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:
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., 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 $V = (\text{lock}, p_1, p_2)$ for the program variables, where $p_1$ and $p_2$ are local program counter variables of the first and second thread, respectively.

We start by representing the program using assertions $\varphi_{\text{init}}$ and $\varphi_{\text{err}}$ over program variables that describe the initial and error states of the program, together with assertions over unprimed and primed program variables $p_1$ and $p_2$ that describe the transition relations for program statements.

$$\varphi_{\text{init}} = (p_1 = a \land p_2 = p \land \text{lock} = 0) ,$$

$$\varphi_{\text{err}} = (p_1 = b \land p_2 = q) ,$$

$$p_1 = (\text{lock} = 0 \land \text{lock}' = 1 \land p_1 = a \land p_1' = b \land p_2') ,$$

$$p_2 = (\text{lock} = 0 \land \text{lock}' = 1 \land p_2 = p \land p_1' = q \land p_2') ,$$

$$p_2' = (p_2 = p_2') .$$

The auxiliary assertions $p_1'$ and $p_2'$ 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 $\text{ARET}$s (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.

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 $\varphi_{\text{err}}$.

First $\text{ARET}$ computation The thread reachability computation for the first thread starts by computing an abstraction of the initial program states $\varphi_{\text{init}}$. 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., $\dot{\mathcal{P}}_1 = \{ p_1 = a, p_1 = b \}$, and is computed as follows: $\dot{\alpha}_1(S) = \{ p \in \dot{\mathcal{P}}_1 \mid \forall V : S \rightarrow p \}$. We obtain the initial abstract state $m_1$ as follows:

$$m_1 = \dot{\alpha}_1(\varphi_{\text{init}}) = (p_c_1 = a) .$$

Next, we compute an abstract successor of $m_1$ with respect to the transition $p_1$ using the strongest postcondition operator $\text{post}$ that is combined with $\dot{\alpha}_1$:

$$m_2 = \dot{\alpha}_1(\text{post}(p_1, m_1)) = (p_c_1 = b) .$$

Similarly, we compute the thread reachability for the second thread. Using predicates over the program counter of the second thread, i.e., $\dot{\mathcal{P}}_2 = \{ p_2 = p, p_2 = q \}$, we compute the following two abstract states:

$$n_1 = \dot{\alpha}_2(\varphi_{\text{init}}) = (p_c_2 = p) ,$$

$$n_2 = \dot{\alpha}_2(\text{post}(p_2, n_1)) = (p_c_2 = q) .$$

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

We stop the $\text{ARET}$ 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 \( "m_1'(V)" \), \( "m_2'(V)" \), 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 \( \varphi_{init} \) 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.

\[
HC_1 = \{ \varphi_{init} \rightarrow "m_1'(V)" \lor "m_2'(V)" \land "n_2'(V)" \}
\]

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 \( \text{Sol} \) that maps each unknown predicate to an assertion of the program variables.

\[
\text{Sol}("m_1'(V)") = (pc_2 = p) \quad \text{Sol}("n_1'(V)") = (pc_1 = a) \\
\text{Sol}("m_2'(V)") = (pc_2 = p) \quad \text{Sol}("n_2'(V)") = (pc_1 = a)
\]

The existence of \( \text{Sol} \) indicates that the discovered evidence is spurious. We use \( \text{Sol} \) 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 \( \mathcal{P}_1 \), and perform a similar step for the second thread. The resulting sets of predicates are shown below.

\[
\mathcal{P}_1 = \{ pc_1 = a, pc_1 = b, pc_2 = p \} \\
\mathcal{P}_2 = \{ pc_2 = p, pc_2 = q, pc_1 = a \}
\]

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

\[
m_1 = (pc_1 = a \land pc_2 = p) \quad n_1 = (pc_1 = a \land pc_2 = p) \\
m_2 = (pc_1 = b \land pc_2 = p) \quad n_2 = (pc_1 = a \land pc_2 = q)
\]

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 \( \rho_1 \) 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 \( \tilde{\alpha}_{1\rightarrow 2} \).

In this function, the double dot indicates that the function abstracts binary relations over states (and not sets of states). The index 1 \( \tilde{\rightarrow} 2 \) 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) \( \mathcal{P}_{1\rightarrow 2} = \emptyset \) to define \( \tilde{\alpha}_{1\rightarrow 2} \).

The environment transition \( e_1 \) is defined as

\[
e_1 = \tilde{\alpha}_{1\rightarrow 2}(m_1 \land \rho_1) = (\varphi_{err} = \text{true})
\]

and it non-deterministically updates the program variables (since \( \text{true} \) 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:

\[
n_3 = \tilde{\alpha}_{2\rightarrow 1}(n_1 \land \rho_2) = (pc_2 = q)
\]

The conjunct \( \rho_2^c \) ensures that the local variable of the second thread is not changed by the environment transition.

Symmetrically, we use a function \( \tilde{\alpha}_{2\rightarrow 1} \) to abstract the effect of applying transitions in the second thread on the thread reachability of the first thread. The application of \( \rho_2 \) on \( n_1 \) results in the environment transition \( e_2 \) such that

\[
e_2 = \tilde{\alpha}_{2\rightarrow 1}(n_1 \land \rho_2) = (pc_1 = b)
\]

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:

\[
m_3 = \tilde{\alpha}_1(n_1 \land \rho_2, m_2) = (pc_1 = b)
\]

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.

\[
HC_2 = \{ \varphi_{init} \rightarrow "m_1'(V)" \lor "m_2'(V)" \land "n_2'(V)" \}
\]

The conjunction of clauses in \( HC_2 \) is satisfiable. We obtain the following satisfying assignment \( \text{Sol} \).

\[
\text{Sol}("m_1'(V)") = \text{true} \quad \text{Sol}("m_2'(V)") = (\text{lock} = 1) \\
\text{Sol}("n_1'(V)") = \text{true} \quad \text{Sol}("n_2'(V)") = (\text{lock} = 1) \\
\text{Sol}("m_3'(V)") = \text{false} \quad \text{Sol}("e_1'(V,V')") = (\text{lock} = 0) \\
\text{Sol}("n_3'(V)") = \text{false} \quad \text{Sol}("e_2'(V,V')") = (\text{lock} = 0)
\]

This solution constrains \( "m_2'(V)" \) and \( "n_2'(V)" \) to states where the lock is held (\( \text{lock} = 1 \)) , 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 (\( \text{lock} = 0 \)).

We add the (transition) predicates that appear in the environment transition \( e_2 \) of the first thread to the set \( \mathcal{P}_{2\rightarrow 1} \) , and, symmetrically, we add the predicates from \( e_1 \) to \( \mathcal{P}_{1\rightarrow 2} \). For the next ARET
LockBit conclude that yields an intersection that is disjoint from the error states, we found. Since each pair of abstract states from different threads LockBit uses an integer variable lock) to record which thread holds the lock. Due to this additional computation we have the following set of predicates:

\[
\begin{align*}
\mathcal{P}_1 &= \{pc_1 = a, pc_1 = b, pc_2 = p, lock = 1\}, \\
\mathcal{P}_2 &= \{pc_2 = p, pc_2 = q, pc_1 = a, lock = 1\}, \\
\mathcal{P}_{102} &= \{lock = 0\}, \\
\mathcal{P}_{201} &= \{lock = 0\}.
\end{align*}
\]

**Last ARET computation** We perform another ARET computation and a subsequent abstraction refinement step. We add the predicate lock' = 1 to both \(\mathcal{P}_{102}\) and \(\mathcal{P}_{201}\) 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:

\[
\begin{align*}
m_1 &= \alpha_1(\varphi_{init}) = (pc_1 = a \land pc_2 = p), \\
n_1 &= \alpha_2(\varphi_{init}) = (pc_1 = a \land pc_2 = p), \\
m_2 &= \alpha_1(post(p_1, m_1)) = (pc_1 = b \land pc_2 = p \land lock = 1), \\
n_2 &= \alpha_2(post(p_2, n_1)) = (pc_1 = a \land pc_2 = q \land lock = 1), \\
e_1 &= \alpha_{102}(m_1 \land p_1) = (lock = 0 \land lock' = 1), \\
e_2 &= \alpha_{201}(m_1 \land p_2) = (lock = 0 \land lock = 1).
\end{align*}
\]

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.

**First ARET computation** Similarly to LockBit, we discover that \(m_3 \land n_3 \land \varphi_{err}\) is satisfiable, and compute the following set of Horn clauses:

\[
\begin{align*}
HC_3 &= \{\varphi_{init} \rightarrow "m_1"(V), "m_1"(V) \land p_1 \rightarrow "m_2"(V), \\
&\quad \varphi_{init} \rightarrow "n_1"(V), "n_1"(V) \land p_2 \rightarrow "n_2"(V), \\
&\quad "m_2"(V) \land "n_2"(V) \land \varphi_{err} \rightarrow false\}.
\end{align*}
\]

One possible satisfying assignment \(\text{Sol}\) is:

\[
\begin{align*}
\text{Sol}("m_1"(V)) &= (pc_2 = p), \\
\text{Sol}("n_1"(V)) &= (pc_1 = a), \\
\text{Sol}("m_2"(V)) &= (pc_2 = p), \\
\text{Sol}("n_2"(V)) &= (pc_1 = a).
\end{align*}
\]

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

\[
\begin{align*}
\mathcal{P}_1 &= \{pc_1 = a, pc_1 = b, lock = 1\}, \\
\mathcal{P}_2 &= \{pc_2 = p, pc_2 = q, lock = 2\}, \\
\mathcal{P}_{201} &= \{lock = 0\}, \\
\mathcal{P}_{102} &= \{lock = 0\}.
\end{align*}
\]

![Figure 4. Tree that shows all the reachable abstract states found during the last ARET computation for LockId.](image)

This assignment uses a predicate \(pc_2 = p\) 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 pc_1 of the first thread, i.e., we have the unknown predicates "m_1"(lock, pc_1) and "m_2"(lock, pc_1). Similarly, for the second thread we obtain "n_1"(lock, pc_2) and "n_2"(lock, pc_2). Instead of \(HC_3\), we use a set of Horn clauses \(HC_4\) shown below:

\[
\begin{align*}
HC_4 &= \{\varphi_{init} \rightarrow "m_1"(lock, pc_1), \\
&\quad "m_1"(lock, pc_1) \land p_1 \rightarrow "m_2"(lock', pc_1), \\
&\quad \varphi_{init} \rightarrow "n_1"(lock, pc_2), \\
&\quad "n_1"(lock, pc_2) \land p_2 \rightarrow "n_2"(lock', pc_2), \\
&\quad "m_2"(lock, pc_1) \land "n_2"(lock, pc_2) \land \varphi_{err} \rightarrow false\}.
\end{align*}
\]

The conjunction of clauses from \(HC_4\) can be satisfied by an assignment \(\text{Sol}\) such that

\[
\begin{align*}
\text{Sol}("m_1"(lock, pc_1)) &= true, \\
\text{Sol}("m_2"(lock, pc_1)) &= (lock = 1), \\
\text{Sol}("n_1"(lock, pc_2)) &= true, \\
\text{Sol}("n_2"(lock, pc_2)) &= (lock = 2),
\end{align*}
\]

which contains only modular predicates.

**Figure 3. Example program LockId.**

![Tree that shows all the reachable abstract states found during the last ARET computation for LockId.](image)
The ARET construction is completed, since
\[ \alpha_1 \text{(post}(e_2 \land \rho_1^e, m_1)) \rightarrow m_1 \], \[ \alpha_2 \text{(post}(e_2, m_2)) = \text{false} \]
\[ \alpha_2 \text{(post}(e_1 \land \rho_2^e, n_1)) \rightarrow n_1 \], \[ \alpha_2 \text{(post}(e_1, n_2)) = \text{false} \].

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

Furthermore no predicate in \( \mathcal{P}_3 \) refers to the local variable of the second thread, the symmetric condition holds for \( \mathcal{P}_3 \), and the predicates in \( \mathcal{P}_{1,2} \) as well as \( \mathcal{P}_{2,1} \) 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 \( N \geq 1 \) concurrent threads. Let \( 1 \ldots N \) be the set \{1, \ldots, N\}. We assume that the program variables \( V = (V_0, V_1, \ldots, V_N) \) are partitioned into global variables \( V_G \) that are shared by all threads, and local variables \( V_1, \ldots, V_N \) that are only accessible by the threads \( 1, \ldots, N \), respectively.

The set of global states \( G \) consists of the valuations of global variables, and the sets of local states \( L_1, \ldots, L_N \) 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 \( \Sigma = G \times L_1 \times \cdots \times L_N \). 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 \( \equiv \) denote the satisfaction relation between (pairs of) states and assertions.

The set of initial program states is denoted by \( \varphi_{init} \), and the set of error states is denoted by \( \varphi_{err} \). For each thread \( i \in 1 \ldots N \) we have a finite set of transition relations \( T_i \), which are abbreviated as transitions. Each transition \( \rho \in T_i \) can change the values of the global variables and the local variables of the thread \( i \). Let \( \rho_i^e \) be a constraint requiring that the local variables of the thread \( i \) do not change, i.e., \( \rho_i^e = (V_i = V_i^0) \). Then, \( \rho \in T_i \) has the form
\[ \rho^\text{update}(V_G, V_i, V_G', V_i') \land \bigwedge_{j \in 1 \ldots N \setminus \{i\}} \rho_j^e \],
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 \( \rho_i \) for the union of the transitions of the thread \( i \) and \( \rho = \bigvee_i \rho_i \). The transition relation of the program is \( \rho_T = \rho_1 \lor \cdots \lor \rho_N \).

Computations A computation of \( P \) is a sequence of program states \( s_1, s_2, \ldots \) 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 \( \rho \) from a program thread, i.e., \( (s_i, s_{i+1}) \models \rho \). A path is a sequence of transitions. We write \( \epsilon \) for the empty sequence.

Let \( \lbrack z/w \rbrack \) be a substitution function such that \( \lbrack z/w \rbrack \) replaces \( w \) by \( z \) in \( \varphi \). Let \( \circ \) be the relational composition function for binary relations given by assertions over unprimed and primed variables such that for assertions over \( \varphi \) and \( \psi \) we have \( \varphi \circ \psi = \exists V': \varphi[V'/V] \land \psi[V'/V] \). 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 \( \rho_{\pi} = \rho_1 \circ \cdots \circ \rho_n \). A path \( \pi \) is feasible if its path relation is not empty, i.e., \( \exists V': \rho_{\pi} \).

A program state is reachable if it appears in some computation. Let \( \varphi_{reach} \) denote the set of reachable states. The program is safe if none of its error states is reachable, i.e., \( \varphi_{reach} \land \varphi_{err} \rightarrow \text{false} \).

For assertions \( R_1, \ldots, R_N \) over \( V \), and \( E_1, \ldots, E_N \) over \( V \land V' \)

\begin{align*}
\text{CS1:} & \quad \varphi_{init} \rightarrow R_i \quad \text{for } i \in 1 \ldots N \\
\text{CS2:} & \quad R_i \land \rho_i \rightarrow R_i' \quad \text{for } i \in 1 \ldots N \\
\text{CS3:} & \quad R_i \land E_i \land \rho_i^e \rightarrow R_i' \quad \text{for } i \in 1 \ldots N \\
\text{CS4:} & \quad (\bigvee_{j \in 1 \ldots N \setminus \{i\}} R_i \land \rho_i) \rightarrow E_j \quad \text{for } j \in 1 \ldots N \\
\text{CS5:} & \quad R_i \land \cdots \land R_N \land \varphi_{err} \rightarrow \text{false} \\
\end{align*}

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 \( \rho \) and a set of states \( \varphi \) we have
\[ \text{post}(\rho, \varphi) = \exists V' : \varphi[V'/V] \land \rho[V'/V'][V/V'] \].

We also extend the logical implication to tuples of equal length, i.e.,
\[ (\varphi_1, \ldots, \varphi_n) \rightarrow (\psi_1, \ldots, \psi_n) = \varphi_1 \rightarrow \psi_1 \land \cdots \land \varphi_n \rightarrow \psi_n \],
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 \( \rightarrow \).

A Horn clause \( b_1(w_1) \land \cdots \land b_n(w_n) \rightarrow b(w) \) consists of relation symbols \( b_1, \ldots, b_n \), and vectors of variables \( w_1, \ldots, w_n, w \). For the algorithm SOLVELINEARHC in Section 7 we only consider Horn clauses over linear arithmetic. We say that \( b \) depends on the relation symbols \( \{b_i \mid i \in 1 \ldots n \land b_i \not= \bot \} \). 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 \( R_1, \ldots, R_N \) that keep track of program states reached by threads \( 1, \ldots, N \) together with their respective environment transitions \( E_1, \ldots, E_N \). 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 \( \rho_i^e \) in CS3 se-
Proof. Let $R_1, \ldots, R_N$ and $E_1, \ldots, E_N$ satisfy the premises CS1, \ldots, CS5. We show that the program is sound, for each reachable state $s \models \varphi_{\text{reach}}$. We prove that $s \models R_1 \land \cdots \land R_N$ by induction over the length $k$ of a shortest computation segment $s_1, \ldots, s_k$ such that $s_1 \models \varphi_{\text{out}}$ 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 $s_k \models \varphi_{\text{reach}}$ and hence $s_k \models R_1 \land \cdots \land R_N$. If $s_k$ does not have any successor, i.e., $\neg(\exists s_{k+1} : (s_k, s_{k+1}) \models \rho_T)$, 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 $s_{k+1} \models R_i$ for each $i \in \{1, N \setminus \{1\}\}$ 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.

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:

$$R_i = \exists V \setminus (V_G \cup V_i) : \varphi_{\text{reach}}, \quad \text{for } i \in 1..N$$

$$E_i = \exists (V \cup \alpha)^I \setminus (V_G \cup V_G^I) : \varphi_{\text{reach}} \land \alpha_T, \quad \text{for } i \in 1..N$$

`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 $\varphi_{\text{reach}}$ 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

$$F(S_1, \ldots, S_N, T_1, \ldots, T_N) =$$

$$(\text{post}(\rho_1 \lor T_1 \land \rho_1^T, S_1), \ldots, \text{post}(\rho_N \lor T_N \land \rho_N^T, S_N), \ldots)$$

$$(\bigvee_{i \in 1..N \setminus \{1\}} S_i \land \rho_1, \ldots, \ldots \bigvee_{i \in 1..N \setminus \{N\}} S_i \land \rho_N).$$

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 $F(R_1, \ldots, R_N, E_1, \ldots, E_N)$ then $R_1, \ldots, R_N, E_1, \ldots, E_N$ satisfies CS2, CS3, and CS4.

We define a distinguished tuple $\bot_F$:

$$\bot_F = (\varphi_{\text{out}}, \varphi_{\text{init}}, \text{false}, \ldots, \text{false})$$

Then, each pre-fixpoint of $F$ that is greater than $\bot_F$ satisfies the premise CS1. By choosing a pre-fixpoint $(R_1, \ldots, R_N, E_1, \ldots, E_N)$ above $\bot_F$ such that $R_1 \land \cdots \land R_N \land \varphi_{\text{err}} \Rightarrow \text{false}$ 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, 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 $\hat{\alpha}_i$ and $\hat{\alpha}_w$, where $i \neq j \in 1..N$, that over-approximate sets and binary relations over programs states, respectively.

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

$$F^\#(S_1, \ldots, S_N, T_1, \ldots, T_N) =$$

$$(\hat{\alpha}_1(\text{post}(\rho_1, S_1)) \lor \hat{\alpha}_1(\text{post}(T_1 \land \rho_1^T, S_1)), \ldots)$$

$$\hat{\alpha}_N(\text{post}(\rho_N, S_N)) \lor \hat{\alpha}_N(\text{post}(T_N \land \rho_N^T, S_N)), \ldots)$$

$$(\bigvee_{i \in 1..N \setminus \{1\}} \hat{\alpha}_i(S_i \land \rho_1), \ldots)$$

$$(\bigvee_{i \in 1..N \setminus \{N\}} \hat{\alpha}_i(S_i \land \rho_N)).$$

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

$$\bot_{F^\#} = (\hat{\alpha}_1(\varphi_{\text{init}}), \ldots, \hat{\alpha}_N(\varphi_{\text{init}}), \text{false}, \ldots, \text{false}).$$

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 $(R_1, \ldots, R_N, E_1, \ldots, E_N)$, 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.
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 refine the abstraction functions used for the ARET computation, see Section 7.

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

```plaintext
function MAIN
input
$P$ - program with $N$ threads
vars
$\overline{P}_i$, $\tilde{\alpha}_i$ - predicates for thread $i$ and corresponding state abstraction function
$\overline{P}_{i,j}$, $\tilde{\alpha}_{i,j}$ - transition predicates for pair of threads $i, j$ and corresponding transition abstraction function
$R_i$ - abstract states of thread $i$
$E_i$ - 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
for each $i \neq j \in 1..N$ do
$P_i := \overline{P}_{i,j} := \emptyset$
repeat
for each $i \neq j \in 1..N$ do
$\tilde{\alpha}_i := \lambda S. \bigwedge \{ \tilde{p} \in \overline{P}_i \mid \forall V : S \rightarrow \tilde{p} \}$
$\tilde{\alpha}_{i,j} := \lambda T. \bigwedge \{ \tilde{p} \in \overline{P}_{i,j} \mid \forall V, V' : T \rightarrow \tilde{p} \}$
ABSTRACTENV()
if exists $S_1 \in R_1, \ldots, S_N \in R_N$ such that
then try
REFINE($S_1, \ldots, S_N$)
with UNSATISFIABLE
$D :=$ some $S_i$ from $\{S_1, \ldots, S_N\}$
return "counterexample MKPATH($D$)"
else return "program $P$ is safe with the proof $\bigwedge R_1, \ldots, \bigwedge R_N, \bigwedge E_1, \ldots, \bigwedge E_N"$
until true
end.
```

Figure 7. Procedure ABSTRACTENV implements ARET computation. We assume that the iterator statements in lines 7 and 9 make an immutable snapshot of their domains $R_i$ and $E_i$, 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 $\tilde{\alpha}_i$ and $\tilde{\alpha}_{i,j}$ 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 ABSTRACTENV See Figure 7 for the procedure ABSTRACTENV that implements ARET computation using the abstraction functions $\tilde{\alpha}_i$ and $\tilde{\alpha}_{i,j}$. We use Parent and ParentTId to maintain information about the constructed trees, and initialize
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 \( i \in 1..N \). \( \mathcal{E}_i \) 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 \( \text{ARET} \) 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 \( \text{Parent} \) is updated to keep track of the child-parent relation between abstract states, while \( \text{ParentTId} \) 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 \( \text{ABSTR\_REACH\_ENV} \) computes sets of abstract states \( \mathcal{R}_1, \ldots, \mathcal{R}_N \) and sets of environment transitions \( \mathcal{E}_1, \ldots, \mathcal{E}_N \).

### 6. Abstraction refinement

**Procedure \text{REFINE}** In Figure 9, we present the procedure \text{REFINE} that takes as argument an error tuple and, if possible, refines the abstraction functions to include the predicate that witnesses the fact that the error state is unreachable. The procedure \text{REFINE} generates a set of Horn clauses corresponding to the error tuple (lines 1–4). Next, the \text{REFINE} algorithm invokes a solving procedure for Horn clauses (lines 5–7). Lastly, the procedure \text{REFINE} updates the abstraction functions using the solution of Horn clauses at lines 8–12. We consider the solution \( \text{Sol} \) and add the atomic predicates that appear in \( \text{Sol}(\text{S'}(V)) \) 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 \( \text{SOLVEHC} \) returns a correct solution to the set of Horn clauses received as argument. In

---

**Function \text{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 \( \varphi_{err} \) : \{“\( S_i(V) \) \( \land \) \( \cdots \) \( \land \) “\( S_N(V) \) \( \land \) \( \varphi_{err} \rightarrow \) false”\}. The other clauses are generated by invoking \text{MKHORNCLAUSES}(S_i) for \( i \in 1..N \). The function \text{MKHORNCLAUSES} generates Horn clauses for transitions considered during \( \text{ARET} \) computation as follows. If the abstract state \( D \) was produced by following a local transition, i.e., \( \text{Parent}(D) = \)
We also replace lines 9–14 from $M$.

Our abstraction refinement algorithm that guarantees the discovery of modular predicates $V$.

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 $\varphi_{init}$ constrains both $cnt$ and $lock$ to the value 0. The transition relation of the first thread is extended with $\rho_0$, 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, $\varphi_{err}$ 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., $m_4 \land n_3 \land \varphi_{err}$ is satisfiable.

From this error tuple, $\text{MKHORNCLAUSES}$ generates Horn clauses following the procedure from Figure 10. These Horn clauses are shown in Figure 13(b). The Horn clauses have unknown states $m_1$, $m_2$, $m_3$, $m_4$, $n_1$, $n_2$, $n_3$, $n_4$, $\varphi_{init}$, $\varphi_{err}$. 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 $(V_2, V_1)$, while the unknown states of thread 2 are restricted to $(V_2, V_3)$. The unknown transitions are $e_1(V, V')$ and $e_2(V, V')$.

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 $b \neq b'$. Second, we assume that $HC$ contains a clause $\ldots \rightarrow false$.

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.

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 $V_i$, rather than the whole set of program variables $V$. Solutions for unknown transitions are restricted to the set of global variables $V_2, V_3$. To implement these changes, we change line 1 from the $\text{REFINE}$ procedure as follows:

$$HC := \{S_1(V_2, V_1) \land \cdots \land S_n(V_2, V_n) \land \varphi_{err}\}$$

We also replace lines 9–14 from $\text{MKHORNCLAUSES}$ with the fragment shown in Figure 11. The rest of the function $\text{MKHORNCLAUSES}$ 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 $\text{MKHORNCLAUSES}$ from Figure 10.
Figure 13. (a) Reachability trees constructed by ARE 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 ANNOPRED, 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 \( \text{Pred}(n) \) 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 \( \text{Pred} \) of \( n \)'s children is computed using a recursive call at line 5. After completing the recursive call, \( \text{Pred}(n) \) is calculated by adding the values of the \( \text{Pred} \) 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 \( \text{Pred} \) attributes of all these nodes. Therefore, at lines 8–9 we compute solutions for each \( b(u) \) in UnkRel by taking conjunction of \( \text{Pred} \) 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:

\[
\begin{align*}
\text{cnt}^3 &\geq 1 \\
\text{Lock}(7) &\text{ Label}(8) &\text{Label}(11)
\end{align*}
\]

Our solver treats each linear inequality as a conjunction of two linear inequalities. The equality \( \text{cnt}^2 = 0 \) is split in two inequalities \( \text{cnt}^2 \leq 0 \land -\text{cnt}^2 \leq 0 \). The proof of unsatisfiability is:

\[
(1 \leq \text{cnt}^3) \land (\text{cnt}^3 \leq \text{cnt}^2) \land (\text{cnt}^2 \leq 0) \land (1 \leq 0).
\]

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 ANNOPRED. 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.

\[
\begin{align*}
\text{ SOL}(\text{m}_1(V)) &= \text{ SOL}(\text{n}_2(V)) = (0 \leq 0) \\
\text{ SOL}(\text{e}_1(V, V')) &= (0 \leq 0) \\
\text{ SOL}(\text{m}_2(V)) &= \text{ SOL}(\text{m}_3(V)) = (1 \leq \text{ cnt}) \\
\text{ SOL}(\text{n}_3(V)) &= \text{ SOL}(\text{e}_2(V, V')) = (\text{cnt} \leq 0)
\end{align*}
\]

**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 \( \text{cnt}^2 \) does not appear in the subtree of the node 9 since Label(9) = \( "e_2(V_3, V_2)" \). Part of this subtree is the node 11. Let us name the variable at this node as cnt.VIII. The proof of unsatisfiability shown above does no longer hold, since the following formula is satisfiable:

\[
(1 \leq \text{cnt}^3) \land (\text{cnt}^3 = \text{cnt}^2) \land (\text{cnt}^\text{VIII} = 0)
\]

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:

\[
\begin{align*}
\text{lock}^2 &= 1 \land \text{lock}^2 = 0 \\
\text{Label}(8) &= (\text{cnt} \leq 0)
\end{align*}
\]

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

\[
\begin{align*}
\text{Pred}(8) &= (1 \leq \text{lock}^2) \\
\text{Pred}(12) &= (\text{lock}^2 \leq 0) \\
\text{Pred}(3) &= (1 \leq \text{lock}^2) \\
\text{Pred}(9) &= (\text{lock}^2 \leq 0) \\
\text{Pred}(2) &= (1 \leq 0)
\end{align*}
\]

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

**8. Experimental results**

In this section, we describe a proof-of-concept implementation of our proposed algorithm as an extension of the model checker ARMIC [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
  1  p, q := new nodes
  2  match g with
  3    | false ->
  4      {b₁(w₁) ∧ ⋯ ∧ bₙ(wₙ) → false, ⋯} := HC
  5  6  σ := [z₁/w₁] ⋯ [zₙ/wₙ]
  7    | b(u) →
  8      {b₁(w₁) ∧ ⋯ ∧ bₙ(wₙ) → b(w), ⋯} := HC
  9  10  σ := [z₁/w₁] ⋯ [zₙ/wₙ][u/z]
  11  12  Label(p) := {b₁(w₁)σ | i ∈ 1..n ∧ bᵢ = (≤)}
  13  14  Children(p) := ∅
  15  16  Children(q) :=
  17  {p} ∪ \bigcup \{MkTree(bᵢ(wᵢ)σ | i ∈ 1..n ∧ bᵢ ≠ (≤))\}
  return q
end

Figure 15. Function MkTree. Fresh copies are created consistent, e.g., fresh copies of \{v₁, v₂\}, \{v₁, v₃\} returns \{f₁, f₂\}, \{f₁f₂f₃\}, where f₁, f₂, f₃ are fresh variables that do not appear anywhere else.

procedure ANNOTPred
input
  n - node of Horn tree
begin
  1  if Children(n) = ∅ then
  2    Pred(n) := \sum \{Proof(b(u))·b(u) | b(u) ∈ Label(n)\}
  3  else
  4    for each n' ∈ Children(n) do
  5      ANNOTPred(n')
  6    Pred(n) := \sum \{Pred(n') | n' ∈ Children(n)\}
  end
end

Figure 16. Procedure ANNOTPred.

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 naive 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 (pc₁ and pcₑ) 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.
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.

<table>
<thead>
<tr>
<th>Benchmark programs</th>
<th>Our algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>LOC</td>
</tr>
<tr>
<td>Fig2-cex[20]</td>
<td>33</td>
</tr>
<tr>
<td>Fig2-fixed[20]</td>
<td>38</td>
</tr>
<tr>
<td>Fig4-cex[20]</td>
<td>175</td>
</tr>
<tr>
<td>Fig4-fixed[20]</td>
<td>168</td>
</tr>
<tr>
<td>Bluetooth2[30]</td>
<td>90</td>
</tr>
<tr>
<td>Bluetooth2-fixed</td>
<td>90</td>
</tr>
<tr>
<td>Bluetooth3-fixed</td>
<td>90</td>
</tr>
<tr>
<td>Scull[6]</td>
<td>451</td>
</tr>
<tr>
<td>Dekker[1]</td>
<td>39</td>
</tr>
<tr>
<td>Peterson[1]</td>
<td>26</td>
</tr>
<tr>
<td>Readers-writer-lock[10]</td>
<td>22</td>
</tr>
<tr>
<td>Time varying mutex[10]</td>
<td>29</td>
</tr>
<tr>
<td>Szymanski[32]</td>
<td>43</td>
</tr>
<tr>
<td>NaïveBakery[22]</td>
<td>22</td>
</tr>
<tr>
<td>Bakery[18]</td>
<td>37</td>
</tr>
<tr>
<td>Lamport[19]</td>
<td>62</td>
</tr>
<tr>
<td>QRCU[23]</td>
<td>120</td>
</tr>
</tbody>
</table>

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 Boon [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