Threader: A Constraint-Based Verifier for Multi-threaded Programs

Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko

Technische Universität München

Abstract. We present a tool that implements Owicki-Gries and rely-guarantee methods for the compositional verification of multi-threaded programs. Our tool computes the requisite auxiliary assertions automatically using an abstraction and refinement procedure. Our procedure is based on a Horn clause encoding of refinement queries and facilitates the discovery of thread-modular proofs when such proofs exist. We present the tool and its evaluation on a collection of benchmarks, including a direct comparison of the effectiveness of the proof rules.

1 Introduction

Software running on our computers is becoming increasingly concurrent, i.e., it consists of several execution threads that process several tasks in parallel and interact with each other during the operation. Increasing concurrency is supported by the state-of-the-art in computer hardware, where modern CPUs have several computing cores and can execute several threads at the same time. However, it is extremely difficult to develop correct concurrent programs that are free of bugs, as evidenced by recent studies [5, 10].

In this paper we present Threader, a tool that automates verification of multi-threaded programs. The algorithms implemented in Threader are rooted in compositional proof rules [9, 13]. Following [6, 7], Threader uses abstraction and abstraction refinement to find adequate auxiliary assertions for verification. In this paper, we investigate the effectiveness of the compositional rules on a collection of benchmarks.

2 Threader Overview

Threader consists of three main modules that interact as shown in Figure 1. First, a C-frontend translates the input C program and its assert statements into a transition system that is represented using constraints over program variables. Next, the program safety is verified by iteratively applying abstract reachability computation and abstraction refinement steps. If no error state is unreachable then Threader reports that the program is safe. Otherwise, i.e., if an error state is discovered by the abstract reachability computation, Threader encodes the error state reachability using a set of recursion free Horn clauses and invokes a Horn solver. If the Horn clauses are not satisfiable then Threader returns a

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 412–417, 2011.

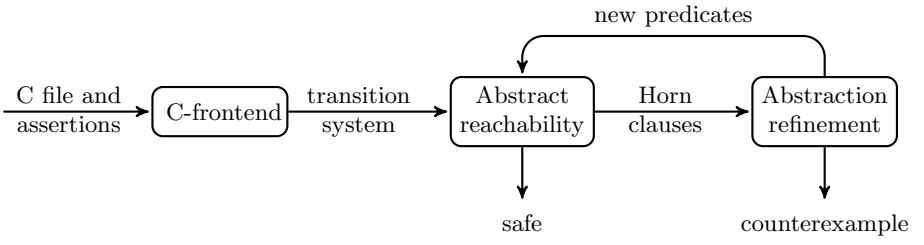

Fig. 1. The main modules of THREADER. The abstract reachability module solves recursive equations (1), (2), or (3). The abstraction refinement module discovers (transition) predicates by solving Horn clauses.

counterexample. Otherwise, a solution of the Horn clauses yields predicate that are needed to refine the abstraction. We describe the modules of THREADER below.

C-frontend. THREADER's frontend is based on the CIL framework [12]. The frontend takes as input a C file containing N functions that represent N threads to be executed in parallel. We assume that the threads interact using shared variables. THREADER does not support recursive functions and relies on inlining to deal with function calls. After inlining, each of the N functions is translated to a constraint-based representation. The frontend outputs a transition system $P = (V, \varphi_{init}, \varphi_{err}, \rho_1, \ldots, \rho_{\mathsf{N}})$ with variables V, initial states φ_{init}, error states φ_{err}, and thread transitions $\rho_1, \ldots, \rho_N$. The program variables $V = (V_G, V_1, \ldots, V_{\mathsf{N}})$ are partitioned into global variables shared by all threads, and local variables of each thread. The set of initial program states φ_{init} is obtained by initializing the global variables. The set of error states φ_{err} is derived from the assert statements in the input C program. Finally, each transition relation ρ_i can only change the values of global variables and local variables of thread i. We use $\rho_i^{=} = (V_i = V_i')$ to make this requirement explicit, i.e., for each $i \neq j \in 1..\mathsf{N}$ we require the validity of $\rho_i \rightarrow \rho_j^{=}$. We assume that each implication assertion in this paper is implicitly universally quantified over its free variables. The transition relation of the entire program is $\rho_1 \vee \cdots \vee \rho_{\mathsf{N}}$.

Abstract reachability (and environment transitions). Given an abstraction function, the abstract reachability module computes an over-approximation of the states reachable during any execution of a multi-threaded program and corresponding environment transitions, as described by the proof rules in Figure 2.

The rule (1) relies on a single, global auxiliary assertion R over program variables V. If R satisfies all three conditions of the proof rule then the program is safe. The first condition ensures that R over-approximates the initial states of the program φ_{init}. The second condition ensures that R is invariant under the

Find an assertion R over V such that:

$$\begin{array}{ll} \dot{\alpha}(\varphi_{init}) & \rightarrow R \\ \dot{\alpha}(post(\rho_1 \vee \cdots \vee \rho_{\mathsf{N}}, R)) & \rightarrow R \\ R \wedge \varphi_{err} & \rightarrow \mathit{false} \end{array} \tag{1}$$

Find assertions $R_1, \ldots, R_{\mathsf{N}}$ over V such that

$$\begin{array}{lll} \dot{\alpha}_i(\varphi_{init}) & \rightarrow R_i & \text{for } i \in 1..\mathsf{N} \\ \dot{\alpha}_i(post(\rho_i, R_i)) & \rightarrow R_i & \text{for } i \in 1..\mathsf{N} \\ \dot{\alpha}_i(post(\rho_j, R_i \wedge R_j)) & \rightarrow R_i & \text{for } i \neq j \in 1..\mathsf{N} \\ R_1 \wedge \cdots \wedge R_{\mathsf{N}} \wedge \varphi_{err} & \rightarrow \mathit{false} & \end{array} \tag{2}$$

Find assertions $R_1, \ldots, R_{\mathsf{N}}$ over V and $E_1, \ldots, E_{\mathsf{N}}$ over V and V' such that

$$\begin{array}{lll} \dot{\alpha}_i(\varphi_{init}) & \rightarrow R_i & \text{for } i \in 1..\mathsf{N} \\ \dot{\alpha}_i(post(\rho_i \vee (E_i \wedge \rho_i^{=}), R_i)) & \rightarrow R_i & \text{for } i \in 1..\mathsf{N} \\ \ddot{\alpha}_{j \triangleright i}(R_j \wedge \rho_j) & \rightarrow E_i & \text{for } i \neq j \in 1..\mathsf{N} \\ R_1 \wedge \cdots \wedge R_{\mathsf{N}} \wedge \varphi_{err} & \rightarrow \mathit{false} & \end{array} \tag{3}$$

Fig. 2. Proof rules for safety of a program $(V, \varphi_{init}, \varphi_{err}, \rho_1, \ldots, \rho_{\mathsf{N}})$. Given abstraction functions, Threader computes the strongest solution for the auxiliary assertions using either (1) "Monolithic", (2) "Owicki-Gries", or (3) "Rely-Guarantee" proof rule.

application of the thread transitions $\rho_1, \ldots, \rho_{\mathsf{N}}$. The last condition requires that R does not intersect the error states φ_{err} .

The rule (2) is a formulation of the "Owicki-Gries" proof rule [13]. The reasoning about reachable states is localized by replacing a global auxiliary assertion R with N thread-reachability assertions $R_1, \ldots, R_{\mathsf{N}}$. Each thread-reachability assertion R_i needs to over-approximate the initial states and needs to be invariant under the transition relation of thread i . In addition, R_i also needs to account for interference from the transition relation of thread j when it is applied to reachable states in R_j . The last condition requires that the intersection of the thread-reachability assertions and φ_{err} is empty.

The rule (3) reasons about the threads individually by relying on environment assertions. Each assertion E_i denotes a binary relation over V and V' that captures how all threads other than i can change the program states. As above, the thread-reachability assertion R_i is required to over-approximate the initial states and be invariant under the transition relation of thread i . Threader accounts for the interference from threads other than i by the environment transition $E_i \wedge \rho_i^{=}$, which does not modify the values of variables local to thread i .

Threader effectively computes the strongest candidate for the auxiliary assertions wrt. a given abstraction. See [7] for a corresponding algorithm for the proof rule (3) (other rules are similar).

In practice, it is crucial to maintain for each thread i an abstraction function $\dot{\alpha}_i$ that approximates the thread-reachability R_i . Environment transitions are approximated using different abstraction functions for different pairs of threads. For an abstraction function $\ddot{\alpha}_{i \triangleright j}$ the double dot indicates that the function $\ddot{\alpha}_{i \triangleright j}$ abstracts binary relations over states (not sets of states) and the index $i \triangleright j$ indicates that this function is used to abstract the effect of the thread i on the reachability of thread j .

THREADER uses predicate and transition predicate abstraction functions that are defined using sets of predicates $\dot{\mathcal{P}}_i$ and transition predicates $\ddot{\mathcal{P}}_{i \triangleright j}$ as follows.

$$\dot{\alpha}_i(S) = \bigwedge \{\dot{p} \in \dot{\mathcal{P}}_i \mid S \rightarrow \dot{p}\} \qquad \ddot{\alpha}_{i \triangleright j}(T) = \bigwedge \{\ddot{p} \in \ddot{\mathcal{P}}_{i \triangleright j} \mid T \rightarrow \ddot{p}\}$$

THREADER discovers the sets of (transition) predicates automatically. Initially, the empty sets are used to compute a coarse approximation of the reachable state space and environment transitions. If, for given abstraction functions, the reachability assertions intersect the set of error states, then the discovered error evidence needs to be checked for spuriousness. The reachability assertions computed so far are used to formulate a query to the abstraction refinement module.

Termination properties. THREADER can prove termination properties based on a "Rely-Guarantee" proof rule and automated construction of transition abstraction functions. More details are reported in [8].

3 Experiments

In this section, we present our experience with applying THREADER on 15 multi-threaded C programs. See Table 1. The name of the benchmark and the number of lines of C code are shown in the first two columns. Columns 3, 4 and 5 present verification results obtained from our implementation of the proof rules (1), (2), and (3), respectively. The source code for the examples together with additional data can be found at http://www.model.in.tum.de/~popeea/research/threader.html.

The programs shown in Table 1 are small but intricate. We are not aware of any automatic tool that can deal with these examples.

The first example from the table illustrates a program used as running example in the paper introducing thread-modular model checking [4]. The program safety can be proven using each of the proof rules. Due to the low complexity of this safety proof, this program illustrates that monolithic verification may conclude faster than localized reasoning.

The second part of the table reports on various algorithms to establish mutual exclusion between a number of threads. For all these examples, we instrumented a safety assertion to check mutual exclusion. Dekker, Peterson, and Szymanski are classical algorithms. Readers-writer-lock and Time-varying-mutex are tests for the Calvin model checker [3]. QRCU [11] is a variant of the read-copy-update algorithm that is used in the Linux kernel, and is the most complex of

Table 1. Applying different proof rules implemented in Threader. All programs are safe. Time is measured in seconds. "T/O" stands for time out after 15 minutes.

Benchmark programs	LOC	"Monolithic"	"Owicki-Gries"	"Rely-Guarantee"
Spin2003	18	0.02s	0.02s	0.1s
Dekker	39	T/O	0.3s	1.1s
Peterson	26	T/O	0.7s	2.3s
Szymanski	43	T/O	1.8s	12.2s
Readers-writer-lock	22	0.03s	0.03s	0.1s
Time-varying mutex	29	0.3s	0.73s	7.5s
NaïveBakery	22	T/O	0.3s	2.4s
Bakery	37	T/O	1.4s	97s
Lamport	62	T/O	11.4s	57s
QRCU-2processes	120	T/O	1.8s	11.3s
QRCU-3processes	148	T/O	89s	T/O
QRCU-4processes	182	T/O	T/O	T/O
Mozilla-fixed-vulnerab	168	T/O	0.8s	0.8s
See-Saw	98	T/O	T/O	7.8s
Scull	451	T/O	T/O	41.2s

the presented mutual exclusion algorithms. We test its simple variant with two processes (one reader and one updater), as well as variants with two and three readers. For all benchmarks, we observed that the monolithic verification cannot cope with the transition relation of the entire program. Furthermore, the "Owicki-Gries" proof rule captures concisely the interference between threads, as the state space is overly constrained by values of the variables establishing the mutual exclusion invariant. Finally, the "Rely-Guarantee" proof rule requires representing intricate environment transitions and therefore it needs to discover supporting transition predicates. We include the example QRCU-4processes that times-out for both "Owicki-Gries" and "Rely-Guarantee" abstraction refinement methods. We note that for the moment Threader does not implement algorithms for symmetry reduction that would be beneficial for the efficiency of a compositional verification approach, as demonstrated in [1].

In the third part of the table, Mozilla-fixed-vulnerab is a fix from the Mozilla CVS repository for a vulnerability described in a study of concurrency bugs [10]. See-Saw is a multi-threaded version of the program reported in [14] where we instrumented the invariants obtained by the StInG prover as assertions in the C program. Scull [2] is a Linux character driver that implements access to a global memory area. Different invocations of the open, read, write, and release functions implemented by the device driver access common variables and these accesses should be performed in a critical section. For these programs, we observed that "Rely-Guarantee" reasoning allows a natural encoding of environment transitions as binary relations. In contrast, the "Owicki-Gries" proof rule is not able to capture the thread interference since the thread-reachability assertions are in this case expressed over sets of states.

To conclude, we note some of the significant advantages of the algorithms implemented in Threader.

- They are 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 analyzes (implicitly) an unbounded number of context switches.
- The proofs constructed by THREADER are not restricted to thread-modular proofs. In addition, the search for new (transition) predicates can be restricted to thread-modular solutions that favor compositional reasoning, as described in [7].
- THREADER allows an experimental comparison of the two state-of-the-art proof rules for compositional verification of multi-threaded programs in a uniform setting.

References

1. Cohen, A., Namjoshi, K.S.: Local proofs for global safety properties. In: FMSD, vol. 34(2), pp. 104–125 (2009)
2. Corbet, J., Rubini, A., Kroah-Hartman, G.: Linux Device Drivers, 3rd edn. O'Reilly Media. Sebastopol (2005)
3. Flanagan, C., Freund, S.N., Qadeer, S.: Thread-modular verification for shared-memory programs. In: Le Métayer, D. (ed.) ESOP 2002. LNCS, vol. 2305, pp. 262–277. Springer, Heidelberg (2002)
4. Flanagan, C., Qadeer, S.: Thread-modular model checking. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 213–224. Springer, Heidelberg (2003)
5. Fonseca, P., Li, C., Singhal, V., Rodrigues, R.: A study of the internal and external effects of concurrency bugs. In: DSN, pp. 221–230 (2010)
6. Gupta, A., Popeea, C., Rybalchenko, A.: Non-monotonic refinement of control abstraction for concurrent programs. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 188–202. Springer, Heidelberg (2010)
7. Gupta, A., Popeea, C., Rybalchenko, A.: Predicate abstraction and refinement for verifying multi-threaded programs. In: POPL, pp. 331–344 (2011)
8. Gupta, A., Popeea, C., Rybalchenko, A.: Transition invariants and environment transitions for proving termination of multi-threaded programs (2011) (under submission)
9. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. 5(4), 596–619 (1983)
10. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: ASPLOS, pp. 329–339 (2008)
11. McKenney, P.: Using Promela and Spin to verify parallel algorithms. LWN.net weekly edition (2007)
12. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: CC 2002. LNCS, vol. 2304, p. 213. Springer, Heidelberg (2002)
13. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs. I. Acta Inf. 6, 319–340 (1976)
14. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constraint-based linear-relations analysis. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 53–68. Springer, Heidelberg (2004)