# Context-Bounded Translations for Concurrent Software: An Empirical Evaluation[⋆]

Naghmeh Ghafari[1], Alan J. Hu[2], and Zvonimir Rakamarić[2]

[1] Critical Systems Labs, Vancouver, BC, Canada
naghmeh.ghafari@cslabs.com
[2] Department of Computer Science, University of British Columbia, Canada
{ajh,zrakamar}@cs.ubc.ca

**Abstract.** Context-Bounded Analysis has emerged as a practical and successful automatic formal analysis technique for fine-grained, shared-memory concurrent software. Two recent papers (in CAV 2008 and 2009) have proposed ingenious translation approaches that promise much better scalability, backed by compelling, but differing, theoretical and conceptual advantages. Empirical evidence comparing the different translations, however, has been lacking. Furthermore, these papers focused exclusively on Boolean model checking, ignoring the also widely used paradigm of verification-condition checking. In this paper, we undertake a methodical, empirical evaluation of the three main source-to-source translations for context-bounded analysis of concurrent software, in a verification-condition-checking paradigm. We evaluate their scalability under a wide range of experimental conditions. Our results show: (1) The newest, CAV 2009 translation is the clear loser, with the CAV 2008 translation the best in most instances, but the oldest, brute-force translation doing surprisingly well. Clearly, previous results for Boolean model checking do not apply to verification-condition checking. (2) Disturbingly, confounding factors in the experimental design can change the relative performance of the translations, highlighting the importance of extensive and thorough experiments. For example, using a different (slower) SMT solver changes the relative ranking of the translations, potentially misleading researchers and practitioners to use an inferior translation. (3) SMT runtimes grow exponentially with verification-condition length, but different translations and parameters give different exponential curves. This suggests that the practical scalability of a translation scheme might be estimated by combining the size of the queries with an empirical or theoretical measure of the complexity of solving that class of query. Taken altogether, our results highlight the crucial importance of extensive experimental evaluation, provide practical guidance for using context-bounded analysis for research or application, and outline pitfalls and questions for further research as these and other translations are developed and improved.

## 1 Introduction

The original application for model checking was concurrent software, in the form of protocols (e.g., [9, 18]), and concurrent software continues to be a major impetus for

[⋆] This work was supported by a Microsoft Research Graduate Fellowship and the Natural Science and Engineering Research Council of Canada.

model checking. With changes in technology, new versions of the software model checking problem emerge. Currently, due to architectural and electrical constraints, Moore's Law is manifesting itself via an exponential growth in processor cores per chip, rather than the formerly exponential improvements in single-threaded performance. The result is a push for vastly greater levels of fine-grained, shared-memory concurrent software — in addition to classical message-passing and coarse-grained protocol-level concurrency — even in the most mundane applications. Such software needs verification.

It is possible, of course, to model check such software directly, and several pioneering systems provide that capability (e.g., [13, 19, 10, 32]). The state space is the cross product of all program variables, stacks, heaps, and program counters for all threads, and this state space can be explored as a transition system. The obvious challenge is extreme state explosion (if variable domains, stacks, and memories are modeled as finite) and/or theoretical undecidability (if any are modeled as infinite).

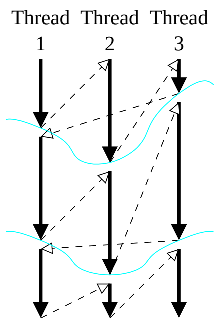
Context-Bounded Analysis (CBA) [26] promises a way around these challenges. Analogously to bounded model checking [8], the user specifies an integer constant that bounds the maximum number of execution contexts (i.e., periods of a thread running between context swaps) to be considered, and all concurrent executions up to that bound are analyzed. The downside, of course, is that if a bug requires more than that bound to manifest, it will be missed. The upside is that CBA reduces the analysis of concurrent software (under the context bound) to the analysis of sequential software. In theory, the advantage is that CBA is NP-complete [25, 23], whereas full concurrent software analysis is undecidable (even with finite variable domains and no heap, due to the call stack). In practice, CBA has proven its ability to detect hard concurrency bugs in real software, and many approaches rely on context-bounding to tackle the complexity of concurrent software (e.g., [26, 27, 24, 17, 30, 21, 20]).

The original CBA paper [26] used a source-to-source translation of concurrent to sequential program text, and subsequent work has followed that approach. The approach enables CBA to exploit all of the tools and algorithms for verification of sequential software, e.g., including the use of logics and decision procedures for reasoning about unbounded data domains, arrays, and heap-allocated memory. Recent papers by Lal and Reps [22] and by La Torre, Madhusudan, and Parlato [31] have proposed two ingenious and radically different source-to-source translations for CBA. These translations are more general than the original, but more importantly, they offer compelling theoretical and conceptual arguments for much better scalability. In Lal and Reps's paper (henceforth referred to as LR in this paper), the key theoretical advance is the elimination of the exponential cross-product of the local states of the threads, at the expense of introducing multiple non-deterministic symbolic variables to guess the values of the shared global variables at context switches. La Torre, Madhusudan, and Parlato's paper (LMP in this paper) retains the theoretical advantage of LR, but adds "laziness" — instead of non-deterministic guesses, variables can assume only those values that are actually possible during a real concurrent execution — at the expense of needing to recompute the values of local variables at context switches. (More on both translations in Section 2.) Both papers support their arguments with runtimes on a handful of small Boolean programs, e.g., the popular "Windows NT Bluetooth driver"-derived example [26].

Given the very different approaches, with differing trade-offs (local state cross-product vs. symbolic variables vs. recomputation), and limited experiments (small Boolean programs model-checked with Moped [15]), it is hard to draw more general conclusions about what will work well in practice, under differing conditions. In particular, finite-state (or PDA) model-checking of highly abstracted Boolean programs (e.g., [2, 15, 7]) is only one of the major approaches for automatic formal software verification. Another main paradigm is verification-condition (VC) generation[3], with the resulting VC checked by a SAT or SMT solver (e.g., [16, 3, 5, 6, 29, 1, 28]). SAT/SMT solvers behave very differently from the BDDs used in Boolean model checking, so experimental results in the VC-checking paradigm are especially needed.

This paper addresses those needs. We undertake a methodical, empirical evaluation of the three main source-to-source translations for context-bounded analysis of concurrent software, in a VC-checking paradigm. We consider the LMP approach, the LR approach, and a straightforward generalization of the translation given in the original CBA paper [26]. We evaluate how they perform under vastly more experimental conditions than previous work, and also measure scalability versus program length, which was not done before. Some of the results are surprising (e.g., older methods outperforming newer ones), and some are disturbing (e.g., the extent that confounding factors can influence results). Taken altogether, our results highlight the crucial importance of extensive experimental evaluation, provide practical guidance for using context-bounded analysis in a VC-checking paradigm for research or application, and outline pitfalls and questions for further research as these and other translations are developed and improved.

## 2 Context-Bounded Translations



We use a standard model of shared-memory concurrent software. There are $T$ threads, each with its own local variables and program code. The only communication between threads is via a set of global shared variables, which all threads can read or write. (Writes to a global occur atomically: when a thread writes to a global, the new value is immediately visible to all threads.) At all times, exactly one thread is running. At a context switch, the current thread relinquishes control to another thread (determined by the scheduling policy), which proceeds to execute starting from wherever it last gave up control, with whatever values its local variables had at that time and the current values of the global variables. Context switches occur non-deterministically at any point in time. The figure to the right shows the concurrent execution of three threads. The program code executed by each thread is depicted by the sequence of dark, vertical arrows. Each of those arrows represents one "context" — an uninterrupted period when one thread runs its code. The dashed arrows represent context switches, which occur non-deterministically, and transfer control to a different thread.

[3] A VC is a logical formula whose validity implies partial correctness of the code for which it was generated. VCs are typically constructed via weakest precondition or symbolic execution.

When a context switch occurs, which thread runs next is determined by a scheduling policy. Two policies are common in CBA: round-robin and arbitrary. In round-robin scheduling, the context switch is always to the numerically next thread, modulo $T$. Hence, execution proceeds in a series of rounds, during which each thread gets a chance to execute once at its turn. In the preceding figure, the schedule is round-robin, and the light (cyan) solid curved lines demarcate the three rounds. In arbitrary scheduling, a context switch can jump to any thread, non-deterministically chosen. Obviously, the schedules permitted by round-robin with $K$ rounds is a subset of the arbitrary schedules with $K \cdot T$ contexts. Conversely, the schedules permitted by round-robin with $K$ rounds is a superset of the arbitrary schedules with $K$ contexts, since a thread can execute zero instructions before another context switch occurs, so each round of round-robin can simulate one context of an arbitrary-scheduled thread. Between these two bounds, neither policy dominates the other.

We now survey the three main source-to-source translations for CBA under this model. For space reasons, we give only some brief intuition for each.

### 2.1 Explicit Program Counter (EPC)

We dub our first translation "Explicit Program Counter" (EPC). This is the obvious, brute-force approach and is a straightforward generalization of the original CBA paper [26] (where they restricted themselves to two context switches in order to permit an efficient implementation via the procedure call mechanism).

For the EPC translation, the state of the sequential program includes all of the local variables, including the program counters, of all of the threads. The code of the sequential program consists of the code of all of the threads combined into a single program. However, at each possible location for a context switch (i.e., between every adjacent pair of accesses to global variables), we insert code that can non-deterministically decide to simulate a context switch. The context switch code consists of choosing the next thread to run (based on the scheduling policy), and then jumping to the correct location in that thread based on its stored program counter. The sequential program starts executing at the beginning of $Thread_1$ for round-robin or with a non-deterministic jump to the beginning of an arbitrary thread for arbitrary scheduling. An auxiliary variable $k$ counts how many contexts have run. The sequential program terminates when $k$ reaches the context bound $K$, or when all threads have executed all of their code.

This translation is simple and has linear static and dynamic code size versus the concurrent program. However, at each point during execution, the program state consists of the cross-product of all local variables and the global variables, potentially producing a complexity blow-up.

### 2.2 Lal-Reps CAV 2008 (LR)

The LR translation eliminates the EPC complexity blow-up, at the expense of introducing symbolic prophesy variables to guess the values of results that are not yet known. The basic construction is to execute each thread one-by-one in its entirety, in sequence, i.e., all of $Thread_1$, then all of $Thread_2$, then all of $Thread_3$, etc. Accordingly, the static and dynamic code size are unchanged from the original program. Furthermore, since each thread executes in its entirety, without interruption from the others, there is no need to keep the local state of a thread after it is done, thereby eliminating the blow-up of the local state cross-product.

The construction in the preceding paragraph would produce wrong results, since it ignores the fact that in the concurrent program, a context switch could occur at any point and change the value of global variables. Worse, because we are executing the threads sequentially one after another, the results computed by the other threads might not be known until much later in the sequential execution!

The solution is to create $K$ copies of the global variables, where $K$ is the bound on the number of round-robin rounds. The $i$th copy contains the values of the global variables during the $i$th scheduling round. Since we will not know what values these variables will contain until the program completes, we initialize all $K$ copies with non-deterministic symbolic values. An auxiliary variable $k$ in each thread keeps track of which round is executing; all accesses to globals are indexed through $k$. A context switch during the execution of a thread, therefore, consists simply of increasing $k$, which results in a switch to the correct set of global variables for that round. Hence, at each possible location for a context switch, we insert code that non-deterministically increases $k$. At the end of the program, we use `assume` statements to enforce that the results in the copy of the globals at the end of round $i$ are equal to the non-deterministic symbolic values we used to initialize the copy of the globals for the start of round $i+1$. In effect, the translation is computing symbolic summaries for each round and stitching them together via `assume` statements at the end.

Note that this construction is intrinsically round-robin. Because the threads are executed in order $1,\ldots,T$, where $T$ is the number of threads, the values of the globals in each round pass automatically from any $Thread_t$ to $Thread_{t+1}$. The symbolic values and stitching are required only between rounds.

## 2.3 La Torre-Madhusudan-Parlato CAV 2009 (LMP)

Because LR is constructing symbolic summaries from unconstrained symbolic values, it might explore expensive, infeasible regions of the state space, only to eliminate them in the end using the `assume` statements. LMP avoids this problem by introducing "laziness" — instead of non-deterministic guesses, variables can assume only those values that are actually possible during a real concurrent execution. (For comparison purposes, the LMP paper also introduces an eager translation similar to the LR translation. In this paper, LMP refers to their lazy translation.)

LMP with a bound of $K$ contexts starts with a non-deterministic schedule $t_1,\ldots,t_K$, where $t_i \in [1,\ldots,T]$ contains the identity of the thread to execute during the $i$th context. As in LR, there are $K$ copies of the global variables, but these are assigned only as their values are computed. Like EPC, execution in the LMP translation follows the same order as the concurrent execution — a context switch is an actual jump to the next thread in the schedule. Unlike EPC, however, the local state of a thread is completely discarded when context-switching away from it, thereby eliminating the local-state cross-product blow-up. How can a thread resume where it left off after a context switch? The solution is to recompute the thread's local state! In other words, the context-switch code is considerably more complicated, as it will re-execute a thread from the beginning each time a context switch to it occurs, but during re-execution, the values of the global variables at earlier context switches are already known. In the presence of non-determinism, correctness of this construction is not obvious, since the recomputed local state might be different from the one that occurred when last executing in this thread — a subtle correctness argument is needed to show that no additional behaviors are introduced. The resulting translation has the best attributes of both LR (no local-state blow-up) and EPC (no exploration of infeasible states), but at a cost: a blow-up in the length of the dynamic code paths that must be executed/analyzed. The LMP paper provides some experimental results showing the translation greatly outperforming their version of LR, but only in the Boolean model checking paradigm, and on only two small examples: an artificial example specifically constructed to illustrate the benefit of laziness,[4] and the aforementioned "Bluetooth driver" benchmark. How the methods compare as program size grows, and under the VC-checking paradigm, are open questions.

## 3 Experimental Methodology and Results

Obviously, the ultimate test of these translations is their performance in the wild on real software. In our experience applying CBA to real, industrial code [21], a crucial factor was scalability to larger code sizes, so the paramount dimension for our experiments is scalability with respect to code length. For scalability benchmarking, however, real code has a fatal flaw: the code length is not scalable.

Accordingly, we crafted a microbenchmark that is scalable along the three key problem parameters: the number of threads $T$, the context bound $K$, and the length of the program code in each thread $L$. To avoid confounding factors, we distilled our benchmark to only the essentials:

$$\texttt{int x} = 0;$$

$$\underbrace{\begin{array}{c|c|c} L\left\{\begin{array}{l} \texttt{x}++; \\ \vdots \\ \texttt{x}++; \end{array}\right. & & \begin{array}{c} \texttt{x}++; \\ \vdots \\ \texttt{x}++; \end{array} \\ & \cdots\cdots & \\ \texttt{assert } 0 < \texttt{x}; & & \texttt{assert } 0 < \texttt{x}; \end{array}}_{T}$$

The benchmark has a shared global variable `x`, which is initialized to 0. Then, $T$ threads are spawned. Each thread consists of $L$ increments of `x` followed by an assertion that checks if `x` is greater than 0. Despite its simplicity, this microbenchmark still has the key elements of the concurrent software model: local state (the program counters), shared global state `x`, and long data-dependency chains that grow with code size and must be inspected in order to prove the assertions, capturing an essential aspect of program scaling. The microbenchmark does not have procedures or loops — in the VC-checking

[4] In their "permutation" example, an interlock serializes two threads, the first thread zeroes out 16 bits, and the second thread computes all permutations of the 16 bits. With laziness, the bits are all zero, so the permutation is vacuous; without laziness, the second thread explodes.

paradigm, these are handled via invariants. Note that this particular microbenchmark could easily be solved via other means, since it has a small finite state space. That is not the point. This is a "test tube" experiment, to identify and eliminate confounding factors while measuring scalability. If a translation scales poorly on this benchmark, it will not fare well on real code.

For each of the three translations, we encoded in BoogiePL [12] multiple instances of this benchmark by varying its $T$, $K$, and $L$ parameters. BoogiePL is the input language of the BOOGIE verifier [3], which generates a verification-condition (VC) from the input program. The VC generation in BOOGIE is performed using a variation of the standard *weakest precondition* transformer [14]. This translation is linear in the input code size; combined with the linear static code-size expansion of the three translations, all three translations produce linear-size VCs, albeit with differing complexity (LMP the most complex; LR the simplest). We checked the VCs generated from our benchmarks using the Z3 SMT solver [11], except where indicated otherwise. The experiments were performed on four identical machines (Intel Xeon 5160 at 3GHz with 2GB RAM), running Z3 continuously for weeks. We report the solver's running times, and the time out is set to 2 hours.

As an additional sanity check, we also performed experiments on the Bluetooth driver example [26]. The original benchmark has two threads: an "adder" and a "stopper". We artificially scale $T$ by adding adder threads. We artificially scale code length by repeating (i.e. copy-pasting) the body of the adder threads; the parameter $L$ denotes the number of such repetitions.

Figs. 1–10 present our main results. Details and interpretation of each experiment are in the accompanying captions. Full experimental data and complete results graphs are available at `http://people.cs.ubc.ca/~naghmehg/spin2010-results`.

## 4 Conclusions

The primary, practical take-away conclusion of this paper is that LMP is not competitive in the VC-checking paradigm. This radical reversal of recently published results highlights the difference of Boolean model checking versus VC-checking. Both are important, and our experiments show that they need different translations.

Our experiments suggest that LR is likely the best translation for the VC-checking paradigm, under most experimental conditions, and with current state-of-the-art SMT solvers. LR is also particularly easy to implement, making it our recommendation for VC-checking-based research prototypes for CBA. Surprisingly, the superiority of LR to LMP holds even with arbitrary schedules, when LR is at an artificial disadvantage. Given that LR with $K$ round-robin *rounds* outperforms LR with $K$ arbitrary contexts, there is no efficiency argument for using arbitrary schedules.

Another surprise was how well the brute-force EPC translation did, beating LMP in almost all experiments and LR in a few. In the VC-checking paradigm, the power of SMT and SAT solvers to quickly prune irrelevant parts of the search space and to propagate information in any direction perhaps lessens the benefit of translation insights like laziness or state space reductions, benefits that might make a big difference for

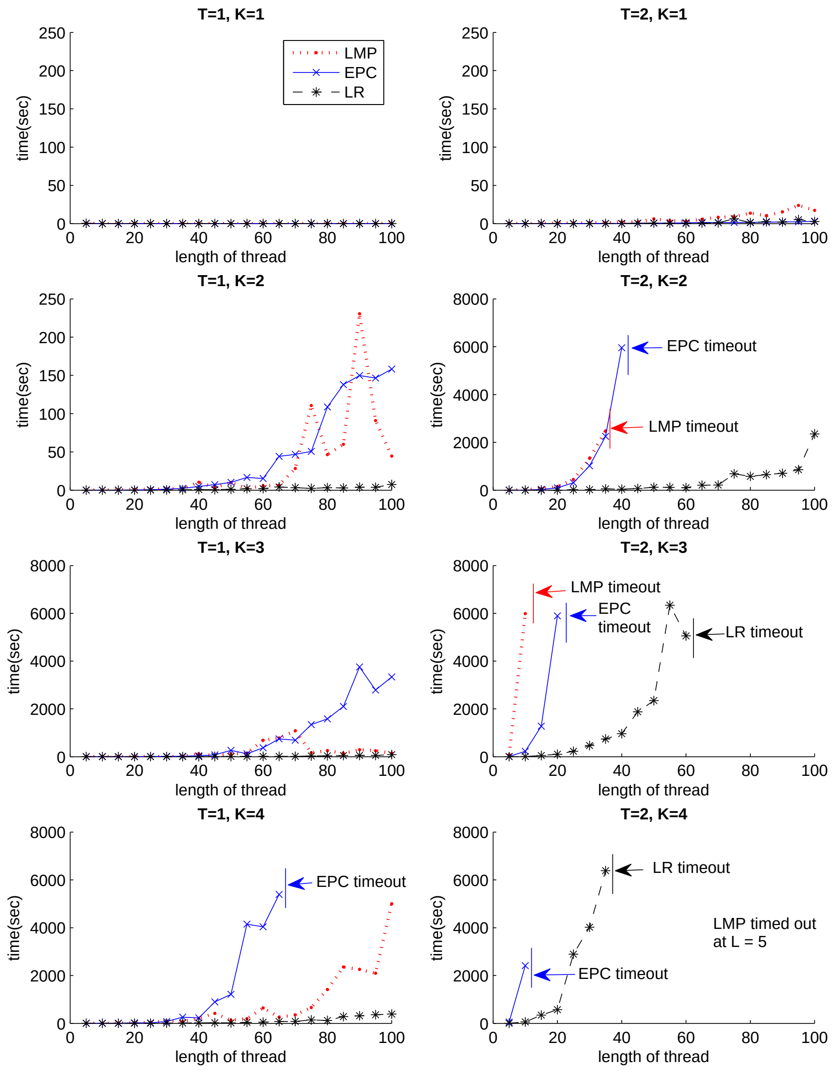


**Fig. 1. Baseline Comparison.** This and Fig. 2 show a representative sample of our baseline comparison (round-robin scheduling, Z3 as SMT solver), with the number of threads $T \in [1, \ldots, 4]$ (going across the pages), the context bound $K \in [1, \ldots, 4]$ (going down the page), and the length of each thread's program code $L$ going from $5, 10, \ldots$ up to 100. (Caption continues on next page.)

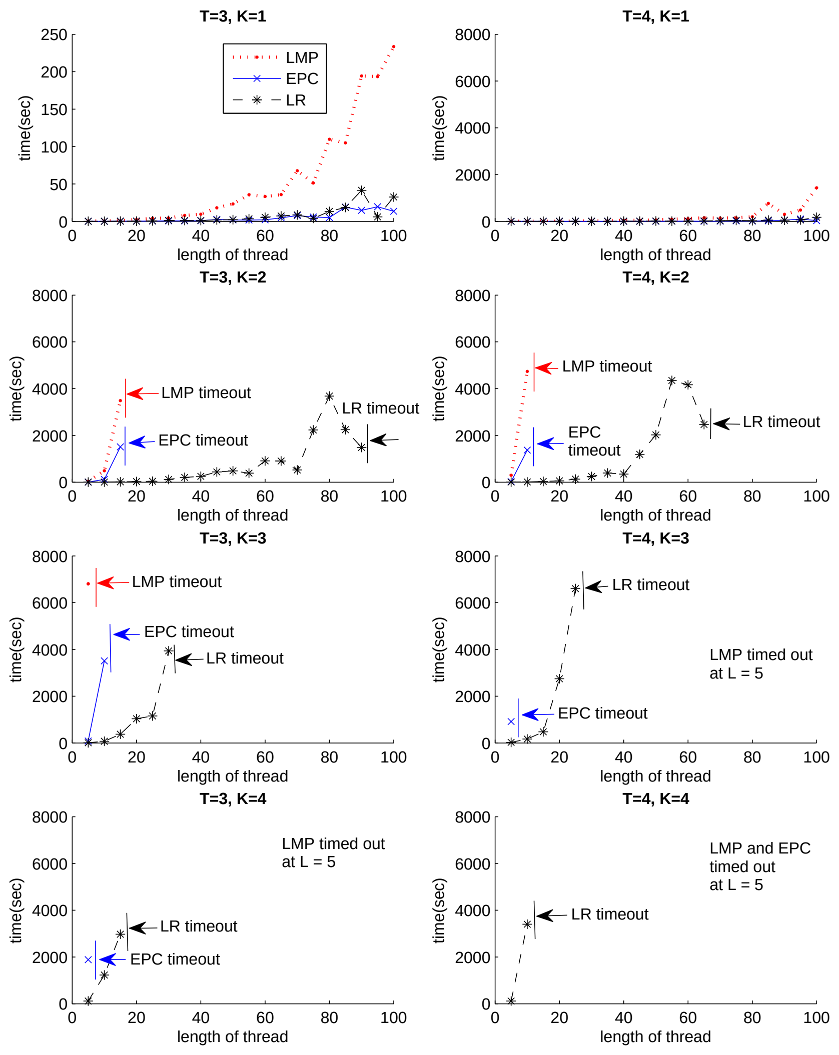
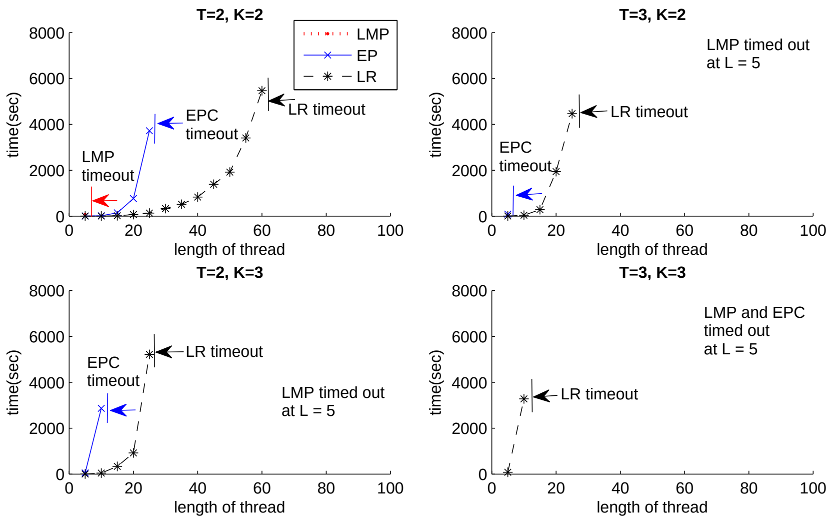


**Fig. 2. Baseline Comparison (cont'd).** Surprisingly, LR beats LMP, contradicting the results in [31], and furthermore, even EPC beats LMP. The different experimental conditions in [31] likely explain much of this reversal: (1) While EPC and LMP do not suffer particularly under round-robin scheduling, LR has an intrinsic advantage. (2) In the VC-checking paradigm, the benefit of laziness is unclear, since the solver can propagate information in any direction. (3) On the other hand, LMP's longer dynamic code paths generate more complex VCs.

**Default Relevancy=2 Results**



**Relevancy=1 Results**

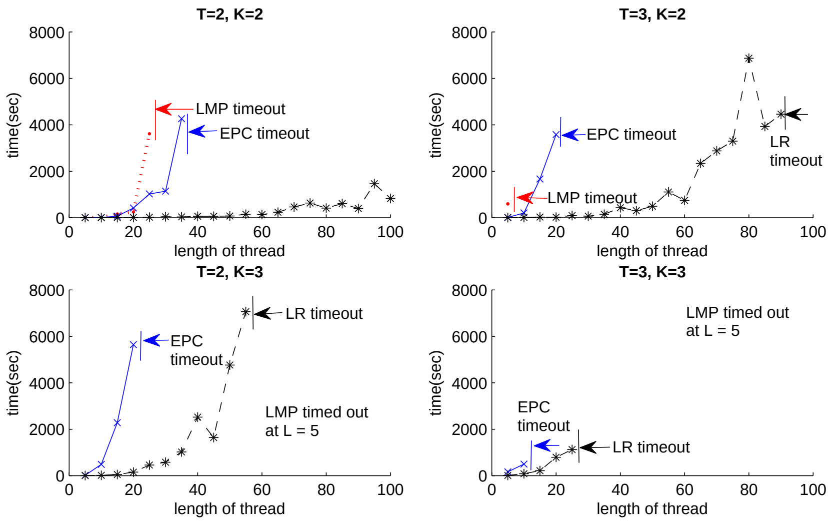


**Fig. 3. Sensitivity to Z3 Parameter Tuning.** For the baseline results, we had performed informal tuning of Z3 parameters. The only setting that we changed from the defaults was "relevancy propagation heuristic" that affects quantifier instantiation and assertion of atoms in the solver. We set it to 0 for best performance. Given the surprising results, though, it was imperative to run with different relevancy settings, to ensure we had not inadvertently biased our experiments. For space reasons, we show only four graphs, for $T, K \in [2,3]$, the smallest non-degenerate ($T$ or $K = 1$) cases. The results are qualitatively the same: LR wins; LMP loses.

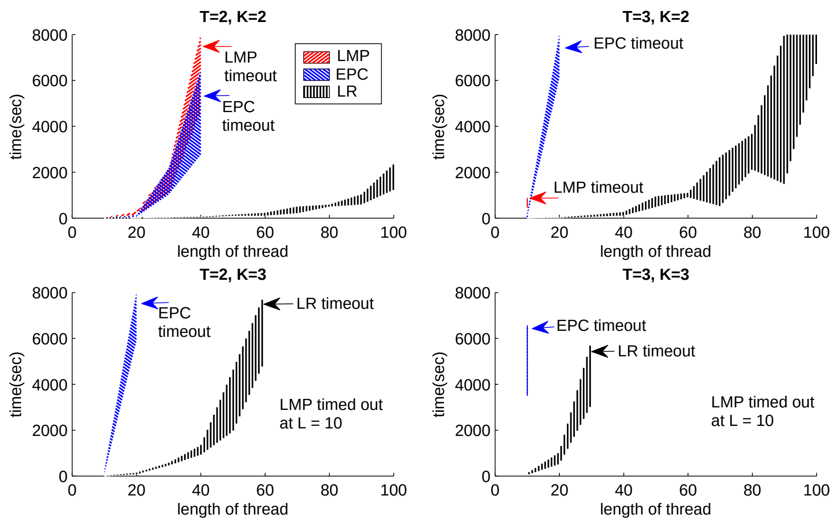


**Fig. 4. Sensitivity to Random Perturbation.** SMT solvers, like SAT solvers, are notoriously temperamental. To assess the robustness of our results, we repeated all of the baseline experiments three additional times, with different random number seeds. These graphs show the range of performance across different seeds. Although times vary considerably, there is almost no overlap between the translations. We can conclude that the baseline results are robust.

a Boolean model checker. However, SMT/SAT solver performance is quirky, highly dependent on heuristics, and hard to predict.

Fortunately, our experiments suggest an avenue for predicting performance. SMT runtimes tend to grow exponentially with VC length, but the different translations and parameters give different exponential curves. This suggests that the practical scalability of a translation scheme might be predicted by combining the size of the queries generated with an empirical or theoretical measure of the complexity of solving that class of query. As a crude heuristic, pick translations that keep VCs short and simple.

Disturbingly, our experiments also highlight experimental pitfalls and confounding factors. For example, using an older, slower SMT solver changes the relative ranking of the translations, implying that practical performance depends more on the interaction of a translation with a given SMT solver than on theoretical properties of the translation in isolation. If another revolution in SMT solving occurred, these translations would need to be re-evaluated. Similarly, the relative ranking of translations was sometimes different in small, degenerate corners of the parameter space. Performing only a few experiments, as if often done in this research area, could easily give misleading results. Our results can be used as a cautionary map, clarifying the landscape (e.g., VC-checking vs Boolean model checking) and highlighting pitfalls (e.g., use the fastest solver, conduct thorough experiments). These are not happy results, but they are critically important, since they challenge the assumptions and methodologies underpinning further research.

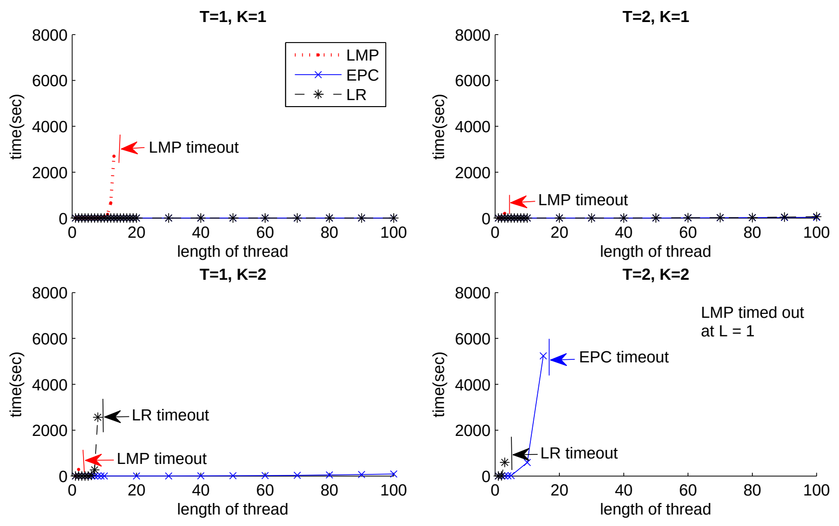


**Fig. 5. Sensitivity to SMT Solver.** A key question is whether our results are specific to Z3. We undertook to re-run our experiments with all publicly available SMT solvers that can handle the needed logic. Because BOOGIE generates VCs with quantifiers, from the 2009 SMT Competition, the only other suitable solver is CVC3 [4]. Performance was much worse, so we have results only for the smallest values of $T$ and $K$. Disturbingly, the performance order changes: EPC is the clear winner in these experiments. The choice of SMT solver determines which translation performs best! Experimental evaluation, therefore, should always include multiple solvers, or at least the fastest one available. Using a slower solver can produce misleading results.

Finally, it is thought-provoking to consider how poorly all three translations scale on our microbenchmark. As noted earlier, the microbenchmark is easily solvable by other means, e.g., even the largest parameter configuration we considered ($T = 4$, $K = 4$, $L = 100$) yields a state space of roughly only 1.6 billion states (100 program counter locations for each of 4 threads times a 4-valued context counter times a 4-valued pointer that indicates the active thread), within the reach of even explicit-state model checking. Yet, none of the translations, under any experimental setup, came anywhere close to those parameter values. There is an odd disparity between the ample empirical proof of the practical success of CBA (including our own work, finding real bugs in real industrial software, using the same LR-BOOGIE-Z3 tool chain, albeit augmented with some abstraction techniques [21]) and its poor scalability here. The explanation, we believe, is that the power of CBA with VC-checking comes from the ability to reason about and soundly abstract software features like large memories, heaps, and recursion, but not from the core exploration of interleavings, which is where classical model checking excels. We speculate there may be promising hybrids between the two approaches, or that CBA will dominate bug-finding in low-level software implementations with explicit-state model checking dominating protocol-level software verification, much as bounded model checking and explicit-state model checking complement each other in hardware verification.

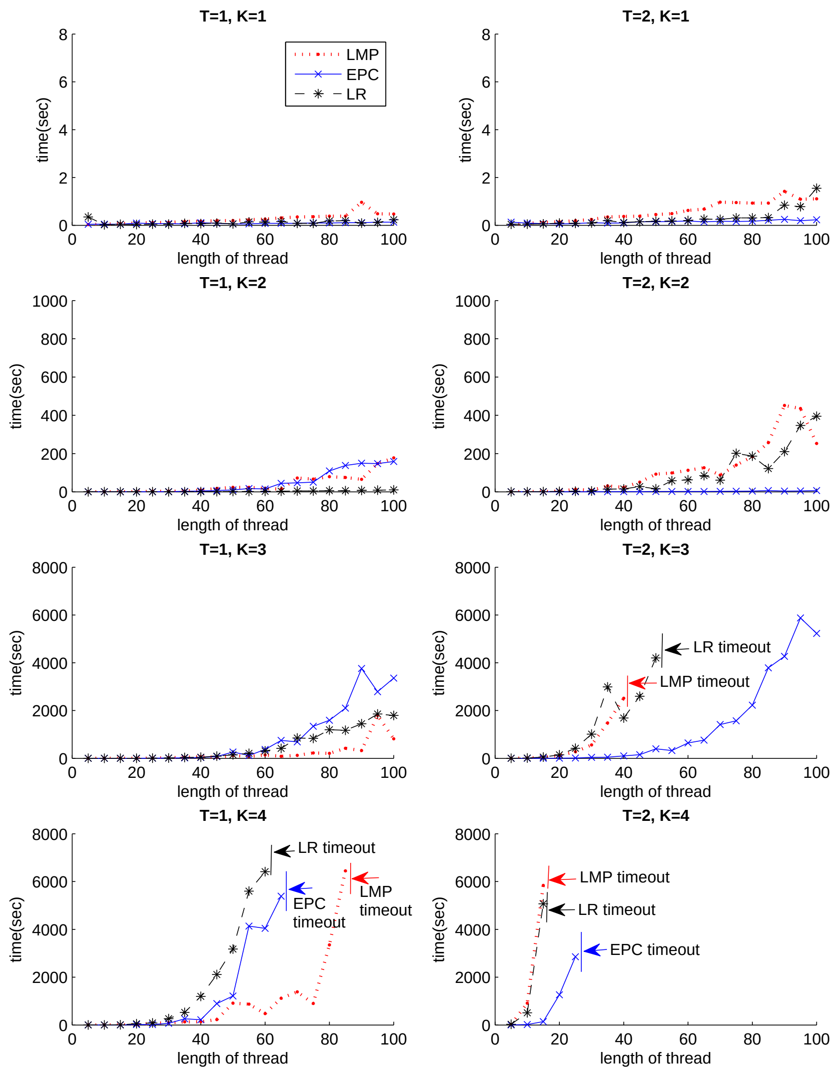


**Fig. 6. Results with Arbitrary Scheduling Policy.** As noted earlier, our baseline results are with round-robin scheduling, and with all translations analyzing the same number of contexts, to be fair. This and Fig. 7 show the results using an arbitrary scheduling policy with $K$ total contexts, also with all translations analyzing the same number of contexts. Now, there is no clear winner. LMP clearly does best in the degenerate $T = 1$ cases. EPC wins in several of the mid-sized configurations. And LR is the last to timeout as $T$ and $K$ grow larger. (Caption continues on next page.)

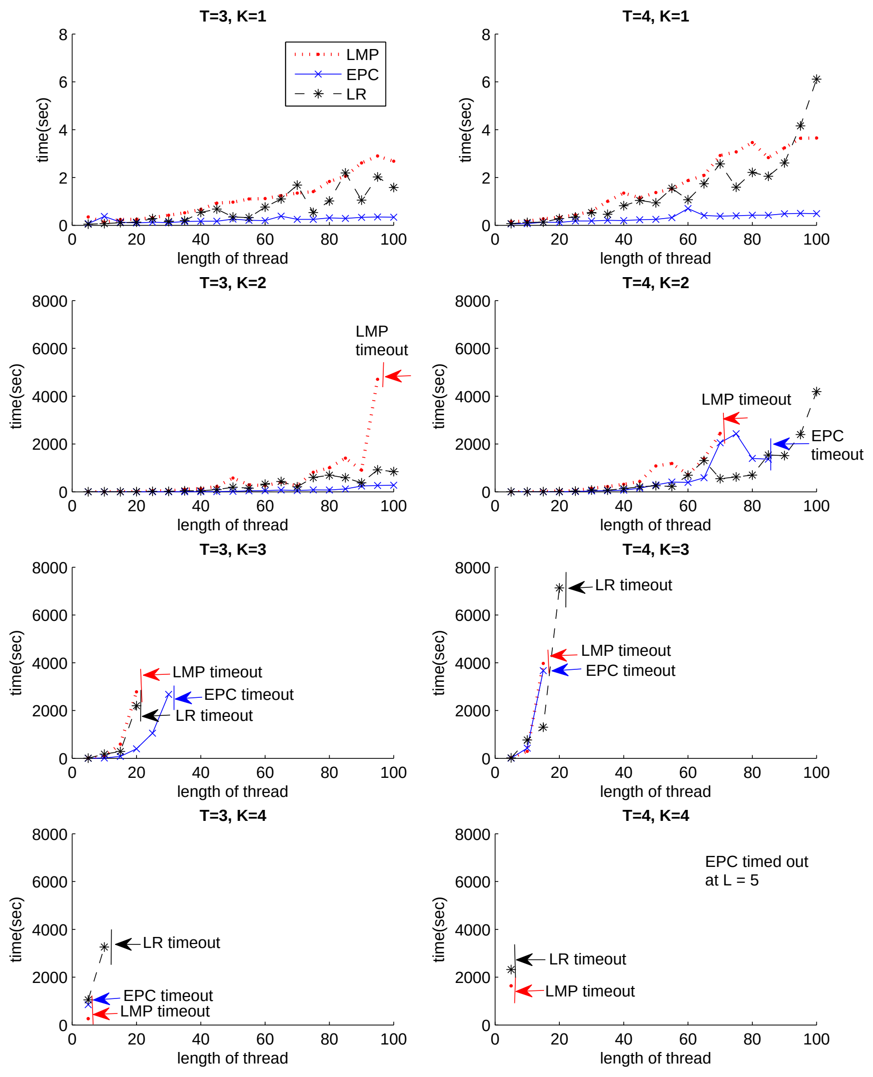


**Fig. 7. Results with Arbitrary Scheduling Policy (cont'd).** It would be easy to draw incorrect conclusions about algorithmic superiority if only small parts of the experimental space are explored, as is often the case. Interestingly, LR with $K$ arbitrary contexts performs worse than LR with $K$ round-robin *rounds*! The explanation is that LR is intrinsically round-robin, so the arbitrary-schedule $K$-context translation is essentially doing a round-robin $K$-round translation, plus extra work to ensure that only one context executes in each round. We present here results for arbitrary-schedule LR only for fairness; it is pointless in practice.

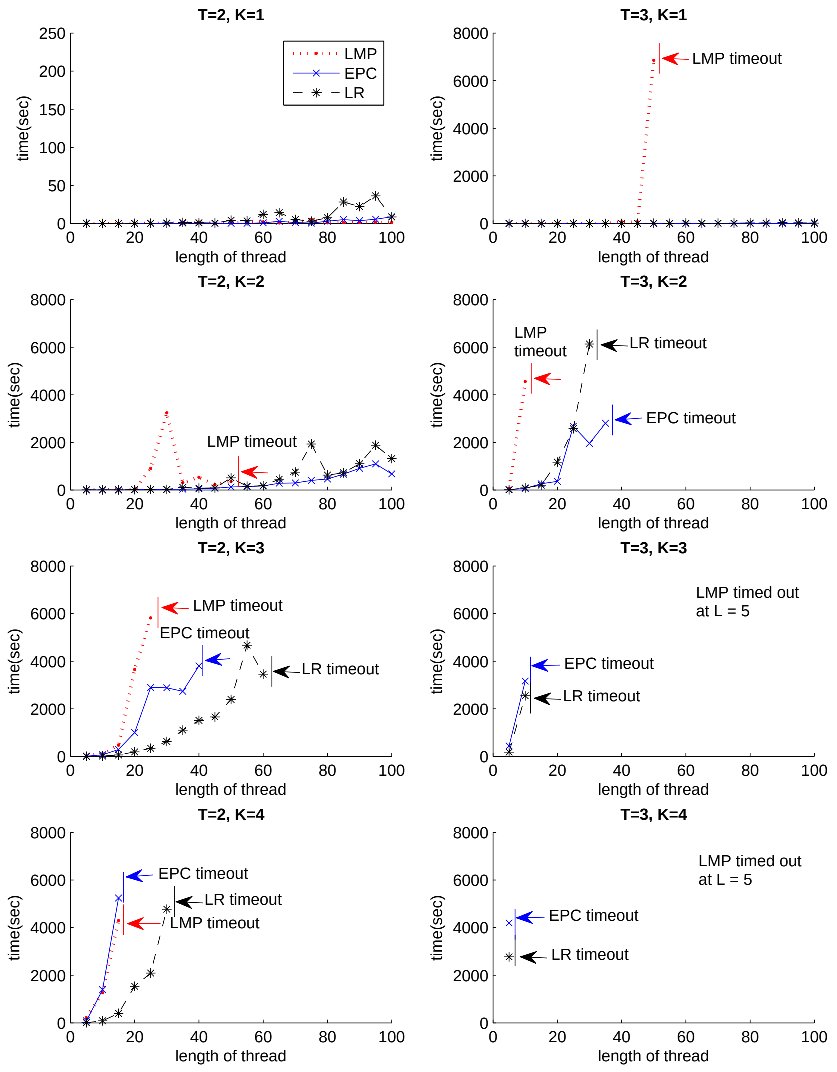


**Fig. 8. Results for Bluetooth Example.** As a check of our results, we also evaluated the three translations on the popular "Bluetooth driver" example. The original example had two threads: an "adder" and a "stopper". To scale the number of threads, we created additional adder threads. $T$ indicates the total number of threads. ($T$ starts at 2 in these graphs because we need at least 1 adder and 1 stopper.) To artificially scale program length, we duplicated the body of the adder threads. $L$ indicates the number of copies of the original code in each thread. The runs are under the baseline conditions: round-robin scheduling with Z3 as the SMT solver. (Caption continues on next page.)

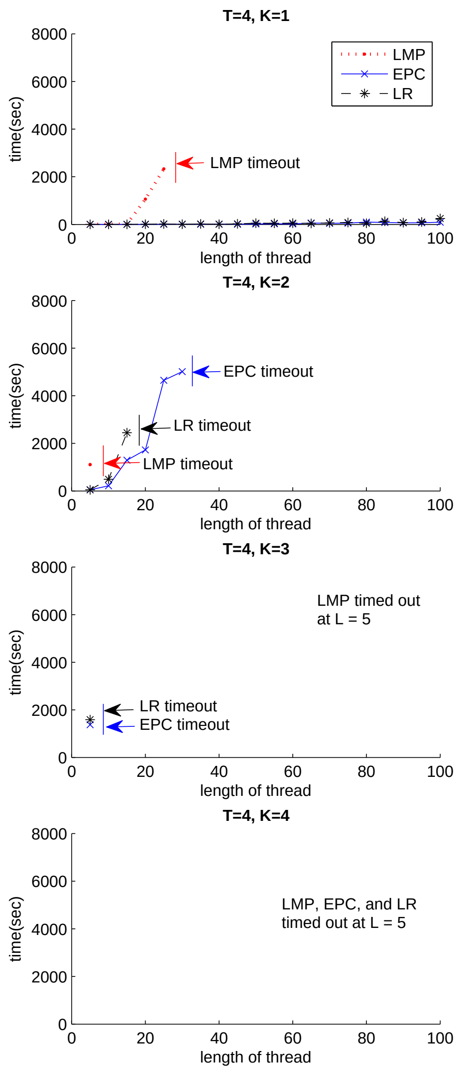


**Fig. 9. Results for Bluetooth Example (cont'd).** The results are similar: LMP clearly loses, but as with the results for arbitrary schedules (Figs. 6 and 7), EPC wins in several configurations.

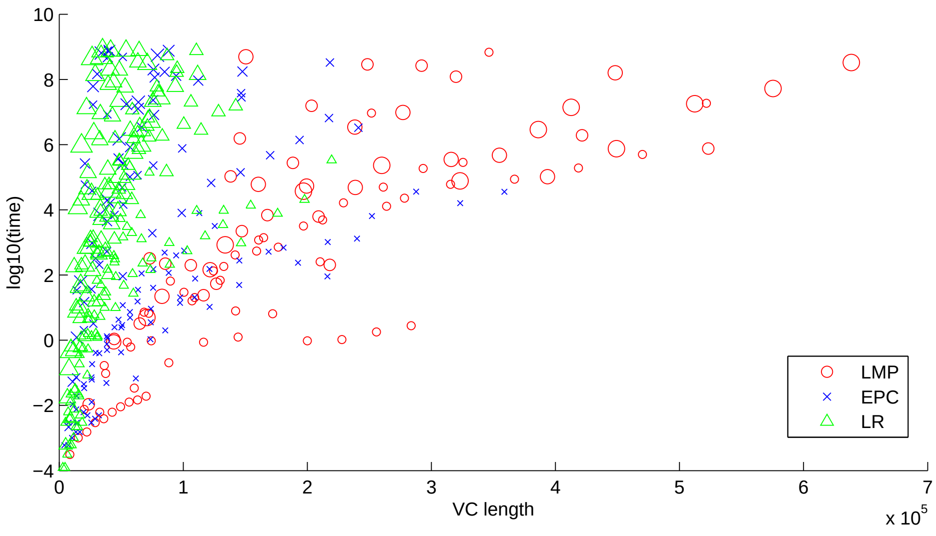


**Fig. 10. Runtime vs. VC Length.** One hypothesis is that VC length (and therefore static code size and complexity produced by the translation) is crucial for performance in the VC-checking paradigm. To test this hypothesis, we collected all the baseline runs, for both the microbenchmark and the Bluetooth queries, and plotted runtimes versus the size of the VC, as reported in "words" by the Unix utility `wc` to normalize for different variable naming conventions. In this plot, each translation is shown with a different symbol (and color). The size of the symbol is proportional to the context bound $K$. The data appear to cluster into multiple straight lines, indicating exponential growth, but with different bases for the different translations and problem instances. For example, LMP compensates somewhat for vastly bigger VCs with a lower-based exponent. There appears to be some correlation between $K$ and the exponential growth rate, although this is imperfect. We observed even less correlation with $T$. We conjecture that some complexity measure, such as number of program paths, could predict which exponential curve a given problem family and translation approach would exhibit. (The graph is much easier to interpret in color.)

## References

1. D. Babić and A. J. Hu. Calysto: Scalable and precise extended static checking. In *International Conference on Software Engineering (ICSE)*, pages 211–220, 2008.
2. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 203–213, 2001.
3. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.
4. C. Barrett and C. Tinelli. CVC3. In *International Conference on Computer Aided Verification (CAV)*, pages 298–302, 2007.
5. S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamarić. A reachability predicate for analyzing low-level software. In *TACAS*, pages 19–33, 2007.
6. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, pages 168–176, 2004.
7. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI–C programs using SAT. *Formal Methods in System Design (FMSD)*, 25:105–127, 2004.

8. E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
9. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, LNCS 131, pages 52–71, 1981.
10. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *International Conference on Software Engineering (ICSE)*, pages 439–448, 2000.
11. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.
12. R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.
13. C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. *Software — Practice and Experience*, 29(7):577–603, 1999.
14. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.
15. J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In *International Conference on Computer Aided Verification (CAV)*, pages 324–336, 2001.
16. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245, 2002.
17. M. K. Ganai and A. Gupta. Efficient modeling of concurrent systems in BMC. In *Intl. SPIN Workshop on Model Checking Software*, pages 114–133, 2008.
18. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
19. G. J. Holzmann and M. H. Smith. Software model checking. In *Formal Methods for Protocol Engineering and Distributed Systems (FORTE)*, volume 156 of *IFIP Conference Proceedings*, pages 481–497. Kluwer, 1999.
20. V. Kahlon, S. Sankaranarayanan, and A. Gupta. Semantic reduction of thread interleavings in concurrent programs. In *TACAS*, pages 124–138, 2009.
21. S. K. Lahiri, S. Qadeer, and Z. Rakamarić. Static and precise detection of concurrency errors in systems code using SMT solvers. In *CAV*, pages 509–524, 2009.
22. A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. In *Conf. on Computer Aided Verification (CAV)*, pages 37–51, 2008.
23. A. Lal, T. Touili, N. Kidd, and T. W. Reps. Interprocedural analysis of concurrent programs under a context bound. In *TACAS*, pages 282–298, 2008.
24. M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, pages 446–455, 2007.
25. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107, 2005.
26. S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 14–24, 2004.
27. I. Rabinovitz and O. Grumberg. Bounded model checking of concurrent programs. In *Conf. on Computer-Aided Verification (CAV)*, pages 82–97, 2005.
28. Z. Rakamarić and A. J. Hu. A scalable memory model for low-level code. In *Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 290–304, 2009.
29. W. Schulte, S. Xia, J. Smans, and F. Piessens. A glimpse of a verifying C compiler (extended abstract). In *C/C++ Verification Workshop (CCV)*, 2007.
30. D. Suwimonteerabuth, J. Esparza, and S. Schwoon. Symbolic context-bounded analysis of multithreaded Java programs. In *SPIN*, pages 270–287, 2008.
31. S. L. Torre, P. Madhusudan, and G. Parlato. Reducing context-bounded concurrent reachability to sequential reachability. In *CAV*, pages 477–492, 2009.
32. W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.