Threader: A Verifier for Multi-threaded Programs
(Competition Contribution)

Corneliu Popeea and Andrey Rybalchenko

Technische Universität München

Abstract. THREADER is a tool that automates verification of safety and termination properties for multi-threaded C programs. The distinguishing feature of THREADER is its use of reasoning that is compositional with regards to the thread structure of the verified program. This paper describes the verification approach taken by THREADER and provides instructions on how to install and use the tool.

1 Verification Approach

THREADER is a tool for verification of C programs based on predicate abstraction and refinement following the counterexample-guided abstraction refinement (CEGAR) paradigm [3]. There is a number of verification tools based on abstraction refinement that are successful for sequential programs [1, 2, 4, 5, 7, 12]. This paper gives a brief description of specific features that were required to handle the concurrency benchmarks from the verification competition. Interested readers can find more details about the theory behind THREADER in [6].

2 Software Architecture

THREADER consists of two main components: a frontend for translating C programs in corresponding transition systems and a model checking back-end. The frontend is implemented in the OCaml language and relies on the CIL library [10]. Additional analyses are implemented in our frontend to handle the competition benchmarks (see next section for details). The model checker automates compositional reasoning of multi-threaded programs by implementing Owicki-Gries and rely-guarantee proof rules [9, 11]. This model checker is implemented in the Prolog language and relies on the constraint solver for linear arithmetic CLP(Q) [8].

3 Discussion

In this section we present our experience in running THREADER on the benchmarks from the Concurrency category.

THREADER supports C programs with calls to Pthread library functions. To handle threads and mutex objects from the Pthread library, we require a pointer analysis that is more precise than the standard flow insensitive analysis available from the CIL library. As a solution to this problem, we implemented a context-sensitive pointer analysis that is explicit about some heap allocated objects and sound for multi-threaded programs.

Creation of threads in loops is another difficulty for THREADER, since our model checker assumes a finite number of threads during verification. To handle this problem, we implemented a frontend analysis to compute the number of loop iterations and consequently the number of threads to be created. For all the competition benchmarks, this analysis is precise and we obtain constant values for the number of threads. As future work we would like to handle cases where the number of threads cannot be precisely computed statically, i.e., to be able to do automatic verification of parameterized systems.

Another difficulty for automatic verifiers is the analysis of array objects. Here THREADER takes a pragmatic approach automating verification for some particular universal properties over the elements of an array. This reasoning is sufficient to handle three benchmarks (`indexer_safe.i`, `stack_unsafe.i` and `stack_safe.i`). Precise results for the four queue benchmarks require invariants that relate contents of different array objects and cannot be currently handled by THREADER.

The set of Concurrency benchmarks contains some benchmarks that are pre-processed using the Simplify CIL module (the `*.cil.c` benchmarks). These benchmarks are presented as three-address-code with a significant number of temporary variables, with 'for' statements transformed into loops with 'goto' statements indicating the loop exit, and with array operations expressed using pointer arithmetic. THREADER benefits from the CIL framework that allows an easy recovery of the high-level information regarding loops and array operations. Therefore we observed (almost) identical verification results and times for both the `*.cil.c` and the `*.i` forms of the benchmarks.

In general our verifier is designed not to miss bugs present in the C programs. We list here some of the significant advantages of THREADER that facilitate a sound analysis of multi-threaded programs.

- THREADER is applicable to arbitrary (or ad-hoc) synchronization patterns, not only nested locking patterns or datarace free code.
- THREADER does not restrict the analysis to a bounded number of context-switches, but instead deals with an unbounded number of context switches.
- THREADER is not restricted to programs with thread-modular proofs and can handle the general case of non-thread-modular proofs required for example by the Fibonacci competition benchmarks.

To summarize, we ran THREADER on the 32 benchmarks from the Concurrency category and obtained a total of 43 out of the 49 points available in this category. THREADER reports SAFE and UNSAFE correctly for 28 benchmarks. For the other four benchmarks (`queue_unsafe.cil.c`, `queue_unsafe.i`, `queue_ok_safe.cil.c`, `queue_ok_safe.i`), THREADER returns UNKNOWN due to limitations in handling quantified array invariants. (We are not aware of any automatic verification tool that can handle these benchmarks.) A SAFE result leads to the creation of an abstract reachability tree that represents a correctness proof (see generated file `art.dot`). An UNSAFE result leads to the creation of a counterexample in dotty format (see generated file `cex.dot`).

4 Tool Setup

Threader can be downloaded from http://www7.in.tum.de/tools/threader/.

Threader is provided as a set of statically compiled binaries for the Linux x86-64 architecture. A script is provided to invoke Threader with predefined options for the competition. The tool should be run as follows: `./threader.sh <file.c>`. The working directory (`PWD`) must be the directory where Threader's files are located.

Acknowledgements We gratefully acknowledge the help of Ashutosh Gupta on designing and implementing various aspects of the previous version of Threader. This research was supported in part by ERC project 308125 VeriSynth.

References

1. T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, 2002.
2. D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *CAV*, 2011.
3. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, 2000.
4. E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *TACAS*, pages 570–574, 2005.
5. S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, and A. Rybalchenko. HSF(C): A software verifier based on Horn clauses - (competition contribution). In *TACAS*, pages 549–551, 2012.
6. A. Gupta, C. Popeea, and A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In *POPL*, pages 331–344, 2011.
7. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.
8. C. Holzbaur. *OFAI clp(q,r) Manual, Edition 1.3.3.* Austrian Research Institute for Artificial Intelligence, Vienna, 1995. TR-95-09.
9. C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.
10. G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC*, pages 213–228, 2002.
11. S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Inf.*, 6:319–340, 1976.
12. A. Podelski and A. Rybalchenko. ARMC: The logical choice for software model checking with abstraction refinement. In *PADL*, pages 245–259, 2007.