Verification of Concurrent Software

Daniel Kröning
University of Oxford, United Kingdom

Problem Context and Rationale Software products are becoming increasingly complex. One of the chief reasons for this is the demand for concurrent software that efficiently exploits multiple execution cores. Such systems, such as Intel's Core Duo, have become ubiquitous over the past two or three years. Unfortunately, developing reliable concurrent programs is a difficult and specialised task, requiring highly skilled engineers, most of whose efforts are spent on the testing and validation phases [1]. As a result, there is a strong economic and strategic incentive for software houses to automate parts of the verification process.

Random simulation and testing, while automated, has severe limitations, particularly in the case of concurrent software, in which the plethora of possible thread interleavings often conspires to conceal design flaws. Formal verification, on the other hand, can also be automated, and tools that implement it check a concurrent program for *all* its possible behaviours. It therefore promises higher gains in productivity, if done the right way.

The promise of automated bug-finding in complex software is not new. The *Verifying Compiler* was identified as a research goal in the 70s, and was recently re-stated as a Grand Challenge for computing research by Sir Tony Hoare. Whilst the problem in general is not yet solved, tools that specialise in very specific aspects of the problem demonstrated significant results. An instance is the SLAM project at Microsoft Research. By focussing on lightweight, control-flow dominated properties, the SLAM toolkit is able to verify Windows device drivers with hundreds of thousands of lines of code [2, 3]. SLAM checks for violations of about 30 rules such as "a thread must not acquire a lock it has already acquired, nor release a lock it does not hold". SLAM is now shipped as Static Driver Verifier (SDV) to developers.

Overview We survey the most important techniques in automated program verification in this section. For a full survey, we refer the reader to our IEEE TCAD paper [4]. *Model checking* [5, 6] is a formal verification technique, whose three originators (Ed Clarke, Allen Emerson, Joseph Sifakis) were awarded the 2007 ACM Turing Award. Model checking has been shown to be especially useful for verifying concurrency-related properties, and identifying bugs related to process schedules. A distinguishing feature of Model Checking is its ability to generate *diagnostic counterexamples* in the case of a bug, which practitioners consider invaluable for understanding and repair of the problem.

However, model checking suffers from the *state-space explosion problem*. In the case of BDD-based symbolic model checking, this problem manifests itself in the form of unmanageably large BDDs [7]. In case of concurrent software, the state-space explosion problem arises from two sources: (i) the model checker has to consider manifold interleavings among the threads, and (ii) software usually operates on a very large set of data variables. The principal technique to address the first problem is *partial-order reduction* [8]. The principal method for addressing the large amount of data is *abstraction*.

Abstraction Abstraction techniques reduce the state space by mapping the set of states of the actual, concrete system to an abstract, and smaller, set of states in a way that preserves the relevant behaviours of the system. Sound use of abstraction on transition systems is formalised in the abstract interpretation framework of [9].

Predicate abstraction [10, 11] is a popular and widely applied method for systematic abstraction of programs. It consists in abstracting data by only keeping track of certain predicates on program variables. Each predicate is represented by a Boolean variable in the abstract program, while the original data variables are eliminated. Verification of a software system with predicate abstraction consists of constructing and evaluating a finite-state system that is an abstraction of the original system with respect to a set of predicates. Typically, the abstract program is created using *existential abstraction* [12]. This method defines the instructions in the abstract program in such a way as to guarantee for it to be an over-approximation of the original program for reachability properties. Thus, in order to show that no erroneous state is reachable in the original program, it is sufficient to show that the abstract model has that property. Existential abstractions are therefore conservative for reachability specifications.

The drawback of such a conservative abstraction is that when model checking of the abstract program fails, it may produce a counterexample that does not correspond to any counterexample on the original program. This is usually called a *spurious counterexample*. When a spurious counterexample is encountered, a *refinement* is performed by adjusting the set of predicates in a way that eliminates this counterexample from the abstract program. This process is iterated until either a genuine counterexample is found, or the safety property is established. The actual steps of the loop follow the *counterexample-guided abstraction refinement* (CEGAR) framework.

Software Model Checkers Several predicate-abstraction based verification tools are available, most of which employ a CEGAR approach, such as SLAM [2, 13], MAGIC [14], BLAST [15], and SATABS [16]. The existing software model checkers, however, are not readily applicable to non-trivial software that uses shared-variable concurrency. BLAST improves the approach implemented in SLAM [2, 17] by using a 'lightweight' inner refinement abstraction loop that refines only relevant parts of the abstract model [15]. Henzinger et al. present a BLAST-based automated race checker for multithreaded C programs [18], which examines each thread separately using assume-guarantee reasoning.

References

[1] S. Lu, S. Park, E. Seo, Y Zhou. *Learning from Mistakes: a Comprehensive Study on Real World Concurrency Bug Characteristics*. I: Procs. of the n ASPLOS; ACM; 2008.

[2] T. Ball, S.K. Rajamani. *Boolean Programs: A Model and Process for Software Analysis*. Technical Report 2000-14; Microsoft Research; 2000.

[3] T. Ball, B. Cook, V. Levin, S. K. Rajamani. *SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft*. In: Procs of IFM; LNCS 2999; Springer; 2004.

[4] V. D'Silva, D. Kroening, G. Weissenbacher. *A Survey of Automated Techniques for Formal Software Verification*. TCAD, 27(7); 2008.

[5] E. Clarke, O. Grumberg, D. Peled. *Model Checking*. MIT Press; 1999.

[6] E. M. Clarke, E. A. Emerson. *Synthesis of Synchronization Skeletons for Branching Time Temporal Logic*. In: Logic of Programs: Workshop; LNCS 131; Springer; 1981.

[7] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang. *Symbolic Model Checking: 1020 States and Beyond*. Inf. Comput., 98(2); 1992.

[8] G. Holzmann, D. Peled. *An Improvement in Formal Verification.* In: Procs. of FORTE; Chapman & Hall; 1994.

[9] P. Cousot. *Abstract Interpretation.* ACM Comput. Surv., 28(2); 1996.

[10] S. Graf and H. Saïdi. *Construction of Abstract State Graphs with PVS.* In: Procs. of CAV; LNCS 1254; Springer; 1997.

[11] M. Colón, T. E. Uribe. *Generating Finite-state Abstractions of Reactive Systems using Decision Procedures.* In: Procs. of CAV; LNCS 1427; Springer; 1998.

[12] E. M. Clarke, O. Grumberg, D. E. Long. *Model Checking and Abstraction.* In: Procs. of POPL; IEEE; 1992.

[13] T. Ball, R. Majumdar, T. Millstein, S. K. Rajamani. *Automatic Predicate Abstraction of C Programs.* In: PLDI; ACM; 2001.

[14] S. Chaki, E. M. Clarke, A. Groce, S. Jha, H. Veith. *Modular Verification of Software Components in C.* IEEE Trans. Software Eng., 30(6); 2004.

[15] T. A. Henzinger, R. Jhala, R. Majumdar, G. Sutre. *Lazy Abstraction.* In: Procs. of POPL; ACM; 2002.

[16] E. M. Clarke, D. Kroening, N. Sharygina, K. Yorav. *Predicate Abstraction of ANSI-C Programs using SAT.* FMSD, 25(2-3); 2004.

[17] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, A. Ustuner. *Thorough Static Analysis of Device Drivers.* In: Procs. of EuroSys; ACM; 2006.

[18] T. A. Henzinger, R. Jhala, R. Majumdar, S. Qadeer. *Thread-modular Abstraction Refinement.* In: Procs. of CAV; LNCS 2725; Springer; 2003.