Predicate Abstraction and Refinement for Verifying Multi-Threaded Programs

Ashutosh Gupta Corneliu Popeea Andrey Rybalchenko

Institut für Informatik, Technische Universität München
Germany
{guptaa,popeea,rybal}@in.tum.de

Abstract

Automated verification of multi-threaded programs requires explicit identification of the interplay between interacting threads, so-called environment transitions, to enable scalable, compositional reasoning. Once the environment transitions are identified, we can prove program properties by considering each program thread in isolation, as the environment transitions keep track of the interleaving with other threads. Finding adequate environment transitions that are sufficiently precise to yield conclusive results and yet do not overwhelm the verifier with unnecessary details about the interleaving with other threads is a major challenge. In this paper we propose a method for safety verification of multi-threaded programs that applies (transition) predicate abstraction-based discovery of environment transitions, exposing a minimal amount of information about the thread interleaving. The crux of our method is an abstraction refinement procedure that uses recursion-free Horn clauses to declaratively state abstraction refinement queries. Then, the queries are resolved by a corresponding constraint solving algorithm. We present preliminary experimental results for mutual exclusion protocols and multi-threaded device drivers.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Algorithms, Languages, Reliability, Verification.

Keywords Multi-threaded programs, safety, proof rule, modular reasoning, environment transitions, (transition) predicate abstraction, abstraction refinement, Horn clauses.

1. Introduction

The ubiquitous availability of parallel computing infrastructures facilitated by the advent of multicore architectures requires a shift towards multi-threaded programming to take full advantage of the available computing resources. Writing correct multi-threaded software is a difficult task, as the programmer needs to keep track of a very large number of possible interactions between the program threads. Automated program analysis and verification tools can support programmer in dealing with this challenge by systematically and exhaustively exploring program behaviours and checking their correctness.

Direct treatment of all possible thread interleavings by reasoning about the program globally is a prohibitively expensive task, even for small programs. By applying rely-guarantee techniques, see e.g. [17, 26], such global reasoning can be avoided by considering each program thread in isolation, using environment transitions to summarize the effect of executing other threads, and applying them on the thread at hand. The success of such an approach depends on the ability to automatically discover environment transitions that are precise enough to deliver a conclusive analysis/verification outcome, and yet do not keep track of unnecessary details in order to avoid sub-optimal efficiency.

In this paper we present a method that automates rely-guarantee reasoning for verifying safety of multi-threaded programs. Our method relies on an automated discovery of environment transitions using (transition) predicate abstraction [12, 28]. It performs a predicate abstraction-based reachability computation for each thread and interleaves it with the construction of environment transitions that over-approximate the effect of executing thread transitions using transition predicates. The success of our method crucially depends on an abstraction refinement procedure that discovers (transition) predicates. The refinement procedure attempts to minimize the amount of details that are exposed by the environment transitions, in order to avoid unnecessary details about thread interaction.

The crux of our refinement approach is in using a declarative formulation of the abstraction refinement algorithm that can deal with the thread reachability, environment transitions, and their mutual dependencies. We use Horn clauses to describe constraints on the desired (transition) predicates, and solve these constraints using a general algorithm for recursion-free Horn clauses. Our formalization can accommodate additional requirements that express the preference for modular predicates that do not refer to the local variables of environment threads, together with the preference for modular transition predicates that only deal with global variables and their primed versions.

We implemented the proposed method in a verification tool for multi-threaded programs and applied it on a range of benchmarks, which includes fragments of open source software, ticket-based mutual exclusion protocols, and multi-threaded Linux device drivers. The results of the experimental evaluation indicate that our declarative abstraction refinement approach can be effective in finding adequate environment transitions for the verification of multi-threaded programs.

This paper makes the following contributions:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL'11, January 26–28, 2011, Austin, Texas, USA.
Copyright © 2011 ACM 978-1-4503-0490-0/11/01…$10.00

1. the automatic, rely-guarantee based method for verifying multi-threaded programs using (transition) predicate abstraction;
2. the novel formulation of abstraction refinement schemes using Horn clauses, and its application for the (transition) abstraction discovery for multi-threaded programs;
3. the algorithm for solving recursion-free Horn clauses over linear arithmetic constraints;
4. the prototype implementation and its evaluation.

The rest of the paper is organized as follows. First, we illustrate our method in Section 2. In Section 3 we present necessary definitions. Section 4 presents a proof rule that provides a basis for our method, and shows how the proof rule can be automated using the connection to fixpoints and abstraction techniques. We present the main algorithm in Section 5. Section 6 focuses on the abstraction refinement using Horn clauses, while Section 7 presents a constraint solving algorithm for Horn clauses over linear inequalities. We discuss the experimental evaluation in Section 8. Related work is presented in Section 9.

2. Illustration

In this section we illustrate our algorithm using two multi-threaded examples. The first example does not have a modular proof, hence our algorithm reasons about relationship between the local variables of different threads. For the second example, our algorithm succeeds in finding a modular proof by applying an abstraction refinement procedure that guarantees the discovery of a modular abstraction whenever it exists.

2.1 Example `LockBit`

See Figure 1 for the program `LockBit` that consists of two threads. The threads attempt to access a critical section, and synchronize their accesses using a global variable `lock` . We assume that initially the lock is not taken, i.e., $\texttt{lock} = 0$, and that the locking statement `take_lock` waits until the lock is released and then assigns the value of its second parameter to `lock` , thus taking the lock. We write $V = (\texttt{lock}, \texttt{pc}_1, \texttt{pc}_2)$ for the program variables, where $\texttt{pc}_1$ and $\texttt{pc}_2$ are local program counter variables of the first and second thread, respectively.

We start by representing the program using assertions φ_{init} and φ_{err} over program variables that describe the initial and error states of the program, together with assertions over unprimed and primed program variables ρ_1 and ρ_2 that describe the transition relations for program statements.

$$\begin{aligned}
\varphi_{init} &= (\texttt{pc}_1 = a \wedge \texttt{pc}_2 = p \wedge \texttt{lock} = 0)\ , \\
\varphi_{err} &= (\texttt{pc}_1 = b \wedge \texttt{pc}_2 = q)\ , \\
\rho_1 &= (\texttt{lock} = 0 \wedge \texttt{lock}' = 1 \wedge \texttt{pc}_1 = a \wedge \texttt{pc}_1{}' = b \wedge \rho_2^{=})\ , \\
\rho_1^{=} &= (\texttt{pc}_1 = \texttt{pc}_1{}')\ , \\
\rho_2 &= (\texttt{lock} = 0 \wedge \texttt{lock}' = 1 \wedge \texttt{pc}_2 = p \wedge \texttt{pc}_2{}' = q \wedge \rho_1^{=})\ , \\
\rho_2^{=} &= (\texttt{pc}_2 = \texttt{pc}_2{}')\ .
\end{aligned}$$

The auxiliary assertions $\rho_1^{=}$ and $\rho_2^{=}$ state that the local variable of the first and second thread, respectively, is preserved during the transition.

To verify `LockBit`, our algorithm computes a sequence of ARETs (Abstract Reachability and Environment Trees). Each tree computation amounts to a combination of i) a standard abstract reachability computation that is performed for each thread and is called thread reachability, and ii) a construction and application of environment transitions. Abstract states represent sets of (concrete) program states, while environment transitions are binary relations of program states.

```
// Thread 1                    // Thread 2
a: take_lock(lock, 1);         p: take_lock(lock, 1);
b: // critical                 q: // critical
```

Figure 1. Example program `LockBit`. Each thread waits until the lock is released, and then assigns the integer 1 to `lock` .

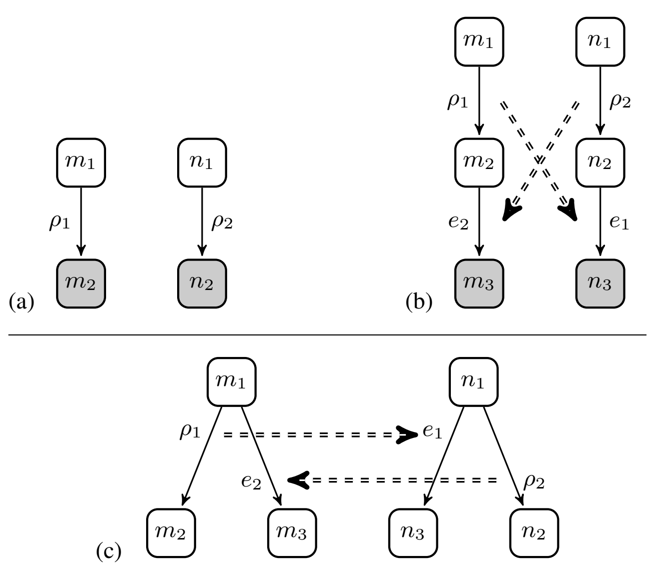


Figure 2. Reachability trees constructed using different abstraction functions. Edges are labeled with a transition. Nodes with gray background represent (spurious) error tuples: (m_2, n_2) from (a) and (m_3, n_3) from (b). No pair of states from (c) intersects φ_{err}.

First* ARET *computation The thread reachability computation for the first thread starts by computing an abstraction of the initial program states φ_{init} . Here, we use an abstraction function $\dot{\alpha}_1$, where the dot indicates that this function over-approximates sets of program states (and not sets of pairs of states, as will take place later) and the index 1 indicates that this abstraction function is used for the first thread. In this example, we assume that the abstraction function only tracks the value of the program counter of the first thread, i.e., $\dot{\mathcal{P}}_1 = \{\texttt{pc}_1 = a, \texttt{pc}_1 = b\}$, and is computed as follows: $\dot{\alpha}_1(S) = \bigwedge\{\dot{p} \in \dot{\mathcal{P}}_1 \mid \forall V : S \rightarrow \dot{p}\}$. We obtain the initial abstract state m_1 as follows:

$$m_1 = \dot{\alpha}_1(\varphi_{init}) = (\texttt{pc}_1 = a)\ .$$

Next, we compute an abstract successor of m_1 with respect to the transition ρ_1 using the strongest postcondition operator $post$ that is combined with $\dot{\alpha}_1$:

$$m_2 = \dot{\alpha}_1(post(\rho_1, m_1)) = (\texttt{pc}_1 = b)\ .$$

Similarly, we compute the thread reachability for the second thread. Using predicates over the program counter of the second thread, i.e., $\dot{\mathcal{P}}_2 = \{\texttt{pc}_2 = p, \texttt{pc}_2 = q\}$, we compute the following two abstract states:

$$\begin{aligned}
n_1 &= \dot{\alpha}_2(\varphi_{init}) &&= (\texttt{pc}_2 = p)\ , \\
n_2 &= \dot{\alpha}_2(post(\rho_2, n_1)) &&= (\texttt{pc}_2 = q)\ .
\end{aligned}$$

For each thread, we organize the computed abstract states in a tree, see Figure 2(a).

We stop the ARET computation since we discover that the error states overlap with the intersection of the abstract state m_2 from the thread reachability of the first thread and n_2 from the second thread, i.e., $m_2 \land n_2 \land \varphi_{err}$ is satisfiable.

First abstraction refinement We treat the pair m_2 and n_2 as a possible evidence that the error states of the program can be reached. Yet, we cannot assert that the program is incorrect, since abstraction was involved when computing m_2 and n_2 .

We check if the discovered evidence is spurious by formulating a constraint that is satisfiable if and only if the abstraction can be refined to exclude the spuriousness. For each abstract state involved in the reachability of and including m_2 and n_2 we create an unknown predicate that denotes a set of program states. We obtain "m_1"(V), "n_1"(V), "m_2"(V), and "n_2"(V) , which correspond to m_1, n_1, m_2, and n_2, respectively. Then, we record the relation between the unknown predicates using constraints in the form of Horn clauses. For example, since m_1 was an abstraction of the initial program states, we require that "m_1"(V) over-approximates φ_{init} as well, and represent this requirement by a Horn clause $\varphi_{init} \rightarrow$ "m_1"(V) . As a result, we obtain the following set of clauses.

$$\begin{aligned} HC_1 = \{&\varphi_{init} \rightarrow \text{“}m_1\text{”}(V),\ \text{“}m_1\text{”}(V) \land \rho_1 \rightarrow \text{“}m_2\text{”}(V'),\\ &\varphi_{init} \rightarrow \text{“}n_1\text{”}(V),\ \text{“}n_1\text{”}(V) \land \rho_2 \rightarrow \text{“}n_2\text{”}(V'),\\ &\text{“}m_2\text{”}(V) \land \text{“}n_2\text{”}(V) \land \varphi_{err} \rightarrow \mathit{false}\} \end{aligned}$$

The last clause in HC_1 requires that the intersection of the refined versions of the abstract states m_2 and n_2 is disjoint from the error states of the program.

We check if the conjunction of the clauses in HC_1 is satisfiable using a SAT-based algorithm presented in [13]. (Section 7 presents an algorithm for solving Horn clauses over linear inequalities.) We obtain the following satisfying assignment SOL that maps each unknown predicate to an assertion of the program variables.

$$\begin{array}{ll} \text{SOL}(\text{“}m_1\text{”}(V)) = (\mathtt{pc}_2 = p) & \text{SOL}(\text{“}n_1\text{”}(V)) = (\mathtt{pc}_1 = a)\\ \text{SOL}(\text{“}m_2\text{”}(V)) = (\mathtt{pc}_2 = p) & \text{SOL}(\text{“}n_2\text{”}(V)) = (\mathtt{pc}_1 = a) \end{array}$$

The existence of SOL indicates that the discovered evidence is spurious. We use SOL to refine the abstraction functions and hence eliminate the source of spuriousness. We collect the predicates that appear in the solution for abstract states from the first thread, add them to the sets of predicates $\dot{\mathcal{P}}_1$, and perform a similar step for the second thread. The resulting sets of predicates are shown below.

$$\begin{aligned} \dot{\mathcal{P}}_1 &= \{\mathtt{pc}_1 = a, \mathtt{pc}_1 = b, \mathtt{pc}_2 = p\}\\ \dot{\mathcal{P}}_2 &= \{\mathtt{pc}_2 = p, \mathtt{pc}_2 = q, \mathtt{pc}_1 = a\} \end{aligned}$$

They guarantee that the same spuriousness will not appear during subsequent ARET computations.

Second* ARET *computation We re-start the ARET computation using the previously discovered predicates. Figure 2(b) shows the two trees computed with the refined abstraction functions where

$$\begin{array}{ll} m_1 = (\mathtt{pc}_1 = a \land \mathtt{pc}_2 = p)\ , & n_1 = (\mathtt{pc}_1 = a \land \mathtt{pc}_2 = p)\ ,\\ m_2 = (\mathtt{pc}_1 = b \land \mathtt{pc}_2 = p)\ , & n_2 = (\mathtt{pc}_1 = a \land \mathtt{pc}_2 = q)\ . \end{array}$$

Due to the first abstraction refinement step, $m_2 \land n_2 \land \varphi_{err}$ is unsatisfiable. The thread reachability computation for each thread does not discover any further abstract states.

The ARET computation proceeds by considering interleaving of the transitions from one thread with the transitions from the other thread. We account for thread interleaving by constructing and applying environment transitions. First, we construct an environment transition e_1 that records the effect of applying ρ_1 on m_1 in the first thread on the thread reachability in the second thread. This effect is over-approximated by using an abstraction function $\ddot{\alpha}_{1\triangleright 2}$. In this function, the double dot indicates that the function abstracts binary relations over states (and not sets of states). The index $1 \triangleright 2$ indicates that this function is applied to abstract effect of the first thread on the second thread. Initially, we use the empty set of transition predicates (over pairs of states) $\ddot{\mathcal{P}}_{1\triangleright 2} = \emptyset$ to define $\ddot{\alpha}_{1\triangleright 2}$. The environment transition e_1 is defined as

$$e_1 = \ddot{\alpha}_{1\triangleright 2}(m_1 \land \rho_1) = \mathit{true}\ ,$$

and it non-deterministically updates the program variables (since $true$ does not impose any restrictions on the successor states of the transition).

Next, we add e_1 to the transitions of the second thread. Then, its thread reachability computation uses e_1 during the abstract successor computation, and creates an abstract state n_3 by applying e_1 on n_2 as follows:

$$n_3 = \dot{\alpha}_2(\mathit{post}(e_1 \land \rho_2^{=}, n_2)) = (\mathtt{pc}_2 = q)\ .$$

The conjunct $\rho_2^{=}$ ensures that the local variable of the second thread is not changed by the environment transition.

Symmetrically, we use a function $\ddot{\alpha}_{2\triangleright 1}$ to abstract the effect of applying transitions in the second thread on the thread reachability of the first thread. The application of ρ_2 on n_1 results in the environment transition e_2 such that

$$e_2 = \ddot{\alpha}_{2\triangleright 1}(n_1 \land \rho_2) = \mathit{true}\ .$$

We apply e_2 to contribute an abstract successor m_3 of the abstract state m_2 to the thread reachability of the first thread:

$$m_3 = \dot{\alpha}_1(\mathit{post}(e_2 \land \rho_1^{=}, m_2)) = (\mathtt{pc}_1 = b)\ .$$

We observe that the intersection of the abstract states m_3 and n_3 contains a non-empty set of error states, i.e., $m_3 \land n_3 \land \varphi_{err}$ is satisfiable, thus delivering a possible evidence for incorrectness.

Second abstraction refinement Similarly to the first abstraction refinement step, we construct a set of Horn clauses HC_2 to check if the discovered evidence is spurious. We consider predicates "m_1"(V), "n_1"(V), "m_2"(V), "n_2"(V), "m_3"(V), and "n_3"(V) that represent unknown sets of program states, together with "e_1"(V, V') and "e_2"(V, V') that represent unknown binary relations over program states.

$$\begin{aligned} HC_2 = \{&\varphi_{init} \rightarrow \text{“}m_1\text{”}(V),\ \text{“}m_1\text{”}(V) \land \rho_1 \rightarrow \text{“}m_2\text{”}(V'),\\ &\varphi_{init} \rightarrow \text{“}n_1\text{”}(V),\ \text{“}n_1\text{”}(V) \land \rho_2 \rightarrow \text{“}n_2\text{”}(V'),\\ &\text{“}m_1\text{”}(V) \land \rho_1 \rightarrow \text{“}e_1\text{”}(V, V'),\\ &\text{“}n_2\text{”}(V) \land \text{“}e_1\text{”}(V, V') \land \rho_2^{=} \rightarrow \text{“}n_3\text{”}(V'),\\ &\text{“}n_1\text{”}(V) \land \rho_2 \rightarrow \text{“}e_2\text{”}(V, V'),\\ &\text{“}m_2\text{”}(V) \land \text{“}e_2\text{”}(V, V') \land \rho_1^{=} \rightarrow \text{“}m_3\text{”}(V'),\\ &\text{“}m_3\text{”}(V) \land \text{“}n_3\text{”}(V) \land \varphi_{err} \rightarrow \mathit{false}\} \end{aligned}$$

The conjunction of clauses in HC_2 is satisfiable. We obtain the following satisfying assignment SOL.

$$\begin{array}{ll} \text{SOL}(\text{“}m_1\text{”}(V)) = \mathit{true} & \text{SOL}(\text{“}m_2\text{”}(V)) = (\mathtt{lock} = 1)\\ \text{SOL}(\text{“}n_1\text{”}(V)) = \mathit{true} & \text{SOL}(\text{“}n_2\text{”}(V)) = (\mathtt{lock} = 1)\\ \text{SOL}(\text{“}m_3\text{”}(V)) = \mathit{false} & \text{SOL}(\text{“}e_1\text{”}(V, V')) = (\mathtt{lock} = 0)\\ \text{SOL}(\text{“}n_3\text{”}(V)) = \mathit{false} & \text{SOL}(\text{“}e_2\text{”}(V, V')) = (\mathtt{lock} = 0) \end{array}$$

This solution constrains "m_2"(V) and "n_2"(V) to states where the lock is held $(\mathtt{lock} = 1)$, while the environment transitions "e_1"(V, V') and "e_2"(V, V') are applicable only in states for which the lock is not held by the respective thread $(\mathtt{lock} = 0)$.

We add the (transition) predicates that appear in the environment transition e_2 of the first thread to the set $\ddot{\mathcal{P}}_{2\triangleright 1}$, and, symmetrically, we add the predicates from e_1 to $\ddot{\mathcal{P}}_{1\triangleright 2}$. For the next ARET

```
// Thread 1                      // Thread 2
a: take_lock(lock, 1);           p: take_lock(lock, 2);
b: // critical                   q: // critical
```

Figure 3. Example program `LockId` .

computation we have the following set of predicates:

$$\begin{aligned}
\dot{\mathcal{P}}_1 &= \{\mathtt{pc}_1 = a, \mathtt{pc}_1 = b, \mathtt{pc}_2 = p, \mathtt{lock} = 1\}\ ,\\
\dot{\mathcal{P}}_2 &= \{\mathtt{pc}_2 = p, \mathtt{pc}_2 = q, \mathtt{pc}_1 = a, \mathtt{lock} = 1\}\ ,\\
\ddot{\mathcal{P}}_{1\triangleright 2} &= \{\mathtt{lock} = 0\}\ ,\\
\ddot{\mathcal{P}}_{2\triangleright 1} &= \{\mathtt{lock} = 0\}\ .
\end{aligned}$$

Last* ARET *computation We perform another ARET computation and a subsequent abstraction refinement step. We add the predicate $\mathtt{lock}' = 1$ to both $\ddot{\mathcal{P}}_{1\triangleright 2}$ and $\ddot{\mathcal{P}}_{2\triangleright 1}$, and proceed with the final ARET computation. Figure 2(c) shows the resulting trees. The application of thread transitions produces the following abstract states and environment transitions:

$$\begin{aligned}
m_1 &= \dot{\alpha}_1(\varphi_{init}) &&= (\mathtt{pc}_1 = a \wedge \mathtt{pc}_2 = p)\ ,\\
n_1 &= \dot{\alpha}_2(\varphi_{init}) &&= (\mathtt{pc}_1 = a \wedge \mathtt{pc}_2 = p)\ ,\\
m_2 &= \dot{\alpha}_1(post(\rho_1, m_1)) &&= (\mathtt{pc}_1 = b \wedge \mathtt{pc}_2 = p \wedge \mathtt{lock} = 1)\ ,\\
n_2 &= \dot{\alpha}_2(post(\rho_2, n_1)) &&= (\mathtt{pc}_1 = a \wedge \mathtt{pc}_2 = q \wedge \mathtt{lock} = 1)\ ,\\
e_1 &= \ddot{\alpha}_{1\triangleright 2}(m_1 \wedge \rho_1) &&= (\mathtt{lock} = 0 \wedge \mathtt{lock}' = 1)\ ,\\
e_2 &= \ddot{\alpha}_{2\triangleright 1}(n_1 \wedge \rho_2) &&= (\mathtt{lock} = 0 \wedge \mathtt{lock}' = 1)\ .
\end{aligned}$$

The environment transitions e_1 and e_2 produce the abstract states m_3 and n_3 whose intersection does not contain any error states.

$$\begin{aligned}
m_3 &= \dot{\alpha}_1(post(e_2 \wedge \rho_{\bar{1}}^{=}, m_2)) = (\mathtt{pc}_1 = a \wedge \mathtt{lock} = 1)\\
n_3 &= \dot{\alpha}_2(post(e_1 \wedge \rho_{\bar{2}}^{=}, n_2)) = (\mathtt{pc}_2 = p \wedge \mathtt{lock} = 1)
\end{aligned}$$

Neither thread nor environment transitions can be applied from the abstract states m_3 and n_3, while no further abstract states are found. Since each pair of abstract states from different threads yields an intersection that is disjoint from the error states, we conclude that `LockBit` is safe. The labeling of the computed trees can be directly used to construct a safety proof for `LockBit`, as Sections 4 and 5 will show.

2.2 Example `LockId`

Our second example `LockId`, shown in Figure 3, is a variation of `LockBit`. `LockId` uses an integer variable `lock` (instead of a single bit) to record which thread holds the lock. Due to this additional information recorded in the global variable, the example `LockId` has a modular proof, which does not refer to any local variables. We show how our algorithm discovers such a proof by only admitting modular predicates in the abstraction refinement step.

`LockId` differs from `LockBit` in its transition relation ρ_2:

$$\rho_2 = (\mathtt{lock} = 0 \wedge \mathtt{lock}' = 2 \wedge \mathtt{pc}_2 = p \wedge \mathtt{pc}_2{}' = q \wedge \rho_{\bar{1}}^{=})\ .$$

First* ARET *computation Similarly to `LockBit`, we discover that $m_2 \wedge n_2 \wedge \varphi_{err}$ is satisfiable, and compute the following set of Horn clauses:

$$\begin{aligned}
HC_3 = \{&\varphi_{init} \rightarrow \text{“}m_1\text{”}(V),\ \text{“}m_1\text{”}(V) \wedge \rho_1 \rightarrow \text{“}m_2\text{”}(V'),\\
&\varphi_{init} \rightarrow \text{“}n_1\text{”}(V),\ \text{“}n_1\text{”}(V) \wedge \rho_2 \rightarrow \text{“}n_2\text{”}(V'),\\
&\text{“}m_2\text{”}(V) \wedge \text{“}n_2\text{”}(V) \wedge \varphi_{err} \rightarrow \mathit{false}\}\ .
\end{aligned}$$

One possible satisfying assignment SOL is:

$$\begin{aligned}
&\mathrm{SOL}(\text{“}m_1\text{”}(V)) = (\mathtt{pc}_2 = p)\ , \quad \mathrm{SOL}(\text{“}n_1\text{”}(V)) = (\mathtt{pc}_1 = a)\ ,\\
&\mathrm{SOL}(\text{“}m_2\text{”}(V)) = (\mathtt{pc}_2 = p)\ , \quad \mathrm{SOL}(\text{“}n_2\text{”}(V)) = (\mathtt{pc}_1 = a)\ .
\end{aligned}$$

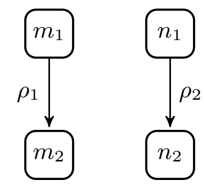


Figure 4. Tree that shows all the reachable abstract states found during the last ARET computation for `LockId`.

This assignment uses a predicate $\mathtt{pc}_2 = p$ over the local variable of the second thread as a solution for the abstract state “m_1”(V) in the thread reachability of the first thread. By collecting and using the corresponding predicates, we will discover a non-modular proof.

To avoid the drawbacks of non-modular proofs, our algorithm does not use HC_3 and attempts to find modular predicates for abstraction refinement instead. We express the preference for modular predicates declaratively, using a set of Horn clauses in which the unknown predicates are restricted to the desired variables, as described in Sections 6. For the abstract states in the first thread, we require that the corresponding solutions are over the global variable `lock` and the local variable $\mathtt{pc}_1$ of the first thread, i.e., we have the unknown predicates “m_1”$(\mathtt{lock}, \mathtt{pc}_1)$ and “m_2”$(\mathtt{lock}, \mathtt{pc}_1)$. Similarly, for the second thread we obtain “n_1”$(\mathtt{lock}, \mathtt{pc}_2)$ and “n_2”$(\mathtt{lock}, \mathtt{pc}_2)$. Instead of HC_3, we use a set of Horn clauses HC_4 shown below:

$$\begin{aligned}
HC_4 = \{&\varphi_{init} \rightarrow \text{“}m_1\text{”}(\mathtt{lock}, \mathtt{pc}_1),\\
&\text{“}m_1\text{”}(\mathtt{lock}, \mathtt{pc}_1) \wedge \rho_1 \rightarrow \text{“}m_2\text{”}(\mathtt{lock}', \mathtt{pc}_1{}'),\\
&\varphi_{init} \rightarrow \text{“}n_1\text{”}(\mathtt{lock}, \mathtt{pc}_2),\\
&\text{“}n_1\text{”}(\mathtt{lock}, \mathtt{pc}_2) \wedge \rho_2 \rightarrow \text{“}n_2\text{”}(\mathtt{lock}', \mathtt{pc}_2{}'),\\
&\text{“}m_2\text{”}(\mathtt{lock}, \mathtt{pc}_1) \wedge \text{“}n_2\text{”}(\mathtt{lock}, \mathtt{pc}_2) \wedge \varphi_{err} \rightarrow \mathit{false}\}\ .
\end{aligned}$$

The conjunction of clauses from HC_4 can be satisfied by an assignment SOL such that

$$\begin{aligned}
\mathrm{SOL}(\text{“}m_1\text{”}(\mathtt{lock}, \mathtt{pc}_1)) &= \mathit{true}\ ,\\
\mathrm{SOL}(\text{“}m_2\text{”}(\mathtt{lock}, \mathtt{pc}_1)) &= (\mathtt{lock} = 1)\ ,\\
\mathrm{SOL}(\text{“}n_1\text{”}(\mathtt{lock}, \mathtt{pc}_2)) &= \mathit{true}\ ,\\
\mathrm{SOL}(\text{“}n_2\text{”}(\mathtt{lock}, \mathtt{pc}_2)) &= (\mathtt{lock} = 2)\ ,
\end{aligned}$$

which contains only modular predicates.

Last* ARET *computation We present the last ARET computation for `LockBit`, which uses on the following (transition) predicates collected so far:

$$\begin{aligned}
\dot{\mathcal{P}}_1 &= \{\mathtt{pc}_1 = a, \mathtt{pc}_1 = b, \mathtt{lock} = 1\}\ ,\\
\dot{\mathcal{P}}_2 &= \{\mathtt{pc}_2 = p, \mathtt{pc}_2 = q, \mathtt{lock} = 2\}\ ,\\
\ddot{\mathcal{P}}_{2\triangleright 1} &= \{\mathtt{lock} = 0\}\ ,\\
\ddot{\mathcal{P}}_{1\triangleright 2} &= \{\mathtt{lock} = 0\}\ .
\end{aligned}$$

Figure 4 shows the resulting abstract reachability and environment trees constructed as follows:

$$\begin{aligned}
m_1 &= \dot{\alpha}_1(\varphi_{init}) &&= (\mathtt{pc}_1 = a)\\
n_1 &= \dot{\alpha}_2(\varphi_{init}) &&= (\mathtt{pc}_2 = p)\\
m_2 &= \dot{\alpha}_1(post(\rho_1, m_1)) &&= (\mathtt{pc}_1 = b \wedge \mathtt{lock} = 1)\\
n_2 &= \dot{\alpha}_2(post(\rho_2, n_1)) &&= (\mathtt{pc}_2 = q \wedge \mathtt{lock} = 2)\\
e_1 &= \ddot{\alpha}_{1\triangleright 2}(m_1 \wedge \rho_1) &&= (\mathtt{lock} = 0)\\
e_2 &= \ddot{\alpha}_{2\triangleright 1}(n_1 \wedge \rho_2) &&= (\mathtt{lock} = 0)
\end{aligned}$$

The ARET construction is completed, since

$$\dot{\alpha}_1(post(e_2 \wedge \rho_{\bar{1}}^{=}, m_1)) \to m_1 \ , \quad \dot{\alpha}_1(post(e_2, m_2)) = \mathit{false} \ ,$$
$$\dot{\alpha}_2(post(e_1 \wedge \rho_{\bar{2}}^{=}, n_1)) \to n_1 \ , \quad \dot{\alpha}_2(post(e_1, n_2)) = \mathit{false} \ .$$

By inspecting pairs of abstract states from different trees we conclude that `LockId` is safe.

Furthermore no predicate in $\dot{\mathcal{P}}_1$ refers to the local variable of the second thread, the symmetric condition holds for $\dot{\mathcal{P}}_2$, and the predicates in $\ddot{\mathcal{P}}_{1\triangleright 2}$ as well as $\ddot{\mathcal{P}}_{2\triangleright 1}$ do not refer to any local variables. Thus, from the trees in Figure 4 we can construct a modular safety proof.

3. Preliminaries

In this section we briefly describe multi-threaded programs, their computations and correctness. We also introduce auxiliary definitions that we apply for reasoning about programs.

Programs We consider a *multi-threaded* program P that consists of $\mathsf{N} \geq 1$ concurrent threads. Let $1..\mathsf{N}$ be the set $\{1, \dots, \mathsf{N}\}$. We assume that the *program* variables $V = (V_G, V_1, \dots, V_\mathsf{N})$ are partitioned into *global* variables V_G that are shared by all threads, and *local* variables $V_1, \dots, V_\mathsf{N}$ that are only accessible by the threads $1, \dots, \mathsf{N}$, respectively.

The set of *global states* G consists of the valuations of global variables, and the sets of *local states* $L_1, \dots, L_\mathsf{N}$ consist of the valuations of the local variables of respective threads. By taking the product of the global and local state spaces, we obtain the set of *program states* $\Sigma = G \times L_1 \times \cdots \times L_\mathsf{N}$. We represent sets of program states using assertions over program variables. Binary relations between sets of program states are represented using assertions over unprimed and primed variables. Let $\models$ denote the satisfaction relation between (pairs) of states and assertions.

The set of *initial* program states is denoted by φ_{init} , and the set of *error* states is denoted by φ_{err} . For each thread $i \in 1..\mathsf{N}$ we have a finite set of transition relations $\mathcal{T}_i$, which are abbreviated as *transitions*. Each transition $\rho \in \mathcal{T}_i$ can change the values of the global variables and the local variables of the thread i . Let $\rho_i^{=}$ be a constraint requiring that the local variables of the thread i do not change, i.e., $\rho_i^{=} = (V_i = V_i')$. Then, $\rho \in \mathcal{T}_i$ has the form

$$\rho^{\text{update}}(V_G, V_i, V_G', V_i') \wedge \bigwedge_{j \in 1..\mathsf{N} \setminus \{i\}} \rho_j^{=} \ ,$$

where the first conjunct represents the update of the variables in the scope of the thread i and the remaining conjuncts ensure that the local variables of other threads do not change. We write ρ_i for the union of the transitions of the thread i , i.e., $\rho_i = \bigvee \mathcal{T}_i$. The transition relation of the program is $\rho_\mathcal{T} = \rho_1 \vee \cdots \vee \rho_\mathsf{N}$.

Computations A *computation* of P is a sequence of program states $s_1, s_2, \dots$ such that s_1 is an initial state, i.e., $s_1 \models \varphi_{init}$, and each pair of consecutive states s_i and s_{i+1} in the sequence is connected by some transition ρ from a program thread, i.e., $(s_i, s_{i+1}) \models \rho$. A *path* is a sequence of transitions. We write ϵ for the *empty* sequence.

Let $[z/w]$ be a substitution function such that $\varphi[z/w]$ replaces w by z in φ . Let $\circ$ be the *relational composition* function for binary relations given by assertions over unprimed and primed variables such that for assertions φ and ψ we have $\varphi \circ \psi = \exists V'' : \varphi[V''/V'] \wedge \psi[V''/V]$. Then, a *path relation* is a relational composition of transition relations along the path, i.e., for $\pi = \rho_1 \cdots \rho_n$ we have $\rho_\pi = \rho_1 \circ \ldots \circ \rho_n$. A path π is *feasible* if its path relation is not empty, i.e., $\exists V, V' : \rho_\pi$.

A program state is *reachable* if it appears in some computation. Let φ_{reach} denote the set of reachable states. The program is *safe* if *none* of its error states is reachable, i.e., $\varphi_{reach} \wedge \varphi_{err} \to \mathit{false}$.

For assertions $R_1, \dots, R_\mathsf{N}$ over V ,
and $E_1, \dots, E_\mathsf{N}$ over V and V'

CS1:	φ_{init}	$\to R_i$	for $i \in 1..\mathsf{N}$
CS2:	$R_i \wedge \rho_i$	$\to R_i'$	for $i \in 1..\mathsf{N}$
CS3:	$R_i \wedge E_i \wedge \rho_i^{=}$	$\to R_i'$	for $i \in 1..\mathsf{N}$
CS4:	$(\bigvee_{i \in 1..\mathsf{N}\setminus\{j\}} R_i \wedge \rho_i)$	$\to E_j$	for $j \in 1..\mathsf{N}$
CS5:	$R_1 \wedge \cdots \wedge R_\mathsf{N} \wedge \varphi_{err}$	$\to \mathit{false}$	

program P is safe

Figure 5. Proof rule REACHENV for compositional safety proofs of multi-threaded programs. R_i' stands for $R_i[V'/V]$. REACHENV yields a pre-fixpoint characterization through Equations (1).

Auxiliary definitions We define a *successor* function *post* such that for a binary relation over states ρ and a set of states φ we have

$$post(\rho, \varphi) \ = \ \exists V'' : \varphi[V''/V] \wedge \rho[V''/V][V/V'] \ .$$

We also extend the logical implication to tuples of equal length, i.e.,

$$(\varphi_1, \dots, \varphi_n) \to (\psi_1, \dots, \psi_n) \ = \ \varphi_1 \to \phi_1 \wedge \cdots \wedge \varphi_n \to \phi_n \ ,$$

where each implication is implicitly universally quantified over the free variables occurring in it. From now on, we assume that tuples of assertions are partially ordered by the above extension of $\to$.

A *Horn clause* $b_1(w_1) \wedge \cdots \wedge b_n(w_n) \to b(w)$ consists of relation symbols $b_1, \dots, b_n$, b, and vectors of variables $w_1, \dots, w_n$, w. For the algorithm SOLVELINEARHC in Section 7 we only consider Horn clauses over linear arithmetic. We say that b *depends* on the relation symbols $\{b_i \mid i \in 1..n \wedge b_i \neq (\leq)\}$. A set of Horn clauses is *recursion-free* if the transitive closure of the corresponding dependency relation is well founded.

4. Proof rule, fixpoints, and abstraction

In this section we develop the foundations for our verification algorithm. We present a compositional proof rule and then derive a corresponding characterization in terms of least fixpoints and their approximations. We present the ability of our proof rule to facilitate modular reasoning, when admitted by the program, without losing the ability for global reasoning otherwise.

Proof rule Figure 5 presents a proof rule REACHENV for compositional verification of program safety. The proof rule is inspired by the existing proof rules for compositional safety reasoning, see e.g. [5, 15, 16, 26]. Our formulation of REACHENV directly leads to a pre-fixpoint characterization, thus, providing a basis for the proof rule automation using abstraction and refinement techniques.

REACHENV relies on *thread reachability* assertions $R_1, \dots, R_\mathsf{N}$ that keep track of program states reached by threads $1, \dots, \mathsf{N}$ together with their respective *environment transitions* $E_1, \dots, E_\mathsf{N}$. The environment transition of each thread keeps track of modifications of program states by other threads. The auxiliary assertions used in our proof rule can refer to all program variables, that is, they are not restricted to a combination of global variables and local variables of a particular thread.

If the provided auxiliary assertions satisfy all premises of the proof rule, i.e., CS1, …, CS5, then the program is safe. The premise CS1 requires that each thread reachability over-approximates the initial program states. CS2 ensures that the thread reachability of each thread is invariant under the application of the thread transitions. In addition, CS3 requires invariance under the environment transitions of the thread. The conjunct $\rho_i^{=}$ in CS3 selects the subset of the environment transition that does not modify the local variables of the thread. Given a thread j, the premise CS4 collects transitions that start from states in the thread reachability of other threads and combines them into the environment transition for j. Finally, CS5 checks that there is no error state that appears in each thread reachability set.

The proof rule REACHENV can be directly used to prove program safety following a two-step procedure. First, we need to identify candidate assertions for the thread reachability and environment transitions. Second, we need to check that these candidate assertions satisfy the premises of proof rule. The correctness of the conclusion is formalized by the following theorems.

THEOREM 1 (Soundness). *The proof rule* REACHENV *is sound.* ■

Proof. Let $R_1, \ldots, R_{\mathsf{N}}$ and $E_1, \ldots, E_{\mathsf{N}}$ satisfy the premises CS1, ..., CS5. We show that the program is safe. To prove safety, for each reachable state $s \models \varphi_{reach}$ we prove that $s \models R_1 \wedge \cdots \wedge R_{\mathsf{N}}$ by induction over the length k of a shortest computation segment $s_1, \ldots, s_k$ such that $s_1 \models \varphi_{init}$ and $s_k = s$.

For the base case $k = 1$, the inclusion holds due to the premise CS1. For the induction step, we assume that the above statement holds for states reachable in $k \geq 1$ steps and prove the statement for their immediate successors. That is, let $s_k \models \varphi_{reach}$ and hence $s_k \models R_1 \wedge \cdots \wedge R_{\mathsf{N}}$. If s_k does not have any successor, i.e., $\neg(\exists s_{k+1} : (s_k, s_{k+1}) \models \rho_{\mathcal{T}})$, then there are no more states to consider. Otherwise, we choose a successor state s_{k+1} of s_k that is reached by taking a transition in a thread i, i.e., $(s_k, s_{k+1}) \models \rho_i$. From CS2 follows that $s_{k+1} \models R_i$.

To show that $s_{k+1} \models R_j$ for each $j \in 1..\mathsf{N} \setminus \{i\}$ we rely on the premises CS4 and CS3. By induction hypothesis $s_k \models R_i$ and due to CS4 we have $(s_k, s_{k+1}) \models E_j$. Now $s_{k+1} \in R_j$ follows from CS3. □

THEOREM 2 (Relative completeness). *The proof rule* REACHENV *is complete relative to first-order reasoning.* ■

Proof. Let P be safe. We define $R_1 = \cdots = R_{\mathsf{N}} = \varphi_{reach}$ and $E_1 = \cdots = E_{\mathsf{N}} = \varphi_{reach} \wedge \rho_{\mathcal{T}}$. Then, the premises CS1, ..., CS5 are immediately satisfied. □

Modular and global proofs Reasoning about multi-threaded program is more complex than reasoning about sequential programs since thread interaction needs to be taken into account. Some programs admit modular reasoning that deals with each thread in isolation, i.e., assertions used in the proof only refer to the global variables and the local variables of one thread at a time.

The proof rule REACHENV facilitates *modular* reasoning about multi-threaded programs. If a program has a modular safety proof, then the following *modular* assertions satisfy the proof rule premises:

$$R_i = \exists V \setminus (V_G \cup V_i) : \varphi_{reach}\ , \qquad \text{for } i \in 1..\mathsf{N}$$
$$E_i = \exists (V \cup V') \setminus (V_G \cup V'_G) : \varphi_{reach} \wedge \rho_{\mathcal{T}}\ . \qquad \text{for } i \in 1..\mathsf{N}$$

REACHENV is *not* restricted to modular proofs. Since the assertions used in REACHENV can refer to each of the program variables, non-modular proofs can be directly used. In fact, the proof of Theorem 2 relies on non-modular assertions, since φ_{reach} may refer to local variables of different threads.

In Section 5 we will present our algorithm that can discover modular assertions for REACHENV if the program admits modular proofs, and delivers non-modular assertions otherwise.

Fixpoints The proof rule REACHENV in Figure 5 directly leads to a fixpoint-based characterization, which defines our algorithm in Section 5.

From the premises CS2, CS3, and CS4 we obtain a function F on N-tuples of assertions over the program variables and N-tuples of assertions over the unprimed and primed program variables such that

$$\begin{aligned} &F(S_1, \ldots, S_{\mathsf{N}}, T_1, \ldots, T_{\mathsf{N}}) \ = \\ &\quad (post(\rho_1 \vee T_1 \wedge \rho_1^{=}, S_1),\ \ldots,\ post(\rho_{\mathsf{N}} \vee T_{\mathsf{N}} \wedge \rho_{\mathsf{N}}^{=}, S_{\mathsf{N}}), \\ &\quad \textstyle\bigvee_{i \in 1..\mathsf{N} \setminus \{1\}} S_i \wedge \rho_i, \qquad \ldots, \bigvee_{i \in 1..\mathsf{N} \setminus \{\mathsf{N}\}} S_i \wedge \rho_i)\ . \end{aligned} \tag{1}$$

We formalize the relation between F and REACHENV as follows.

LEMMA 1. *Each pre-fixpoint of F satisfies the premises* CS2, CS3, *and* CS4 *of* REACHENV. *That is, if*

$$F(R_1, \ldots, R_{\mathsf{N}}, E_1, \ldots, E_{\mathsf{N}}) \rightarrow (R_1, \ldots, R_{\mathsf{N}}, E_1, \ldots, E_{\mathsf{N}})$$

then $R_1, \ldots, R_{\mathsf{N}}, E_1, \ldots, E_{\mathsf{N}}$ satisfies CS2, CS3, *and* CS4. ■

We define a distinguished tuple $\bot_F$:

$$\bot_F \ = \ (\varphi_{init}, \ldots, \varphi_{init}, \underbrace{\mathit{false}, \ldots, \mathit{false}}_{\mathsf{N}\text{ times}})\ . \tag{2}$$

Then, each pre-fixpoint of F that is greater than $\bot_F$ satisfies the premise CS1. By choosing a pre-fixpoint $(R_1, \ldots, R_{\mathsf{N}}, E_1, \ldots, E_{\mathsf{N}})$ above $\bot_F$ such that $R_1 \wedge \ldots, R_{\mathsf{N}} \wedge \varphi_{err} \rightarrow \mathit{false}$ we will satisfy all premises of the proof rule REACHENV, and hence prove the program safety.

Fixpoint abstraction Computing pre-fixpoints of F that satisfy CS1 and CS5 is a difficult task. We automate this computation using the framework of abstract interpretation [7], which uses over-approximation to strike a balance between reasoning precision and efficiency. To implement required over-approximation functions, we will use a collection of abstraction functions $\dot\alpha_i$ and $\ddot\alpha_{i \triangleright j}$, where $i \neq j \in 1..\mathsf{N}$, that over-approximate sets and binary relations over programs states, respectively.

We define a function $F^\#$ that over-approximates F using given abstraction functions:

$$\begin{aligned} &F^\#(S_1, \ldots, S_{\mathsf{N}}, T_1, \ldots, T_{\mathsf{N}}) \ = \\ &\quad (\dot\alpha_1(post(\rho_1, S_1)) \vee \dot\alpha_1(post(T_1 \wedge \rho_1^{=}, S_1))\ , \\ &\qquad \ldots \\ &\quad \dot\alpha_{\mathsf{N}}(post(\rho_{\mathsf{N}}, S_{\mathsf{N}})) \vee \dot\alpha_{\mathsf{N}}(post(T_{\mathsf{N}} \wedge \rho_{\mathsf{N}}^{=}, S_{\mathsf{N}}))\ , \\ &\quad \textstyle\bigvee_{i \in 1..\mathsf{N} \setminus \{1\}} \ddot\alpha_{i \triangleright 1}(S_i \wedge \rho_i)\ , \\ &\qquad \ldots \\ &\quad \textstyle\bigvee_{i \in 1..\mathsf{N} \setminus \{\mathsf{N}\}} \ddot\alpha_{i \triangleright \mathsf{N}}(S_i \wedge \rho_i))\ . \end{aligned} \tag{3}$$

Let $\bot_{F^\#}$ be an over-approximation of F such that

$$\bot_{F^\#} \ = \ (\dot\alpha_1(\varphi_{init}), \ldots, \dot\alpha_{\mathsf{N}}(\varphi_{init}), \underbrace{\mathit{false}, \ldots, \mathit{false}}_{\mathsf{N}\text{ times}})\ . \tag{4}$$

The least pre-fixpoint of $F^\#$ above $\bot_{F^\#}$ can be used to prove program safety by applying the following theorem, and is the key outcome of our algorithm in Section 5.

THEOREM 3 (Abstract fixpoint checking). *If the least pre-fixpoint of $F^\#$ above $\bot_{F^\#}$, say $(R_1, \ldots, R_{\mathsf{N}}, E_1, \ldots, E_{\mathsf{N}})$, satisfies the premise* CS5 *then the program is safe.* ■

Proof. The theorem follows directly from the soundness of the proof rule REACHENV, Lemma 1, and over-approximations introduced by the applied abstraction functions. □

function MAIN
input
P - program with N threads
vars
$\dot{\mathcal{P}}_i, \dot{\alpha}_i$ - predicates for thread i and corresponding state abstraction function
$\ddot{\mathcal{P}}_{i \triangleright j}, \ddot{\alpha}_{i \triangleright j}$ - transition predicates for pair of threads i, j and corresponding transition abstraction function
$\mathcal{R}_i$ - abstract states of thread i
$\mathcal{E}_i$ - abstract environment transitions of thread i
$Parent$ - parent function for abstract states and environment transitions
$ParentTId$ - parent thread function for abstract states and environment transitions
begin
1 **for** each $i \neq j \in 1..\mathsf{N}$ **do**
2 $\quad \dot{\mathcal{P}}_i := \ddot{\mathcal{P}}_{i \triangleright j} := \emptyset$
3 **repeat**
4 $\quad$ **for** each $i \neq j \in 1..\mathsf{N}$ **do**
5 $\quad\quad \dot{\alpha}_i := \lambda S. \bigwedge \{\dot{p} \in \dot{\mathcal{P}}_i \mid \forall V : S \rightarrow \dot{p}\}$
6 $\quad\quad \ddot{\alpha}_{i \triangleright j} := \lambda T. \bigwedge \{\ddot{p} \in \ddot{\mathcal{P}}_{i \triangleright j} \mid \forall V, V' : T \rightarrow \ddot{p}\}$
7 $\quad$ ABSTREACHENV()
8 $\quad$ **if** exists $S_1 \in \mathcal{R}_1, \dots, S_{\mathsf{N}} \in \mathcal{R}_{\mathsf{N}}$ such that
9 $\quad\quad \exists V : S_1 \wedge \dots \wedge S_{\mathsf{N}} \wedge \varphi_{err}$
10 $\quad$ **then**
11 $\quad\quad$ **try**
12 $\quad\quad\quad$ REFINE$(S_1, \dots, S_{\mathsf{N}})$
13 $\quad\quad$ **with** UNSATISFIABLE
14 $\quad\quad\quad D :=$ some S_i from $\{S_1, \dots, S_{\mathsf{N}}\}$
15 $\quad\quad\quad$ **return** "counterexample MKPATH(D)"
16 $\quad$ **else**
17 $\quad\quad$ **return** "program P is safe with the proof
18 $\quad\quad\quad \bigvee \mathcal{R}_1, \dots, \bigvee \mathcal{R}_{\mathsf{N}}, \bigvee \mathcal{E}_1, \dots, \bigvee \mathcal{E}_{\mathsf{N}}$"
19 **until** $true$
end.

Figure 6. Function MAIN for verifying safety of the multi-threaded program P.

The choice of the abstract domains, i.e., the range sets of the abstraction functions, determines if the least fixpoint of $F^\#$ yields a modular proof. Our abstraction discovery algorithm in Section 6 automatically chooses the abstraction such that modular proofs are preferred.

5. Thread reachability and environment transitions

In this section, we present our rely-guarantee based verification algorithm for proving safety properties of multi-threaded programs. The algorithm is based on Theorem 3 and consists of three main steps. The first step computes for each thread a tree that is decorated by abstract states and environment transitions, so-called ARET, and analyses the discovered abstract states. If an intersection with the error states of the program is found, then the second step generates a set of corresponding Horn clauses, see Section 6. At the third step, we solve the constraint defined by the conjunction of the generated Horn clauses and use the solutions to the refine the abstraction functions used for the ARET computation, see Section 7.

procedure ABSTREACHENV
begin
1 $Parent := ParentTId := \bot$ $\quad$ ($*$ the empty function $*$)
2 **for** each $i \in 1..\mathsf{N}$ **do**
3 $\quad \mathcal{R}_i := \{\dot{\alpha}_i(\varphi_{init})\}$
4 $\quad \mathcal{E}_i := \emptyset$
5 **repeat**
6 $\quad finished := true$
7 $\quad$ **for** each $i \in 1..\mathsf{N}$ and $S \in \mathcal{R}_i$ **do**
8 $\quad\quad$ ($*$ states $*$)
9 $\quad\quad$ **for** each $\rho \in \mathcal{T}_i \cup \mathcal{E}_i$ **do**
10 $\quad\quad\quad S' :=$ **if** $\rho \in \mathcal{T}_i$ **then** $\dot{\alpha}_i(post(\rho, S))$
11 $\quad\quad\quad\quad$ **else** $\dot{\alpha}_i(post(\rho \wedge \rho_i^{=}, S))$
12 $\quad\quad\quad$ **if** $\neg(\exists S'' \in \mathcal{R}_i\ \forall V : S' \rightarrow S'')$ **then**
13 $\quad\quad\quad\quad \mathcal{R}_i := \{S'\} \cup \mathcal{R}_i$
14 $\quad\quad\quad\quad Parent(S') := (S, \rho)$
15 $\quad\quad\quad\quad ParentTId(S') := i$
16 $\quad\quad\quad\quad finished := false$
17 $\quad\quad$ **done**
18 $\quad\quad$ ($*$ environment transitions $*$)
19 $\quad\quad$ **for** each $\rho \in \mathcal{T}_i$ and $j \in 1..\mathsf{N} \setminus \{i\}$ **do**
20 $\quad\quad\quad \rho' := \ddot{\alpha}_{i \triangleright j}(S \wedge \rho)$
21 $\quad\quad\quad$ **if** $\neg(\exists \rho'' \in \mathcal{T}_j \cup \mathcal{E}_j\ \forall V, V' : \rho' \rightarrow \rho'')$ **then**
22 $\quad\quad\quad\quad \mathcal{E}_j := \{\rho'\} \cup \mathcal{E}_j$
23 $\quad\quad\quad\quad Parent(\rho') := (S, \rho)$
24 $\quad\quad\quad\quad ParentTId(\rho') := i$
25 $\quad\quad\quad\quad finished := false$
26 $\quad\quad$ **done**
27 $\quad$ **done**
28 **until** $finished$
end

Figure 7. Procedure ABSTREACHENV implements ARET computation. We assume that the iterator statements in lines 7 and 9 make an immutable snapshot of their domains $\mathcal{R}_i$ and $\mathcal{E}_i$, respectively. For example, this implies that each addition of S' in line 12 is unnoticed in line 7 until the next iteration of the **repeat** loop.

***Function* MAIN** The main function of our algorithm MAIN is shown in Figure 6. MAIN takes as input the multi-threaded program P . The **repeat** loop iterates through the three main steps of the algorithm. First, we construct the abstraction functions $\dot{\alpha}_i$ and $\ddot{\alpha}_{i \triangleright j}$ at lines 4–6 from a given set of (transition) predicates, which is empty initially. Next, the ARET computation is performed in line 7 using these abstraction functions. In lines 8–9, the abstract states in the computed ARET's are analyzed wrt. the safety property. In case of a positive outcome of this check, MAIN constructs and returns a safety proof in lines 17–18. If the safety check fails, then REFINE is executed on the violating abstract states. If REFINE terminates normally, and hence succeeds in eliminating the violation by refining the abstraction functions, then MAIN continues with the next iteration of the **repeat** loop. In case an UNSATISFIABLE exception is thrown, MKPATH from Figure 8 constructs a counterexample path that we report to the user.

***Procedure* ABSTREACHENV** See Figure 7 for the procedure ABSTREACHENV that implements ARET computation using the abstraction functions $\dot{\alpha}_i$ and $\ddot{\alpha}_{i \triangleright j}$. We use $Parent$ and $ParentTId$ to maintain information about the constructed trees, and initialize

```
function MkPath
input
  D - abstract state
begin
  match Parent(D) with
   | (S, ρ) ->
     π := MkPath(S)
     match Parent(ρ) with
      | (O, ρ_O) -> return π · ρ_O
      | ⊥ -> return π · ρ
   | ⊥ -> return ε                    (* the empty sequence *)
end
```

Figure 8. Function MkPath takes as input an abstract state D and returns a sequence of transitions that lead to D.

them with the empty function $\perp$ in line 1. $\mathcal{R}_i$ and $\mathcal{E}_i$ keep track of abstract states and environment transitions for a thread $i \in 1..\mathsf{N}$. $\mathcal{E}_i$ is initialized to an empty set in line 4, while $\mathcal{R}_i$ contains the abstraction of the initial program states computed for the thread i . The Aret computation is performed iteratively in the **repeat** loop, see lines 5–27.

The first part of the loop (see lines 7–17) implements a standard, least fixpoint computation over reachable states. At line 7, the algorithm picks an already reachable states $S \in \mathcal{R}_i$ in order to compute its abstract successors. After computing at lines 10–11 one successor of S , line 12 implements a fixpoint check, which succeeds if S' contains program states that have not been reached yet. The new states reachable in thread i are stored in $\mathcal{R}_i$. At line 14, the function Parent is updated to keep track of the child-parent relation between abstract states, while $\mathit{ParentTId}$ maps the new reachable state to its parent thread.

The second part of the loop (see lines 19–26) performs a least fixpoint computation over environment transitions. Each time a transition from a thread i is picked at line 7, the abstraction of its effect computed in line 20 is propagated to each other thread j . Note however, that the propagation only happens for environment transitions that are not subsumed by the existing ones, which is checked in line 21. Additional environment transitions are recorded in line 22. Environment transitions are taken into consideration when computing abstract state reachability, see line 9.

Upon termination, which is guaranteed by the finiteness of our abstract domains, the function AbstReachEnv computes sets of abstract states $\mathcal{R}_1, \ldots, \mathcal{R}_\mathsf{N}$ and sets of environment transitions $\mathcal{E}_1, \ldots, \mathcal{E}_\mathsf{N}$.

6. Abstraction refinement

***Procedure* Refine** In Figure 9, we present the procedure Refine that takes as argument an error tuple and, if possible, refines the abstraction functions to include predicates that witness the fact that the error state is unreachable. The procedure Refine generates a set of Horn clauses corresponding to the error tuple (lines 1–4). Next, the Refine algorithm invokes a solving procedure for Horn clauses (lines 5–7). Lastly, the procedure Refine updates the abstraction functions using the solution of Horn clauses at lines 8–12. We consider the solution Sol and add the atomic predicates that appear in Sol("S"(V) to the set of predicates $\dot{\mathcal{P}}_i$. The index i is chosen to be that of the thread where S originated from. Similarly, the procedure updates the transition abstraction functions at line 12. Here, we only assume that SolveHC returns a correct solution to the set of Horn clauses received as argument. In

```
   procedure Refine
   input
     S_1, ..., S_N - abstract error tuple
   begin
1    HC := {"S_1"(V) ∧ ··· ∧ "S_N"(V) ∧ φ_err → false}
2           ∪ MkHornClauses(S_1)
3           ...
4           ∪ MkHornClauses(S_N)
5    Sol := SolveHC(HC,
6                  {"S"(V) | i ∈ 1..N ∧ S ∈ R_i} ∪
7                  {"ρ"(V, V') | i ∈ 1..N ∧ ρ ∈ E_i})
8    for each i ∈ 1..N and S ∈ R_i do
9      Ṗ_i := PredsOf(Sol("S"(V))) ∪ Ṗ_i
10   for each j ∈ 1..N and ρ ∈ E_j do
11     i := ParentTId(ρ)
12     P̈_{i▷j} := PredsOf(Sol("ρ"(V, V'))) ∪ P̈_{i▷j}
   end
```

Figure 9. Procedure Refine. The quotation function "·" creates a relation symbol from a given abstract state/abstract transition. The function $\mathit{PredsOf}$ extracts atomic predicates from the solutions to the set of Horn clauses *HC*.

```
function MkHornClauses
input
  D - abstract state
begin
  i := ParentTId(D)
  return
    match Parent(D) with
     | (S, ρ) ->
       MkHornClauses(S) ∪
       begin
         match Parent(ρ) with
          | (O, ρ_O) ->
            {"O"(V) ∧ ρ_O → "ρ"(V, V') ,
             "S"(V) ∧ "ρ"(V, V') ∧ ρ_i^= → "D"(V')}
            ∪ MkHornClauses(O)
          | ⊥ -> {"S"(V) ∧ ρ → "D"(V')}
       end
     | ⊥ -> {φ_init → "D"(V)}
end
```

Figure 10. Function MkHornClauses.

Section 7, we present a solving algorithm for recursion-free Horn clauses over the linear arithmetic domain.

***Function* MkHornClauses** The generation of the Horn clauses is started from lines 1–4 of Figure 9. One clause requires that the solutions corresponding to the abstract states from the error tuple do not intersect φ_{err} : $\{``S_1"(V) \wedge \cdots \wedge ``S_\mathsf{N}"(V) \wedge \varphi_{err} \rightarrow \mathit{false}\}$. The other clauses are generated by invoking $\textsc{MkHornClauses}(S_i)$ for $i \in 1..\mathsf{N}$. The function MkHornClauses generates Horn clauses for transitions considered during Aret computation as follows. If the abstract state D was produced by following a local transition, i.e., $\mathit{Parent}(D) =$

```
9         {"O"(V_G, V_i) ∧ ρ_O → "ρ"(V_G, V'_G) ,
10         "S"(V_G, V_i) ∧ "ρ"(V_G, V'_G) ∧
11         ρ_i^= → "D"(V'_G, V'_i)}
12       ∪ MkHornClauses(O)
13     | ⊥ -> {"S"(V_G, V_i) ∧ ρ → "D"(V'_G, V'_i)}
14   end
15 | ⊥ -> {φ_init → "D"(V_G, V_i)}
```

Figure 11. Modifications to MKHORNCLAUSES to admit only modular solutions.

```
// Thread 1                    // Thread 2
x:  cnt++;                     p: take_lock(lock,1);
    assume(cnt >= 1);          q: // critical
a:  take_lock(lock,1);
b:  // critical
```

$$\begin{aligned}
\varphi_{init} &= (\mathtt{pc}_1 = x \wedge \mathtt{pc}_2 = p \wedge \mathtt{lock} = 0 \wedge \mathtt{cnt} = 0)\\
\rho_1^{=} &= (\mathtt{pc}_1 = \mathtt{pc}_1{}' \wedge \mathtt{cnt} = \mathtt{cnt}')\\
\rho_2^{=} &= (\mathtt{pc}_2 = \mathtt{pc}_2{}')\\
\rho_0 &= (\mathtt{pc}_1 = x \wedge \mathtt{pc}_1{}' = a \wedge \mathtt{lock} = \mathtt{lock}' \wedge\\
&\qquad \mathtt{cnt} + 1 = \mathtt{cnt}' \wedge 1 \le \mathtt{cnt}' \wedge \rho_2^{=})\\
\rho_1 &= (\mathtt{pc}_1 = a \wedge \mathtt{pc}_1{}' = b \wedge \mathtt{lock} = 0 \wedge \mathtt{lock}' = 1 \wedge\\
&\qquad \mathtt{cnt} = \mathtt{cnt}' \wedge \rho_2^{=})\\
\rho_2 &= (\mathtt{pc}_2 = p \wedge \mathtt{pc}_2{}' = q \wedge \mathtt{lock} = 0 \wedge \mathtt{lock}' = 1 \wedge \rho_1^{=})\\
\varphi_{err} &= (\mathtt{pc}_1 = b \wedge \mathtt{pc}_2 = q)
\end{aligned}$$

Figure 12. `LockBitCnt` is an expanded version of the program `LockBit` .

(S, ρ) and $Parent(\rho) = \bot$, then one Horn clause corresponds to the application of the local transition at line 12: $\{$"S"$(V) \wedge \rho \rightarrow$ "D"$(V')\}$. Additional Horn clauses are generated recursively for the parent state S at line 5. If the abstract state D was produced by following an environment transition, i.e., $Parent(D) = (S, \rho)$ and $Parent(\rho) = (O, \rho_O)$, then two Horn clauses correspond to the generation of the environment transition (line 9) and to the application of the environment transition (line 10). Finally, if $Parent(D) = \bot$, then one Horn clause constrains the solution of "D"(V) at line 14: $\{\varphi_{init} \rightarrow$ "D"$(V)\}$. Note that solutions for unknown states are expressed in terms of all program variables V, while solutions for unknown transitions are expressed in terms of V and V' . Consequently, these solutions may lead to non-modular proofs even for a set of Horn clauses that has modular solutions.

Discovery of Modular Predicates We present modifications to our abstraction refinement algorithm that guarantee the discovery of modular solutions whenever they exist. With these modifications, solutions for unknown states originating in thread i can only be expressed in terms of V_G, V_i rather than the whole set of program variables V . Solutions for unknown transitions are restricted to the set of global variables V_G, V'_G . To implement these changes, we change line 1 from the REFINE procedure as follows:

```
1   HC := {"S_1"(V_G, V_1) ∧ ··· ∧ "S_N"(V_G, V_N) ∧ φ_err}
```

We also replace lines 9–14 from MKHORNCLAUSES with the fragment shown in Figure 11. The rest of the function MKHORNCLAUSES is unchanged. If the resulting Horn clauses have no solution, i.e., SOLVEHC throws an UNSATISFIABLE exception, then it may still be possible that a non-modular solution exists. In this case, we invoke the abstraction refinement once again, this time generating Horn clauses using the unmodified function MKHORNCLAUSES from Figure 10.

Example We illustrate the generation of Horn clauses using an expanded version of `LockBit` shown in Figure 12. This example contains an additional variable `cnt` local to the first thread. The initial symbolic state of the program φ_{init} constrains both `cnt` and `lock` to the value 0 . The transition relation of the first thread is extended with ρ_0, which increments `cnt` by 1 and assumes that the incremented value is greater than or equal to 1 . Similar to the example from Section 2, φ_{err} encodes the violation of the mutual exclusion property. We show in Figure 13(a) the reachability trees as computed by the ARET computation. The error tuple consists of m_4 and n_3 , i.e., $m_4 \wedge n_3 \wedge \varphi_{err}$ is satisfiable.

From this error tuple, MKHORNCLAUSES generates Horn clauses following the procedure from Figure 10. These Horn clauses are shown in Figure 13(b). The Horn clauses have unknown states "m_1"(V), "m_2"(V), "m_3"(V), "m_4"(V), "n_1"(V), "n_2"(V) , and "n_3"(V) . The unknown transitions are "e_1"(V, V') and "e_2"(V, V') .

Comparatively, we show in Figure 13(c) the Horn clauses generated with preference for modular solutions. The solutions for the unknown states of thread 1 can refer only to (V_G, V_1) , while the unknown states of thread 2 are restricted to (V_G, V_1) . The unknown transitions are "e_1"(V_G, V'_G) and "e_2"(V_G, V'_G) .

THEOREM 4 (Progress of abstraction refinement). *The procedure* REFINE *guarantees progress of abstraction refinement, i.e., the same set of Horn clauses is never discovered twice.* ■

7. Solving Horn clauses over linear inequalities

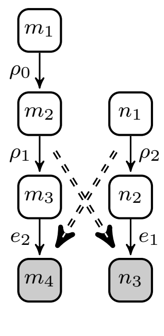
As presented in the previous section, REFINE calls the function SOLVEHC. In this section we present a function SOLVELINEARHC that can be used as an implementation of SOLVEHC takes as input a set of clauses *HC* over linear inequalities that is recursion-free.

To simplify the presentation of the algorithm, we make two additional assumptions on *HC*. First, we assume that for each pair of clauses $(\ldots) \rightarrow b(w)$ and $(\ldots) \rightarrow b'(w')$ from *HC* we have $b \neq b'$ and $b \neq (\le) \neq b'$. Second, we assume that *HC* contains a clause $(\ldots) \rightarrow \mathit{false}$.

The additional assumptions are satisfied by the clauses generated in Section 6. In case SOLVELINEARHC is applied on a set of recursion-free Horn clauses over linear arithmetic that violates the two assumptions above, we can apply a certain renaming of relation symbols and introduction of additional clauses to meet the assumptions.

***Function* MKTREE** The function MKTREE generates a tree representation for a set of Horn clauses and is shown in Figure 15. For every relation appearing in the Horn clauses, the algorithm generates a corresponding tree node. The children of a node are maintained in a function *Children* as follows. Nodes that correspond to linear arithmetic relations have no children, see lines 11–12. A node that corresponds to an unknown relation with a relation symbol b has as children those nodes that represent relation symbols that depend on b . The *Label* attribute of the tree nodes is initialized to a linear arithmetic constraint for leaves of the tree in line 11, and to an unknown relation for internal tree nodes in line 13.

***Function* SOLVELINEARHC** See Figure 14 for the pseudocode of the procedure SOLVELINEARHC . This procedure creates a tree representation for *HC* at line 2. At line 3, we build a set containing



(a)

$$\varphi_{init} \rightarrow \text{“}m_1\text{”}(V)$$
$$\text{“}m_1\text{”}(V) \wedge \rho_0 \rightarrow \text{“}m_2\text{”}(V')$$
$$\text{“}m_2\text{”}(V) \wedge \rho_1 \rightarrow \text{“}m_3\text{”}(V')$$
$$\text{“}m_2\text{”}(V) \wedge \rho_1 \rightarrow \text{“}e_1\text{”}(V, V')$$
$$\text{“}m_3\text{”}(V) \wedge \text{“}e_2\text{”}(V, V') \wedge \rho_1^{=} \rightarrow \text{“}m_4\text{”}(V')$$

$$\varphi_{init} \rightarrow \text{“}n_1\text{”}(V)$$
$$\text{“}n_1\text{”}(V) \wedge \rho_2 \rightarrow \text{“}n_2\text{”}(V')$$
$$\text{“}n_1\text{”}(V) \wedge \rho_2 \rightarrow \text{“}e_2\text{”}(V, V')$$
$$\text{“}n_2\text{”}(V) \wedge \text{“}e_1\text{”}(V, V') \wedge \rho_2^{=} \rightarrow \text{“}n_3\text{”}(V')$$

$$\text{“}m_4\text{”}(V) \wedge \text{“}n_3\text{”}(V) \wedge \varphi_{err} \rightarrow \mathit{false}$$

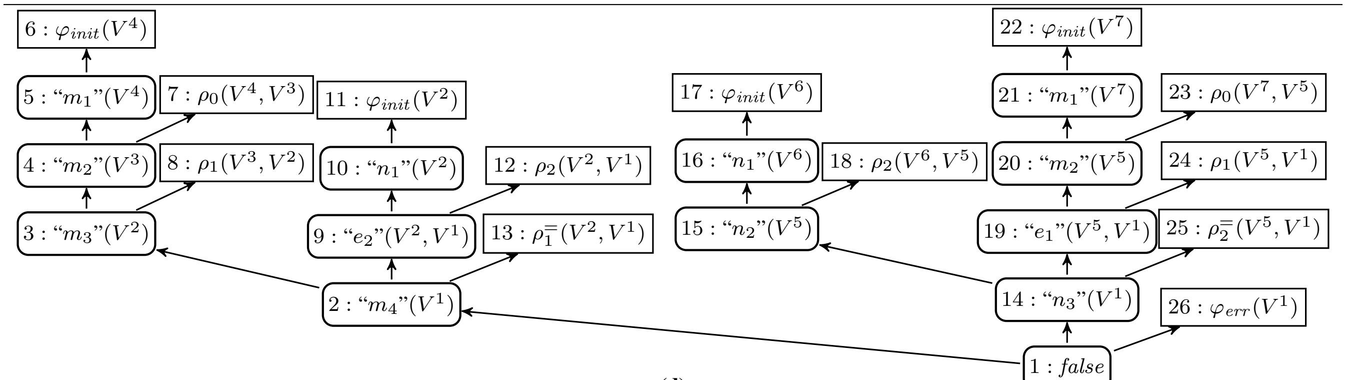
(b)

$$\varphi_{init} \rightarrow \text{“}m_1\text{”}(V_G, V_1)$$
$$\text{“}m_1\text{”}(V_G, V_1) \wedge \rho_0 \rightarrow \text{“}m_2\text{”}(V'_G, V'_1)$$
$$\text{“}m_2\text{”}(V_G, V_1) \wedge \rho_1 \rightarrow \text{“}m_3\text{”}(V'_G, V'_1)$$
$$\text{“}m_2\text{”}(V_G, V_1) \wedge \rho_1 \rightarrow \text{“}e_1\text{”}(V_G, V'_G)$$
$$\text{“}m_3\text{”}(V_G, V_1) \wedge \text{“}e_2\text{”}(V_G, V'_G) \wedge \rho_1^{=} \rightarrow \text{“}m_4\text{”}(V'_G, V'_1)$$

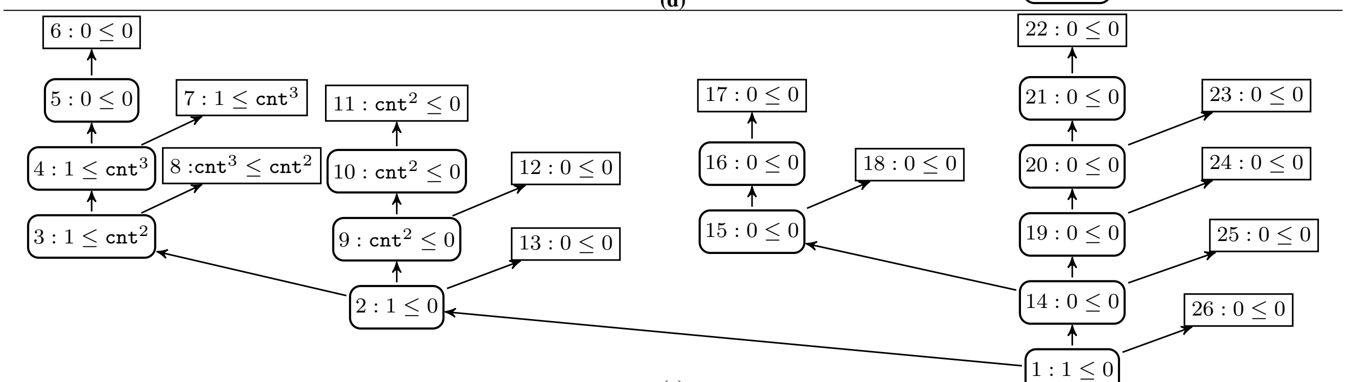
$$\varphi_{init} \rightarrow \text{“}n_1\text{”}(V_G, V_2)$$
$$\text{“}n_1\text{”}(V_G, V_2) \wedge \rho_2 \rightarrow \text{“}n_2\text{”}(V'_G, V'_2)$$
$$\text{“}n_1\text{”}(V_G, V_2) \wedge \rho_2 \rightarrow \text{“}e_2\text{”}(V_G, V'_G)$$
$$\text{“}n_2\text{”}(V_G, V_2) \wedge \text{“}e_1\text{”}(V_G, V'_G) \wedge \rho_2^{=} \rightarrow \text{“}n_3\text{”}(V'_G, V'_2)$$

$$\text{“}m_4\text{”}(V_G, V_1) \wedge \text{“}n_3\text{”}(V_G, V_2) \wedge \varphi_{err} \rightarrow \mathit{false}$$

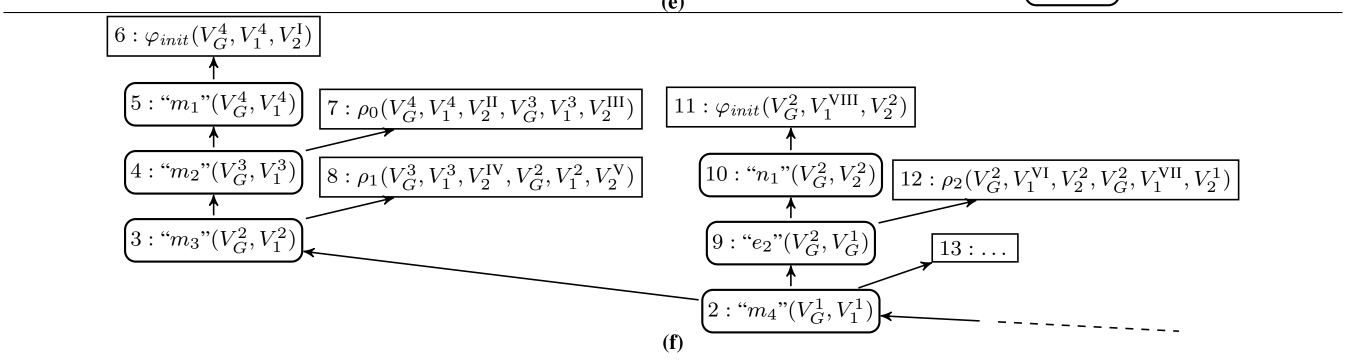
(c)



(d)



(e)



(f)

Figure 13. (a) Reachability trees constructed by ARET computation. (b) Corresponding Horn clauses generated using MKHORNCLAUSES. (c) Horn clauses generated with preference for modular solutions. (d) A tree representation of the clauses from (b) as generated by MKTREE. Each node shows its *Label* attribute. The superscript of the node name identifies the set of variables appearing in the attribute of the node. (e) The *Pred* map generated by SOLVELINEARHC from the clauses in (b). (f) The *Label* map generated by MKHORNCLAUSES from the clauses in (c).

all the *Label* attributes of leaf nodes and store this set in $Atoms$. The input set of Horn clauses is satisfiable if and only if $\bigwedge Atoms$ is unsatisfiable. If $\bigwedge Atoms$ is unsatisfiable, the test at lines 4–5 succeeds and returns a proof of unsatisfiability in the form of weights for each linear inequality. This test can be implemented using some linear arithmetic constraint solver. If the constraint solver fails to find a proof, an exception UNSATISFIABLE is thrown at line 12.

At line 7, SOLVELINEARHC calls the procedure ANNOTPRED , which is presented in Figure 16. This procedure recursively traverses the input tree in postorder. If this procedure is invoked for a leaf node n , it directly computes the value of $Pred(n)$ as a linear combination of atomic formulas with weights given by the *Proof* function (see line 2). If this procedure is invoked for an internal node n , the attribute *Pred* of n's children is computed using a recursive call at line 5. After completing the recursive call, $Pred(n)$ is calculated by adding the values of the *Pred* attributes of n's children.

Since there may be multiple nodes in the tree corresponding to the same unknown relation, the algorithm has to account for the *Pred* attributes of all these nodes. Therefore, at lines 8–9 we compute solutions for each $b(w)$ in *UnkRel* by taking conjunction of *Pred* of each node of the tree that is labeled with $b(u)$ for some u .

THEOREM 5. SOLVELINEARHC *computes a solution for a set of Horn clauses HC if and only if the conjunction of the clauses in HC is satisfiable.*

Example We illustrate the solving procedure using the same example from the previous section. Given the Horn clauses from Figure 13(b), MKTREE constructs a tree that is shown in Figure 13(d). This tree contains nodes which we label for convenience with identifiers from 1 to 26 . In Figure 13(d) we show the *Label* map of the tree. A witness of the unsatisfiability of $\bigwedge Atoms$ is given by the following atomic formulas:

$$\begin{array}{ccccc}(\texttt{cnt}^3 \geq 1) & \wedge & (\texttt{cnt}^3 = \texttt{cnt}^2) & \wedge & (\texttt{cnt}^2 = 0)\\ \rotatebox{-90}{$\in$} & & \rotatebox{-90}{$\in$} & & \rotatebox{-90}{$\in$}\\ Label(7) & & Label(8) & & Label(11)\end{array}$$

Our solver treats each linear equality as a conjunction of two linear inequalities. The equality $\texttt{cnt}^2 = 0$ is split in two inequalities $\texttt{cnt}^2 \leq 0 \wedge -\texttt{cnt}^2 \leq 0$. The proof of unsatisfiability is:

$$(1 \leq \texttt{cnt}^3) + (\texttt{cnt}^3 \leq \texttt{cnt}^2) + (\texttt{cnt}^2 \leq 0) = (1 \leq 0).$$

This is encoded in the *Proof* map with values of 1 at locations corresponding to the three atomic formulas above and values of 0 for all the other atomic formula. Next, we show in Figure 13(e) the values for the *Pred* map as computed by ANNOTPRED . The final solution of the Horn clauses is built by a conjunction of the *Pred* attributes for nodes with the same unknown label. The resulting solution SOL is shown below.

$$\begin{array}{lcll}\textsc{Sol}(\text{“}m_1\text{”}(V)) & = & \textsc{Sol}(\text{“}n_2\text{”}(V)) & = \ (0 \leq 0)\\ \textsc{Sol}(\text{“}e_1\text{”}(V, V')) & = & & = \ (0 \leq 0)\\ \textsc{Sol}(\text{“}m_2\text{”}(V)) & = & \textsc{Sol}(\text{“}m_3\text{”}(V)) & = \ (1 \leq \texttt{cnt})\\ \textsc{Sol}(\text{“}n_1\text{”}(V)) & = & \textsc{Sol}(\text{“}e_2\text{”}(V, V')) & = \ (\texttt{cnt} \leq 0)\end{array}$$

Solving the clauses shown in Figure 13(c) Given the Horn clauses shown in this figure, MKTREE returns a tree representation with a similar *Children* map structure but with different *Label* attributes. The part of the tree that contributes to the proof of unsatisfiability is shown in Figure 13(f). The variable $\texttt{cnt}^2$ does not appear in the subtree of the node 9 since $Label(9) = \text{“}e_2\text{”}(V_G^2, V_G^1)$. Part of this subtree is the node 11. Let us name the variable at this node as $\texttt{cnt}^{\text{VIII}}$. The proof of unsatisfiability shown above does no longer hold, since the following formula is satisfiable:

$$(1 \leq \texttt{cnt}^3) \wedge (\texttt{cnt}^3 = \texttt{cnt}^2) \wedge (\texttt{cnt}^{\text{VIII}} = 0)$$

However, the conjunction of the elements from the $Atoms$ set is still unsatisfiable, indicating that a modular solution exists. We find that the following atoms contribute to a proof of unsatisfiability:

$$\begin{array}{ccc}\texttt{lock}^2 = 1 & \wedge & \texttt{lock}^2 = 0\\ \rotatebox{-90}{$\in$} & & \rotatebox{-90}{$\in$}\\ Label(8) & & Label(12)\end{array}$$

After splitting the equalities in equivalent inequalities, our algorithm computes the following solution:

$$\begin{array}{ll}Pred(8) = (1 \leq \texttt{lock}^2) & Pred(12) = (\texttt{lock}^2 \leq 0)\\ Pred(3) = (1 \leq \texttt{lock}^2) & Pred(9) = (\texttt{lock}^2 \leq 0)\\ Pred(2) = (1 \leq 0) & \end{array}$$

From this *Pred* map, our algorithm derives a solution SOL in lines 8–9 and succeeds in computing modular predicates.

---

function SOLVELINEARHC
input
  HC - recursion-free Horn clauses over linear inequalities
  UnkRel - unknown relations
vars
  Label - map from node to attribute
  Children - map from node to a set of nodes
  Pred - map from node to an atomic predicate
  Proof - weight function for inequalities
begin
1 $Label$:= $Children$:= $\perp$ (∗ the empty function ∗)
2 $root$:= MKTREE($false$)
3 $Atoms$:= $\bigcup\{Label(n) \mid Children(n) = \emptyset\}$
4 **if** exists $Proof : Atoms \rightarrow \mathbb{Q}_{\geq 0}$ such that
5    $\sum\{Proof(b(u)) \cdot b(u) \mid b(u) \in Atoms\} = (1 \leq 0)$
6 **then**
7   ANNOTPRED($root$)
8   **for** each $b(w) \in UnkRel$ **do**
9   SOL($b(w)$) := $\bigwedge\{Pred(n)[w/u] \mid Label(n) = b(u)\}$
10   **return** SOL
11 **else**
12   **throw** UNSATISFIABLE
end

Figure 14. Function SOLVELINEARHC returns a solution for a set of recursion-free Horn clauses over linear arithmetic.

8. Experimental results

In this section, we describe a proof-of-concept implementation of our proposed algorithm as an extension of the model checker ARMC [29].

Tool description The verifier we built takes as input a number of functions (written in the C language) representing threads that should execute concurrently. The input file also contains the description of an initial state and a number of assertions to be proven correct. Our tool uses a frontend based on the CIL infrastructure [25] to translate a C program to its corresponding multi-threaded transition system that is formalized in Section 3. The main compo-

function MKTREE
input
  g - relation, either $b(u)$ or $false$
begin
  p, q := new nodes
  match g **with**
   | $false$ ->
     $\{b_1(w_1) \wedge \cdots \wedge b_n(w_n) \rightarrow false,\ \ldots\}$:= HC
     $z_1, \ldots, z_n$:= fresh copies of $w_1, \ldots, w_n$
     σ := $[z_1/w_1] \cdots [z_n/w_n]$
   | $b(u)$ ->
     $\{b_1(w_1) \wedge \cdots \wedge b_n(w_n) \rightarrow b(w),\ \ldots\}$:= HC
     $z_1, \ldots, z_n, z$:= fresh copies of $w_1, \ldots, w_n, w$
     σ := $[z_1/w_1] \cdots [z_n/w_n][u/z]$
  $Label(p)$:= $\{b_i(w_i)\sigma \mid i \in 1..n \wedge b_i = (\leq)\}$
  $Children(p)$:= $\emptyset$
  $Label(q)$:= g
  $Children(q)$:=
    $\{p\} \cup \bigcup\{$MKTREE$(b_i(w_i)\sigma) \mid i \in 1..n \wedge b_i \neq (\leq)\}$
  return q
end

Figure 15. Function MKTREE. Fresh copies are created consistent, e.g., fresh copies of $\{v_1, v_2\}$, $\{v_3, v_1\}$ returns $\{f_1, f_2\}$, $\{f_3 f_1\}$, where f_1, f_2, f_3 are fresh variables that do not appear anywhere else.

procedure ANNOTPRED
input
  n - node of Horn tree
begin
  if $Children(n) = \emptyset$ **then**
    $Pred(n)$:= $\sum\{Proof(b(u)) \cdot b(u) \mid b(u) \in Label(n)\}$
  else
    for each $n' \in Children(n)$ **do**
      ANNOTPRED(n')
    $Pred(n)$:= $\sum\{Pred(n') | n' \in Children(n)\}$
end

Figure 16. Procedure ANNOTPRED.

nent of our tool is an implementation of our algorithm done using SICStus Prolog [33].

An important design decision in our implementation concerns the treatment of control-location and data variables. Even if both control-location variables and data variables can be handled uniformly by our algorithm, we found that different abstraction domains and refinement for the two domains can lead to significant improvement. In our implementation, the REFINE procedure first splits the constraints into data variable constraints and control-location constraints. The splitting procedure preserves satisfiability/unsatisfiability of the original constraint since there is no atomic formula in the program transitions that relates both control variables and data variables. If the data constraints are satisfiable, the algorithm proceeds as in Figure 9. If the data constraints are unsatisfiable, our implementation relies on a specialized refinement procedure (described in [13]) that takes advantage of the simpler form of control counterexamples. For these counterexamples, control variables range over a finite domain and no atomic formula from the program transitions involves different control variables.

Benchmark programs We tested our prototype implementation using a collection of programs that have intricate correctness proofs for their safety assertions. The first four programs shown in Table 1 are derived from two buggy examples highlighted as figures in [20], together with their fixes from the MOZILLA CVS repository. The property to verify is that two operations performed by different threads are executed in the correct order. The next three examples model the stopping procedure of a Windows NT Bluetooth driver [30]. BLUETOOTH2 contains two threads, one worker thread and another thread to model the stopping procedure of the driver. BLUETOOTH2-FIXED and BLUETOOTH3-FIXED are the fixed versions of the model with two and respectively three threads. SCULL [6] is a Linux character device driver that implements access to a global memory area. The property to verify is that read and write operations are performed in critical section.

We also include some examples which are not particularly favorable to a modular reasoning approach. These examples are algorithms that establish mutual exclusion and mainly deal with global variables (no local computation is included in the critical region). The mutual exclusion property of the naïve version of the Bakery algorithm [22] holds only when assuming assignments are performed atomically. (Our verifier was able to confirm the bug present in the code without such atomicity assumption.) BAKERY [18] is the complete version of the Bakery algorithm, while LAMPORT [19] is an algorithm with an optimized path in the absence of memory contention. QRCU [23] is an algorithm implementing the Read-copy-update synchronization algorithm. It is an alternative to a readers-writer lock having wait-free read operations.

Performance of our tool To explain our experimental results, we first articulate a working hypothesis. This hypothesis suggests that, when verifying a program that does not have a modular proof, the algorithm with preference for modular solutions (denoted as *verification with bias*) is expected to pay a penalty by insisting to search for modular solutions that do not exist. On the other hand, for cases where a modular proof does exist, the non-biased verification could fail to find a modular proof and instead return a more detailed non-modular proof. Therefore, the hypothesis suggests that in these cases the biased verification is expected to succeed faster compared to the non-biased verification.

We report statistical data for each of the programs in Table 1. We show the number of lines of code (LOC) and whether a modular proof exists for a program (see Column 3). Our implementation has two modes. Column 4 shows the verification results, when using our algorithm with a preference for modular solutions. The last column of the table shows the verification results for the non-biased implementation of our algorithm. The results demonstrate that our approach to verification of multi-threaded programs is feasible and that the constraint solving procedure with bias is able to produce modular proofs more often than the non-biased verification. Furthermore, without the bias, the verification procedure times-out for Scull and QRCU examples showing the benefits of modular proofs.

As another experiment, we tested some of our smaller examples using two state-of-the-art model checkers for sequential C programs, Blast [14] and ARMC[31]. For each of the tested programs (Fig2-fixed, Fig4-fixed, Dekker, Peterson, and Lamport), we instrumented the program counter as explicit program variables ($\mathtt{pc}_1$ and $\mathtt{pc}_2$) and obtained a sequential model of the multi-threaded examples. Both Blast and ARMC eagerly consider all interleavings and obtained timeouts after 30 minutes for both Fig4-fixed and Lamport. Comparatively, our tool exploits the thread structure of these programs and obtains conclusive verification results fast.

Benchmark programs			Our algorithm			
Name	LOC	Has a modular proof?	With bias		No bias	
Fig2-cex[20]	33	No	×	0.2s	×	0.2s
Fig2-fixed[20]	38	Yes	✓-Modular	0.8s	✓-Modular	0.7s
Fig4-cex[20]	175	No	×	4.5s	×	3.7s
Fig4-fixed[20]	168	Yes	✓-Modular	1.5s	✓	11.1s
Bluetooth2[30]	90	No	✓	29.1s	✓	11.2s
Bluetooth2-fixed	90	No	✓	3.7s	✓	0.4s
Bluetooth3-fixed	90	No	✓	135s	✓	9.7s
Scull[6]	451	Yes	✓-Modular	128.5s	T/O	
Dekker[1]	39	No	✓	11.1s	✓	6s
Peterson[1]	26	No	✓	4.7s	✓	3.9s
Readers-writer-lock[10]	22	Yes	✓-Modular	0.2s	✓	0.4s
Time varying mutex[10]	29	No	✓	11.8s	✓	3.1s
Szymanski[32]	43	No	✓	32s	✓	8.8s
NaïveBakery[22]	22	Yes	✓-Modular	2.5s	✓	3s
Bakery[18]	37	No	✓	105.4s	✓	101s
Lamport[19]	62	No	✓	120.8s	✓	97s
QRCU[23]	120	Yes	✓-Modular	34.5s	T/O	

Table 1. "Has a modular proof?" indicates whether the program has a modular proof of correctness. "✓" and "×" indicate whether the program is proven safe or a counterexample is returned, while "T/O" stands for time out after 15 minutes. ✓-Modular indicates that a modular proof is found by our tool.

9. Related work

The main inspiration for our work draws from the rely-guarantee reasoning method [16, 17] and automatic abstraction refinement approach to verification [4].

The seminal work on rely-guarantee reasoning [16, 17] initially offered an approach to reason about multi-threaded programs by making explicit the interference between threads. Subsequently, rely-guarantee reasoning was used to tackle the problem of state explosion in verification of multi-threaded programs. Rely-guarantee reasoning was mechanized and firstly implemented in the Calvin model checker [10] for Java shared-memory programs. Calvin reduces the verification of the multi-threaded program to the verification of several sequential programs with the help of a programmer specified environment assumption. In [9], thread-modular model checking was proposed to infer automatically environment assumptions that propagate only global variable changes to other threads. The algorithm has low complexity, polynomial in the number of threads, but is incomplete and fails to discover environment assumptions that refer to the local states of a thread. Thread-modular verification is formalized by [21] in the framework of abstract interpretation as Cartesian product of sets of states.

The method of [15] uses a richer abstraction scheme that computes contextual thread reachability, where the context in which a thread executes includes information on both global and local states of threads. The context (or environment) is computed using bisimilarity quotients in steps that are interleaved with abstract reachability computations. The verification starts with the strongest possible environment assumption and, by refinement, the environment is weakened until it over-approximates the transitions of the other threads. In contrast, our approach refines iteratively the environment based on over-approximation, starting with the weakest environment and strengthening it at every iteration. For abstraction refinement, a counterexample from [15] is reduced to a concrete sequential path by replacing environment transitions with their corresponding local transitions.

The approach of [5] presents another solution to overcome the incompleteness of local reasoning. Guided by counterexamples, it refines the abstraction by exposing a local variable of a thread as a global variable. This refinement recovers the completeness of reasoning, but is applicable to finite-state systems and may compute an unnecessarily precise abstraction. In contrast, our refinement procedure relies on interpolation and includes predicates on local variables as needed during verification.

Another approach to overcome the state explosion problem of monolithic reasoning over multi-threaded programs is to translate the multi-threaded program to a sequential program assuming a bound on the number of context switches. This scheme was initially proposed and implemented in KISS [30], a multi-threaded checker for C programs, and later evolved to handle and reproduce even difficult to find Heisenbugs [24]. Monolithic reasoning can be greatly facilitated by using techniques evolved from partial-order reduction [11], like dynamic partial-order reduction [8] or peephole partial order reduction [34]. Yet another technique to fight state explosion is to factor out redundancy due to thread replication as proposed in counter abstraction [27] and implemented in the model checker Boom [2, 3]. We view these techniques as paramount in obtaining practical multi-threaded verifiers, but orthogonal to our proposal for automatic environment inference.

Acknowledgments

The first author was supported by the DFG Graduiertenkolleg 1480 (PUMA). We thank Byron Cook, Ruslán Ledesma Garza, and Peter O'Hearn for comments and suggestions.

References

[1] Y. Bar-David and G. Taubenfeld. Automatic discovery of mutual exclusion algorithms. In *DISC*, pages 136–150, 2003.

[2] G. Basler, M. Mazzucchi, T. Wahl, and D. Kroening. Symbolic counter abstraction for concurrent software. In *CAV*, pages 64–78, 2009.

[3] G. Basler, M. Hague, D. Kroening, C.-H. L. Ong, T. Wahl, and H. Zhao. Boom: Taking boolean program model checking one step further. In *TACAS*, pages 145–149, 2010.

[4] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.

[5] A. Cohen and K. S. Namjoshi. Local proofs for global safety properties. *FMSD*, 34(2):104–125, 2009.

[6] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.

[7] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[8] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121, 2005.

[9] C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN*, pages 213–224, 2003.

[10] C. Flanagan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *ESOP*, pages 262–277, 2002.

[11] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. PhD thesis, University of Liege, Computer Science Department, 1994.

[12] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.

[13] A. Gupta, C. Popeea, and A. Rybalchenko. Non-monotonic refinement of control abstraction for concurrent programs. In *ATVA*, pages 188–202, 2010.

[14] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.

[15] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI*, pages 1–13, 2004.

[16] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[17] C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[18] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[19] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, 1987.

[20] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, pages 329–339, 2008.

[21] A. Malkis, A. Podelski, and A. Rybalchenko. Thread-modular verification is cartesian abstract interpretation. In *ICTAC*, pages 183–197, 2006.

[22] Z. Manna and A. Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag, 1995.

[23] P. McKenney. Using Promela and Spin to verify parallel algorithms. *LWN.net weekly edition*, 2007.

[24] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *OSDI*, pages 267–280, 2008.

[25] 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.

[26] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Inf.*, 6:319–340, 1976.

[27] A. Pnueli, J. Xu, and L. D. Zuck. Liveness with (0, 1, infty)-counter abstraction. In *CAV*, pages 107–122, 2002.

[28] A. Podelski and A. Rybalchenko. Transition predicate abstraction and fair termination. In *POPL*, pages 132–144, 2005.

[29] A. Podelski and A. Rybalchenko. ARMC: The logical choice for software model checking with abstraction refinement. In *PADL*, pages 245–259, 2007.

[30] S. Qadeer and D. Wu. KISS: keep it simple and sequential. In *PLDI*, pages 14–24, 2004.

[31] A. Rybalchenko. The ARMC tool. *Available from http://www7.in.tum.de/˜rybal/armc/*.

[32] B. K. Szymanski. A simple solution to Lamport's concurrent programming problem with linear wait. In *ICS*, pages 621–626, 1988.

[33] The Intelligent Systems Laboratory. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, 2001. Release 3.8.7.

[34] C. Wang, Z. Yang, V. Kahlon, and A. Gupta. Peephole partial order reduction. In *TACAS*, pages 382–396, 2008.