Thread-modular Abstraction Refinement*

Thomas A. Henzinger[1], Ranjit Jhala[1], Rupak Majumdar[1], and Shaz Qadeer[2]

[1] University of California, Berkeley
[2] Microsoft Research, Redmond

Abstract. We present an algorithm called TAR ("Thread-modular Abstraction Refinement") for model checking safety properties of concurrent software. The TAR algorithm uses thread-modular assume-guarantee reasoning to overcome the exponential complexity in the control state of multithreaded programs. Thread modularity means that TAR explores the state space of one thread at a time, making assumptions about how the environment can interfere. The TAR algorithm uses counterexample-guided predicate-abstraction refinement to overcome the usually infinite complexity in the data state of C programs. A successive approximation scheme automatically infers the necessary precision on data variables as well as suitable environment assumptions. The scheme is novel in that transition relations are approximated from above, while at the same time environment assumptions are approximated from below. In our software verification tool BLAST we have implemented a fully automatic race checker for multithreaded C programs which is based on the TAR algorithm. This tool has verified a wide variety of commonly used locking idioms, including locking schemes that are not amenable to existing dynamic and static race checkers such as ERASER or WARLOCK.

1 Introduction

Many model-checking tools for software analysis fall into two categories. The first category of tools, pioneered by SPIN [15], relies on the user-guided representation of a program as an abstract model, which is then checked for the desired properties. Several tools in this category support model extraction from code [7, 13, 16, 21], but they still rely on the user to define the granularity of the abstraction. The second category of tools, pioneered by SLAM [4], automatically refines an extracted model to the necessary precision. While highly successful, both approaches have their limitations. The SLAM approach is fully automatic but, so far, has been limited to sequential programs. This is unfortunate, because the traditional strength of model checking lies in the analysis of concurrent systems, such as multithreaded programs, in which errors are notoriously difficult to reproduce. The SPIN approach is well-suited for concurrency but relies on the insights of the user to find appropriate abstractions. This is also unfortunate, because the other traditional strength of model checking is its "push-button"

* This work was supported in part by the NSF grants CCR-0085949 and CCR-0234690, the DARPA grant F33615-00-C-1693, and the MARCO grant 98-DT-660.

characteristics, which in the realm of hardware verification has enabled the routine application of model-checking tools. We present an algorithm and tool that for the first time uses fully automatic abstraction refinement on concurrent, multithreaded programs.

The first ingredient of our algorithm is *thread-modular assume-guarantee reasoning*. The main source of complexity in multithreaded programs is the interaction between threads that operate on shared data. A model checker must analyze all possible interleavings of the actions of the various threads, resulting in the dreaded state-explosion problem. One method for controlling state explosion is thread-modular reasoning, first proposed by Jones [17] and first implemented in the CALVIN checker [9, 11] for multithreaded Java programs. To check a 2-threaded program $T_1||T_2$, CALVIN allows the programmer to specify suitable abstractions G_1 and G_2 for the transition relations T_1 of thread 1 and T_2 of thread 2, and then separately analyzes $T_1||G_2$ and $G_1||T_2$. This reasoning is "thread-modular" if the abstractions G_1 and G_2 constrain only the shared variables, and consequently the analyzed systems $T_1||G_2$ and $G_1||T_2$ each depend on the private variables of at most one thread. The abstraction G_2 is an environment assumption for thread 1, and G_1 constrains the environment of thread 2. Thread-modular reasoning, therefore, is a form of assume-guarantee reasoning [1, 2], which is sound for safety properties:

> If no error states are reachable in $T_1||G_2$ nor in $G_1||T_2$, and $T_1||G_2 \subseteq G_1$ and $G_1||T_2 \subseteq G_2$, then no error states are reachable in $T_1||T_2$. (AG)

While thread-modular reasoning is not complete for safety properties [18], it suffices for establishing the safety of many concurrency-control mechanisms commonly used in multithreaded programming [9]. The main obstacle to thread-modular reasoning is the annotation burden involved in specifying the environment assumptions G_1 and G_2. Although these assumptions can be inferred automatically for finite-state systems [10], in the presence of unbounded data they have to be supplied manually for tools such as CALVIN.

This limitation motivates the second ingredient of our algorithm, *counterexample-guided predicate-abstraction refinement*. Many relations between data values are irrelevant for the task of proving a desired property, but in general it is difficult to divine the relevant relations ("predicates" [12]). Indeed, the process of manually finding a suitable abstract model, which is neither too detailed to choke the model checker nor too coarse to invalidate the specification, often dominates the verification effort. This process has been automated by the abstract-check-refine paradigm [3, 6]: one starts with a very coarse abstraction, and if the model checker finds an abstract error trace that has no concrete counterpart, then one uses that trace to automatically refine the abstraction, and repeats the process. In particular, a theorem prover can be used to automatically discover predicates that rule out spurious counterexamples and thus are good candidates for abstraction refinement [14]. This approach to automatic abstraction refinement has been very successful for sequential programs [4], but has its problems when dealing with concurrency. Suppose that we try to find appropriate abstractions of the two thread transition relations T_1 and T_2, as well as abstractions of the two environment assumptions G_1 and G_2, by iteratively approximating all four relations from above until condition (AG) holds. We will see in Section 2 that this approximation scheme fails in many cases of practical interest: even if suitable thread-modular assumptions G_1 and G_2 exist, one may end up with approximations $t_1 \supseteq T_1$, $t_2 \supseteq T_2$, $g_1 \supseteq t_1||g_2$, and $g_2 \supseteq g_1||t_2$ such that error states are reachable in $t_1||g_2$ or $g_1||t_2$, but no new predicates on the shared variables can be discovered to remove these error traces. We overcome this hurdle by approximating environment assumptions from *below*, rather than above. The algorithm TAR ("Thread-modular Abstraction Refinement") operates with approximations of four relations: $t_1 \supseteq T_1$ and $t_2 \supseteq T_2$ are traditional overapproximations of the two thread transition relations, and are successively refined by the addition of new predicates. The approximations g_1 and g_2 of the environment assumptions, however, start out empty (which corresponds to there being no environment) and are successively weakened until $g_1 \supseteq t_1||g_2$ and $g_2 \supseteq g_1||t_2$. By interleaving the strengthening of the t_i relations and the weakening of the g_i relations in an appropriate way, we can prove that if environment assumptions G_1 and G_2 that satisfy condition (AG) exist, then they will be found in the limit, that is, on finite state spaces they are guaranteed to be found in a finite number of iterations. In other words, the TAR algorithm provides a sound combination of thread-modular assume-guarantee reasoning and counterexample-guided abstraction refinement which, in the case of finite-state systems, is also complete in the sense that thread-modular environment assumptions are found if they exist.

We have implemented the TAR algorithm in the model checker BLAST [14], which was originally designed for the verification of sequential C programs. BLAST deals with all aspects of C, such as function calls, structures, and pointer aliasing, and uses an on-the-fly algorithm for integrated reachability analysis and predicate discovery ("lazy abstraction"). The TAR extension of BLAST is targeted to find *data races* in multithreaded C programs. Unlike much of the exposition in this paper, which considers for simplicity only two threads, the tool deals with any number of concurrent threads. Data races are states in which either two different threads update a shared data variable (as opposed to a lock variable), or one thread updates and another thread reads the variable. Race detection can therefore be formulated as a safety verification problem. We have focused on race detection for two reasons. First, race checking requires no code annotations or specifications from the user, and is therefore particularly appealing to practicioners. Second, the absence of race conditions is a prerequisite for establishing a variety of more complicated correctness requirements.

Section 2 presents the TAR algorithm in a generic setting, together with soundness and termination (completeness) arguments. Section 3 develops a version of the algorithm in the specific setting of race detection for multithreaded C programs. It gives a producer-consumer example for which the TAR algorithm proves the absence of race conditions, whereas all existing race checkers available to us, both dynamic ("lockset"-based) [19] and static ("type"-based) [5, 8, 20], report false positives. Section 4 briefly discusses the implementation of the

TAR algorithm in BLAST. Section 5 summarizes our experience with the tool. We have applied the race checker with success to a number of examples that capture a variety of synchronization idioms commonly used in operating systems and databases, including schemes where the same variable is protected at different times by different locks, which again lie outside the scope of applicability of existing race checkers. We have used BLAST to check several thousand lines of multithreaded C code, such as Linux and Windows device drivers, and have verified the absence of race conditions. In the case of one Windows driver, a race was found, and although benign, its inspection led to the discovery of a real concurrency error in the code.

2 A Thread-modular Abstraction Refinement Algorithm

Thread-modular reasoning. A *2-threaded program* $\Pi = (Q, Q_0, T_1, T_2, E)$ consists of the following components. The state space has the form $Q = S \times P_1 \times P_2$, where S is the shared state space, and P_i is the private state space of thread i, for $i = 1, 2$. We write $Q_i = S \times P_i$ for the state space of thread i. There is a set $Q_0 \subseteq Q$ of initial states, a transition relation $T_i \subseteq Q_i^2$ for each thread i, and a set $E \subseteq Q$ of error states. Given a relation $t \subseteq Q^2$, a *t-trace* $q_0, q_1, \ldots, q_n$ is a finite sequence of states $q_j \in Q$ such that (1) $q_0 \in Q_0$ and (2) $(q_j, q_{j+1}) \in t$ for all $0 \leq j < n$. A state $q \in Q$ is *t-reachable* iff there is a t-trace whose last state is q. We write $R(t)$ for the set of t-reachable states. Given two relations $t_1 \subseteq Q_1^2$ and $t_2 \subseteq Q_2^2$, the interleaving $(t_1||t_2) \subseteq Q^2$ is the relation defined by $((s, p_1, p_2), (s', p_1', p_2')) \in (t_1||t_2)$ iff either $((s, p_1), (s', p_1')) \in t_1$ and $p_2 = p_2'$, or $((s, p_2), (s', p_2')) \in t_2$ and $p_1 = p_1'$. We write $\sigma[t_i] \subseteq Q_i^2$ for the relation defined by $((s, p_i), (s', p_i')) \in \sigma[t_i]$ iff $((s, \hat{p}_i), (s', \hat{p}_i')) \in t_i$ for some private states $\hat{p}_i, \hat{p}_i' \in P_i$. The relation $\sigma[t_i] \supseteq t_i$ on the state space of thread i conforms with t_i on the shared state, but updates the private state in arbitrary ways. We write $\rho_i[t_1, t_2] \subseteq Q_i^2$ for the relation defined by $((s, p), (s', p')) \in \rho_i[t_1, t_2]$ iff there is some state (s, p_1, p_2) that is $(t_1||t_2)$-reachable and $((s, p_1), (s', p_1')) \in t_i$ for some p_1'. The relation $\rho_i[t_1, t_2]$ is the restriction of $\sigma[t_i]$ to the $(t_1||t_2)$-reachable states.

The program Π is *safe* iff $R(T_1||T_2) \cap E = \emptyset$. Safety means that there is no $(T_1||T_2)$-trace that ends in an error state. The program Π is *safe in a strongly thread-modular way* iff $R(T_1||\sigma[T_2]) \cap R(\sigma[T_1]||T_2) \cap E = \emptyset$. Strongly thread-modular safety means that there are no $(T_1||\sigma[T_2])$- and $(\sigma[T_1]||T_2)$-traces that end in the same error state. In the strongly thread-modular case, suitable thread-modular environment assumptions $\sigma[T_i]$ can be obtained noninductively, by existential quantification of the private variables of thread i. The general thread-modular case permits stronger environment assumptions, which may be obtained inductively by taking into account reachability information. Formally, the program Π is *safe in a thread-modular way* [9] iff there are environment assumptions G_1 and G_2 such that $G_1 \supseteq \rho_1[T_1, G_2]$ and $G_2 \supseteq \rho_2[G_1, T_2]$ and $R(T_1||G_2) \cap R(G_1||T_2) \cap E = \emptyset$. If a program is safe in a thread-modular way, then it can be proved without considering the product transition relation $T_1||T_2$;

it suffices to reason about thread 1 together with an environment assumption about thread 2 that concerns only the shared state, and vice versa.

Proposition 1a. Every program that is safe in a strongly thread-modular way is safe in a thread-modular way. Every program that is safe in a thread-modular way is safe. □

Proposition 1b. There is a program that is safe but not in a thread-modular way. There is a program that is safe in a thread-modular way but not in a strongly thread-modular way. □

The second part of Proposition 1a follows by circular assume-guarantee reasoning [2]. For Proposition 1b, the existence of a program that is safe but not in a thread-modular way follows from the incompleteness of thread-modular reasoning [10]. The following example shows a program that is safe in a thread-modular way but not in a strongly thread-modular way.

Example 1. Consider a 2-threaded program with shared variables m and x, both initially 0. The variable m represents a lock; its value is 0 if no thread holds the lock, and i if thread i holds it. Thread i executes the following code:

```
while (1) do { acquire(m); x=i; release(m); }
```

We model the program counter of thread i by a private variable pc_i initialized to 0. The transition relation T_i of thread i is

$$(pc_i = 0 \wedge pc'_i = 1 \wedge m' = m \wedge x' = x) \ \vee \ (pc_i = 1 \wedge m = 0 \wedge pc'_i = 2 \wedge m' = 1 \wedge x' = x)$$
$$\vee \ (pc_i = 2 \wedge pc'_i = 3 \wedge m' = m \wedge x' = i) \ \vee \ (pc_i = 3 \wedge pc'_i = 0 \wedge m' = 0 \wedge x' = x).$$

The error states E are those in which a data race occurs: $pc_1 = 2 \wedge pc_2 = 2$. For $i = 1, 2$:

$$\begin{aligned}
\sigma[T_i] : \ & (m' = m \wedge x' = x) \ \vee \ (m = 0 \wedge m' = i \wedge x' = x) \ \vee \\
& (m' = m \wedge x' = i) \ \vee \ (m' = 0 \wedge x' = x) \\
\rho_i[T_1, T_2] : \ & (m \neq i \wedge m' = m \wedge x' = x) \ \vee \ (m = 0 \wedge m' = i \wedge x' = x) \ \vee \\
& (m = i \wedge m' = m \wedge x' = i) \ \vee \ (m = i \wedge m' = 0 \wedge x = i \wedge x' = x)
\end{aligned}$$

The error state $pc_1 = 2 \wedge x = 0 \wedge m = 0 \wedge pc_2 = 2$ is reachable by the transition sequences $T_1, T_1, \sigma[T_2]$ and $T_2, T_2, \sigma[T_1]$; hence the program is not safe in a strongly thread-modular way. However, the program is safe in a thread-modular way, because the extra conjuncts in the last clauses of $\rho_1[T_1, T_2]$ and $\rho_2[T_1, T_2]$ ensure that all states in $R(T_1 || \rho_2[T_1, T_2])$ with $pc_1 = 2$ have $m = 1$, and all states in $R(\rho_1[T_1, T_2] || T_2)$ with $pc_2 = 2$ have $m = 2$, thus making the intersection empty. □

Abstraction refinement. To establish the correctness and termination of our algorithm, we treat abstraction refinement as a black box (oracle). A *refinement function* Θ is given two relations $(t_1, t_2) \subseteq Q_1^2 \times Q_2^2$ such that (1) either $T_1 \subseteq t_1$ and $T_2 \subset t_2$, or $T_1 \subset t_1$ and $T_2 \subseteq t_2$, and (2) there are $(t_1 || \sigma[t_2])$- and $(\sigma[t_1] || t_2)$-traces that end in error states, but none of them is a $(T_1 || T_2)$-trace. The function Θ returns a pair $(r_1, r_2) \subseteq Q_1^2 \times Q_2^2$ of relations such that (1) $T_i \subseteq r_i \subseteq t_i$ for $i = 1, 2$, and (2) either $r_1 \subset t_1$ or $r_2 \subset t_2$. Our implementation of Θ will look at the reasons why the abstract counterexamples (i.e., traces to error states) cannot be concretized, and will add new predicates that rule out some abstract counterexamples by removing at least one transition from t_1 or t_2, thus obtaining the new, refined abstract transition relations r_1 and r_2.

Naive abstraction refinement. Abstraction refinement usually proceeds from a very coarse overapproximation of the transition relations, which are successively refined until either all abstract counterexamples are ruled out, or a concrete counterexample is found. In our setting, this can be accomplished by the following algorithm.

Algorithm A. Initially $t_i := Q_i^2$ for $i = 1, 2$. Loop:

> If $R(t_1||t_2) \cap E = \emptyset$, then return "safe." If some $(t_1||t_2)$-trace that ends in an error state is a $(T_1||T_2)$-trace, then return "unsafe." Let $(t_1, t_2) := \Theta(t_1, t_2)$.

Algorithm A computes with relations on the product space $S \times P_1 \times P_2$. A thread-modular algorithm computes only with relations on $S \times P_1$ and $S \times P_2$. Algorithm B is a thread-modular version of A. It uses overapproximations t_i of T_i, and also thread-modular environment assumptions $g_i \subseteq Q_i^2$, for $i = 1, 2$.

Algorithm B. Initially $t_i := Q_i^2$ and $g_i := Q_i^2$ for $i = 1, 2$. Loop:

> If $R(t_1||g_2) \cap R(g_1||t_2) \cap E = \emptyset$, then return "safe." If some $(t_1||g_2)$- or $(g_1||t_2)$-trace that ends in an error state is also a $(T_1||T_2)$-trace, then return "unsafe." If $(t_1, t_2) = (T_1, T_2)$, then return "don't know." Let $(t_1, t_2) := \Theta(t_1, t_2)$ and let $(g_1, g_2) := (\sigma[t_1], \sigma[t_2])$.

Proposition 2a. If Algorithm B returns "safe," then the program is safe in a strongly thread-modular way. If Algorithm B returns "unsafe," then the program is not safe. □

Proposition 2b. If the state space Q is finite, then Algorithm B terminates. If additionally, the program is safe in a strongly thread-modular way, then Algorithm B terminates returning "safe." □

In other words, Algorithm B succeeds exactly for the (finite-state) programs that are safe in a strongly thread-modular way; if the state space is finite, but the program is not safe in a strongly thread-modular way, then the algorithm always terminates with either "unsafe" or "don't know." Unfortunately, as Example 1 showed, standard locking schemes are not strongly thread-modular. In particular, Algorithm B terminates with "don't know" on Example 1. To find the environment assumptions for programs that are thread-modular but not in a strong way, such as Example 1, we need to use a different approach.

Thread-modular abstraction refinement. The correctness and termination of Algorithms A and B were straightforward to prove, because both transition relations (t_1 and t_2) and both environment assumptions (g_1 and g_2) were approximated from above. The following algorithm approximates the transition

Iteration 1	
$t_1:$	$(pc_1 = 0 \wedge pc'_1 = 1) \ \vee\ (pc_1 = 1 \wedge pc'_1 = 2) \vee\ (pc_1 = 2 \wedge pc'_1 = 3) \ \vee\ (pc_1 = 3 \wedge pc'_1 = 0)$
$t_2:$	$(pc_2 = 0 \wedge pc'_2 = 1) \ \vee\ (pc_2 = 1 \wedge pc'_2 = 2) \vee\ (pc_2 = 2 \wedge pc'_2 = 3) \ \vee\ (pc_2 = 3 \wedge pc'_2 = 0)$
$g_1:$	$\mathit{false},\ g_2:\ \mathit{false}$
Iteration 2	
$t_1:$	$(pc_1 = 0 \wedge pc'_1 = 1 \wedge m' = m) \ \vee\ (pc_1 = 1 \wedge pc'_1 = 2 \wedge m = 0 \wedge m' = 1) \ \vee$
	$(pc_1 = 2 \wedge pc'_1 = 3 \wedge m' = m) \ \vee\ (pc_1 = 3 \wedge pc'_1 = 0 \wedge m' = 0)$
$t_2:$	$(pc_2 = 0 \wedge pc'_2 = 1 \wedge m' = m) \ \vee\ (pc_2 = 1 \wedge pc'_2 = 2 \wedge m = 0 \wedge m' = 2) \ \vee$
	$(pc_2 = 2 \wedge pc'_2 = 3 \wedge m' = m) \ \vee\ (pc_2 = 3 \wedge pc'_2 = 0 \wedge m' = 0)$
$g_1:$	$\mathit{false},\ g_2:\ \mathit{false}$
Iteration 3	
$t_1,\ t_2:$	same as in iteration 2
$g_1:$	$(m' = m) \ \vee\ (m = 0 \wedge m' = 1) \ \vee\ (m = 1 \wedge m' = 0)$
$g_2:$	$(m' = m) \ \vee\ (m = 0 \wedge m' = 2) \ \vee\ (m = 2 \wedge m' = 0)$

Table 1. Running Algorithm Tar on Example 1.

relations from above, but approximates the environment assumptions from below, and stops with success only once the successively coarsened environment assumptions move above the successively refined transition relations.

Algorithm Tar. Initially $t_i := Q_i^2$ and $g_i := \emptyset$ for $i = 1, 2$. Loop:

> If $R(t_1||g_2) \cap R(g_1||t_2) \cap E = \emptyset$ and $\rho_1[t_1, g_2] \subseteq g_1$ and $\rho_2[g_1, t_2] \subseteq g_2$, then return "safe." If some $(t_1||g_2)$- or $(g_1||t_2)$-trace that ends in an error state is also a $(T_1||T_2)$-trace, then return "unsafe." If $R(t_1||g_2) \cap R(g_1||t_2) \cap E = \emptyset$, then let $(g_1, g_2) := (\rho_1[t_1, g_2], \rho_2[g_1, t_2])$; otherwise, if $(t_1, t_2) = (T_1, T_2)$, then return "don't know," else let $(t_1, t_2) := \Theta(t_1, t_2)$ and $(g_1, g_2) := (\emptyset, \emptyset)$.

Theorem 3a. If Algorithm Tar returns "safe," then the program is safe in a thread-modular way. If Algorithm Tar returns "unsafe," then the program is not safe. □

This is proved by circular assume-guarantee reasoning, because the environment assumption of each thread must be discharged against the abstract (overapproximate) transition relation of the other thread.

Theorem 3b. If the state space Q is finite, then Algorithm Tar terminates. If additionally, the program is safe in a thread-modular way, then Algorithm Tar terminates returning "safe." □

In other words, Algorithm Tar succeeds exactly for the (finite-state) programs that are safe in a thread-modular way. It automatically finds suitable thread-modular environment assumptions, provided they exist. If the state space is finite but the program is not safe in a thread-modular way, then the algorithm always terminates with either "unsafe" or "don't know."

Algorithm Tar **on Example 1**. The initial abstraction contains the program counter of each thread (predicates of the form $pc_i = 1$, $pc_i = 2$, etc.).

The resulting existential overapproximations of T_1 and T_2 are t_1 and t_2. In this abstraction, the error state $pc_1 = 2 \wedge x = 0 \wedge m = 0 \wedge pc_2 = 2$ is reachable. The counterexample analysis reveals that we should track the predicates $m = 0$, $m = 1$, and $m = 2$ (note that $m' = m$ is shorthand for $(m = 0 \Rightarrow m' = 0) \wedge (m = 1 \Rightarrow m' = 1) \wedge (m = 2 \Rightarrow m' = 2)$). At the end of iteration 2, there is no counterexample, because $R(t_1||g_2) \cap R(g_1||t_2) \cap E = \emptyset$. However, the test $\rho_1[t_1, g_2] \subseteq g_1$ fails, because g_1 is *false*, and $\rho_1[t_1, g_2]$ is $(m' = m) \vee (m = 0 \wedge m' = 1) \vee (m = 1 \wedge m' = 0)$. We update g_1 to $\rho_1[t_1, g_2]$ and g_2 to $\rho_2[g_1, t_2]$ for iteration 3. In iteration 3, again there is no error, as $R(t_i||g_{3-i})$ implies $pc_i = 2 \Rightarrow m = i$, for $i = 1, 2$. Moreover $\rho_1[t_1, g_2] = g_1$ and $\rho_2[g_1, t_2] = g_2$. Thus the g_i's are suitable environment assumptions, and the program is safe (in a thread-modular way). □

3 Race Detection

Data races. We now define the race-detection problem on a 2-threaded program $\Pi = (Q, Q_0, T_1, T_2, E)$. For the remainder of the paper, i ranges over $\{1, 2\}$. The program has a set *Shared* of variables visible to both threads, and sets $Private_i$ of variables visible only to thread i. Hence the shared (resp. private) state space S (resp. P_i) is the set of all valuations to the variables in *Shared* (resp. $Private_i$). The program counter of thread i is included in $Private_i$. For each variable $x \in Shared$, let $Write_i(x) \subseteq Q_i$ (resp. $Read_i(x) \subseteq Q_i$) denote the set of states from which thread i writes (resp. reads) x. These states can be computed by a simple syntactic analysis of the program. For instance, in the program of Example 1, $Write_i(x)$ is defined by the predicate $pc_i = 2$, and $Read_i(x)$ is *false*. The *race-detection problem* for a variable $x \in Shared$ asks if a state in $E_x = \cup_{i \in \{1,2\}}((Write_i(x) \cup Read_i(x)) \cap Write_{3-i}(x))$ is $(T_1||T_2)$-reachable. Intuitively, there is a data race on the variable x if the program can reach a state in which both threads have enabled actions that access (read or write) x, and at least one of these accesses is a write. For the program of Example 1, the set E_x is $pc_1 = 2 \wedge pc_2 = 2$.

Havoc abstractions. We have implemented a variant of Algorithm Tar to check for data races: we further approximate each environment assumption g_i by predicates that use the domain (first coordinate) of the relation g_i, but ignore the effect of a transition (second coordinate). For each thread i, the algorithm maintains a *havoc* map h_i on *Shared*, such that $h_i(x) \subseteq Q_i$ for every variable $x \in Shared$. Every havoc assumption $h_i(x)$ gives an approximation of the set of states from which thread i can write to x. The environment assumption g_i induced by the havoc map h_i may change x to any arbitrary value ("havoc x") in a shared state s if there is some private state p_i such that $(s, p_i) \in h_i(x)$; otherwise g_i leaves x unchanged. The initial havoc map assigns to every shared variable the empty set (represented by the predicate *false*). Hence, each thread initially assumes that the other threads do not modify any shared variable. Each iteration of Algorithm Tar updates the havoc map as follows. Whenever the overapproximations t_1 and t_2 of the thread transition relations are refined

```
thread Producer                      thread Consumer
 1':while (1) {                       1: while (1) {
 2':   while (flag != 0) {};          2:    while (flag != 1) {};
 3':   data = get_new_data();         3:    copy = data;
 4':   flag = 1;                      4:    flag = 0;
     }                                    }
```

Fig. 1. A producer-consumer system.

(by Θ), then all havoc assumptions are reinitialized to *false*. Otherwise, $h_1(x)$ (resp. $h_2(x)$) is updated to $R(t_1||g_2) \cap \mathit{Write}_1(x)$ (resp. $R(g_1||t_2) \cap \mathit{Write}_2(x)$).

Example 1 with havoc. In Example 1, $\mathit{Write}_i(m)$ is $(pc_i = 1 \wedge m = 0) \vee pc_i = 3$. After iteration 2, the havoc assumption $h_1(m)$ is $(pc_1 = 1 \wedge m = 0) \vee (pc_1 = 3 \wedge m = 1)$, meaning that when analyzing thread 2, the environment may change the value of m arbitrarily in states where $m = 0 \vee m = 1$. □

Algorithm TAR **with havoc.** The input is a multithreaded program, a set $X \subseteq \mathit{Shared}$ of variables on which we check for data races, and a set of predicates that include control information about each thread.

Initialization ("Seed assumption") Set the havoc assumption for each thread and each shared variable to *false*, i.e., the environment does nothing. The initial transition relation for each thread is the (existential) predicate abstraction of the thread's transition relation w.r.t. the given predicates.

Step 1 ("Reachability analysis") Compute an abstraction R_i of the reachable states of each thread i, based on the current havoc assumptions about the behavior of the other threads, and the present set of predicates.

Step 2 ("Counterexample analysis") Check if the reach sets R_i computed in step 1 contain states with data races, i.e., check if $R_1 \cap R_2 \cap E_x$ is nonempty for some variable $x \in X$. If the intersection is empty for all $x \in X$, then go to step 3. Otherwise, check if the abstract counterexample corresponds to a concrete program trace. If it does, then report the bug ("unsafe"); otherwise (a) discover new predicates that rule out the spurious counterexample and add them to the set of predicates, thus refining the transition relation of each thread, and (b) reset the havoc assumptions for each thread to *false*, and then go to step 1.

Step 3 ("Discharge assumptions") Check if the havoc assumptions are sound, i.e., for each thread i and variable $x \in \mathit{Shared}$ check if $h_i(x) \subseteq R_i \cap \mathit{Write}_i(x)$. If so, report "safe"; otherwise update the havoc assumptions w.r.t. the present reach sets as discussed above, and then go to step 1.

Example 2. The 2-threaded program of Figure 1 has a `Producer` thread that produces new data items by writing to the variable `data`, and a `Consumer` thread that consumes the data items by reading from `data`. We illustrate how our algorithm verifies the absence of races on the shared variable `data`. The example is correct, because `Producer` writes to `data` only when it knows that `flag` is 0, which happens after `Consumer` has removed the item that was put there last; and `Consumer` reads only when `Producer` is done with putting new data in, which it knows has happened when `flag` is set to 1. As the race-freedom depends on the value of the variable `flag`, and not on explicitly declared locks (as in Example 1, where m was a lock), existing static and dynamic race checkers report false positives for this program.

Initialization The havoc assumptions for both threads are set to *false*. In other words, when analyzing `Consumer` we assume that `Producer` does not affect the shared state in any way, and vice versa. The initial set of predicates is empty, but we track control flow (i.e., program counter values) explicitly.

Iteration 1

Step 1_1 Since we do not track any predicates, reachability analysis finds that for both threads the entire state space is reachable.

Step 2_1 To check if the current reach sets are error-free, we check if they contain a state with `Consumer` in location `3` and `Producer` in location `3'` (where `data` is accessed). For this, we check the intersection of the regions reachable in locations `3` and `3'` for emptiness. Since the respective reachable regions are *true*, we find the intersection nonempty. From the reachability analysis, we have a trace for `Consumer` that leads to `3` with region *true*, and a corresponding trace for `Producer`. We submit both finite traces to counterexample analysis (this routine is discussed in the next section), which reports that the predicates $\mathit{flag} = 1$ and $\mathit{flag} = 0$ are important. We reset the havoc assumptions for both threads to *false* and add these two predicates to be tracked.

Iteration 2

Step 1_2 Once the reach set for `Consumer` is recomputed using the enlarged set of predicates, we find that the region reachable in locations `3` and `4` is $\mathit{flag} = 1$. Hence, `Consumer` reads `data` or modifies `flag` only when $\mathit{flag} = 1$. This stems from the assumption that `Producer` never modifies `flag`, which follows from the current havoc assumption *false*. Similarly, in the recomputed reach set for `Producer`, the variables `data` or `flag` are modified only when $\mathit{flag} = 0$.

Step 2_2 We check if the present reach sets contain races. As the intersection of the states when `Consumer` and `Producer` access `data` is $\mathit{flag} = 1 \wedge \mathit{flag} = 0$, which is empty, we conclude that there are no races on `data`.

Step 3_2 We check the soundness of the havoc assumptions by checking if for each thread, the havoc assumptions for `data` and `flag` contain the set of states where the thread modifies the variable. This is not the case, as the havoc assumptions are *false*, but the region where `Consumer` (resp. `Producer`) modifies the variables is $\mathit{flag} = 1$ (resp. $\mathit{flag} = 0$). Hence we update the havoc assumptions to the new reach sets. In particular, the new havoc assumption of `flag` for `Consumer` (resp. `Producer`) is $\mathit{flag} = 1$ (resp. $\mathit{flag} = 0$), that is, when analyzing the `Consumer` (resp. `Producer`) we now assume that the environment havocs `flag` only when $\mathit{flag} = 0$ (resp. $\mathit{flag} = 1$).

Iteration 3

Step 1_3 We recompute the reach set of each thread with the new havoc assumptions. When `Consumer` breaks out of the loop at location `1`, the state is $\mathit{flag} = 1$. Subsequently, the environment cannot havoc `flag`, because the current havoc assumption is $\mathit{flag} = 0$. Thus, the state at which `Consumer` accesses `data` has still $\mathit{flag} = 1$.

Step 2_3 As the reach sets are the same as in iteration 2, there is no race.

Step 3_3 This time we find that the havoc assumptions are sound, because they contain the reach sets. Thus we conclude that the current reach sets are over-approximations of the reachable states of the program, and hence there are no races on `data`. □

4 Implementation in BLAST

We have implemented Algorithm TAR in the model checker BLAST [14]. The checker abstracts transition relations and havoc maps using predicate abstraction [12]. The program counter and stack of each thread are kept concrete, but the shared and private state of a thread are abstracted to a boolean combination of a finite set of predicates over program variables.

Counterexample analysis. An *atomic region* for a thread consists of the program counter, the stack, and a boolean combination of predicates. For each atomic region a in the reach set $R(t_i||g_{3-i})$, the checker maintains a path $\mathit{path}(a)$ from an initial region to a, comprising of t_i and g_{3-i} transitions. There is a possible data race on x if there is a pair of atomic regions $a \in R(t_i||g_{3-i})$ and $a' \in R(g_i||t_{3-i})$ such that $a \subseteq \mathit{Read}_i(x) \cup \mathit{Write}_i(x)$ and $a' \subseteq \mathit{Write}_{3-i}(x)$ and $a \cap a' \neq \emptyset$. If the intersection of the regions a and a' is nonempty, then the checker analyzes $\mathit{path}(a)$ and $\mathit{path}(a')$. Our heuristic refinement procedure ignores thread interleavings and symbolically executes $\mathit{path}(a)$ (resp. $\mathit{path}(a')$) replacing the abstract t_i (resp. t_{3-i}) transitions with concrete T_i (resp. T_{3-i}) transitions. It then uses a theorem prover to check if the conjunction of the resulting symbolic stores is satisfiable. If the conjunction is satisfiable, then it reports a possible race. Note that because the tool ignores thread interleavings, it may return false positives (so far, we have not encountered false positives in practice). If the conjunction is unsatisfiable, then the proof of unsatisfiability is mined for new predicates that are used to refine the abstract transition relations [14].

Thread symmetry. If both threads execute the same code, but differ only in the thread identifier, we can optimize the algorithm by computing the reachable set of states for one thread, and obtaining the reachable set of states for the other thread by syntactic renaming. For example, in a variant of Example 1, if both threads run the code of thread 1, we can compute the set of reachable states of thread 2 from those of thread 1 by simultaneously substituting $m = 2$ for $m = 1$, $m = 1$ for $m = 2$, and pc_1 by pc_2. This enables our implementation to deal with an unbounded number of threads running the same code.

Benchmark	LOC	Iterations outer	Iterations inner	Theorem prover queries	Time (sec)
Simple	27	3	97	693	1.208
Simple (buggy)	36	2	131	466	0.220
Producer-Consumer	73	4	505	4349	5.048
Producer-Consumer (buggy)	73	1	149	331	0.403
Time-varying	58	4	612	11219	9.653
Time-varying (buggy)	58	1	61	190	0.259
aironet	1513	3	30955	227735	713.71
packet	4085	3	654	4336	11.610

Table 2. Race-checking benchmarks. LOC is lines of code. The number of outer iterations is the number of times the environment assumptions are reset to *false*. The number of inner iterations is the number of times the environment assumptions are updated after the last time they are reset to *false*. Theorem prover queries is the total number of theorem prover queries. Time is the total running time for the tool in seconds on a 700MHz Linux PC with 1GB RAM.

5 Experience

Locking schemes. We applied BLAST to a number of small examples that implement different synchronization idioms commonly used in systems code. These include the locking example from Section 2, the producer-consumer synchronization from Section 3, and a time-varying synchronization example from [9]. In the time-varying example, the threads use different locks to access the same shared data at different points in the execution, based on the current state of the program. For each benchmark, we created a second, buggy version by deliberately introducing a bug. In each case, our tool was able to detect the absence or presence of races correctly in a few seconds. Table 2 shows the results of running BLAST on the three benchmarks.

Device drivers. We also ran the tool on much larger Linux and Windows NT device drivers. As these drivers can be called by any number of clients concurrently, we are checking for the absence of races on shared variables (typically the state variables maintained by the driver) in an unbounded number of parallel threads. We checked the dispatch routines `readrid` and `writerid` of the aironet4500 Linux device driver. For this, we modeled the functions `spin_lock` and `spin_unlock` of the Linux kernel manually, as in Example 1, and kernel calls to the driver by an infinite loop that calls the methods `readrid` and `writerid` nondeterministically. Our tool reported that these routines are safe (no data races). We also ran our tool on a network packet driver from the Microsoft Windows DDK. We found two race conditions in this program, which has more than 4,000 lines of C code. The race conditions were benign, because the operations modifying the shared state were atomic. However, further inspection of the traces revealed a real concurrency bug.

References

1. M. Abadi, L. Lamport. Conjoining specifications. *ACM TOPLAS* 17:507–534, 1995.
2. R. Alur, T.A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, 1999.
3. R. Alur, A. Itai, R.P. Kurshan, M. Yannakakis. Timing verification by successive approximation. *Information and Computation*, 118:142–157, 1995.
4. T. Ball, S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proc. POPL*, pp. 1–3. ACM, 2002.
5. C. Boyapati, M. Rinard. A parameterized type system for race-free Java programs. In *Proc. OOPSLA*, pp. 56–69, 2001.
6. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. CAV*, LNCS 1855, pp. 154–169. Springer, 2000.
7. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, R.S. Laubach, H. Zheng. BANDERA: extracting finite-state models from Java source code. In *Proc. ICSE*, pp. 439–448. IEEE, 2000.
8. C. Flanagan, S.N. Freund. Detecting race conditions in large programs. In *Proc. PASTE*, pp. 90–96. ACM, 2001.
9. C. Flanagan, S.N. Freund, S. Qadeer. Thread-modular verification for shared-memory programs. In *Proc. ESOP*, LNCS 2305, pp. 262–277. Springer, 2002.
10. C. Flanagan, S. Qadeer. Thread-modular model checking. In *Proc. SPIN*, LNCS 2648, pp. 213–224. Springer, 2003.
11. C. Flanagan, S. Qadeer, S.A. Seshia. A modular checker for multithreaded programs. In *Proc. CAV*, LNCS 2404, pp. 180–194. Springer, 2002.
12. S. Graf, H. Saïdi. Construction of abstract state graphs with PVS. In *Proc. CAV*, LNCS 1254, pp. 72–83. Springer, 1997.
13. K. Havelund, T. Pressburger. Model checking Java programs using Java Pathfinder. *Software Tools for Technology Transfer*, 2:72–84, 2000.
14. T.A. Henzinger, R. Jhala, R. Majumdar, G. Sutre. Lazy abstraction. In *Proc. POPL*, pp. 58–70. ACM, 2002.
15. G.J. Holzmann. The SPIN model checker. *IEEE TSE*, 23:279–295, 1997.
16. G.J. Holzmann. Logic verification of ANSI-C code with SPIN. In *Proc. SPIN*, LNCS 1885, pp. 131–147. Springer, 2000.
17. C.B. Jones. Tentative steps toward a development method for interfering programs. *ACM TOPLAS* 5:596–619, 1983.
18. S. Owicki, D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.
19. S. Savage, M. Burrows, C.G. Nelson, P. Sobalvarro, T. Anderson. ERASER: a dynamic data race detector for multithreaded programs. *ACM TOCS*, 15:391–411, 1997.
20. N. Sterling. WARLOCK: a static data race analysis tool. In *Proc. USENIX Technical Conference*, pp. 97–106, 1993.
21. E. Yahav. Verifying safety properties of concurrent Java programs using three-valued logic. In *Proc. POPL*, pp. 27–40. ACM, 2001.