Race Checking by Context Inference*

Thomas A. Henzinger Ranjit Jhala
EECS Department, UC Berkeley, U.S.A.
{tah,jhala}@eecs.berkeley.edu

Rupak Majumdar
CS Department, UC Los Angeles, U.S.A.
rupak@cs.ucla.edu

ABSTRACT

Software model checking has been successful for *sequential* programs, where predicate abstraction offers suitable models, and counterexample-guided abstraction refinement permits the automatic inference of models. When checking *concurrent* programs, we need to abstract threads as well as the contexts in which they execute. Stateless context models, such as predicates on global variables, prove insufficient for showing the absence of race conditions in many examples. We therefore use richer context models, which combine (1) predicates for abstracting data state, (2) control flow quotients for abstracting control state, and (3) counters for abstracting an unbounded number of threads. We infer suitable context models automatically by a combination of counterexample-guided abstraction refinement, bisimulation minimization, circular assume-guarantee reasoning, and parametric reasoning about an unbounded number of threads. This algorithm, called CIRC, has been implemented in BLAST and succeeds in checking many examples of NESC code for data races. In particular, BLAST proves the absence of races in several cases where previous race checkers give false positives.

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: Languages, Verification, Reliability.
Keywords: Software model checking, race conditions.

1. INTRODUCTION

Data races are a major source of errors in concurrent programs. Race detection tools enable the construction of robust concurrent systems by finding, or confirming the absence of, races. They also allow more aggressive programming by detecting redundant synchronizations (by verifying the safety of the program without the synchronizations). Existing race checkers fall into two major categories: dynamic, lockset-based tools [25, 6] and static, type-based tools [4, 11]. Programmers, however, often use synchronization idioms that cause false positives for these tools (*i.e.,* the tool reports a possible race when there is none). Consider, for example, the "test-and-set" NESC program taken from [13] in Figure 1. Lockset- and type-based approaches falsely flag this program as potentially buggy, as it uses the *value* of the variable `state` instead of explicitly declared locks to guarantee race-freedom. The first thread to enter the atomic block sets its local variable `old` to 0 and the global `state` to 1, and gets to access `x`. The other threads copy the value 1 into their local copy of `old`, and the check `old` = 0 before accessing `x` precludes the possibility of a race on `x`. In many programs, the problem is harder as the accesses to the "protected" variable happen in procedures other than the ones where the variable `state` is toggled, and often happen only if the function that changes the "state" variable returns a particular value ("conditional locking"). Other synchronization mechanisms, such as the enabling and disabling of certain interrupts, are also beyond the scope of methods based on locks. A more precise path and interleaving sensitive analysis that tracks the values of variables is required to verify the absence of races.

Race detection is a safety verification problem for concurrent programs: a race occurs when two threads can access (read or write) a data variable simultaneously, and at least one of the two accesses is a write. The program is race-free if no such state is reachable. Thus, in principle, races can be detected (and their absence proved) using model checking. Concurrency, however, is a major practical obstacle to model checking: the interleaving of concurrent threads causes an exponential explosion of the control state, and if threads can be dynamically created, the number of control states is unbounded.

One approach [23] is to consider the system as comprising a "main" thread and a *context* which is an abstraction of all the other threads in the system, and then verifying (a) that this composed system is safe ("assume") and (b) that the context is indeed a *sound* abstraction ("guarantee"). Once the appropriate context has been divined, the above checks can be discharged by existing methods [14, 8, 22, 16, 12]. Additionally, the remaining data abstraction can be performed automatically using counterexamples [3, 20, 5]. Note that either check may fail due to imprecision in the context, leaving us with no information about whether the system is safe or not.

* This research was supported in part by the NSF grants CCR-0085949, CCR-0234690, and ITR-0326577.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLDI'04, June 9–11, 2004, Washington, DC, USA.
Copyright 2004 ACM 1-58113-807-5/04/0006 ..$5.00

Consequently, the main issues are: (a) what is a model for the *context* that is simultaneously (i) abstract enough to permit efficient checking and (ii) precise enough to preclude false positives as well as yield real error traces when the checks fail, and (b) how can we infer such a context automatically.

In [19], we addressed these issues as follows: (a) We chose as context model, a *relation* R on the global variables, which represents the possible effects that the other threads may have on the global state between any two transitions of the *main* thread, *i.e.,* at any point, the context could change the global variables from s to s' so long as $(s, s') \in R$. (b) We inferred such a context using counterexample-guided abstraction refinement.

Experiments showed that this *stateless* context model lacks the precision required to prove the safety of programs such as the ones described earlier, and to produce error traces for buggy programs. As context threads change the global variables depending on their local states, statelessness leads to false positives. Also, to generate error traces (and to refine abstractions) we must be able to check if an abstract trace corresponds to *some* concrete interleaving of the program's threads. This is difficult if the context has no information about the other threads' local states. For these reasons, the context must track the *local state* of its threads. Unfortunately, statefulness brings the burden of tracking the state of *each* of the arbitrarily many context threads. We present in this paper a richer model for contexts that solves both the above problems, and a generalization of the algorithm from [19] that constructs these richer context models automatically.

Stateful Contexts. First, we represent each context thread by an *abstract control flow automaton* (ACFA). Each ACFA location corresponds to a set of control locations of the thread, and we keep ACFAs minimal by computing weak bisimilarity quotients [7]. Each ACFA location is labeled by a formula over the globals, which constrains the possible values of the global variables. Second, we track the state of each of arbitrarily many context ACFAs by labeling each ACFA location with an integer counter (possibly ω), which represents the *number* of threads at that location ("counter ACFA"). Thus, our context models combine three forms of abstraction: predicates for data abstraction, weak bisimilarity quotients for control abstraction, and counters for abstracting multiple threads.

Context Inference. Suppose, for simplicity, that all threads run the same code as *main*. In general, our algorithm requires that each of the threads be running one of finitely many pieces of code, and that the threads do not reference each other. The inference of context models proceeds in two nested loops. The outer loop sets the context model to be the *strongest* model (which does not interfere with *main*) and then executes the inner loop. Given a predicate abstraction of *main*, and a counter ACFA that represents the multithreaded context, the inner loop iteratively weakens the context model until either (i) an abstract error is found, or (ii) the resulting counter ACFA overapproximates (simulates) the program. If (i) happens, we break out of the inner loop and analyze the abstract counterexample. If it is real we report the bug and exit, if it is spurious we add new predicates or refine the counter, and repeat the outer loop. If (ii) happens, we conclude (by assume-guarantee reasoning) that the program is free of races and exit. Otherwise (*i.e.,* neither (i) or (ii)), we weaken the context model by transforming the current reach set of *main* into a new ACFA and repeat the inner loop with the new, weaker context model. The whole process stops when either a concrete race is found, or the absence of races is proved using a context which overapproximates the program.

While our method applies to verifying any safety property of concurrent programs, we have focused on race detection for two reasons. First, race checking requires no code annotations or specifications from the user. Second, the absence of race conditions is a prerequisite for establishing a variety of more complicated correctness requirements.

Experimental Results. To demonstrate the practicality of the method, we have implemented this algorithm, called CIRC, in our C model checker BLAST [20]. The use of stateful contexts (ACFAs), their minimization, and the treatment of an unbounded number of threads using counters are new to BLAST . We ran the method on several networked embedded systems applications [13] which use the synchronization idioms mentioned above. We were able to find potential races in some cases and prove the absence of races in others.

Related Work. Type based race detectors [11, 4] provide strong type systems that guarantee the absence of races, but require code annotated with locking information. Additionally, control flow information may be used for more precision [26]. However, none of these methods can prove absence of races in programs with complex state-based synchronization idioms. Dynamic race detectors [25, 6] are effective in finding bugs but cannot guarantee their absence.

Software model checkers like SLAM [3] and BLAST [20] check sequential programs. Verisoft [14], Bandera [8], Feaver [22], Magic [5], and Java Pathfinder [16] check concurrent programs with a fixed finite number of threads. Verisoft runs on the concrete semantics of the program, the others require a user supplied abstraction. 3VMC [27] treats an unbounded number of threads, but again requires a user supplied abstraction. Calvin [12] requires that a suitable abstract context is provided. Magic [5] checks a finite number of concurrent threads communicating by message-passing. Since communication is explicit, abstraction and minimization are done independently of the other threads, *i.e.,* reachability information is not required. Parametric verification methods [24, 9, 2, 10] handle arbitrarily many threads using counters, but assume finite-state processes.

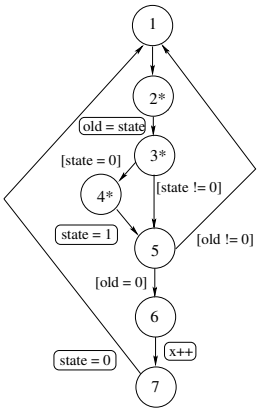
2. AN EXAMPLE

We begin with describing how our algorithm works on an example. The formal development of the technique is postponed to the next section. Consider the fragment of code shown in Figure 1, taken from a NESC program [13]. This fragment describes the behavior of a single thread; `x` and `state` are global variables and each thread has a local variable called `old`. The multithreaded program $\mathcal{P}$ has an arbitrary number of threads running this code concurrently. We wish to verify that there are no races on `x` in $\mathcal{P}$, *i.e.,* that $\mathcal{P}$ never reaches a state where two (or more) threads are about to access (read or write) `x`, and one of the accesses is a write.

2.1 Threads

Threads. We represent each thread as a *Control Flow Automaton* (CFA). The CFA is essentially the control flow

```
int x, state;
Thread() {
  int old;
1: while (1) {
     atomic{
2:     old := state;
3:     if (state = 0){
4:        state := 1;
       }
     }
5:   if (old = 0){
6:     x := x+1;
7:     state := 0;
     }
  }
}
```



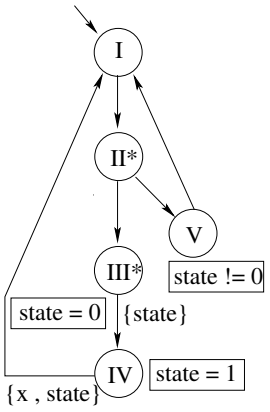


Figure 1: (a) Thread (b) CFA (c) ACFA

graph of the thread, with instructions labeling the edges instead of the vertices. A CFA consists of: (1) integer variables, local and global, that are accessed by the thread, (2) control locations, some of which are *atomic* and one of which is the distinguished *start location*, and (3) directed edges that connect the vertices. Each edge is labeled by either an assignment that is executed when the thread moves along the edge or by an *assume predicate* which must be true for control to move along the edge.

EXAMPLE 1: [Thread] Instead of a formal definition, consider the CFA shown in the middle in Figure 1 for the thread shown on the left in the same figure. The assignments are in the boxes and the assume predicates are labeled with $[\cdot]$. The vertices marked with $*$ are atomic locations. The `atomic` construct of NESC allows a sequence of operations to occur without preemption; atomic locations model this. If in a multithreaded program, a thread is at an atomic location, only that thread is allowed to execute. □

Informal Semantics. A multithreaded program is a set of threads where each thread is represented by a CFA. A *state* of a multithreaded program is a valuation for all the variables, including the global variables shared by all threads and each thread's local variables (*e.g.,* program counter). We shall assume for clarity that all threads have the same CFA. In the initial state, each thread is at the start location, and all the variables have value 0. The system evolves as follows. (1) A thread is scheduled: if some thread is at an atomic location, it gets to run, otherwise some thread is chosen non-deterministically. (2) The scheduled thread picks one of the out-edges of the location it is at and executes it and proceeds to the target of the edge. If the edge is an assume, this happens only if the state satisfies the predicate and the variables remain unchanged; if the edge is an assignment `x:=e` then the expression `e` is evaluated and written into `x`, and then the program moves to the target location of the edge. It can be checked that if the start location is not atomic, then in any reachable state at most one thread is at an atomic location.

Data Races. There is a *data race* on the variable `x` if the program can reach a state in which two or more threads have enabled actions that read or write `x`, and at least one of these accesses is a write. We say a thread can write (read) `x` if there is an out-edge from its location where `x` is assigned (read). Thus a state has a race on `x` if (1) no thread is at an atomic location, and (2) one thread is at a location where `x` may be written and another is at a location that may access `x`. In the program comprising threads of Figure 1, there are no races on `x` if in every reachable state, at most one thread is at location `6`.

2.2 Thread-Context Programs

We analyze a multithreaded program as a thread-context program (TCP), which comprises a *main* thread executing in a *context* which represents all the other threads. We model each of the context threads using an abstract thread.

Abstract Threads. An abstract thread is represented by an abstract control flow automaton (ACFA). An ACFA is a directed graph, whose vertices are *abstract control locations* labeled by predicates on the global variables of the program, and optionally by *atomic*, and whose edges are labeled by sets of *havoced* global variables. When the automaton moves from one location to the next, the havoced variables on the traversed edge are written to with arbitrary values, but the successor state is constrained to satisfy the predicate labeling the successor location.

EXAMPLE 2: [Abstract Thread] Figure 1(c) shows an ACFA for the thread of the example. Locations labeled "*" are atomic, and if there is an (abstract) thread at an atomic location, then only that (abstract) thread is scheduled. Each location is also labeled by a predicate inside a box, locations not labeled explicitly have the label `true`. Note this abstraction captures the essence of the behavior of the thread: first, it enters the atomic block, then if `state` is 0, it havocs `state` subject to the constraint that `state` is 1 in the next state. It then proceeds to access `x`, as it will have set its `old` to 0, and then havocs `state` to any arbitrary value. Alternately, if `state` is not 0 when the thread entered the block, then it would set its `old` to a non-zero value and thus loop back without writing to `x` or `state`. □

Informal Semantics. A TCP is a set $\{C\} \cup A^{\omega}$ comprising a main thread, represented by a CFA C, and a context which is an arbitrary number of abstract threads A. The semantics of a TCP are similar to that of a multithreaded program. At the initial location, the main thread is at the start location of C and each context thread is at the start location of

A. At each time step, either the main or a context thread is scheduled, and the scheduled thread makes a transition according to one of the out-edges of its current location.

2.3 Verification by Abstraction

Our method works by analyzing a TCP which is an abstraction of the program we wish to verify.

Abstraction Components

A precise analysis of the reachable states of a multithreaded program must abstract the state space to counter the infinite data valuations as well as the exponential number of possible program location tuples. Accordingly, we present three orthogonal abstractions.

1. Data Abstraction. The number of valuations for the program variables is infinite. To deal with this, we use predicate abstraction [15], where instead of tracking the exact values of variables, we track relationships between program variables captured by boolean formulae over a finite set of predicates over the variables. Any local variable in a predicate refers to the main thread's copy of the local.

2. Control Abstraction. The number of configurations of other "context" threads is exponential in the number of program locations of each thread. To ameliorate this exponential blowup, we represent each context thread as an *abstract thread* which is a state machine that (1) has fewer locations than, and (2) overapproximates the behavior of (*e.g.,* simulates), the thread it represents. All the predicates labeling the ACFA vertices are over the globals; information pertaining to the local state of context threads is encoded in the ACFA location.

3. Counters. There may be an arbitrary number of context threads. To make our analysis sound in this setting, we must model the ACFA location of each of the arbitrarily many context threads. To do this, we track the *number* of abstract threads that are at each of the finitely many ACFA control location [24]. Since this representation is infinite, we use a *counter abstraction*: we track the number precisely so long as it is less than or equal to a parameter k, and any number greater than k is abstracted to ω, meaning an arbitrary number of threads is at that abstract control location.[1]

Abstract Multithreaded Programs

Given a set of predicates P, and a natural number k, we say that $\hat{\mathcal{P}} = ((C, P), (A, k))$ is an *abstract multithreaded program*, which represents an abstraction of $\{C\} \cup A^\omega$. An abstract state of $\hat{\mathcal{P}}$ is the tuple $((pc, \varphi), \Gamma)$, where pc is the *main* thread's control location, φ is a boolean formula over the predicates P (local variables refer to the main thread's copy of the local variable), and Γ is a map from A's vertices to $\{0, \ldots, k, \omega\}$. The operations enabled at an abstract state are the operations enabled at pc and at each location n of A s.t. $\Gamma.n > 0$, so long as none of the above mentioned locations is atomic, otherwise, the enabled operations are the operations enabled at the (single) atomic location.

In the initial abstract state, the main thread is in the initial location of C, Γ is ω for the initial abstract location, and 0 elsewhere, and φ is the predicate abstraction (w.r.t. P) of the state where all variables are 0. For an abstract state $\hat{s} = ((pc, \varphi), \Gamma)$ and an operation op, the successor abstract state $\mathsf{post}.\hat{s}.\mathsf{op} = ((pc', \varphi'), \Gamma')$ is computed as follows. If the operation is the main thread's operation, then pc' is the target of the CFA edge taken, φ' is the predicate abstraction (w.r.t. P) of the strongest postcondition of φ w.r.t. the operation [15, 20], and $\Gamma' = \Gamma$. If it is a context ACFA moving across an abstract edge $n \to n'$, then $pc' = pc$, φ' is the predicate abstraction (w.r.t. P) of $(\exists y_1 \cdots y_k.\varphi) \wedge r.n'$ where $y_1 \cdots y_n$ are havoced on edge $n \to n'$ and n' is labeled with the predicate $r.n'$, and Γ' maps n to $\Gamma.n - 1$, n' to $\Gamma.n' + 1$, and all other n'' to $\Gamma.n''$.

Abstract Reachability. We build the set of reachable abstract states by iterating post from the initial abstract state until a fixpoint. We check if there are races by checking if any reachable state contains a race. If so, the reachability procedure returns an abstract error trace. We say that G is an *abstract reachability graph* (ARG) for $\hat{\mathcal{P}} = ((C, P), (A, k))$, if it is an ACFA that overapproximates the behavior of C in $\hat{\mathcal{P}}$.[2] The reachability procedure also computes an ARG G for $\hat{\mathcal{P}}$ which we use to guarantee the soundness of A. If A is an overapproximation of (*e.g.,* can simulate) G then we know that A is sound. If not, a minimized version of G gives us a better abstraction of the individual threads, which we use in the subsequent analysis.

2.4 The Algorithm CIRC

Given a CFA C, and a global variable $\mathtt{x}$, we wish to verify that in the multithreaded program comprising arbitrarily many copies of C running concurrently, there are no races on $\mathtt{x}$. In addition, the user may supply an initial set of predicates P (the default is $\emptyset$), and an initial counter parameter k (the default is 1).

Initialization ("Initial context") Set the initial ACFA A to be the empty ACFA, *i.e.,* the context does nothing.

Step 1 ("Reachability: Assume") Assuming that the context is made of threads behaving as A, compute the set of abstract reachable states of C using the present set of predicates P. Simultaneously build an abstract *reachability graph* (ARG) which is an ACFA G overapproximating the behavior of C in the current context (Algorithm ReachAndBuild in Section 3).

Step 2 ("Counterexample analysis") Check if the reachable states computed above contain states with races on $\mathtt{x}$. If there are no such states, **go to** step 3. Otherwise, check whether this trace is real by first generating a concrete sequence of interleaved thread operations (from the sequence of thread/ACFA operations) and then checking if the interleaved trace is feasible. The concretization of the ACFA trace is done using the ARG of which the ACFA is the minimized version. Hence, every sequence of ACFA operations, corresponds to a (possibly infeasible) path through the underlying CFA. If (a) it was not possible to generate the concrete trace as the counter was too low, increment k, (b) the concrete trace is infeasible, infer new predicates [18] and add them to the set of predicates P, (c) the concrete trace *is feasible* then **return** UNSAFE with the genuine error trace. Reset A to the empty context and **go to** step 1.

Step 3 ("Guarantee") Check that the A assumed in step 1 was sound by checking that it overapproximates G

[1] Note: $k + 1 = \omega$, $\omega + 1 = \omega$, and $\omega - 1 = \omega$

[2] We make this notion precise in Section 3

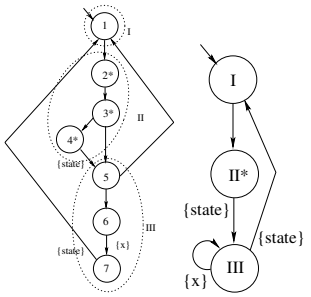


Figure 2: (a) ARG G_1 (b) Min. ARG A_1

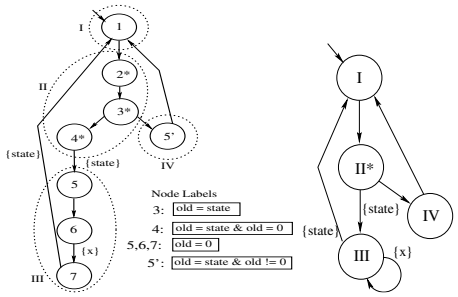


Figure 3: (a) ARG G_3 (b) Min. ARG A_3

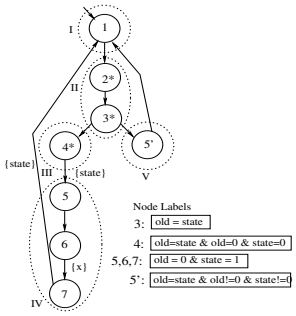


Figure 4: ARG G_5

computed in step 1 (Algorithm CheckSim). If so, **return** SAFE, else, set A to be the bisimulation minimization of G (Algorithm Collapse), and **go to** step 1.

Running CIRC. We shall now run the algorithm on the example of Figure 1. Recall that there is no race on `x`. The first thread that goes inside the `atomic` block sets `state` to 1 and subsequent threads always set their `old` to 1 and so do not write `state` or `x`. Once the original thread has set `state` back to 0 the other threads can make another attempt, in which they set their `old` to 0, set `state` to 1 and then access `x`.

Initialization The initial ACFA A_0 is set to be the empty ACFA. The initial set of predicates P_0 is empty, but control flow is explicitly tracked.

Iteration 1

Step $1_1, 2_1$ The ARG G_1 of ReachAndBuild is shown in Figure 2(a). All the control locations are reachable and the state is just `true`, *i.e.,* we know nothing about the values of the variables of C. The reachability is trivially free of races as the context threads do nothing.

Step 3_1 Since A_0 was empty, Algorithm CheckSim detects that A_0 does not overapproximate G_1 and hence A_0 is unsound. Thus, we minimize G_1 to get the new ACFA A_1 shown in Figure 2(b). The dotted circles denote the sets of G_1 states that are merged into a single A_1 state. The minimized ACFA starts at a non-atomic location, then moves into an atomic location, in which it havocs `state` and moves to a non-atomic location from which it again havocs {`x`, `state`} and returns to the start location. The locations `I,II` are not collapsed together as we wish to preserve atomicity, the same holds for `II,III`. Locations `I,III` are not collapsed as `x` can be written only in `III`. We repeat the loop setting A to be A_1.

Iteration 2

Step 1_2 On redoing reachability assuming the context threads behave as A_1 we find a race where one of the context threads moves two steps to reach the abstract location `III` (Figure 2(b)), following which the main thread moves to the concrete location `6`.

Step 2_2 We concretize the abstract trace described above and find that the thread followed an infeasible path: $\mathtt{1} \rightarrow \mathtt{2} \rightarrow \mathtt{3} \rightarrow \mathtt{5} \rightarrow \mathtt{6}$, *i.e.,* the trace is infeasible without even considering the other thread. From this trace, we learn the predicates $old = state$ and $old = 0$ are required to rule out this infeasible path. We add these to get the new set of predicates P_2, set the context ACFA A_2 to be the empty ACFA, and go back to step 1.

Iteration 3

Steps $1_3, 2_3, 3_3$ We repeat the reachability using A_2 and P_2, to get the ARG G_3, shown in Figure 3(a). Notice that this time, the only path to the location where the write is enabled is a feasible path for each thread. Again, the reach set is trivially error free. As G_3 is not overapproximated by A_2, the latter being the empty ACFA, we set A to be A_3 which is the result of minimizing G_3. This is shown in Figure 3(b). Note that the path that leads to `III` where to the write to `x` is enabled is feasible for the individual threads.

Iteration 4

Step 1_4 We recompute the reachability assuming the context has threads behaving as A_3, and the predicates P_2. The same abstract race as in step 1_2 is possible again.

Step 2_4 We concretize the trace from the previous step. This time, we get the feasible path $\mathtt{1} \rightarrow \mathtt{2} \rightarrow \mathtt{3} \rightarrow \mathtt{4} \rightarrow \mathtt{5} \rightarrow \mathtt{6}$ for the individual threads, but find that the composed trace, where the context thread follows the above path and *waits at* `6` then the main thread follows the same path to `6` is infeasible. This is because the first thread will set `state` to 1, and so the second thread cannot take the assume edge $\mathtt{3} \rightarrow \mathtt{4}$. The analysis reveals the predicates $state = 0$ and $state = 1$ rule out this behavior and we add these to our set to get P_4, set A_4 to be the empty ACFA and return to step 1.

Iteration 5

Steps $1_5, 2_5, 3_5$ We repeat the reachability using A_4 and P_4, to get the ARG G_5, shown in Figure 4. Notice that this time, the vertices in G_5 contain the values of `state`. The reach set is error free, but G_5 is not overapproximated by A_4, the latter being the empty ACFA, so we set A to be A_5 which is the result of minimizing G_5. This is the same as the ACFA shown in Figure 1(c). Notice that `II,III` are not collapsed as they differ on the values of predicate $state = 0$. Notice also, that in A_5, the various locations are labeled by predicates describing the value of `state` when the abstract thread is at that location. In particular, when a thread is at `IV`, the value of `state` is non-zero, thus preventing other threads from writing `x`.

Iteration 6

Step $1_6, 2_6$ We compute the ARG with the new ACFA A_5 with counter parameter still 1. We find a few more states, *e.g.,* after a thread sees in its atomic block that `state` is 1, it may see that it has been havoced, but this is not essential

as the thread still just returns to the head of the loop (since its old is still 0). There is no error possible as if a context ACFA goes first, it keeps state at 1 till after it has written x: so when the main thread takes the assume edge $3 \rightarrow 4$ ([state = 0]) the abstract state is empty ($state = 0 \wedge state = 1$ is unsatisfiable) meaning that edge is not behavior is not possible. Similarly, if the main thread gets in first, when a context thread attempts to take the abstract edge $2' \rightarrow 3'$, the abstract state is empty. The resulting ARG is G_6, and we proceed to step 3.

Step 3_6 We find that in fact G_6 is overapproximated by A_5 and so the context approximation is sound. We conclude the system is free of races.

3. MULTITHREADED PROGRAMS AND ABSTRACTIONS

In this section we shall define our model for multithreaded programs. First we define the semantics of such programs abstractly using transition systems. Then we define the syntactic representations of threads (CFAs) and abstract threads (ACFAs) and define their transition systems, and finally describe the semantics of multithreaded programs.

3.1 Shared Variable Transition Systems

Given a set X of variables, an X-state is a valuation of the variables in X. Let $\mathcal{V}_X$ be the set of all X-states. An X-transition relation is a subset of $\mathcal{V}_X \times \mathcal{V}_X$. A multithreaded program $\mathcal{P}$ is a set $\{(\rightsquigarrow_1, At_1), (\rightsquigarrow_2, At_2), \ldots\}$ where $\rightsquigarrow_i$ is an X_i transition relation and $At_i \subseteq \mathcal{V}_{X_i}$ is an atomic predicate. The set of variables of $\mathcal{P}$ is $X = \cup X_i$. The set of global variables of $\mathcal{P}$ is $\mathcal{P}.X_G = \{x \mid \exists i \neq j : (x \in X_i \cap X_j)\}$. The set of states of $\mathcal{P}$ is $\mathcal{V}_X$, and the semantics of $\mathcal{P}$ are given by an X-transition relation $\rightsquigarrow_{\mathcal{P}}$ and an atomic predicate $At \subseteq \mathcal{V}_X$ defined as follows: Define the predicate $\mathtt{En}.s.i$ where s is an X-state and i a thread as: $\mathtt{En}.s.i \equiv ((\exists j : (At_j.s)) \Rightarrow At_i.s)$. The atomic predicate $At.s \equiv \exists i : (At_i.s)$. The transition relation is defined as: $s \rightsquigarrow_{\mathcal{P}} s'$ iff there is an i such that (1) $\mathtt{En}.s.i$ (2) $t \rightsquigarrow_i t'$ (3) $\forall x \in X_i : (s.x = t.x \wedge s'.x = t'.x)$ (4) $\forall x \notin X_i : (s.x = s'.x)$. That is, if the thread i is enabled, then it updates its variables according to its transition relation, and all the other variables remain unchanged. The initial state s_0 of $\mathcal{P}$ maps every variable to 0. Let $\rightsquigarrow^*_{\mathcal{P}}$ be the reflexive transitive closure of $\rightsquigarrow_{\mathcal{P}}$. We define $[\![\mathcal{P}]\!] = \{s \mid s_0 \rightsquigarrow^*_{\mathcal{P}} s\}$ to be the set of reachable states of $\mathcal{P}$.

3.2 Threads: Control Flow Automata

Given a set of variables X, the set $\mathtt{Exp}.X$ is the set of arithmetic expressions over the variables X, the set $\mathtt{Pred}.X$ is the set of boolean expressions (arithmetic comparisons) over X, and the set $\mathtt{Op}.X$ is the set of instructions containing: (1) assignments x := e, where $\mathtt{x} \in X$ and $\mathtt{e} \in \mathtt{Exp}.X$, and (2) *assume predicates* asm [p], where $\mathtt{p} \in \mathtt{Pred}.X$, representing a condition that must be true for the edge to be taken.

A *control flow automaton* (CFA) is a tuple $\langle Q, q_0, X, \rightarrow, Q^* \rangle$, where (1) Q is a finite set of control locations, (2) $q_0 \in Q$ is the initial control location, (3) X is a set of variables, partitioned into X_G and X_L, disjoint sets of global and local variables, respectively, (4) $\rightarrow \subseteq (Q \times \mathtt{Op}.X \times Q)$ is a finite set of directed edges labeled with operations, and (5) $Q^* \subseteq Q$ is a set of "atomic" locations. An edge $(q, \mathtt{op}, q') \in \rightarrow$ is also written as $q \xrightarrow{\mathtt{op}} q'$.

For clarity we describe our method only for CFAs without function calls; we implement function calls in our tool.

A CFA $C = \langle Q, q_0, X, \rightarrow, Q^* \rangle$ induces a state space $\mathcal{V}_{X.C}$ where $X.C = X \cup \{pc\}$.[3] The atomic predicate $At.C$ of the CFA is $\{s \mid s.pc \in Q^*\}$, that is, a state is atomic if the thread is at an atomic location. We say that $s \overset{\mathtt{op}}{\rightsquigarrow} s'$ if: (1) if op is asm p then $s \models p$ and $s'.y = s.y$ for all $y \in X$, and (2) if op is x := e then $s'.x = s.e$ and for all $y \in X \setminus \{x\}$, $s'.y = s.y$. The transition relation $\rightsquigarrow_C$ is defined as follows: $s \rightsquigarrow_C s'$ if there exists some op such that $s.pc \xrightarrow{\mathtt{op}} s'.pc$ in the CFA C, and $s \overset{\mathtt{op}}{\rightsquigarrow} s'$.

A set of states π over the set of variables X is called a *data region*. A predicate over X represents a data region consisting of all valuations that satisfy the predicate. We lift the transition relation to sets of states by defining the *strongest postcondition* operation $sp.\pi.(q, \mathtt{op}, q') = \exists x' : \pi[x'/x] \wedge x = e[x'/x]$ if op is x := e and $\pi \wedge p$ if op is asm p. Note that $sp.\pi.(q, \mathtt{op}, q')$ represents the set $\{s' \in \mathcal{V}_X \mid \exists s \in \pi : (s \overset{\mathtt{op}}{\rightsquigarrow} s')\}$.

We can now describe multithreaded programs and their semantics. For clarity we shall restrict ourselves to *symmetric* multithreaded programs where each thread runs the same code, *i.e.,* has the same CFA C. Let C^ω denote the symmetric multithreaded program running an arbitrary number of copies of the CFA C. Formally, C^ω is the program $\{(\rightsquigarrow_1, At_1), (\rightsquigarrow_2, At_2), \ldots\}$ where each pair $(\rightsquigarrow_i, At_i)$ is defined as follows. Let C_i be the CFA C with each local variable $x \in X_L \cup \{pc\}$ renamed to x_i. Then $\rightsquigarrow_i = \rightsquigarrow_{C_i}$ and $At_i = At.C_i$.

3.3 Thread-Context Programs

We shall consider multithreaded programs in which a main thread runs in a context of an arbitrary number of abstract threads. For this, we define abstract threads, and Thread-Context programs (TCPs).

Abstract Threads: Abstract Control Flow Automata. An *Abstract CFA* (ACFA) is a tuple $\langle Q, q_0, X, \rightarrow, Q^*, r \rangle$, where (1) Q is a finite set of abstract locations, (2) $q_0 \in Q$ is a *start location*, (3) X is a set of variables partitioned into X_G and X_L, disjoint sets of global and local variables, respectively, (4) $\rightarrow \subseteq (Q \times 2^X \times Q)$ is a finite set of directed *havoc edges* labeled with subsets of X, (5) $Q^* \subseteq Q$ is a set of atomic abstract locations, and (6) $r : Q \rightarrow \mathtt{Pred}.X$ is a location labeling function labeling each location with a an abstract data region. An edge $(q, \mathtt{Y}, q')$ is also written as $q \xrightarrow{\mathtt{Y}} q'$,

An ACFA $A = \langle Q, q_0, X, \rightarrow, Q^*, r \rangle$ induces a state space $S.A \subseteq \mathcal{V}_{X.A}$ where $X.A = X \cup \{pc\}$ and $s \in S.A$ iff $s \models r.(s.pc)$. The atomic predicate $At.A$ of the ACFA is $\{s \mid s.pc \in Q^*\}$, that is, a state is atomic if the abstract thread is at an atomic location. The transition relation $\rightsquigarrow_A \subseteq S.A \times S.A$ of an ACFA A is defined as $s \rightsquigarrow_A s'$ if: (1) $s.pc \xrightarrow{\mathtt{op}} s'.pc$ and (2) if op is Y then for each $x \notin \{pc\} \cup \mathtt{Y}$, we have $s'.x = s.x$, and $s' \models r.(s'.pc)$ The sp operator for sets of states and ACFA operations is defined as: $sp.\pi.(q, \mathtt{Y}, q') = (\exists y \in \mathtt{Y}.\pi) \wedge A.r.q'$.

Thread-Context Programs. A thread-context program $\{C\} \cup A^\omega$ consists of a CFA C and a context A^ω of an arbitrary number of copies of an ACFA A. Formally, $\{C\} \cup A^\omega$ is the program $\{(\rightsquigarrow_C, At.C)\} \cup \{(\rightsquigarrow_1, At_1), (\rightsquigarrow_2, At_2), \ldots\}$

[3] We abuse notation to identify Q and $\mathbb{Z}$ and q_0 with 0.

where each pair $(\rightsquigarrow_i, At_i)$ is defined as follows. Let A_i be the ACFA A with each local variable $x \in X_L \cup \{pc\}$ renamed to x_i. Then $\rightsquigarrow_i = \rightsquigarrow_{A_i}$ and $At_i = At.A_i$.

3.4 Abstractions

As mentioned in Section 2, to make the analysis tractable, we make the state space small and finite by abstracting the system along several orthogonal dimensions.

1. Data Abstraction. First, we combat the infinite data space by abstracting the state space of each thread using predicates [15, 3, 20, 5]. For a set of predicates $P \subseteq \texttt{Pred}.X$ and a formula φ over X, let $\mathsf{Abs}.P.\varphi$ denote the smallest (in the inclusion order) set of data regions expressible as a boolean formula over atomic predicates from P.

A *thread abstraction* is a pair (C, P) where $C = \langle Q, q_0, X, \rightarrow, Q^* \rangle$ a CFA and $P \subseteq \texttt{Pred}.X$ is a set of predicates. An abstract thread state is a pair (q, φ) where $q \in Q$ and φ is a boolean formula over atomic predicates from P. The set of abstract thread states is $S.(C, P)$.

2. Control Abstraction. Second, we must track the local states of *each* thread separately. Just the control states of each thread suffice to overwhelm the analysis (since their size of the state space grows as the product of the number of control locations of each thread). Thus, we approximate the behavior of each thread using a *abstract thread*.

Our algorithm incrementally builds abstract threads until it has one that overapproximates the behavior of the thread in the context. To know when the above happens, we require a notion of when abstract thread overapproximates another. We formalize this notion as a *simulation* $\preceq$ relation [17, 7]. Given two ACFAs $A = \langle Q, q^0, X, \rightarrow, Q^*, r \rangle$ and $A_1 = \langle Q_1, q_1^0, X_1, \rightarrow, Q_1^*, r_1 \rangle$, $\preceq$ is the largest subset of $Q_1 \times Q$ such that: if $q_1 \preceq q$ then (1) $r_1.q_1 \Rightarrow r.q$, and, (2) For every $q_1 \xrightarrow{\mathtt{Y}} q_1'$ there exists a $q \xrightarrow{\mathtt{Y}} q'$ such that $\mathtt{Y} \subseteq \mathtt{Y}'$ and $q_1' \preceq q'$. We say $A_1 \preceq A$ if $q_1^0 \preceq q^0$.

3. Counter Abstraction. Third, for soundness, in many situations we must assume that there are arbitrarily many context threads running concurrently. Thus, we have to track the local state of an infinite number of context threads, in addition to the values of the global variables. We model each context thread with an ACFA whose whose local information is encoded in the ACFA location. In other words, the variables appearing in predicates labeling the ACFA locations and on the ACFA edge labels are all global variables. Now, instead of tracking the control location of each thread separately, we shall *count* the number of threads at each control location. This still leads to an infinite number of possibilities so we shall use a *counter abstraction*, where given a parameter k we shall abstract any number greater than k to be ω. Formally, for every $k \in \mathbb{N} \cup \{\omega\}$, we define the counter abstraction function $\alpha.k : \mathbb{N} \rightarrow \{0, \ldots, k, \omega\}$ as $\alpha.k.j = j$ if $j \leq k$ and ω otherwise. A *context abstraction* is a pair (A, k) where $A = \langle Q, q_0, X, \rightarrow, Q^*, r \rangle$ an ACFA and $k \in \mathbb{N}$ is a natural number. An abstract context state is a function $\Gamma : A.Q \rightarrow \{0 \ldots k, \omega\}$ mapping states to counter values in $\{0, \ldots, k, \omega\}$. The set of abstract context states is denoted $S.(A, k)$.

Abstract Multithreaded Programs. A thread abstraction and a context abstraction defines an *abstract (multithreaded) program* $\hat{\mathcal{P}} = ((C, P), (A, k))$ representing an abstraction of the thread-context program $\{C\} \cup A^\omega$. The variables of the ACFA A are the global variables of C.

An abstract program defines the following transition system. The set of abstract program states is $S.(C, P) \times S.(A, k)$. A particular abstract program state is $((q, \pi), \Gamma)$ where q is the control location of the thread, π is a boolean formula over P, and Γ is an abstract context state, *i.e.,* a map from $A.Q$ to $\{0 \ldots k, \omega\}$. An abstract program state represents a set of states of the multithreaded program $\{C\} \cup A^\omega$. The initial abstract state is $\hat{s}_0 = ((C.q_0, \mathtt{true}), \Gamma_0)$ where Γ_0 maps the initial state $A.q_0$ of the ACFA to ω and maps q for $q \neq A.q_0$ to 0.

The operations of a location $q \in C.Q$ ($q' \in A.Q$) are the operations labeling the out-edges of the location. For an abstract state $\hat{s} = ((q, \pi), \Gamma)$, the set of locations is $L.\hat{s} = \{q\} \cup \{q' \in A.Q \mid \Gamma.q' > 0\}$, *i.e.,* the set of (abstract) locations containing threads, and the the set of atomic locations is the set $AL.\hat{s} = \{q' \in A.Q^* \mid \Gamma.q' > 0\} \cup (\{q\} \cap C.Q^*)$, *i.e.,* it is the set of (abstract) atomic locations containing threads. The set of operations *enabled* in the abstract state $\hat{s}$ is defined as follows: (1) If $|AL.\hat{s}| = 0$, then the enabled operations are the operations of the locations in $L.\hat{s}$. (2) If $|AL.\hat{s}| = 1$, then the enabled operations are the operations of the unique location in $AL.\hat{s}$. (3) Otherwise, no operations are enabled in $\hat{s}$.[4]

For an abstract context state Γ, let $\Psi.\Gamma = \wedge\{A.r.n \mid \Gamma.n > 0\}$. The abstract transition relation is defined by the operator post that takes a abstract state and an operation $\mathtt{o}$, and produces the successor abstract state. For the operation $\mathtt{o} = (q, \mathtt{op}, q')$ of C at the state $((q, \pi), \Gamma)$, we compute the successor state $\mathsf{post}.((q, \pi), \Gamma).(q, \mathtt{op}, q')$ as $((q', \pi'), \Gamma')$, where $\Gamma' = \Gamma$ and $\pi' = \mathsf{Abs}.P.(sp.\pi.(q, \mathtt{op}, q')) \wedge \Psi.\Gamma'$. For the operation $\mathtt{o} = (q_1', \{y_1, \ldots, y_n\}, q_2')$ of A, the successor state $\mathsf{post}.((q, \pi), \Gamma).(q_1', \{y_1, \ldots, y_n\}, q_2') = ((q, \pi'), \Gamma')$ where $\Gamma'.q_1' = \alpha.k.(\Gamma.q_1' - 1)$, $\Gamma'.q_2' = \alpha.k.(\Gamma.q_2' + 1)$, and for all $q' \in A.Q \setminus \{q_1', q_2'\}$, $\Gamma'.q' = \Gamma.q'$. and $\pi' = \mathsf{Abs}.P.(sp.\pi.(q_1', \mathtt{Y}, q_2'')) \wedge \Psi.\Gamma'$.

We say $\hat{s} \hat{\rightsquigarrow} \hat{s}'$ if there exists an operation $\mathtt{o}$ which is enabled in $\hat{s}$ such that $\mathsf{post}.\hat{s}.\mathtt{o} = \hat{s}'$. If $\mathtt{o} = (q, \mathtt{op}, q')$ is an operation of C, then we say $\hat{s} \hat{\rightsquigarrow} \hat{s}'$ via a program edge. If $\mathtt{o} = (q', \mathtt{Y}, q'')$ is an operation of A, then we say $\hat{s} \hat{\rightsquigarrow} \hat{s}'$ via an environment edge. The reachable states $[\![\hat{\mathcal{P}}]\!] = \{\hat{s} \mid \hat{s}_0 \hat{\rightsquigarrow}^* \hat{s}\}$.

Given the operation post, we can construct a reachability tree for an abstract multithreaded program by starting from a root node labeled with $\hat{s}_0$ and constructing successors for each node by computing post for each enabled operator. Precisely, the reachability tree for an abstract multithreaded program is a labeled tree where each node is labeled with an abstract state, and each edge is labeled with an operation. The root of the tree is labeled with $\hat{s}_0$. For each node n marked with s, and each operation $\mathtt{op}$ enabled at s, there is a child n' of n marked with $\mathsf{post}.s.\mathtt{op}$.

Abstract Reachability Graph. An abstract reachability graph for an abstract multithreaded program $((C, P), (A, k))$ is an ACFA $G = \langle Q, q_0, X, \rightarrow, Q^*, r \rangle$ such that there is a function $f : S.(C, P) \times S.(A, k) \rightarrow Q$ such that (1) for all $\hat{s} = ((pc, \pi), \Gamma)$, we have π implies $G.r.(f.\hat{s})$, (2) $f.\hat{s}_0 = q_0$ for the initial state $\hat{s}_0$ of the abstract multithreaded program, (3) if $\hat{s} \hat{\rightsquigarrow} \hat{s}'$ via a program edge $(q, \mathtt{op}, q')$, then $f.\hat{s} \xrightarrow{\mathtt{Y}} f.\hat{s}'$ where $\mathtt{Y}$ contains all the variables written in $\mathtt{op}$, and (4) if $\hat{s} \hat{\rightsquigarrow} \hat{s}'$ via an environment edge, then $f.s = f.s'$. An ARG

[4] So long as the initial locations are not atomic, this will never arise

for $((C,P),(A,k))$ is an ACFA that is an abstraction (overapproximation) of the behavior of C in the TCP $\{C\} \cup A^\omega$. In other words, the ARG represents (an overapproximation of) the set of transitions of C reachable in $\{C\} \cup A^\omega$.

4. SAFETY VERIFICATION

4.1 The Race Detection Problem

Given a multithreaded program $\mathcal{P} = \{T_1, T_2, \ldots\}$ with the variables X, and a set of error states $\mathcal{E} \subseteq \mathcal{V}_X$, the multithreaded safety verification problem is to check if $[\![\mathcal{P}]\!] \cap \mathcal{E} = \emptyset$. A multithreaded program $\mathcal{P}$ is *safe* w.r.t. $\mathcal{E}$ if $[\![\mathcal{P}]\!] \cap \mathcal{E} = \emptyset$, and *unsafe* otherwise.

A specific instance of the above is the *race detection problem*. For each global variable $x \in \mathcal{P}.X_G$, let $Write.i.x \subseteq \mathcal{V}_X$ (resp. $Read.i.x \subseteq \mathcal{V}_X$) denote the set of states from which thread i has an enabled operation that writes (resp. reads) x. An operation writes x if either it is a CFA edge, and the operation is an assignment to x, or it is an ACFA edge, and the variable x is in the set of havocs of the edge. An operation reads x if it is a CFA edge, and the operation is an assignment $\mathtt{y} := e$ and x is a variable of e, or an assume $\mathtt{asm}\ [p]$ and x is a variable of p. Notice that $Read.i.x = \emptyset$ if T_i is an ACFA. The *race-states* $\mathcal{E}_x$ for a variable $x \in \mathcal{P}.X_G$ are X-states where two distinct threads have accesses to x enabled, and one of the accesses is a write, *i.e.,* $\mathcal{E}_x = \cup_{i \neq j}(Write.i.x \cup Read.i.x) \cap Write.j.x$. The *race-detection problem* for a program $\mathcal{P}$ and a global variable x is to check $[\![\mathcal{P}]\!] \cap \mathcal{E}_x = \emptyset$. We say a program $\mathcal{P}$ has no races on variable x iff the program $\mathcal{P}$ is safe w.r.t. $\mathcal{E}_x$.

4.2 Checking

Suppose that we are given the CFA C, and a global variable x of the CFA, we would like to check if there are races on x in C^ω. In addition, suppose we have a set of predicates P, and abstract thread A that purportedly describes succinctly the behavior of C, as well as a number k with which to abstract the context. We now describe how these various objects can be used to check that $[\![C^\omega]\!] \cap \mathcal{E} = \emptyset$ (Algorithm Check). There are two main steps:

1. **Assume** that A is a sound approximation of the behavior of C when C is composed with infinitely many copies of itself. Compute the set of abstract states reachable in the abstract multithreaded program $((C,P),(A,k))$ and check that this set does not contain any races. If an error is reached, return "possibly UNSAFE." This step is implemented by procedure ReachAndBuild, which does a reachability analysis and also builds an abstract reachability graph G describing the behavior of C when its context is an arbitrary number of abstract threads A running concurrently.

2. **Guarantee** that the abstract thread A is indeed a sound approximation of the behavior of C in this context, by checking that abstract reachability graph G computed in the previous step is overapproximated by A, or more precisely, that $G \preceq A$. If the check succeeds, return SAFE, else return "possibly UNSAFE." This is implemented by procedure CheckSim.

The soundness of the above follows via inductive "assume-guarantee" reasoning [23, 1]. We now describe ReachAndBuild and CheckSim in greater detail.

Algorithm 1 Algorithm ReachAndBuild

Require: A thread abstraction (C,P), error states $\mathcal{E}$
Require: A context abstraction (A,k)
1: **Output:** An ACFA A' or raises exception Exception(τ)
2: $L := \{((C.q_0, \mathtt{true}), \Gamma_0)\}$, $Seen := \emptyset$, $G := \emptyset$
3: **while** $L \neq \emptyset$ **do**
4: pick and remove state $((q,\pi),\Gamma)$ from L
5: **if** not $(((q,\pi),\Gamma) \in Seen)$ **then**
6: $Seen := Seen \cup \{((q,\pi),\Gamma)\}$
7: **if** $((q,\pi),\Gamma) \cap \mathcal{E} \neq \emptyset$ **then**
8: $\tau := \mathsf{FindPath}.Seen.((C.q_0, \mathtt{true}), \Gamma_0).((q,\pi),\Gamma)$
9: **raise** Exception(τ)
10: **else**
11: **for each** enabled operation $\mathtt{o}$ **do**
12: $((q',\pi'),\Gamma') := \mathsf{post}.((q,\pi),\Gamma).\mathtt{o}$
13: Connect.$G.((q,\pi),\mathtt{o},(q',\pi'))$
14: $L := L \cup \{((q',\pi'),\Gamma')\}$
15: **return** G

Algorithm 2 Algorithm Connect

Require: Augmented ACFA $G = (Q, \rightarrow, r, S)$, where $S : Q \rightarrow 2^{S.(C,P)}$
Require: (r, op, r') where $r, r' \in S.(C,P)$ and $\mathtt{o}$
1: $n := \mathsf{Find}.G.r$; $n' := \mathsf{Find}.G.r'$
2: **if** $\mathsf{op} \equiv \xrightarrow{\mathtt{x}:=e}$ **then**
3: **if** $n \xrightarrow{\mathtt{Y}} n'$ is in G for some $\mathtt{Y}$ **then**
4: Replace $(n \xrightarrow{\mathtt{Y}} n')$ with $(n \xrightarrow{\mathtt{Y} \cup \{\mathtt{x}\}} n')$ in G
5: **else**
6: Add $n \xrightarrow{\{\mathtt{x}\}} n'$ to G
7: **else if** $\mathsf{op} \equiv \xrightarrow{\mathtt{asm}\ [p]}$ **then**
8: **if** $n \xrightarrow{\mathtt{Y}} n'$ is not in G for some $\mathtt{Y}$ **then**
9: Add $n \xrightarrow{\emptyset} n'$ to G
10: **else**
11: Union.$G.(n,n')$

Procedure ReachAndBuild is shown in Algorithm 1. It is a standard worklist based reachability algorithm [7], but additionally builds an abstract reachability graph G summarizing the reachability information. The main loop of lines 3–14 runs the reachability construction, using the worklist L. At each step, a state is chosen from the worklist. If it has not been seen before (line 5), it is added to the set of explored states (line 6), and checked for possible errors. If an error state has been hit (line 7), the procedure finds an (abstract) interleaved error trace to the error state, and raises an exception containing the error trace. Otherwise, the current state is expanded. For this, we construct the successor of the current state for each operation enabled from it (line 12), and connect the current state and its successor as an edge in the abstract reachability graph G, using the procedure Connect described shortly. Finally, the successor states are added to the worklist. Note that the locations of the abstract reachability graph G correspond to abstract thread states, that is, we drop the context state information.

Procedure Connect adds edges between the abstract states computed by the reachability analysis (Algorithm 2). It takes as argument the augmented ACFA G that is being constructed, abstract thread states r and r' (the successor of r), and an operation $\mathtt{o}$. Each location of G corresponds to a set of abstract thread states. The ACFA G is augmented with a map S that maps a location n to the set $G.S.n$ of thread states mapped to n. Connect first finds locations n, n' corresponding to r and r' respectively by invoking the procedure Find. When Find is called with abstract thread

Algorithm 3 Algorithm Find

Require: Aug. ACFA $G = (Q, \rightarrow, R, S)$ and $r \in S.(C, P)$
1: **Output:** A location $q \in Q$
2: **if** $\exists q \in Q : r \in S.q$ **then**
3: **return** q
4: **else**
5: $q =$ fresh location
6: $G.Q := G.Q \cup \{q\}$
7: $G.R.q := r$
8: $G.S.q := \{r\}$
9: **return** q

Algorithm 4 Algorithm Union

Require: Aug. ACFA $G = (Q, \rightarrow, R, S)$ and $q, q' \in Q$
1: **if** $(q \neq q')$ **then**
2: $G.R.q = G.R.q \vee G.R.q'$
3: $G.S.q := G.S.q \cup G.S.q'$
4: **for each** $q'' \xrightarrow{*} q'$ **do**
5: Add $q'' \xrightarrow{*} q$ to G
6: **for each** $q' \xrightarrow{*} q''$ **do**
7: Add $q \xrightarrow{*} q''$ to G
8: **return** q

state r it checks if a there exists a location n with $r \in G.S.n$; If so, it returns that location and if not, it returns a new location n where $G.S.n = \{r\}$. The location is atomic, if $r = (q, \pi)$ where $q \in C.Q^*$. An invariant maintained is that $G.R.n = \cup G.S.n$. The edges of the graph G are added depending on the type of the operation o. There are two cases: o is either a thread operation, or a context operation. If the thread move is an assignment $x := e$, then we add the edge $n \xrightarrow{\{x\}} n'$ (the $\{x\}$ reflecting the fact that x is updated along the edge); however if an edge $n \xrightarrow{Y} n'$ is already present in G, we replace it with $n \xrightarrow{Y \cup \{x\}} n'$. If the thread move is an assume asm $[p]$ then we add the edge $n \xrightarrow{\emptyset} n'$, (in this case, no variable is added to the havoc edge), unless there is already an edge $n \xrightarrow{Y} n'$. Finally, if the operation is a context edge, then the two locations n and n' are unified by procedure Union which creates a single location n'' which is obtained by "merging" the two locations: $G.S.n'' = G.S.n \cup G.S.n'$, $G.R.n'' = G.R.n \cup G.R.n'$ and the edges of n'' are the union of the edges of n, n'.

Procedure CheckSim checks that the abstract reachability graph G returned by the procedure ReachAndBuild is simulated by the ACFA A (the guarantee part). The procedure CheckSim implements a variation of the standard simulation checking algorithm [17]. This check ensures soundness.

PROPOSITION 1. **[Soundness]** *If Algorithm* Check *with input thread abstraction* (C, P)*, context abstraction* (A, k)*, and error states* $\mathcal{E}$ *terminates and returns* SAFE *then* $[\![C^\omega]\!] \cap \mathcal{E} = \emptyset$.

4.3 Inference

In general, the abstract thread A that succinctly summarizes the behavior of a thread, and is simultaneously precise enough to show the absence of concurrency errors, is not available. Therefore, we must construct this abstraction automatically via an inference algorithm. Algorithm 4.3 shows our inference algorithm CIRC. The initial abstraction has no predicates ($P = \emptyset$), and the counter abstraction uses $k = 0$.

Algorithm 5 The inference algorithm CIRC

Require: CFA C, error states $\mathcal{E}$
1: $P := \emptyset, k := 0$
2: **while** true **do**
3: **try**
4: $G = \emptyset$
5: **repeat**
6: $\langle A, \mu \rangle :=$ Collapse.G
7: $G :=$ ReachAndBuild.$(C, P).(A, k).\mathcal{E}$
8: **until** $(G \preceq A)$
9: **return** SAFE
10: **with** (Exception(τ)) $\rightarrow$
11: **if** Refine.$C.A.G.\mu.\tau =$ REAL$(\bar{s})$ **then**
12: **return** UNSAFE$(\bar{s})$
13: **else**
14: $(P', k') :=$ Refine.$C.A.G.\mu.\tau$
15: $P := P \cup P'$
16: $k := k'$
17: **done**

We start off by assuming the that each thread in the context does nothing, *i.e.,* G is set to the empty abstract thread (line 4). We then minimize the abstract reachability graph G with the procedure Collapse (line 6). Collapse takes abstract reachability graph G and returns its weak bisimulation quotient ACFA A [7], together with a map μ that maps each state of G to its equivalence class (state) in A. In the first round, this is still empty. At each round, A will be the "current" approximation of the context threads, and will be used to make the context of C. We then call ReachAndBuild to see how C behaves in this context, the result being the new abstract reachability graph G (line 7). If the present approximation A simulates the new abstract reachability graph G then it means that A was a sound approximation (*i.e.,* meets the "guarantee"), and we break out of the loop and return SAFE (lines 8, 9). This check is performed using the procedure CheckSim. If on the other hand, we find that A was not a good approximation (fails to meet the guarantee) then we repeat the loop with the new G, which now gives us a better approximation of each context thread (the **repeat**...**until** loop of lines 5–8).

At any point, the procedure ReachAndBuild may raise an exception claiming it has an abstract error trace to a race state. We trap this exception and analyze the counterexample to see if it is genuine, and if not, obtain a more precise set of predicates or increment the counter parameter (lines 10–16). The exception Exception(τ) is caught in line 10, and checked in procedure Refine. Procedure Refine takes as input a CFA C, an ACFA A, the abstract reachability graph G such that A is the bisimilarity quotient of G, the map μ mapping states of G to those of A, and an abstract error trace τ of $\{C\} \cup A^\omega$. If τ can be realized in the concrete program, Refine returns a real error REAL together with a concrete interleaved trace $\bar{s}$ in C^ω that reaches the error state. In this case, the algorithm returns UNSAFE together with the real error $\bar{s}$. On the other hand, if the current error path τ does not have a concrete realization in C^ω, Refine returns a refinement of the abstraction by returning new predicates P' and a new value for the counter k'. The algorithm updates the thread and context abstractions by adding these new predicates to P, and by updating the maximum value k of a counter in the context abstraction to k' respectively (lines 14–16). If the abstraction is updated, we reset the current approximation of the context G back to the empty

context, and repeat starting with the new abstraction parameters (P, k). In the next section, we describe in detail the remaining subroutines of the algorithm, Collapse, and Refine, and mention some optimizations.

THEOREM 1. *If Algorithm* CIRC *on input CFA C and error states $\mathcal{E}$ terminates and returns* SAFE *then C^ω is safe w.r.t. the error states $\mathcal{E}$; if it returns* UNSAFE(τ) *then C^ω is unsafe w.r.t. the error states $\mathcal{E}$.*

The algorithm is not complete in general, since reachability is undecidable already for single-threaded programs. For finite-state systems and just predicate and control state abstractions, completeness follows from finiteness of the state space. It is not obvious that even for finite-state threads, Algorithm CIRC is complete, since we consider unboundedly many threads. We show in Appendix A that our method using counters is complete if the threads are finite-state. This implies that the only way the procedure can loop forever is if either the threads are recursive, or we keep discovering new predicates.

5. DETAILS

Procedure Collapse**.** The procedure Collapse takes an abstract reachability graph G and constructs a weak bisimulation quotient A of G with respect to the global predicates. It returns the weak bisimulation quotient A, along with a mapping μ from $G.Q$ to $A.Q$ mapping each state of G to its equivalence class in A. We do this in two steps. First, for each location $q \in G.Q$, we replace the state $G.r.q$ with the state obtained by quantifying out all local variables as follows: in the formula $G.r.q$, which is a boolean combination of predicates in P, we replace with "unknown" each atomic predicate containing a local variable. We remove all local variables from the havoc sets labeling the edges. We then run a standard weak bisimilarity algorithm [7] with the resulting predicates (now over global variables) labeling states of G as observables. The bisimilarity procedure also constructs the required mapping. Whenever in G we have $n \xrightarrow{\mathtt{Y}} n'$, and the bisimilarity collapses n, n' to the same location n'' in A, we ensure that A has a self loop edge $n'' \xrightarrow{\mathtt{Y}'} n''$ with $\mathtt{Y} \subseteq \mathtt{Y}'$. The result is an ACFA with only global variables in its location predicates and on the edges. This is important as we want that any local variable appearing in the analysis refers to the local of the main thread.

Procedure Refine**.** The procedure Refine analyzes abstract counterexamples, to either extract genuine error traces or to refine the abstraction to eliminate the false positive. An abstract trace is a sequence of operations of the main thread and the context ACFA. The input to Refine is a CFA C, an ACFA A, a reachability graph G such that A is the weak bisimilarity quotient of G, a map μ from states of G to states of A, and an abstract trace τ. It returns either a real error trace $\bar{s}$ or, if τ is infeasible, a refinement of the abstraction (P', k') where P' is a set of predicates, and k' is a new counter value. Procedure Refine works in two steps:

1. Computing an Interleaving. A scan over the entire trace suffices to check if the parameter k is large enough. If not, the only refinement is that k is incremented. If k is large enough, we compute the *number* of context threads that participate in the counterexample. Each operation in the abstract trace is a either a main thread operation, or an abstract operation by a specific abstract context thread. To generate the concrete interleaving, we get a concrete sequence of thread operations from the abstract context operation, by using the underlying reachability tree of the ACFA.

T1: `I` $\rightarrow$ `II`	`skip`	`true`
T1: `II` $\rightarrow$ `III`	`old := state`	$old_1 = state_1$
	`asm [state = 0]`	$state_1 = 0$
	`state := 1`	$state_2 = 1$
	`asm [old = 0]`	$old_1 = 0$
T0: `skip`	`skip`	`true`
T0: `old := state`	`old := state`	$old_2 = state_2$
T0: `asm [state = 0]`	`asm [state = 0]`	$state_2 = 0$
T0: `state := 1`	`state := 1`	$state_3 = 1$
T0: `asm [old = 0]`	`asm [old = 0]`	$old_2 = 0$

Figure 5: Abstract trace, concrete interleaving, TF

2. Analyzing an Interleaving. Given an interleaved trace, we must check if it is feasible. We first compute a *trace formula* (TF) which is a version of the strongest postcondition of the trace. Each operation of the trace yields a clause and the TF is the conjunction of all the clauses. The trace is feasible, and hence, the counterexample genuine, iff the TF is satisfiable, which can be checked by querying a decision procedure. If it is not satisfiable, the *proof of unsatisfiability* of the TF can be mined for predicates using an extension of the technique described in [18].

EXAMPLE 3: **[Refinement]** In Figure 5 the left, middle and right columns show respectively, the abstract trace, concrete trace, and the unsatisfiable TF for the error trace from iteration 4 of the example from Section 2. The proof of unsatisfiability yields the predicates $state = 0$ and $state = 1$. □

ω-check. The procedure ReachAndBuild.$(C, P).(A, k)$ is expensive, so we implement the following optimization of the algorithm CIRC. Let ReachAndBuild^k be an instance of the reachability algorithm that uses the initial state $((C.q_0, \mathtt{true}), \Gamma_0^k)$ where $\Gamma_0^k.q = k$ if $q = q_0$ and 0 if $q \neq q_0$. In our modified algorithm, we run ReachAndBuild^k with the current value of variable k in line 7. This has the effect of running a multithreaded program where there are exactly k threads in the context. If the loop terminates with $k = k_0$, we have an ACFA A that succinctly represents a context with k_0 threads. At this point, we check if the CFA A is also a succinct description for a context with arbitrarily many threads. One way is to check if ReachAndBuild$^\omega.(C, P).(A, k_0) \preceq A$. While sound, this reachability is also very expensive. Instead we perform the following check.

Suppose at termination, the values of A, G, μ and k in the algorithm are $\hat{A}$, $\hat{G}$, $\hat{\mu}$, and k_0 respectively, so that $\langle \hat{A}, \hat{\mu} \rangle =$ Collapse.$\hat{G}$. We first compute the set of reachable states $R \subseteq S.(\hat{A}, k_0)$ by running the reachability algorithm on $\hat{A}^\omega$. This gives us all possible configurations that the environment can reach. From this, we compute the set of possible environment actions as follows. For each state $q \in \hat{A}.Q$, we construct the set of enabled transitions at q given that the main thread is at location q. A transition $q' \dashrightarrow q''$ is enabled at q if there is a $\Gamma \in R$ with $\Gamma.q > 0$, and either $\Gamma.q' > 0$ and $q \neq q'$ or $\Gamma.q' > 1$ and $q = q'$. Intuitively, when the main thread is in location q, if the tran-

sition $q' \dot{\rightarrow} q''$ is enabled, then it is possible for some thread in the environment to execute this transition. A location $n \in \hat{G}.Q$ is *good* for a transition $e = q' \xrightarrow{\{\mathtt{x}_1,\ldots,\mathtt{x}_n\}} q''$ of $\hat{A}$ if (1) e is enabled at $\hat{\mu}.n$, and (2) the result of executing the context action e from $\hat{G}.r.n$ is contained in $\hat{G}.r.n$, that is, $(\exists \mathtt{x}_1 \ldots \mathtt{x}_n.(\hat{G}.r.n)) \wedge (\hat{A}.r.q'') \Rightarrow (\hat{G}.r.n)$. We check that all locations $n \in \hat{G}.Q$ are good for all context transitions in $\hat{A}$ that are enabled in $\hat{\mu}.n$. This check is sound: if the check succeeds then $\hat{A}$ simulates a context with arbitrarily many threads. If the check fails, we increment k_0 and rerun the main loop. We call this algorithm ω-CIRC. We have found that in practice, ω-CIRC is considerably faster than CIRC.

THEOREM 2. *If Algorithm ω-CIRC on input CFA C and error states $\mathcal{E}$ terminates and returns* SAFE *then C^ω is safe w.r.t. the error states $\mathcal{E}$; if it returns* UNSAFE*(τ) then C^ω is unsafe w.r.t. the error states $\mathcal{E}$.*

Memory Model. So far we have described our algorithms assuming all variables are of type integer. In our implementation, we extend the basic algorithm to deal with pointer variables and aliasing. The problem is that we cannot infer the global memory address being accessed syntactically by looking at the name of the lvalue. Thus, for the error check, we ask for every pair of lvalues l_1, l_2 at a state, if the addresses of l_1 and l_2 can be the same, and in addition if there is a race between l_1 and l_2. As an optimization, we use a flow insensitive alias and escape analysis to curtail the possible aliasing relationships to be explored. We omit the details for lack of space.

6. EXPERIENCES

NESC [13] is a programming language for networked embedded systems. It is used to implement event driven applications in the TinyOS operating system [21]. TinyOS has two sources of concurrency: *tasks* and *events*. When an interrupt occurs an event is fired, which may in turn fire other events. As other interrupts can occur while this is happening, events can preempt each other. Events may also post tasks, which are run when nothing else is happening. A task may be preempted by events, but is never preempted by another task. The presence of concurrent execution leads to potential data races on the shared state. Since tasks are nonpreemptible, there is no data race on variables accessed only in tasks, but there may be be races between events and tasks, or between two events.

As it is essential to avoid data races, NESC provides *atomic sections* in the language with an `atomic` keyword. Code in an atomic section is executed atomically. The NESC compiler implements a flow based static analysis to catch race conditions on shared data variables. It runs an alias analysis to detect which global variables are accessed (transitively) by interrupt handlers, and then checks that each such access occurs within an `atomic` section. However, this analysis precludes the use of some common programming idioms (*e.g.,* the example from Section 2) which cause the analysis to return false positives. For this, NESC provides a `norace` annotation that the programmer must provide if she believes that there is no race condition on a data variable. In practice, almost all shared accesses are put in atomic sections to prevent compiler warnings, even though there may be no actual race condition. Since atomic sections are implemented by interrupt disabling, this may make the system less responsive. Thus, NESC programs gave us a suitable application for a precise race checker like CIRC: first, they critically require the absence of data races, and second, they use several non-trivial synchronization idioms.

Running CIRC on NESC Programs. We focused on the variables that had raised false alarms with the flow-based analysis, and which subsequently were flagged with the `norace` qualifier. NESC programs are compiled into C and event fires translate to function calls. We modeled the NESC applications as an arbitrary number of threads each executing a big while-loop that triggered hardware interrupts non-deterministically (as long as interrupts were enabled, modeled by adding a special global) or called tasks non-deterministically (as long as nothing else was running).

Our results on some of the largest NESC applications are summarized in (Table 1). The examples requiring no predicates are ones that were trivially safe as they were accessed in atomic sections or only by tasks and our tool finds this quickly. "Line" is the number of lines in the compiled C source code. "Preds" is the number of predicates discovered to prove safety, "ACFA" is the number of states in the final ACFA. The counter parameter was always 1.

State Variable based Synchronization. Many of the variables `gTxByteCnt, gTxRunningCRC` were protected by a state variable much like the example in Section 2, and CIRC is able to show there are no races, by finding the appropriate abstraction. `gTxState` is protected in a similar manner but is accessed in a more complicated pattern: CIRC first reported a violation on it in secureTosBase. On inspection we found that the variable was accessed at several places inside a function, in most places *before* a call that changed the state variable, but at the point of conflict, it was accessed *after* changing the state variable. On moving the access to before the call, CIRC reported the system was safe. There was another "unprotected" access, that occurred when a certain function call returned the value 0, but CIRC verified that in that context, the function never returned 0. `gRxHeadIndex` uses a complicated synchronization on multiple values of a state variable along with "conditional" accesses.

Split-phase based Synchronization. The variable `rec_ptr` in surge was accessed by an interrupt handler (event) (I) and by a task (T) in the following manner: the handler fired only when I was enabled. It then disabled the interrupt I, posted the task T and then wrote to `rec_ptr`. The task, when it got to run, wrote to the variable, and then re-enabled the interrupt. This is an instance of a *split-phase* operation, used to break up long tasks. When we modeled this interrupt precisely by tracking its status in a global flag, CIRC was able to report the absence of races after inferring the appropriate ACFA. (Since the C code does not match up interrupt bits with handlers, we had to refer to the underlying hardware model.) A more complicated form of synchronization was in sense where the variable `tosPort` was protected by a combination of this and a state variable. We discovered this as CIRC found a race where an interrupt fired which reset the state after one thread had already set it and was about to write to `tosPort` thus letting another thread come in and access `tosPort`. The programmer pointed out that the malicious middle interrupt was only enabled after

Name	Variable	♯ Preds	ACFA size	Time
secureTosBase (9539 lines)	gTxState	11	23	7m38s
	gTxByteCnt	4	13	1m41
	gTxRunningCRC	4	13	1m50s
	gTxProto	0	9	12s
	gRxHeadIndex	8	64	20m50s
	gRxTailIndex	0	5	2s
surge (9697 lines)	rec_ptr	4	23	1m18s
	gTxByteCnt	4	15	1m34
	gTxRunningCRC	4	15	1m45s
	gTxState	11	35	9m54s
sense (3019 lines)	tosPort	6	26	16m25s

Table 1: Experimental results with CIRC on a 2GHz IBM T30 with 512M RAM

the first thread had finished writing to tosPort. On modeling this interrupt, the tool was able to prove safety.

Acknowledgments. We are indebted to David Gay for patiently educating us about nesC.

7. REFERENCES

[1] R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[2] T. Ball, S. Chaki, and S. K. Rajamani. Parameterized verification of multithreaded software libraries. In *TACAS 01: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 2031, pp. 158–173. Springer, 2001.

[3] T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pp. 1–3. ACM, 2002.

[4] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA 02: Object-Oriented Programming, Systems, Languages and Applications*, pp. 211–230. ACM, 2002.

[5] S. Chaki, J. Ouaknine, K. Yorav, and E.M. Clarke. Automated compositional abstraction refinement for concurrent C programs: A two-level approach. In *SoftMC 03: Software Model Checking*, 2003.

[6] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI 2002: Programming Languages Design and Implementation*, pp. 258–269. ACM, 2002.

[7] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[8] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In *ICSE 00: Software Engineering*, pp. 439–448. IEEE, 2000.

[9] G. Delzanno, J.-F. Raskin, and L. Van Begin. Towards the automated verification of multithreaded Java programs. In *TACAS 02: Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2280, pp. 173–187. Springer, 2002.

[10] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *LICS 99: Logic in Computer Science*, pp. 352–359. IEEE, 1999.

[11] C. Flanagan and S.N. Freund. Detecting race conditions in large programs. In *PASTE 01: Program Analysis for Software Tools and Engineering*, pp. 90–96. ACM, 2001.

[12] C. Flanagan, S. Qadeer, and S.A. Seshia. A modular checker for multithreaded programs. In *CAV 02: Computer-Aided Verification*, LNCS 2404, pp. 180–194. Springer, 2002.

[13] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI 2003: Programming Languages Design and Implementation*, pp. 1–11. ACM, 2003.

[14] P. Godefroid. Model checking for programming languages using Verisoft. In *POPL 97: Principles of Programming Languages*, pp. 174–186. ACM, 1997.

[15] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV 97: Computer-aided Verification*, LNCS 1254, pp. 72–83. Springer, 1997.

[16] K. Havelund and T. Pressburger. Model checking Java programs using Java Pathfinder. *Software Tools for Technology Transfer (STTT)*, 2(4):72–84, 2000.

[17] M.R. Henzinger, T.A. Henzinger, and P.W. Kopke. Computing simulations on finite and infinite graphs. In *FOCS 95: Foundations of Computer Science*, pp. 453–462. IEEE, 1995.

[18] T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *POPL 04: Principles of Programming Languages*, pp. 232–244. ACM, 2004.

[19] T.A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *CAV 03: Computer-Aided Verification*, LNCS 2725, pp. 262–274. Springer, 2003.

[20] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL 02: Principles of Programming Languages*, pp. 58–70. ACM, 2002.

[21] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *ASPLOS 00: Architectural Support for Programming Languages and Operating Systems*, pp. 93–104. ACM, 2000.

[22] G.J. Holzmann. Logic verification of ANSI-C code with SPIN. In *SPIN 00: Spin Model Checking and Software Verification*, LNCS 1885, pp. 131–147. Springer, 2000.

[23] C.B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[24] B.D. Lubachevsky. An approach to automating the verification of compact parallel coordination programs I. *Acta Informatica*, 21:125–169, 1984.

[25] S. Savage, M. Burrows, C.G. Nelson, P. Sobalvarro, and T.A. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[26] C. von Praun and T. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *PLDI 03: Programming Languages Design and Implementation*, pp. 115–128. ACM, 2003.

[27] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *POPL 01: Principles of Programming Languages*, pp. 27–40. ACM, 2001.

APPENDIX

A. COMPLETENESS OF COUNTER ABSTRACTIONS

For finite-state threads, counterexample-guided refinement using counter abstractions terminates.

Let X be a set of global variables, and let pc be a variable not in X. Let $T = (\rightsquigarrow, At)$, where $\rightsquigarrow$ is an $X \cup \{pc\}$-transition relation, be a thread. As before, let T^ω be the multithreaded program $\{(\rightsquigarrow_i, At_i) \mid i \in \mathbb{N}\}$ running an unbounded number of copies of the thread T, where $(\rightsquigarrow_i, At_i)$ is obtained from $(\rightsquigarrow, At)$ by renaming pc to pc_i. Notice that we assume pc_i is the only local variable of thread i. A thread T is *finite-state* if each variable in $X \cup \{pc\}$ takes values over a finite domain. The parameterized multithreaded program T^ω is finite-state if T is finite-state.

A state (s, Γ) of the multithreaded program T^ω can be described by an X-state s, together with a map $\Gamma : Q \to \mathbb{N} \cup \{\omega\}$, where Q is the finite range of the variable pc. Intuitively, the X-state s gives the valuations to the global variables, and $\Gamma.q$ for $q \in Q$ counts the number (possibly infinite) of threads T_i with $pc_i = q$.

We now define counter abstractions (T, k) of the multithreaded program T^ω. Let γ be a variable not in X, such that γ ranges over functions in $Q \to \mathbb{N} \cup \{\omega\}$. Given $T = (\rightsquigarrow, At)$ over the variables $X \cup \{pc\}$, define (T, k) to be the program $\{(\rightsquigarrow^k, At^k)\}$, over the variables $X \cup \{\gamma\}$ as follows.

1. The initial state is s_0, where $s_0.\gamma.q = \omega$ if $q = 0$ and 0 otherwise, and $\forall x \in X$, we have $s_0.x = 0$.
2. The set At^k is the set of $X \cup \{\gamma\}$-states s s.t. there exist $q \in Q$ and $t \in At$ with (a) $s.\gamma.q > 0$, (b) $t.pc = q$, and (c) for all $x \in X$, $s.x = t.x$.
3. $\rightsquigarrow^k$ is an $X \cup \{\gamma\}$-transition relation defined as follows: $s \rightsquigarrow^k s'$ iff $\exists q, q' \in Q$ and $t \rightsquigarrow t'$ s.t. (a) for all $x \in X$ we have $s.x = t.x$ and $s'.x = t'.x$, (b) $t.pc = q$ and $t'.pc = q'$, (c) $s.\gamma.q > 0$, (d) $s'.\gamma.q' = \alpha.k.(s.\gamma.q' + 1)$, $s'.\gamma.q = s.\gamma.q - 1$, and, for all $q'' \notin \{q, q'\}$, $s'.\gamma.q'' = s.\gamma.q''$, and, (e) $s \in At^k$ implies $t \in At$.

For sets X, Y of variables, and an X-state s and a Y-state t, we say $s \approx t$ iff $\forall x \in X \cap Y$ we have $s.x = t.x$. For S a set of X-states and T a set of Y-states, we say $S \lesssim T$ iff for each $s \in S$, there exists $t \in T$ such that $s \approx t$. We say $S \approx T$ if $S \lesssim T$ and $T \lesssim S$. Notice that $\approx$ is an equivalence relation and $\lesssim$ is reflexive and transitive.

LEMMA 1. **[Soundness]** *For every $T = (\rightsquigarrow, At)$ and for every $k \in \mathbb{N}$ we have $[\![T^\omega]\!] \approx [\![(T, \omega)]\!] \lesssim [\![(T, k+1)]\!] \lesssim [\![(T, k)]\!]$.*

Let T^ω be a finite-state multithreaded program over $X \cup \{pc\}$. Let $\mathcal{E}$ be a set of X-states, specifying possible error states. The *parameterized* safety verification problem is to check if $[\![T^\omega]\!] \lesssim \mathcal{V}_X \setminus \mathcal{E}$. If so, we say that T^ω is *safe w.r.t. $\mathcal{E}$*, otherwise, T^ω is unsafe. A sequence of states $\tau = t_0 \dots t_m$ is a *$\mathcal{P}$-trace* for a program $\mathcal{P}$ over the globals X if there exists $\sigma = s_0 \dots s_m$ such that: (1) s_0 is the initial state of $\mathcal{P}$, (2) for all $0 \le j < m$, $s_j \rightsquigarrow_{\mathcal{P}} s_{j+1}$, and (3) for all $0 \le k \le m$, $s_j \approx t_j$. Additionally, τ is called a $\mathcal{E}$-counterexample if $\{t_m\} \lesssim \mathcal{E}$. Notice that if $\tau = t_0 \dots t_m$ is a (T, k)-trace and $m \le k$, then τ is also a T^ω trace.

Algorithm 6 Counter-Guided Parameterized Verification

Require: Finite-state thread $T = (\rightsquigarrow, At)$ over variables $X \cup \{pc\}$, set of X-states $\mathcal{E}$.

```
1: k := 0
2: while true do
3:   try
4:     ModelCheck.(T, k).E
5:     return SAFE
6:   with (Exception(t_0 ... t_m)) →
7:     if m ≤ k then
8:       return UNSAFE(t_0 ... t_m)
9:     else
10:      k := k + 1
11: done
```

LEMMA 2. **[Completeness]** *Let $T = (\rightsquigarrow, At)$, where $\rightsquigarrow$ is an $(X \cup \{pc\})$-transition relation, be a finite-state thread. Let $\mathcal{E}$ be a set of X-states.*

1. *If T^ω is safe w.r.t. $\mathcal{E}$, then there exists a k s.t. (T, k) is safe w.r.t. $\mathcal{E}$.*
2. *If T^ω is unsafe w.r.t. $\mathcal{E}$, then there exists a k and a sequence τ of $(X \cup \{\gamma\})$-states s.t. τ is a (T, k)-trace, a T^ω trace, and an $\mathcal{E}$-counterexample.*

For the first case, we use a result from [10], that says the set of states that can reach the error states ($pre^*(\mathcal{E})$) is computable by backward iteration. We then show that there is a large enough k such that if s is a state of (T, k) and no "concretization" of s is in $pre^*(\mathcal{E})$, then inductively, every successor of s also has the same property, and thus (T, k) is safe. For the second case, let σ be the shortest sequence of X-states that is both a T^ω-trace and a $\mathcal{E}$-counterexample. Let k be the length of the sequence σ. Notice that for each $s_j \in \sigma$, there are never more than k threads at location q for any q other than 0. Hence, this sequence has a corresponding sequence of $X.pc$-states which is a (T, k)-trace that is also an $\mathcal{E}$-counterexample.

We now give a counterexample-guided algorithm for the parameterized safety verification problem for finite-state multithreaded programs (Algorithm 6). The algorithm proceeds by iteratively refining the counter abstraction of the parameterized multithreaded program, until either the program is proved safe, or a genuine counterexample is found. The procedure ModelCheck takes two arguments, a finite-state program $\mathcal{P}$ and a set of states $\mathcal{E}$ and checks whether $[\![\mathcal{P}]\!]$ is safe w.r.t. $\mathcal{E}$. If so, it simply returns, and if not, it raises an exception which is the *shortest* $\mathcal{P}$-trace that is an $\mathcal{E}$-counterexample. This procedure can be implemented using standard finite-state model checking techniques [7]. The procedure to check if a counterexample is genuine simply compares the length of the trace with the counter parameter, more sophisticated refinement schemes can also be used. The termination and correctness of the Algorithm 6 follow from Lemmas 1 and 2.

THEOREM 3. **[Termination and Correctness]** *When Algorithm 6 is given a finite-state thread T and a set of error states $\mathcal{E}$, it terminates. If it returns* SAFE *then T^ω is safe w.r.t. $\mathcal{E}$. If it returns* UNSAFE*(τ) then T^ω is not safe w.r.t. $\mathcal{E}$ and τ is a T^ω-trace that is an $\mathcal{E}$-counterexample.*