

Wolf - Bug Hunter for Concurrent Software using Formal Methods

Sharon Barner, Ziv Glazberg and Ishai Rabinovitz

IBM Haifa Research Lab, Haifa, Israel
{sharon,glazberg,ishai}@il.ibm.com

Abstract. Wolf is a “push-button” model checker for concurrent C programs developed in IBM Haifa. It automatically generates both the model and the specification directly from the C code. Currently, Wolf uses BDD-based symbolic methods integrated with a guided search framework. According to our experiments, these methods complement explicit exploration methods of software model checking.

1 Introduction

Wolf is a software bug hunter that uses formal methods in order to discover bugs in a concurrent C program. Concurrency related bugs may depend on a specific rare interleaving of the program, and are likely to be missed by standard testing tools. In addition, even when the presence of a bug is known, it is hard to reproduce it, or locate its source. When using formal methods, we can explore all possible interleavings; hence, detecting the presence of a bug. In addition, we can generate the trace that led to the bug.

Wolf is a “push-button” bug hunter, that is, it does not require intervention by the user. Unlike with hardware verification, users need not be experts in verification. Users who are familiar with verification can use their knowledge of the software to tune the model or verify additional specification. Wolf operates the IBM model checker RuleBase PE [4] using special software algorithms [3] on an automatic generated model and cleanliness properties. Cleanliness properties are those that every program should obey. For example, a dangling pointer should never be dereferenced, there should never be a deadlock situation, and no assert should fail. Programmers can validate functional properties of a program using the assert mechanism with which they are already familiar.

Wolf displays the bug trace in a debugger-like GUI named Vet. Vet translates the trace into an easy-to-understand graphical display which is intuitive for the average programmer.

Explicit model checkers use various reductions to reduce the branches that they explore, such as partial order reductions [6], and they handle the dereferencing of pointers and the dynamic allocation of resources well. However, they have a hard time coping with non-deterministic behaviors that result from non-deterministic inputs.

SAT-based methods generate a formula that represents bounded length executions. SAT’s main drawback is that it does not cope well with long formulas. SAT-based software model checkers try to compact the formula by creating basic

blocks [11] or single assignment formulas [7]. Neither [11] nor [7] supports concurrent programs. A new approach, taken by [12], tries to extend this approach for such programs.

We believe that there is no ultimate software model checking algorithm. Each different kind of software and bug has a different winning algorithm. We designed Wolf as an easy to upgrade framework, which enables us to add different algorithms in the future.

Currently, we are exploring symbolic algorithms. In contrast to Bebop [2] and JPF [9], which combine symbolic and explicit methods, Wolf uses a pure symbolic exploration. Other software model checkers use SAT-based algorithms such as CBMC [7] and NEC [11], or explicit algorithms such as ZING [1], SPIN [10] and BOGOR [13]. BDD-based symbolic methods are less bounded than SAT-based methods and, unlike explicit methods symbolic methods, support non-deterministic choices naturally. In this sense, symbolic algorithms complement SAT-based and explicit algorithms.

Wolf’s symbolic BDD algorithm uses partial disjunctive partitions [3]. This method uses the software property of a few changes in each ”cycle” to enhance the speed of image computation by several orders of magnitude.

We acknowledge the fact that the verification of concurrent software is a hard task, due to the enormous number of possible states. This can cause state explosion to occur when dealing with big and complex models. In order to avoid this problem and find a bug prior to explosion, we use automatically generated ”hints” and ”guides” to manipulate the model checker into examining ”interesting” states. This way, we enter bug-pattern knowledge [8] into Wolf and allow it to search for specific occurrences of these patterns.

Section 2 describes the internal modules within Wolf. Section 3 presents our experimental results, and in section 4, we discuss possible future directions.

2 Structure of Wolf

Wolf is composed of three modules: a C-to-model translator, software verification algorithms integrated into RuleBase PE, and Vet.

Translator

The translator receives a concurrent C program and generates a finite model. This translation, described in [3], is not trivial. We model software concurrent control using two variables: TC (thread chooser) and PC (program counter vector). These signals point to the next command to be executed. In each cycle, only TC, PC, and one additional variable may change their values in order to allow the use of disjunctive partitions [3]. Moreover, the translator models the synchronization primitives¹. Similarly to the method described in [12], the modeling of these primitives removes some unnecessary interleavings from the model without changing its behavior. The translator also bounds the number of threads, the size of the heap, of the stack, and of all data types, making the problem decidable. In addition to that, it generates cleanliness properties such as assert

¹ Currently, we support concurrent programs that use Pthread libraries.

never fails, no use of/set to dangling pointer, no deadlock occur, no livelock, no data race, etc. The translator also generates hints and guides that will guide the software model checker toward the bug. These hints and guides are generated according to the program and the user’s preferences.

Software Model Checker:

Wolf uses a specialized version of RuleBase PE [4] as its underlying model checker. As mentioned, we are currently focusing on symbolic model checking using a BDD based model checker. Our algorithm uses disjunctive partitions [3], which enable faster image computation for software models. We implemented a dynamic BDD reordering algorithm especially designed to reduce the reordering time of software disjunctive BDDs.

Since we are using guided search, we implemented two different ways to divert the model checker to examine interesting states: hints and guides. In [5], the hints method was presented: in each step, the image computation is intersected with the current hint. When a fixed point is reached, the hint is replaced with the next hint. When all hints are used, another search is done from the reachable states of the last iteration without any constraints. We implemented this method with one change: our hints are cyclic, meaning that when the last hint results in a fixed-point, the first hint is rechecked and so forth. Only when all hints do not discover a new state, the final search is performed. Our hints are usually used to force a round-robin execution of the threads: when the i -th hint is active, we intersect the image with $TC = i$. We observed that using such hints kept the BDDs relatively small.

Guides are another method to direct the model checker toward a bug. After each K iterations, the model checker finds the highest priority guide that is satisfied in the frontier², and continues to step forward only from the states in the frontier that satisfies this guide. If no bug is found after one round-robin, the model checker backtracks and explores other states. For example, when looking for a deadlock, a guide may be *one thread is locked* and a higher priority guide may be *two threads are locked*.

Vet: Wolf’s debugger:

Finally, Vet displays the trace in a debugger-like GUI, highlighting the code line that caused the bug³. Vet allows the user to traverse the trace forward and backward. Also, the user can display watches over the variables. Since the trace is of concurrent software, Vet is designed in such a way that all the threads are displayed side-by-side, and the next line to be executed in each thread is marked. Figure 1 shows a trace as it is presented by Vet.

3 Experimental Results

We ran Wolf on two Linux drivers and synchronization code taken from an IBM’s software group. We detected access violation bugs, data races, and deadlocks.

² A frontier is the set of new states which were discovered in the last iteration.

³ The model checker exports information about the specific state that led to the property failure. Vet translates this into a specific code line.

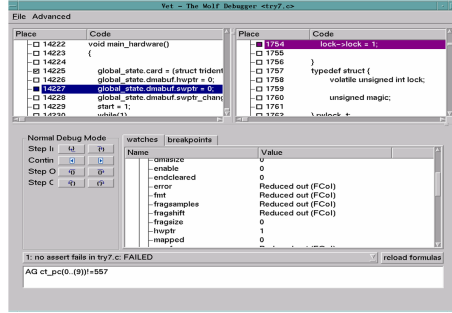


Fig. 1. Vet - wolf’s debugger

In addition, we tried to estimate the potential that symbolic model checking for software has with respect to explicit model checking. We compared Wolf with Zing [1] on three different types of programs. This comparison is problematic because it could be done only on different platforms. However, it can be seen in Table 1 that symbolic model checking for software scales better when bugs occur only in rare interleavings, especially when the program has nondeterministic inputs.

| Example | Number of Threads | Wolf Symbolic MC | Zing Explicit MC |
|--|-------------------|------------------|------------------|
| Deterministic input common interleavings | 6 | 2100s | 374s |
| Deterministic input rare interleavings | 8 | 21907s | 614s |
| | 9 | 31200s | > 10h |
| Non-deterministic input rare interleavings | 7 | 724s | 360s |
| | 8 | 794s | > 10h |
| | 9 | 1373s | > 10h |

Table 1. Comparison between Wolf and Zing with of different types of programs.

4 Future Work

We are now working on new guides and hints based on known bug-patterns [8]. We believe that bug-hunting for hard to detect bugs is worthwhile for our costumers. Simple bug-patterns, such as lock one thread then lock other thread on the same mutex, are already implemented in Wolf and show great promise. In addition, we are interested in introducing other model checking algorithms to the Wolf platform such as explicit and SAT-based model checking.

References

1. T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *CAV*, pages 484–487, 2004.
2. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proc. 7th International SPIN Workshop*, 2000.
3. S. Barner and I. Rabinovitz. Efficient symbolic model checking of software using partial disjunctive partitioning. In *CHARME*, pages 35–50, 2003.

4. I. Beer, S. Ben-David, C. Eisner, and A. Landver. RuleBase: An industry-oriented formal verification tool. In *Design Automation Conference*, pages 655–660, 1996.
5. R. Bloem, K. Ravi, and F. Somenzi. Symbolic guided search for CTL model checking. In *Design Automation Conference*, pages 29–34, June 2000.
6. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT press, 1999.
7. E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ansi-c programs. In *TACAS*, pages 168–176, 2004.
8. E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *IPDPS*, page 286, 2003.
9. K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA pathfinder. *STTT*, 2(4):366–381, 2000.
10. G. Holzmann. On the fly, LTL model checking with SPIN: Simple Spin manual. At <http://cm.bell-labs.com/cm/cs/what/spin/Man/Manual.html>.
11. F. Ivancic, Z. Yang, A. Gupta, M. K. Ganai, and P. Ashar. Efficient SAT-based bounded model checking for software verification, ISoLA, 2004.
12. I. Rabinovitz and O. Grumberg. Bounded model checking of concurrent programs, submitted to CAV, 2005.
13. Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *ESEC/SIGSOFT FSE*, 2003.