Boom: Taking Boolean Program Model Checking One Step Further[⋆]

Gerard Basler, Matthew Hague, Daniel Kroening,
C.-H. Luke Ong, Thomas Wahl, and Haoxian Zhao

Oxford University Computing Laboratory, Oxford, United Kingdom

Abstract. We present Boom, an analysis tool for Boolean programs. We focus in this paper on model-checking non-recursive *replicated* programs. Boom implements a recent variant of *counter abstraction*; efficiency is achieved using thread counters in a program context-aware way. While designed for bounded counters, this method also integrates well with the *Karp-Miller tree* construction for vector addition systems, resulting in a reachability engine for Boolean programs with *unbounded thread creation*. The concurrent version of Boom is implemented using BDDs and includes partial order reduction methods. Boom is intended for model checking system-level code via predicate abstraction. We present experimental results for the verification of Boolean device driver models.

1 Introduction

Over the past decade, *predicate abstraction* has established itself as a viable strategy for model checking software, as witnessed by the success of device driver verification in Microsoft's Slam project. The input program is converted into a finite-state *Boolean program*, whose paths overapproximate the original behavior.

Recently, *concurrent* software has gained tremendous stimulus due to the advent of multi-core computing architectures. The software is executed by asynchronous parallel threads, communicating, in the most general case, through fully shared variables. Bugs in such programming environments are known to be subtle and hard to detect by means of testing, strongly motivating formal analysis techniques for concurrent programs.

In this paper, we present Boom, a model checker for Boolean programs. While Boom has many features that make it useful for sequential programs [4], we focus here on analyzing the set of reachable states of a *replicated* non-recursive Boolean program. Replication often induces symmetry, which can and must be exploited for the analysis to scale. We present our implementation of a *context-aware* form of *counter-abstraction* [3], and compare its performance to alternative reduction techniques also implemented in Boom, and to other tools.

[⋆] Work supported by the Swiss National Science Foundation (SNF) under grant 200021-109594 and by the Engineering and Physical Sciences Research Council (EPSRC) under grant no. EP/G026254/1 and EP/D037085/1.

Replication materializes in practice as dynamic thread creation. Even without a bound on the number of running threads, the reachability problem for non-recursive concurrent Boolean programs is decidable. We have extended Boom by a variant of the *Karp-Miller tree*, which operates directly on Boolean programs. We demonstrate that our implementation performs much better in practice than the worst-case complexity of the construction seems to suggest. The result is a practically useful and **exact** reachability analysis for realistic concurrent Boolean programs with arbitrarily many threads.

2 Concurrent Boolean Program Analysis with Boom

Boom is capable of analyzing the reachable state space of *replicated* programs, where threads may dynamically create other threads during the execution. The Boolean variables are declared either *local* or *shared*. Each thread has its own private copy of the local variables. The shared variables, in contrast, are fully accessible to every thread. Shared-variable concurrency is very powerful and able to simulate many other communication primitives, such as locks.

Concurrent Boolean programs with replicated threads are naturally *symmetric*: the set of transitions of a derived Kripke model is invariant under permutations of the threads. Symmetry is *the* concurrency-related cause for state explosion in replicated Boolean programs. Boom tackles this problem using a form of *counter abstraction*: global states are represented as vectors of counters, one per local state. Each counter tracks the number of threads in the corresponding local state. A transition by a thread translates into an update of the counters for the source (-1) and target ($+1$) local state.

The idea of using process counters has long proved useful for verifying replicated *local-state transition diagrams*. Unfortunately, statically converting a concurrent Boolean program into such a diagram suffers from the *local state explosion problem*: the number of local states is exponential in the program text size. We recently proposed **context-awareness** as a solution [3]: at exploration time, the context in which a statement is executed is known and exposes the local-state counters that need to be updated. As a natural optimization, a global state in Boom only keeps counters for local states that at least one thread resides in. The number of occupied local states is obviously bounded by the number of running threads, which tends to be a tiny fraction of all local states.

Extending Boom to Unbounded Thread Creation

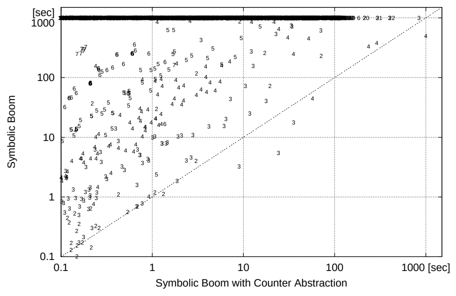
If there is no limit to how many threads may be running, the thread counters become unbounded non-negative integers. The induced transition system is an instance of a *vector addition system with control states* (VASS); the control state represents the values of the shared program variables. The reachability of a *thread state* (s, l) (combination of shared and local state) in a concurrent Boolean program is easily reducible to a VASS *coverability* problem. The latter problem is decidable, using a tree construction proposed by Karp and Miller [8].

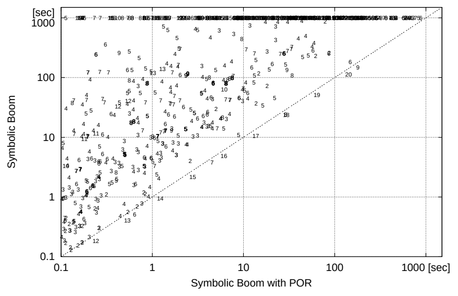
Boom uses the Karp-Miller construction as the starting point for an algorithm to decide thread-state reachability. The local state explosion problem materializes here as the *dimensionality problem* for VASS and Petri nets. Fortunately, our earlier solution of a context-aware, on-the-fly translation carries over quite seamlessly to the unbounded case. Our implementation can be seen as a version of the Karp-Miller procedure that operates directly on a Boolean program. Bypassing the VASS gives us the opportunity to avoid the blowup that a static translation into any type of addition system invariably entails. Furthermore, to ameliorate the exponential-space complexity of the Karp-Miller construction, we exploit the special form of vector-addition systems derived from Boolean programs. For example, our implementation keeps a copy of those tree nodes that are *maximal* with respect to the covering relation as partial order. Newly discovered nodes are compared against these maximal nodes only.

3 Results

Boom and our benchmarks are available at `http://www.cprover.org/boom`; we refer the reader to this website for more details on the tool and the benchmarks.

The left chart below compares a plain symbolic exploration of the concurrent Boolean program against Boom's implementation of (bounded) counter abstraction. Each data point specifies the numbers of threads running. The message of this chart is obvious. The right chart compares plain exploration against Boom's implementations of partial-order reduction. Comparing left and right, we see that counter abstraction performs somewhat better. In other experiments (not shown), we observed that combining the two gives yet better performance.



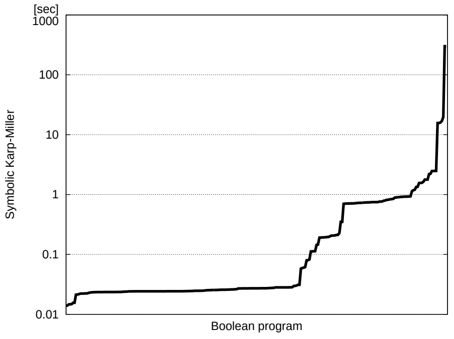


The chart on page 4 (left) compares the bounded version of Boom with counter abstraction against the *lazy* version of GetaFix [9] (which performs better than eager). GetaFix targets *recursive* Boolean programs with a bounded number of context-switches. To compare with Boom, we chose a non-recursive example provided on the GetaFix website. The time for GetaFix to convert the example into a sequential program is tiny and omitted. The table illustrates the time to explore the sequentialized program using Moped-1, for different context-switch bounds. Note that Boom, in contrast, explores any interleaving.

The graph on page 4 (right) shows our preliminary thread-state analysis of Boolean programs with unbounded thread creation. We see that for many examples, the running times are very small. On the other hand, 301 of 570 cases did not terminate within 60 min. We observed no more than 43 non-zero counters in any global state, despite millions of conceivable local states.

n	BOOM	GETAFIX/cont. bd. [sec]					
	[sec]	1	2	3	4	5	6
2	< 0.1	0.1	0.4	2.0	8.7	41	139
3	0.1	0.1	1.0	0.6	4.8	30	187
4	1.2	0.1	1.9	1.2	12.2	146	1318
5	12.1	0.14	2.8	2.3	30.6	426	—
6	88.8	0.2	3.9	3.1	51.7	901	—

Benchmarks on Intel 3GHz, with timeout 60 mins, memory-out 4 GB.



4 Related Work and Conclusion

There are a few tools for the analysis of sequential Boolean programs [2, 6]. When extended to multiple threads, the problem becomes undecidable. To allow an algorithmic solution, BOOM disallows recursion. There are many tools available for the analysis of VASS. Closest to our work are the applications of these tools to Java [5, 7] and to Boolean programs [1]. These tools compile their input into an explicit-state transition system, which may result in a high-dimensional VASS. Our experiments with explicit-state encodings (not shown) indicate that encoding Boolean programs symbolically is mandatory. We believe BOOM to be the first **exact** tool able to analyze non-recursive concurrent Boolean programs with *bounded* replication efficiently, and to extend the underlying technique to the *unbounded* case with reasonable performance.

References

1. T. Ball, S. Chaki, and S. Rajamani. Parameterized verification of multithreaded software libraries. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2001.
2. T. Ball and S. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *Model Checking of Software (SPIN)*, 2000.
3. G. Basler, D. Kroening, M. Mazzucchi, and T. Wahl. Symbolic counter abstraction for concurrent software. In *Computer-Aided Verification (CAV)*, 2009.
4. G. Basler, D. Kroening, and G. Weissenbacher. SAT-based summarization for boolean programs. In *SPIN*, 2007.
5. G. Delzanno, J.-F. Raskin, and L. V. Begin. Towards the automated verification of multithreaded Java programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2002.

6. J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In *Computer-Aided Verification (CAV)*, 2001.
7. G. Geeraerts, J.-F. Raskin, and L. V. Begin. Expand, enlarge and check... made efficient. In *Computer Aided Verification (CAV)*, 2005.
8. R. Karp and R. Miller. Parallel program schemata. *Computer and System Sciences*, 1969.
9. S. L. Torre, P. Madhusudan, and G. Parlato. Reducing context-bounded concurrent reachability to sequential reachability. In *CAV*, 2009.

Appendix

A Presentation

The presentation will be split roughly half-and-half into background information (Boolean programs, counter abstraction and the extension to the Karp-Miller construction) and tool demonstration. The presentation will feature the different algorithms and options offered by Boom in increasing difficulty and power, applied to a single, more or less easily readable Boolean program. At each stage, we will demonstrate Boom's performance, by comparing it to alternative implementations and tools, as we have in part already done in this submission. The demonstration will at least cover the following capabilities:

- how SatAbs generates a Boolean program from a C program:
  `http://www.cprover.org/boolean-programs/example1`
  While this is not done using Boom, it clearly is part of predicate abstraction and required for understanding the whole story.
- a comparison of how the performance of the different algorithms (sequential exploration, symbolic concurrent exploration, counter abstraction, karp-miller) with and without partial-order reduction.
  This emphasizes the capabilities of Boom.
- for longer computations, how statistical information is printed along the way:
  `http://www.cprover.org/boom/screenshots.shtml#run`
  This emphasizes the user-friendliness of Boom.
- explanations of the statistics indicated after the analysis:
  `http://www.cprover.org/boom/screenshots.shtml#statistics`

B Development and Applications of Boom

Boom has been developed in the context of the PhD thesis of one of the authors of this paper. The goal was a competitive reachability checker for the SatAbs-framework. It started as the first tool that implemented the summarization algorithm for Boolean programs using solvers for SAT and QBF formulas. Currently, MiniSat, Quantor and sKizzo are supported.

In the course of about the past two years, support for concurrent Boolean programs has been added; we focused here on BDD-based data structures. Beginning with a simple symbolic exploration engine, we added various forms of partial-order reduction to optimize the performance. This was followed by our implementation of counter abstraction, which is conceptually easy, but technically non-trivial in a symbolic setting, especially when tackling the local-state explosion problem mentioned in this paper. From there, it was a relatively small step to extend the method to unbounded threads, realizing that converting a Boolean program to a vector-addition system is tantamount to applying unbounded counter abstraction to it. The challenge is rather to make the exploration of the resulting counter machine efficient.

Currently, researchers from ETH Zurich and Oxford University are building tools that use Boom as the underlying reachability engine for Boolean programs. Among these are the SystemC simulator Scoot (`http://www.cprover.org/scoot`) and SatAbs (`http://www.cprover.org/satabs`). Embedded in the DDverify driver harness (`http://www.cprover.org/ddverify`), the combination of SatAbs and Boom is able to verify properties on a large set of concurrent Linux drivers.

C Availability of Boom

Boom is available at `http://www.cprover.org/boom`. This website contains

- detailed instructions on how to download and install Boom,
- screen-shots of using the tool, as mentioned in Section A.

A detailed explanation of the syntax and semantics of Boolean programs including a parser skeleton and a benchmark set is available at:
`http://www.cprover.org/boolean-programs`.