SMT-based Symbolic Model Checking for Multi-Threaded Programs

Zijiang Yang
Department of CS
Western Michigan University

Karem Sakallah
Department of EECS
University of Michigan

1 Problem Statement

Today, hyper-threading and multi-core hardware have become ubiquitous, putting us at a fundamental turning point in software development. In order for software applications to benefit from the continued exponential throughput advances in multi-core processors, applications will need to be multi-threaded software programs that are highly reliable. However, multi-threaded programs are notoriously difficult to debug due to their non-deterministic behaviors. To completely verify program behavior for a given test input, *all execution traces* permissible under that input must be examined. This is not possible in current testing environments since users have no control over the scheduling of threads. Even if it were possible to control thread scheduling, it would still be infeasible to explicitly test all interleavings: the number of possible interleavings of a multi-threaded program with n threads each executing at most k steps can be as large as $(nk)!/(k!)^n \geq (n!)^k$, a dependence that is exponential in both n and k.

Despite recent advances in automatic software verification (based on various flavors of model checking), most software developers still rely on manual testing and interactive debugging. This is partly explained by the lack of scalability of automated verification methods even with the most advanced model checking techniques, and partly by the reluctance of the developers to change methodologies. *In this position paper, we advocate the integration of a powerful symbolic reasoning engine within an interactive debugging environment.* Restricted to a given test input, the symbolic reasoning engine can implicitly verify all thread interleavings without imposing excessive computational overheads. This addresses the scalability issue that has hindered the application of unrestricted model checking to large software applications. Additionally, retaining the interactive flavor of debugging lessens the burden on users by not requiring them to switch code development and debugging methodologies.

We envision an "enhanced" symbolic debugger that does intensive behind-the-scenes analysis to assist the user in quickly detecting bugs or proving the absence of them. This may seem like an intractable task considering the fact that the number of interleavings is exponential. However, under the right circumstances this is quite feasible. First, we analyze program behavior under user-specified test inputs. Since only a small fraction of program statements are executed on any given run, the symbolic analysis of only those statements is tractable and within the capability of current symbolic constraint solvers. Second, we combine dynamic execution with symbolic analysis that can implicitly verify inter-thread interactions without explicitly enumerating all thread interleavings.

$$
\begin{array}{lcll}
\delta_{t,s1}[i] & \equiv & T[i]=t \wedge pc_t[i]=loc_1 & \rightarrow \; pc_t[i+1]=loc_2 \wedge v[i+1]=e[i] \wedge (VS_t \backslash v)[i+1]=(VS_t \backslash v)[i] \\
\delta_{t,s2}[i] & \equiv & T[i]=t \wedge pc_t[i]=loc_1 \wedge c[i] & \rightarrow \; pc_t[i+1]=loc_2 \wedge VS_t[i+1]=VS_t[i] \\
\tau_{t,idle}[i] & \equiv & T\,[i] \neq t & \rightarrow \; pc_t\,[i+1]=pc_t\,[i] \wedge V_t\,[i+1]=V_t\,[i] \\
\tau_{t,done}[i] & \equiv & pc_t\,[i]=\perp & \rightarrow \; T\,[i] \neq t \\
\end{array}
$$

$$\rho\,[i] \;\equiv\; T\,[i]=t \wedge L_t\,[i]=l \;\rightarrow\; y\,[i] \leqslant a$$

Figure 1: Examples for program transition constraint δ, thread control constraint τ, and property constraint ρ.

2 Symbolic encoding of multi-threaded program testing

The approach we are proposing is to capture thread interleavings implicitly as a set of constraints, belonging to the family of quantifier-free first-order formulas, for which highly efficient and scalable SMT solvers [5, 4] were recently developed.

Let V_{global} and V_t be the set of global variables and the local variables in thread t, respectively. The set of variables visible to t is $VS_t = V_{global} \cup V_t$. In addition to program variables, we introduce a program counter pc_t for each thread. To model nondeterminism in the scheduler, we add a variable T whose domain is the set of thread indices. A transition in thread t is executed only when $T = t$. At every transition step we add fresh copies of the set of variables. Let $v[i]$ denote the copy of v at the i-th step. In the following we list three types of constraints:

Program transition constraints δ: Fig. 1 shows some example constraints for program statements at the i-th step, where $\delta_{t,s1}$ is for an assignment $s1 : v := e$ from control locations loc_1 to loc_2 in thread t, $\delta_{t,s2}$ is for a branch statement $s2 : if(c)$ from loc_1 to loc_2 if c holds. Overall, the constraint that consider all possible interleavings up to k steps for a multi-threaded program with N threads is defined as $\delta^k \equiv \prod_{i=1}^{k} \prod_{t=1}^{N} \prod_{s} \delta_{t,s}[i]$.

Thread control constraints τ: $\tau_{t,idle}$ in Fig. 1 insures that the local state of a thread (the values of its local variables) remains unchanged when the thread is not executing, and $\tau_{t,done}$ insures that the thread cannot be selected for execution after it has terminated. The constraint that consider thread control constraints up to k steps for a multi-threaded program with N threads is defined as $\tau^k \equiv \prod_{i=1}^{k} \prod_{t=1}^{N} (\tau_{t,idle}[i] \wedge \tau_{t,done}[i])$. Note that additional thread control constraints can be easily included to model particular scheduling policies.

Property constraints ρ: ρ specifies the correctness conditions that we would like to check for validity under all possible executions. In Fig. 1 $\rho[i]$ encodes `assert(y<=a)` at line l in thread t. The predicate $\rho[i]$ evaluates to true at step i if (1) thread t is not executing line l at step i, or (2) thread t is executing line l at step i *and* the assertion $y \leq a$ is true. The complete property constraint (up to k steps) can be specified as $\rho^k \equiv \prod_{i=1}^{k} \rho[i]$.

3 Trace-Driven Verification Flow

The counterexample guided abstraction refinement (CEGAR) framework [6, 2] which has been found to be quite effective in a variety of verification scenarios of complex hardware and software systems. In our research we propose a trace-driven verification flow for multi-threaded software programs, as illustrated in Fig. 2:

1. Run the program under a given user input to obtain an initial execution trace π.

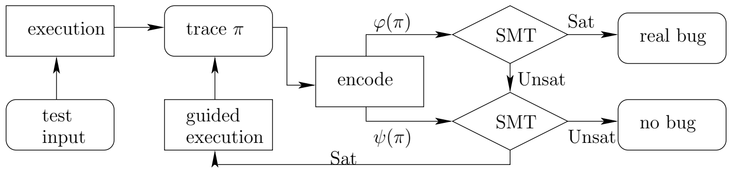

Figure 2: Trace-driven verification flow.

2. Using a encoding along the lines illustrated in Section 2, construct a symbolic SMT formula $\varphi(\pi)$.
3. Using a state-of-the-art SMT solver, check the satisfiability of $\varphi(\pi)$.
   - If $\varphi(\pi)$ is found to be satisfiable, a real bug is found. Based on the solution to $\varphi(\pi)$ we can report to the user the specific scheduling that exposes the bug.
   - If $\varphi(\pi)$ is found to be unsatisfiable, we relax $\varphi(\pi)$ by removing the trace enforcement constraints to obtain $\psi(\pi)$. This allows us to examine *sibling* traces, i.e., traces that conform to the same input but cover different statements.
     - If $\psi(\pi)$ is found to be unsatisfiable, we can conclude that the property holds under all possible thread interleavings under the given test input.
     - If $\psi(\pi)$ is found to be satisfiable, the SMT solver returns a counter-example, which is used to guide new executions that are guaranteed to touch new statements that have not appeared in previous executions.

4 Optimizations

There are many optimizations in this flow to streamline the approach. Following are two optimization directions:

- *Encoding Optimizations:* The encoding illustration in Section 2 is quite inefficient as it ignores a number of obvious, as well as some not-so-obvious, optimizations. Producing an efficient encoding, i.e., an encoding with as few variables as possible and with just the right type of constraints, is critical since it affects the performance of the constraint solver. By performing static analysis during encoding, we will be able to (1) reduce the number of constraints by encoding at a coarser level (e.g., transaction level); (2) reduce the number of variables based on the Static Single Assignment (SSA) format; (3) reduce the size of each constraint by considering variable scope.

- *SMT Solving Optimizations:* Much of the intractability in large combinatorial problems can be traced to symmetries in the search space that lead to unnecessary duplication of computational effort. If we are able to identify symmetric interleavings at either the program or the SMT formula levels we can theoretically avoid exploring symmetric regions of the search space possibly leading to a drastic reduction in the amount of computation needed. We will also explore the extraction of symmetries by mimicking the techniques that were shown to be quite effective in our previous research in the context of Boolean satisfiability and CSP applications [1, 3], and in the context of two-threaded programs [7].

5 Conclusion

Our approach involves modeling thread executions using suitable classes of constraints and reducing various analysis problems to constraint solving. The symbolic analysis is based on satisfiability modulo theories (SMT), which benefit from recent significant advances in Boolean satisfiability (SAT) and SMT solvers. The advantages of the approach include:

- reducing the complexity of testing/debugging multi-threaded programs to a level comparable to that of testing/debugging sequential programs (which is still non-trivial, but significantly simpler), and
- complete coverage of all thread interleavings under the specified inputs.

References

[1] Fadi A. Aloul, Karem A. Sakallah, and Igor L. Markov. Efficient symmetry breaking for boolean satisfiability. In *Proc. 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 271–282, Acapulco, Mexico, 2003.

[2] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[3] Paul T. Darga, Mark H. Liffiton, Karem A. Sakallah, and Igor L. Markov. Exploiting structure in symmetry detection for cnf. In *Proc. 41st IEEE/ACM Design Automation Conference (DAC)*, pages 530–534, San Diego, California, 2004.

[4] Leonardo de Moura and Nikolaj Bjorner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[5] Bruno Dutertre and Leonardo de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proceedings of the 18th Computer-Aided Verification conference*, volume 4144 of *LNCS*, pages 81–94. Springer-Verlag, 2006.

[6] Robert P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach.* Princeton University Press, Princeton, NJ, USA, 1994.

[7] Chao Wang, Zijiang Yang, V. Kahlon, and Aarti Gupta. Peephole partial order reduction. In *International Conference on tools and algorithms for the construction and analysis of systems (TACAS'08)*, 2008.