

    Operation Sort_Q(updates Q: Queue);
        ensures Is_Conformal_w (LEQV, Q)
                and IsPermutation(#Q,Q);
end Sort_Capability;
```

Figure 8: Modified Specification for Sort_Q Procedure

eral (and simple) specification of sorting. The predicate Is_Conformal_w takes an arbitrary binary relation and a string; it is a more general version of IsNondecreasing. The predicate Universally_Relates_to takes an arbitrary binary relation and two strings. It is a more general version of IsPreceding.

```
Begin String_Theory

...

Lemma PermutationCommutative
  IsPermutation (x o y) (y o x)
     ...

Definition Is_Conformal_w (R: (x: G, y: G) : B,
                           a : Str(G)) : B
  = for all x, y.
        If Is_Substring(<x> o <y>, a)
         Then R(x,y)

...

Definition Universally_Relates_to(
                           R: (x: G, y: G) :B,
                           a : Str(G),
                           b : Str(G))i : B
  = for all x, y. If Is_Substring(<x>, a)
                     and Is_Substring(<y>, b)
                  Then R(x,y)

...

...
End String_Theory
```

Figure 9: Definitions of Is_Conformal_w and Universally_Relates_to

The definitions of these predicates are shown in Figure 9. Given the theory development for these general definitions, we do not foresee any additional automation problems due to the generalization.

3. DISCUSSION

An excellent summary of several verification efforts may be found in [12], and details of efforts more directly related to the present paper may be found in [7]. Here, we present only a summary of most related efforts for full, automated verification of programs.

To our knowledge there is no system available today for automated and modular verification of full behavior of object-based programs. Using the πVC Verification System [1], Bradley, Manna, and Sipma have verified several sorting algorithms (of integer arrays) written in the pi programming language. A few examples of the sorting algorithms proved in πVC are: insertion sort, merge sort, bubble sort, and quick sort for integer arrays in the pi programming language. The verification of the various sorting algorithms that they implemented each required less than 20 seconds to complete. Unlike our work, their focus is on using specialized data types and decision procedures.

The Why software verification system [5] can be used to generate verification conditions, similar to the ones generated by our tool. Unlike our effort, the Why tool focuses primarily on functional programs annotated with assertions and typically relies on built-in types (ie, arrays).

Verisoft [13] is a verification system that relies on Isabelle to perform the actual proofs, as we have done. It is based on a "clean" subset of the C programming language, C0 and it aims for full verification. Since it is focused on C, it does not address modular verification of programs using objects.

4. CONCLUSION

We have presented our experiences with the verification of a sorting algorithm implementation using the proof assistant Isabelle. The lessons learned from this exercise include information about the types of errors programmers are likely to make in writing the annotated code, and some of the possible trouble areas for a proof assistant such as Isabelle. We plan to continue this work in order to address the identified issues with Isabelle. At least as importantly, we have also learned how automation can lead to better software engineering and sophisticated theory development along dimensions that would not have come into focus without attempting automated verification in the first place. Finally, the case study illustrates at least a set of minimum requirements for an integrated language for developing verified software.

Acknowledgments

We would like to thank Jeremy Avigad for his guidance in connecting our verification effort with the Isabelle prover and Bill Ogden for his help in formulating the assertions used in this paper. Our thanks are also due to Harvey Friedman and Bruce Weide for various discussions relating to the topics of this paper. We would also like to thank Bruce Adcock, Derek Bronish, Wayne Heym, Joan Krone, Tim Long, Brandon Mintern, and Anna Wolf for their suggestions.

This work was supported in part by the National Science Foundation under grants DMS-0701187, DMS-0700174, DMS-0701260 and DUE-0633055.

5. REFERENCES

[1] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *VMCAI*, pages 427–442, 2006.

[2] P. Bucci. Osu rsrg component catalog, 2008. `http://www.cse.ohio-state.edu/rsrg/sce/rcpp/RESOLVE_Catalog-HTML/index.%html`.

[3] P. Bucci, J. E. Hollingsworth, J. Krone, and B. W. Weide. Part iii: implementing components in resolve. *SIGSOFT Softw. Eng. Notes*, 19(4):40–51, 1994.

[4] S. H. Edwards, W. D. Heym, T. J. Long, M. Sitaraman, and B. W. Weide. Part ii: specifying components in resolve. *SIGSOFT Softw. Eng. Notes*, 19(4):29–39, 1994.

[5] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *19th International Conference on Computer Aided Verification*, volume 4590/2007 of *LNCS*, pages 173–177, Berlin, Germany, July 2007. Springer-Verlag.

[6] D. Harms and B. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Transactions on Software Engineering*, 17(5):424–435, May 1991.

[7] H. Harton, J. Krone, and M. Sitaraman. Formal program verification. *Encyc. of Computer Science and Engineering*, 2008 (to appear).

[8] T. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.

[9] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[10] G. Kulczycki. *Direct Reasoning*. PhD thesis, Clemson University, 2004.

[11] G. Kulczycki, M. Sitaraman, N. Yasmin, and K. Roche. Formal specification. *Encyc. of Computer Science and Engineering*, 2008 (to appear).

[12] G. T. Leavens, J.-R. Abrial, D. Batory, M. Butler, A. Coglio, K. Fisler, E. Hehner, C. Jones, D. Miller, S. Peyton-Jones, M. Sitaraman, D. R. Smith, and A. Stump. Roadmap for enhanced languages and methods to aid verification. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 221–236, New York, NY, USA, 2006. ACM.

[13] D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a c0 compiler: Code generation and implementation correctnes. *sefm*, 0:2–12, 2005.

[14] R. A. D. Millo, R. J. Lipton, and A. J. Perlis. Social processes and proofs of theorems and programs. *Commun. ACM*, 22(5):271–280, 1979.

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

[16] W. F. Ogden, M. Sitaraman, B. W. Weide, and S. H. Zweben. Part i: the resolve framework and discipline: a research synopsis. *SIGSOFT Softw. Eng. Notes*, 19(4):23–28, 1994.

[17] S. Schulz. System Description: E 0.81. In D. Basin and M. Rusinowitch, editors, *Proc. of the 2nd IJCAR, Cork, Ireland*, volume 3097 of *LNAI*, pages 223–228. Springer, 2004.

[18] M. Sitaraman, S. Atkinson, G. Kulczycki, B. W. Weide, T. J. Long, P. Bucci, W. D. Heym, S. M. Pike, and J. E. Hollingsworth. Reasoning about software-component behavior. In *ICSR-6: Proceedings of the 6th International Conerence on Software Reuse*, pages 266–283, London, UK, 2000. Springer-Verlag.

[19] B. W. Weide, M. Sitaraman, H. K. Harton, B. Adcock, P. Bucci, D. Bronish, W. D. Heym, J. Kirschenbaum, and D. Fraiser. Incremental benchmarks for software verification tools and techniques. Technical Report RSRG-08-02, School of Computing, Clemson University, 2008.

[20] C. Weidenbach. Spass - version 0.49. *J. Autom. Reason.*, 18(2):247–252, 1997.

[21] J. Woodcock and R. Banach. The verification grand challenge. *Journal of Universal Computer Science*, 13(5):661–668, may 2007. `http://www.jucs.org/jucs_13_5/the_verification_grand_challenge`.