Comparing Symbolic and Explicit Model Checking of a Software System

Cindy Eisner[1] and Doron Peled[2]

[1] IBM Haifa Research Laboratory
Haifa, Israel
[2] Dept. of Elec. and Comp. Eng.
The University of Texas at Austin
Austin, TX 78712

Abstract. There are two main paradigms for model checking: symbolic model checking, as is performed by the tool RuleBase, and explicit state model checking, as is performed by Spin. It is often accepted that the former is better for verifying hardware systems, while the latter has advantages for verifying software. We examine this piece of common wisdom in light of experience in verifying the software of a disk controller using both symbolic and explicit state model checking.

1 Introduction

Hardware and software model checkers are often distinct. The former are usually based on a symbolic search strategy [14] that examines sets of states in each step, while the latter are often based on explicit state search, where progress is made one state at a time. More importantly, hardware model checkers usually use a synchronous model of execution, in which all the enabled processes progress simultaneously, while the execution model of software model checkers is usually asynchronous (not to be confused with asynchronous message passing), where the execution of independent transitions can be interleaved in all possible orders.

In this paper we report on a verification project performed at the IBM Haifa Research Laboratory. In this project, we verified the software for a disk controller using both symbolic as well as explicit model checking. This verification project stands on the cutting edge of verification technology for the following reasons:

- The size of the project. We deal with actual software, over 2,500 lines of code, with minimal number of abstractions. The attempt is to avoid manual modeling, and verify the software *almost as is*. We used abstraction only when absolutely necessary, in particular, when pointer manipulation is involved. The checked software involves several concurrent processes with shared variables, each consisting of tens of recursive procedures.
- The state space involved. The system has a parametric number of processes. Even a single process of this system involves a huge number of states due to its interaction with the environment.
- The modeling technique, which involved direct compilation of the C code into the target languages used by the two model checkers. Since several different efficiency issues were involved, we constructed a parametric automatic translator and compared the efficiency of the different translation modes. For this reason, the usual methodology of repeatedly refining the model using

the counterexample found during verification was augmented by repeatedly refining the translator. We believe that the translation techniques and the tool generated for this project can be useful for similar projects, especially taking into account the popularity of the C language.

The symbolic part of the work was performed using RuleBase [4, 3]. RuleBase was originally based on a version of SMV [14]. After eight years of development, the original SMV code is a small part of the whole. Alternative model checking algorithms [17, 16, 1, 6] have been added, the original engine has been optimized [15, 10], the temporal logic has been enhanced [2], and features [5] supporting debugging of formulas (as opposed to debugging of the model) have been developed. The explicit model checker used in the work described in this paper was Spin [11].

2 The Verified Software

The software verified is a distributed storage subsystem software application. In other words, it is the software that runs on hardware that connects one or more computers with one or more magnetic disks. It is a distributed application that runs on several nodes, providing full data availability after the failure of a single node, and at least partial service in the event of simultaneous failures. Like most storage subsystems, it contains a cache to speed up read and write operations. The algorithms used to implement the software cache are the subject of the work described in this report.

A software cache, like a hardware cache, holds a copy of the data stored elsewhere (on disk in the case of a software cache, in main memory in the case of a hardware cache). The copy can be either *clean*, which means that the data is identical to that on disk, or *dirty*, which means that the cached copy is different from (and newer than) that on disk. If the cached copy is dirty, then the cache algorithms must ensure that any read of the dirty address gets the newer (cached) copy, and that the new data is written back to disk before the address is discarded from the cache. Because the system is distributed, the cache algorithms must also provide the coordination between multiple nodes. In the case of multiple cached copies, all are identical. Because the system must ensure high availability, there is an additional functionality, which is to ensure that there are always at least two copies of dirty data; a second copy is used in the case of a failure of the node holding the first copy.

The software consists of approximately 20,000 lines of C code; the part that was verified in this project is approximately 2,500 lines. The code is structured as a large number (slightly more than 100) small functions and procedures, which pass pointers to *work buffers*, each of which contains information regarding a single request being processed. The procedures are (non-tail) recursive, although a bound on the depth of the stack is known. A set of procedures operating on one request communicate using shared variables.

3 Issues in Model Translation

Issues in representing C programs for RuleBase

The input language of hardware model checkers is usually very simple, reflecting the simple semantics of hardware description languages. EDL, the input

language of the RuleBase model checker, is no exception. An EDL model basically consists of a set of next state functions, describing a synchronous model in which the next state of each variable is calculated in parallel with all the others. The basic idea behind translating software into EDL follows directly from [13, 8], and was described in [9]. A simple example will suffice to explain the main idea. Consider first the C function getmax() shown below.

```
  getmax (){
    int max, a;
0   a = max = 0;
1   do {
2     if (a > max)
3       max = a;
4     a = input();
5   } while(a);
6   return(max);
7 }
```

We start by annotating the code with the value of the program counter (pc). We then restrict the integers a and max to a finite range, say 0 through 3. If we interpret the call to $a = input()$ on line 4 as a non-deterministic assignment to the variable a, it is a simple process to rewrite getmax() in terms of next-state functions of the variables, as shown below.

```
next(a) = if pc=0 then 0
                  else if pc=4 then {0,1,2,3}
                  else a
next(max) = if pc=0 then 0
                    else if pc=3 then a
                    else max
next(pc) = if pc=0 then 1
                   else if pc=1 then 2
                   else if pc=2 then if a>max then 3 else 4
                   else if pc=3 then 4
                   else if pc=4 then 5
                   else if pc=5 then if a then 1 else 6
                   else if pc=6 then 7
                   else if pc=7 then 7
```

With minor syntactic changes and the addition of state variable declarations, this is a complete EDL program, and can be model checked using RuleBase.

The code verified in this project contains many constructs not covered by the basic solution described in [9] and presented above. Among these are: use of complex data types, such as structures and pointers, overloading of memory location names through the use of unions, and support for local variables and parameters. Structures, pointers, and unions were solved manually, by modifying the C code before the automatic translation into EDL: structures were flattened into sets of individual variables; pointers were converted into integer indices into an array of variables of the types pointed to; unions were resolved so that only one name was used for each data location. Support for local variables and paramters were solved automatically, by modifying the tool, with one caveat: one of the characteristics of local variables is that a recursive function call gets its own copy (unless the variables are declared as static). Because the software verified in this project contains no local variables which need this behavior to function correctly, this feature of local variables was ignored.

Issues in representing C programs in Spin

The input language of software model checkers is usually much richer than that of hardware model checkers. However, even software model checkers usually allow

a restricted syntax for representing the model of the checked system. The design of such a syntax is influenced by several objectives:

Expressiveness. The class of systems that may be verified is quite large. It can include hardware designs, as well as sequential or concurrent programs.
Simplicity. Modeling is perhaps the most difficult task of the verification. Thus, it should be made as easy as possible for the user.
Efficiency. The translation from the syntactic representation of the model into the internal representation, used by the tool, should be as fast and as space efficient as possible.

SPIN's modeling language PROMELA has a rich syntax. It includes C expressions, Dijkstra's *guarded command* sequential constructs and Hoare's CSP communication operators, representing synchronous and asynchronous communication. It also allows a mode of lossy communication channels, and the use of shared variables. Thus, PROMELA allows modeling a wide range of software systems. Still, translation is often required, since the verified system can be represented using any programming language. Because of the wealth of programming languages, automatic translation is usually not available.

The easiest part of the translation is changing the C syntax into PROMELA. Another straightforward problem is the representation of procedures. PROMELA does not have a special construct for representing procedures. However, as shown in the SPIN manual [12], we can simulate procedures with processes. Consider for example the following C code:

```
int firstproc (int num)
{
   int local;
   ...
   sv=secondproc(d);
   ...
   return local;
   ...
}
```

We simulate procedure `firstproc` with a process of the same name. In order not to allow concurrency between the process representing the calling procedure (`firstproc`) and the process representing the called procedure (`secondproc`), the calling procedure waits until it receives the returned value from the called procedure. If no value is returned in the original code, we return a dummy value. Thus, the PROMELA code for `firstproc` is as follows:

```
proctype firstproc(int num; chan ret)
{
   int local;
   chan mychan = [1] of {int};
   ...
   run secondproc(d, mychan);
   mychan?sv;
   ...
   ret!local;
   goto end_firstproc;
   ...
   end_firstproc:skip}
```

We use the same channel name in all processes that simulate a procedure. According to our translation, the channel is called `mychan` by the calling procedure, while the called procedure recognizes the channel by the name `ret` (since the channels *it* uses to communicate with the procedures it calls are called `mychan`).

We also include the directives `xs ret` and `xr mychan` at the beginning of each process in order to report that the process exclusively sends messages on `ret` and exclusively receives messages on `mychan`. These directives allow the partial order reduction algorithm used in SPIN to better reduce the number of states and transitions explored. However, when a procedure calls more than one other procedure, then it can receive messages from the multiple corresponding processes on the same channel. In this case, sending to the channel `mychan` (recognized as `ret` in the called procedures) is not exclusive. It can be shown that because of the sequentiality in the execution of these procedures, the partial order reduction could be executed correctly albeit the non exclusiveness. However, SPIN is not programmed to deal with such a special case. One solution is to manually remove such directives when they do not truly reflect exclusivity. Experiments we have performed in doing that show that the reduction obtained by adding such directives in our code was negligible anyhow, and thus eliminating the need for a complicated solution.

A trickier problem involves the spawning of multiple copies of a process. The checked system is designed to work with a parameterized number of processes. Each copy starts with one main procedure, `main`, which calls other procedures recursively. Each `main` needs to communicate with the processes it has spawned using a set of shared variables. We model these shared variables in PROMELA as a set of global variables, one per main process. Say, for instance, that we want to declare the integer variable `glb` for 3 processes. We declare it as an array variable using `int glb[3]`. Access to these global variables now depends on the process number, which we pass as a parameter from the main process to its called processes. Thus, we execute `run main(1)`, `run main(2)`, `run main(3)`, and convert a process `firstproc(parm)` that calls `secondproc(moreparm)` into `firstproc(id,parm)` calling `secondproc(id,moreparm)`.

Tradeoffs in the translation process

Instead of the solution presented above for spawning multiple copies of a process, we can choose to define the fixed processes `main_1`, `main_2`, ..., and similarly, `firstproc_1`, `firstproc_2`, etc. We then declare copies of the global variables, e.g., `glb_1`, `glb_2`, ... This generates, for n processes, n times more code, and more entries in the process table. This can be a disadvantage, since SPIN keeps several internal enumerating mechanisms limited to one byte.

For non-recursive procedures, we also have an alternative to the representation of procedures as processes as described above. For procedures which are not recursive, we can replace the call for that procedure with its code, as in macro expansion. There is an obvious tradeoff in doing so. When a procedure is represented as a process, extra memory is allocated in each state for the program counter and its local variables. These include in particular the channels through which the process communicate with the processes representing the procedures calling it or called by it. This is avoided when a macro expansion is performed. The macro expansion usually involves only an increase in the address space of the program counter of the process that includes this procedure. In this case, the trace of the counterexample loses the information about the original chain of procedure calls. For this matter, some additional assignments can be performed in order to annotate the counterexamples with the necessary information.

State size vs. state space size

There are two separate measures of the size of the model under model checking. One is the number of bits needed to represent a single state (state size), while the other is the number of states in reachable state space (state space size). While related, these two measures are not tightly coupled. In particular, if we model a digital circuit with n bits, there can be no more than 2^n reachable states. In practice, there are many dependencies, thus the number of reachable states is usually much less than 2^n.

In symbolic model checking, the number of bits needed to represent a single state is limited by the internal data structures. However, because BDDs give a compact way to represent sets of states, the number of reachable states is not a problem: the size of a BDD is not proportional to the number of states it represents. In explicit model checking, both state size and the state space size are important.

4 Experimental Results

4.1 Results with symbolic model checking under RuleBase

Using the translation process described in Section 3, the RuleBase model was built. Due to size problems, only a 2-process system could be model checked. In that configuration, problems were found. Of those, nine were modeling errors, one was deemed a possible bug, one a real bug, and one a problem in a portion of the code not yet relevant to the current release.

All other rules passed for the 2-process configuration. Run time [3], memory usage, and model size is summarized in Figure 1. There are two surprises in

CPU time to compile the EDL model	3.5 minutes
memory use to compile the EDL model	625M
CPU time to perform the model checking	28 hours
memory to perform the model checking	249M
lines of pseudo-code in C model	2478
number of state variables in EDL model	362
number of reachable states	10^{150}

Fig. 1. Run time, memory usage, and model size for a 2-process system under RuleBase

Figure 1. First of all, the fact that the memory requirements of the compilation phase of RuleBase were 625M, while the memory requirements of the model checking itself were only 249M. Model checking is usually much more expensive in terms of memory requirements than any pre-processing phase. A reduction algorithm dedicated to software could probably do a better job at reduction

[3] All run times in this paper are for an IBM RS/6000 workstation model 270

while reducing the memory requirements considerably. The second surprise is that despite the long run time (28 CPU hours), the memory requirements of the model checking itself were quite small - there was no state space explosion. This was not true of early runs of the model, which exploded quickly. Additions of hand-coded hints [16] achieved this impressive result. The hints controlled the interleaving of the concurrent runs of the two work buffers, so that a fixed-point was first reached with control belonging only to the first work buffer, then only to the second, and so on. Only after a number of iterations were the hints released so that more complicated interleavings could be examined.

4.2 Results with explicit model checking under SPIN

The SPIN model was built using the translation process described in Section 3, The hope was to be able to verify a larger system than the 2-process configuration which was the limit for RULEBASE. Obviously, with the number of states reported in the table in Figure 1, we did not expect to be able to cover even a small fraction of the states. Despite this, it was decided to attempt model checking of a 3-process system using SPIN in the hopes of encountering new behavior which might uncover problems not seen in the 2-process system. Several runs were done, each restricting the environment (inputs to the system) in a different way. All of these runs spaced out at 10^8 states without finding any new problems. Run time, memory usage, and model size for a 3-process configuration under SPIN is summarized in the table in Figure 2.

CPU time to compile the Promela model	negligible
memory use to compile the Promela model	negligible
CPU time to perform the model checking	1 cpu sec. per 20K st.
memory to perform the model checking	spaced out at 2G
lines of pseudo-code in C model	2478
number of state variables in Promela model	419
number of states seen	10^8

Fig. 2. Run time, memory usage, and model size for a 3-process system under SPIN

5 Discussion

The size of the state space for a 2-process model as shown in Figure 1 was 10^{150}. Why then did we expect to be able to veirfy something even larger by explicit state model checking? The answer is that things are not as straightforward as they seem, even in a simple matter like counting the number of states.

In explicit state model checking, we perform a search (usually depth first search) through the states of the checked system. The search proceeds state by state (although some reduction techniques allow us to skip some states or to step

through representatives of equivalence classes of states). The search is performed in the *forward* direction, from a state to its successors, unless the search strategy requires backtracking. Thus, it is straightforward to assess the number of states encountered during the verification. Moreover, all the states participating in the search are reachable from the initial state(s).

In symbolic model checking, we keep a compressed representation of the state space (usually, using a binary decision diagram, or BDD [7]). At each stage, the representation holds a collection of states. We can apply a transformation to that set of states, finding the set of states reachable from them with one transition (forward search) or the set of states from that can transform to the given set within one transition (backward search). Then we can append the given set of states or restrict it depending on the search strategy and the property checked.

Symbolic model checking can be performed using the classic algorithm [14] in which state traversal is performed backward from the set of states violating the property checked. In backwards traversal, we either hit an initial state, in which case we have found a violation of the property being checked, or we have reached a fixed-point, in which case we know that there is no violation. In backwards traversal, we do not calculate the set of reachable states, hence we do not report the number of reachable states.

The application of symbolic model checking described in this paper was done using *on-the-fly* symbolic model checking [6], in which we use forward traversal, just like explicit state model checking. Since we know how to count the number of states seen in each symbolic step, it is straightforward, as with explicit state space traversal, to assess the number of states encountered during the verification. Moreover, as with explicit state space traversal, all the states participating in the search are reachable from the initial state(s). The 10^{150} states report in Figure 1 are states reachable from the initial state in the constructed RuleBase model.

However, because of its particular way of compressing the state space, it is often the case that the BDD representation requires less memory when *more* states are added. In fact, it may be beneficial to 'pad' the state space with extra states in order to reduce the size of the BDD. This is done, in fact, in our EDL representation of the checked system. One example of this is that since the recursive stack is represented on the state space, its unused slots are padded with all possible values. Thus, the number of reachable states in an EDL model does not mean that an equivalent model in Promela necessarily reaches the same number of states. Thus, our initial optimism even in the face of the huge number of reachable states reported by RuleBase.

As expected, when applying Spin, we encountered an explosion in the number of states. The state space problem was more severe with the explicit state model checking with Spin. We could only check 10^8 states before spacing out at 2G memory, and our gut feeling was that this was a small part of the real state space of the model. In order to check our intuition, we ran RuleBase on a 2-process model, after modifying the code in order to cancel out the states that were 'padded' for RuleBase performance reasons as described above. This analysis showed that the Spin model could be expected to contain at least 10^{65} reachable states. Even with advanced hashing modes, which allow representing each state as a small number of bits (even as one bit), we cannot expect to cover this number of states with the explicit search strategy. On the other hand, because of the limit on state size, RuleBase could not represent a state large enough to include the information needed for more than a 2-process configuration. In

that respect, RuleBase gave us a better assurance about the verified system for a low number of processes, while Spin allowed us to simulate executions with more processes.

References

1. J. Baumgartner, T. Heyman, V. Singhal, and A. Aziz. Model checking the IBM Gigahertz Processor: An abstraction algorithm for high-performance netlists. In *Proc.* 11^{th} *International Conference on Computer Aided Verification (CAV)*, LNCS 1633, pages 72–83. Springer-Verlag, 1999.
2. I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The temporal logic Sugar. In G. Berry, H. Comon, and A. Finkel, editors, *Proc.* 13^{th} *International Conference on Computer Aided Verification (CAV)*, LNCS 2102. Springer-Verlag, 2001.
3. I. Beer, S. Ben-David, C. Eisner, D. Geist, L. Gluhovsky, T. Heyman, A. Landver, P. Paanah, Y. Rodeh, G. Ronin, and Y. Wolfsthal. RuleBase: Model checking at IBM. In *Proc.* 9^{th} *International Conference on Computer Aided Verification (CAV)*, LNCS 1254. Springer-Verlag, 1997.
4. I. Beer, S. Ben-David, C. Eisner, and A. Landver. RuleBase: an industry-oriented formal verification tool. In *Proc.* 33^{rd} *Design Automation Conference (DAC)*, pages 655–660. Association for Computing Machinery, Inc., June 1996.
5. I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in temporal model checking. *Formal Methods in System Design*, 18(2), 2001.
6. I. Beer, S. Ben-David, and A. Landver. On-the-fly model checking of RCTL formulas. In *Proc.* 10^{th} *International Conference on Computer Aided Verification (CAV)*, LNCS 1427, pages 184–194. Springer-Verlag, 1998.
7. R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.
8. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
9. C. Eisner. Model checking the garbage collection mechanism of SMV. In S. D. Stoller and W. Visser, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier Science Publishers, 2001.
10. D. Geist and I. Beer. Efficient model checking by automated ordering of transition relation partitions. In *Proc.* 6^{th} *International Conference on Computer Aided Verification (CAV)*, LNCS 818, pages 299–310. Springer-Verlag, 1994.
11. G. Holzmann. On the fly, ltl model checking with spin: Simple spin manual. In *http://cm.bell-labs.com/cm/cs/what/spin/Man/Manual.html*.
12. G. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.
13. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.
14. K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
15. K. Ravi, K. McMillan, T. Shiple, and F. Somenzi. Approximation and decomposition of binary decision diagrams. In *Proc.* 35^{th} *Design Automation Conference (DAC)*. Association for Computing Machinery, Inc., June 1998.
16. K. Ravi and F. Somenzi. Hints to accelerate symbolic traversal. In *Proceedings 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, LNCS 1703, Bad Herrenalb, Germany, September 1999. Springer-Verlag.
17. O. Shtrichman. Pruning techniques for the SAT-based bounded model checking problem. In T. Margaria and T. F. Melham, editors, *CHARME*, volume 2144 of *Lecture Notes in Computer Science*. Springer, 2001.

This article was processed using the LaTeX macro package with LLNCS style