Appeared in *IEEE International Conference on Computer Design*, September 2000.

Source-Level Transformations for Improved Formal Verification*

Brian D. Winters and Alan J. Hu
Department of Computer Science
University of British Columbia
(bwinters,ajh)@cs.ubc.ca

Abstract

A major obstacle to widespread acceptance of formal verification is the difficulty in using the tools effectively. Although learning the basic syntax and operation of a formal verification tool may be easy, expert users are often able to accomplish a verification task while a novice user encounters time-out or space-out attempting the same task. In this paper, we assert that often a novice user will model a system in a different manner — semantically equivalent, but less efficient for the verification tool — than an expert user would, that some of these inefficient modeling choices can be easily detected at the source-code level, and that a robust verification tool should identify these inefficiencies and optimize them, thereby helping to close the gap between novice and expert users. To test our hypothesis, we propose some possible optimizations for the Murφ verification system, implement the simplest of these, and compare the results on a variety of examples written by both experts and novices (the Murφ distribution examples, a set of cache coherence protocol models, and a portion of the IEEE 1394 Firewire protocol). The results support our assertion — a nontrivial fraction of the Murφ models written by novice users were significantly accelerated by the very simple optimization. Our findings strongly support further research in this area.

1. Introduction

A major obstacle to widespread acceptance of formal verification is the difficulty in using the tools *effectively*. Ideally, the tools should be accessible to non-experts, so that formal verification can be used as just another aid to design and verification. While some learning effort is expected, the user must not be expected to have a deep understanding of the specific algorithms and heuristics used by the tool. In reality, the situation is markedly different. Although the syntax and basic operation of formal verification tools might be easy to learn, there is usually considerable subtlety in using the tools effectively on large problems. Expert users with a deep understanding of a tool are able to achieve impressive results; novice users are all too frequently frustrated by exploding memory and CPU usage.

A key difference between novice and expert is that the expert has a detailed understanding of the algorithms and heuristics used by a tool and can use that understanding to choose an efficient way to present the verification problem to the tool. When applying formal verification, many choices must be made: how to model parts of the system, which parts of a system to abstract, how to perform that abstraction, and so forth. Differing choice can result in an equivalent verification problem, but some choices will be more or less efficient than others for the particular verification tool. When the novice has the chance to consult with the expert, advice often takes the form, "Don't model it like that; do this instead. Otherwise,...," where the consequence reflects detailed knowledge of the tool: the BDD will have to represent all permutations, the state space will lose symmetry, the reduction heuristic won't work, and so forth. This situation is clearly undesirable; the novice is forced to become an expert on and adapt to the idiosyncrasies of the tool. Even for the expert, being forced to adapt to the tool can be suboptimal, since the description style that most clearly matches the verification problem might not be the style that best suits the verification tool.

We believe that the tool should adapt to the user. A robust, practical verification tool should help close the gap between novice and expert, allowing the novice to get useful results without a lengthy learning curve. For novice and expert alike, the tool should free the user to match the description to the problem being verified, rather than to specific quirks of the verification tool.

One might argue that recent commercial tools are much easier to use than their academic forebears. This is true. It's worth noting, however, that the most commercially successful formal verification techniques, e.g., combinational equivalence checking, are precisely those which place the least burden on novice users to understand the inner workings of the tool. In this light, one can view this paper as a step towards making more formal verification techniques commercializable.

We hypothesize that often a novice will model a system differently (less efficiently for the tool) than an expert would, that many of these inefficient modeling choices can

*This work was supported in part by the National Science and Engineering Research Council of Canada.

be easily detected in the source code of the model description, and that the tool should be able to optimize away these differences. This paper presents a preliminary test of the hypothesis: we are demonstrating proof-of-concept of a general principle, rather than advocating a specific optimization.

To test our hypothesis, however, we must be more concrete: for a specific verification tool, what optimizations are possible, and are they effective in practice on models written by novice users? We have chosen the Murφ verification system [2, 1] as the target for our test. The reasons for this choice are twofold. On one hand, Murφ is simple: the description language is small, the verification algorithm is explicit state reachability, for which heuristic optimizations are easier to understand than for BDD-based methods, and the compiler source code is publicly available.[1] On the other hand, Murφ has been widely used for a variety of applications (e.g. [3, 11, 8, 4, 9, 5]), so realistic verification examples exist, and any optimizations we implement are useful. The choice of Murφ is fairly arbitrary — our ideas should apply in general, although specific optimizations, obviously, apply only to verification tools with similar features.

2. Source-Level Transformations

The Murφ verifier has it's own input language for describing the system being verified. A Murφ program consists of two main parts: declarations and rules. The declaration section declares constants, types, global variables, and procedures. The rules define the transitions of the system. At any given time, the system state is determined by the values of the global variables. As the system executes, a rule is chosen non-deterministically and executes atomically, updating the global variables to a new state.

Murφ provides several features to simplify writing scalable descriptions of large systems. The normal description style uses numerous subrange types, which can be scaled easily. Of particular interest are the special `scalarset` and `multiset` types, which provide automatic symmetry reduction [7]. Scalarsets are like subranges, but without order. Multisets are like arrays, except that the array elements are unordered. Appropriate use of these data types greatly reduces the size of the state space. To simplify writing rules for all values of a subrange or scalarset, Murφ provides a `ruleset` construct, which generates a copy of all enclosed rules for each possible value of its formal parameter. Similarly, the `choose` construct selects an item from a multiset. Figure 1 shows some portions of a Murφ model for a cache coherence protocol.

We have identified a few possible source-code transformations which may improve models written by novice users. These are ruleset rearrangement, scalarset identification, and variable clearing.

[1] http://verify.stanford.edu/dill/murphi.html

```
Const
  ProcCount: 2; -- # of processors
  AddressCount: 2; -- # of addresses
  ValueCount: 2; -- # of distinct values
  ...
Type
  Pid: Scalarset(ProcCount);
  Address: Scalarset(AddressCount);
  Value: Scalarset(ValueCount);
  ...
  ProcState: Record
             ...
             cache: Array[Address] of
                      Record
                        ...
                        v: Value;
                      End;
           End;
  ...
Var
  procs: Array[Pid] of ProcState;
  ...

Ruleset p: Pid Do
  Alias me: procs[p] Do
    Ruleset a: Address Do
      Ruleset v: Value Do
        Rule "Evict shared data"
          (me.cache[a].state = Shared)
        ==>
          me.cache[a].state := Invalid;
          Undefine me.cache[a].v;
        Endrule;
        ...
        Rule "Change exclusive copy"
          (me.cache[a].state = Exclusive)
        ==>
          me.cache[a].v := v;
        Endrule;
      Endruleset;
    Endruleset;
  Endalias;
Endruleset;
  ...
```

Figure 1. Portions of a Murφ Model

Ruleset Rearrangement

Murφ allows users to group rules within `ruleset`, `choose` and `alias` statements, which can be nested arbitrarily. These groupings are primarily for convenience; grouping of rules may be the best match for the problem semantics or may make it easier for the user to understand the model. Grouping rules together under ruleset, choose or alias statements is logically equivalent to placing each rule under separate identical ruleset, choose or alias statements.

If a rule is enclosed inside a ruleset or choose statement on which it doesn't depend, the verifier will needlessly execute the rule for each possible value of the enclosing ruleset or choose parameter. The set of reachable states is unaffected, as each firing of that rule for variations of the independent variable will lead to the same state, but every extra rule firing adds to the run time of the model.

Obviously, we should move rules outside the scope of irrelevant rulesets and choose statements. A straightforward implemention method starts by ungrouping all rules, so that each rule is by itself within its enclosing rulesets, chooses, and aliases. Then, for each rule, we remove enclosing ruleset and choose statements upon which the rule doesn't depend. Removing alias statements does not impact performance, but it simplifies the dependency analysis of ruleset and choose statements. For example, in Figure 1 the eviction rule does not depend on the ruleset over *v*.

Scalarset Identification

Using scalarsets instead of subranges in cases where ordering within the range does not matter provides a tremendous reduction in state space size. Subranges, however, cannot always be replaced by scalarsets: any operation that relies on order, arithmetic, or distinguishing special elements in the range breaks symmetry and precludes the use of scalarsets.

Novices might not realize all cases when a scalarset can be used. A trivial possible transformation is to check for each subrange type whether it could be redefined as a scalarset.

A more interesting and powerful transformation is possible if we consider common programming practice. Data types often get reused for various purposes if it seems appropriate. An integer type, for example, might be used for a counter, for arithmetic, and for ID numbers. Returning to the example in Figure 1, suppose the user had defined an "integer" type as a subrange, and used this type for processor IDs, addresses, values, as well as for some counters. In that model, we could not apply the scalarset symmetry reduction to processors, addresses, and values because the same type is also being used for counters, which have order. Even without the counters, having a single scalarset type for processors, addresses, and values would not allow as much symmetry reduction as having a separate scalarset for each.

To handle this problem, we propose a much more powerful way to identify possible scalarsets. A syntactic check of the Murφ program can determine which variables interact and must have the same type versus which variables have the same type simply for the user's convenience. If we construct a graph with a node for each variable and edges indicating which variables must be type-compatible, then each connected component can be assigned a distinct type. Each of these types can then be checked separately for possible conversion to a scalarset.

Variable Clearing

In most models, not all variables are holding important data at all times. For example, if a cache line is invalid, its contents don't matter, or if a queue is modeled as an array and a tail pointer, the array elements past the tail pointer don't matter.

Although the contents of these variables may not matter to the user or to the accuracy of the model, they do matter to the verification tool. Two states that differ only in the values of don't-matter variables are still considered two distinct states by the verification tool. If the user is not careful to clear any leftover values out of variables whenever they no longer matter, the result is a needless explosion in the number of states. For example, in Figure 1, if the user omitted the `Undefine` statement, the resulting model would have a much larger set of reachable states, because different values leftover in the `me.cache[a].v` field would generate additional states.

Tracking exactly which variables are no longer needed and making sure to clear out their values is tedious for the expert user and extremely difficult for the novice, who may not even realize the importance of doing so. Fortunately, live variables analysis, a standard program analysis technique (e.g., [10]), can determine automatically which variables are dead (not needed) at each point in a program. Adapting live variables analysis to Murφ programs could determine where `Undefine` statements are needed. The source-code transformation would insert the `Undefine` statements automatically, greatly improving the efficiency of verification.

3. Experimental Set-Up and Results

At this point, we have implemented only the ruleset rearrangement transformation. We are enthusiastic about the other transformations, and their implementation presents no research challenges. Actually coding them, however, is impractical in the current Murφ compiler. Accordingly, we chose to evaluate the easiest-to-implement transformation on a wide variety of Murφ programs. If our ideas have merit we should observe some improvement from even one transformation.

We used three sets of Murφ programs for our experiments.[2] The first set is the 27 distinct examples included in the Murφ 3.1 distribution. These examples are written by expert Murφ users, so our hypothesis predicts little or no improvement from the optimization. Indeed, the results on these runs are uninteresting and have been omitted for brevity. In only one example did runtime change by more than 5% (`ldash` improved 8.32% and 7.32% with and without hash compaction). In three other examples, runtime changed by more than 2% (`list6` and `sci` improved, `ns` worsened). In the remaining 23 examples, runtime changed by less than 2% both with and without hash compaction.

The second set consists of eleven implementations of a simple, fictitious directory-based cache coherence protocol, each developed independently by students in a formal verification class. The students had no previous experience with

[2] Our implementation is a modification of Murφ 3.1. Compiling a Murφ model is a two step process — translating the Murφ model into a C++ program that is a verifier for the model, and then compiling the C++ program. For all experiments, we enabled bit compaction and tested both with and without hash compaction. The C++ code was compiled using egcs version 2.91.66 with "-O4 -mpentiumpro" optimization. The Murφ hash table size was set to 96 MB with 30% for the active states. Hash compaction used the default 40-bit hash size. Our experiments were run on a 400 MHz Pentium II workstation with 128 MB of RAM, running SuSE Linux 6.1. We report runtimes in seconds for the original model and the model after our transformation. The runtimes shown are the medians of five runs. For brevity, we report results only without hash compaction; results with hash compaction differ immaterially. Compile times (Murφ translation plus C++ compilation) are not reported because our transformation did not affect them significantly.

model	lines	states	$time_{orig}$	$time_{opt}$	% imprv
student1	587	1828	2.88	2.88	0.00
student2	641	36984	45.31	31.50	30.48
student3*	653	8204	4.38	4.39	-0.23
student4	710	501446	397.91	399.20	-0.32
student5	496	3522	3.17	3.17	0.00
student6	642	36995	44.21	30.94	30.02
student7	947	3647	3.52	3.48	1.14
student8	669	9071	6.87	6.83	0.58
student9*	714	52771	54.62	54.35	0.49
student10	568	4403	3.97	4.00	-0.76
student11	538	1828	2.22	2.22	0.00

Table 1. Cache coherence protocol models

Murφ. They were given a tabular description of the cache coherence protocol and a partial Murφ model that included declarations, but no rules, so they were free to write the rules in whatever manner was most natural for them. This set of programs represents a large number of novice users independently tackling the same verification task. Our hypothesis predicts that some of these programs should show significant improvement, corresponding to when a particular user happens to chose a writing style that happens not to be well-suited to the internals of the tool. The results (Table 1[3]) again match the hypothesis — the optimization greatly speeds up two of the eleven programs. From the enormous variations in code size, state count, and run times, we see that different users naturally express themselves differently. A robust formal verification tool should adapt to this diversity, rather than forcing users to write code specifically tailored to the verification tool's idiosyncrasies.

The third experiment is a model of part of the physical layer of the IEEE 1394 [6] High Performance Serial Bus. The model implements the reset and tree identification portions of the physical layer. The model is significantly larger than the others and was written by a Murφ novice. Large, real models written by novices are not widely available, but such models provide the best test of our hypothesis because large real models written by novices are precisely those that we seek to improve. In this case, the most natural way for the user to model the system did not give the best arrangement of rules and rulesets for efficient Murφ execution. The model had 824 lines and 2060216 reachable states. Our transformation improved runtime 27.08%, from 372.23 to 271.43 seconds.

4. Conclusion

As we have seen, even an extremely weak source-level transformation was able to significantly improve several models written by novices. This result supports our assertions that novice users are likely to model systems differently and less efficiently for formal verification, that many of these inefficient modeling choices can be easily detected at the source-code level, and that a verification tool can optimize away these inefficient modeling choices, thereby boosting the productivity of novice users. A tool user should not need to understand the inner details of the tool, nor adapt to those details, in order to use it effectively. This work is a step towards solving the problem.

We have implemented one simple source-level transformation — our goal was to illustrate how verification tools can adapt to novice users, rather than to advocate a specific optimization for a specific tool. Nevertheless, the most obvious direction for future work is to try the other transformations we have proposed and measure their effectiveness.

The more general direction for future work, and the promise of greater impact, is to apply these ideas to other verification tools and languages. Possible questions to investigate include what source-level changes might reduce BDD size, what aspects of Verilog or VHDL might highlight easy-to-implement optimizations, and how might a future hardware description language or verification language be best designed to support robust ease-of-use. Considerable further research needs to be done.

[3]Models were run with 2 processors, 3 addresses, 2 data values, and communication channels scaled as necessary, except for those marked with asterisks, which blew up with 3 addresses, so we used only 2 addresses.

References

[1] D. L. Dill. The Murφ verification system. *Computer-Aided Verification: 8th Int'l Conf*, pp. 390–393. Springer-Verlag, 1996. Lecture Notes in Computer Science 1102.

[2] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. *Int'l Conf on Computer Design*, 1992.

[3] D. L. Dill, S. Park, and A. G. Nowatzyk. Formal specification of abstract memory models. *Research on Integrated Systems: Proc of the 1993 Symp*, pp. 38–52. MIT Press, 1993.

[4] A. J. Hu, M. Fujita, and C. Wilson. Formal verification of the HAL S1 system cache coherence protocol. *Int'l Conf on Computer Design*, 1997, pp. 438–444.

[5] A. J. Hu, R. Li, X. Shi, and S. Vuong. Model checking a secure group communication protocol: A case study. *Joint Int'l Conf on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification (FORTE/PSTV)*. IFIP TC6/WG6.1, 1999.

[6] *IEEE Standard for a High Performance Serial Bus*. IEEE Std 1394-1995, August 1996.

[7] C. N. Ip and D. L. Dill. Efficient verification of symmetric concurrent systems. *Int'l Conf on Computer Design*, 1993, pp. 230–234.

[8] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murphi. *IEEE Symp on Security and Privacy*, 1997, pp. 141–151.

[9] J. C. Mitchell, V. Shmatikov, and U. Stern. Finite-state analysis of SSL 3.0. *7th USENIX Security Symposium*, 1998.

[10] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997. pp. 443–446.

[11] L. Yang, D. Gao, J. Mostoufi, R. Joshi, and P. Loewenstein. System design methodology of UltraSPARC-I. *32nd Design Automation Conference*, 1995, pp. 7–12.