Race Checking by Context Inference

Thomas A. Henzinger Ranjit Jhala Rupak Majumdar

EECS Department, University of California at Berkeley, U.S.A.

{tah,jhala,rupak}@eecs.berkeley.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, minimization, circular assume-guarantee reasoning, and parametric reasoning. 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.

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 [24, 5] and static, type-based tools [3, 10]. 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 [12] 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. In real 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 [22] 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 [13, 7, 21, 15, 11] and [2, 19, 4] which additionally automatically perform the remaining data abstraction using counterexamples. 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.

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 [18], 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 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, with statefulness comes 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 [18] 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 bisimilarity quotients. 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, 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*. 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 [19]. 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 [12] 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 [10, 3] provide strong type systems that guarantee the absence of races, but require code annotated with locking information. [25] additionally use control flow information for a more precise analysis. Dynamic race detectors [24, 5] use a *lockset* algorithm which is effective in finding bugs but cannot guarantee their absence. None of the above can prove absence of races in programs with complex state-based synchronization idioms.

Software model checkers like SLAM [2] and Blast [19] check sequential programs. Verisoft [13], Bandera [7], Feaver [21], and Java Pathfinder [15] 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. Calvin [11] requires that a suitable abstract context is provided. Magic [4] checks a finite number of concurrent threads communicating by message-passing. Since communication is explicit in the model, abstraction and bisimulation minimization are done independently of the other threads, *i.e.,* reachability information is not required. Parametric verification [23, 8, 1, 9] consider arbitrarily many threads using counters, but assume a finite state abstraction for each thread is given.

```
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++;
7:     state = 0;
     }
   }
}
```

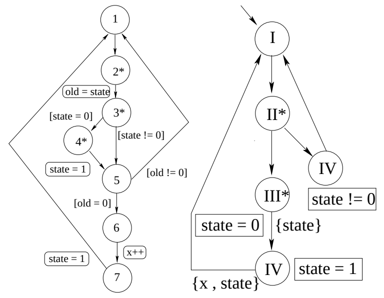

Figure 1: (a) Thread (b) CFA (c) ACFA

2 An Example

We begin with describing how our algorithm works on an example. Consider the fragment of code shown in Figure 1, taken from a NESC program [12]. This fragment describes the behaviour 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 like `Thread` running concurrently and 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 `x`, and one of the accesses is a write.

2.1 Threads and Strands

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

Strands. An abstract thread or *strand* 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 global variables that are *havoced i.e.,* written to with arbitrary values when the automaton moves from one location to the next, but the successor state is constrained to satisfy the predicate labelling the successor location.

EXAMPLE 2: [Strand] Figure 1(c) shows an ACFA for the thread of the example. Nodes labeled "*" are atomic, and if there is an (abstract) thread at an atomic location, then only that (abstract) thread is scheduled. Each node is also labeled by a predicate inside a box, nodes not labeled implicitly 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 not 0 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 multithreaded program is a set of threads where each thread is represented by a CFA. 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 its next state. 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. We say that a write or read on `x` is *enabled* at a state if in that state some thread is at a location one of whose outedges assigns to or reads the value of `x` respectively and either that location is atomic or no thread is at an atomic location. 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.

When the programs are given as CFAs the above criterion reduces to checking that the program never enters a state where (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 Verification by Abstraction

We analyze the program one thread at a time, as one *main thread* executing together with a *context* made up of all the other threads. A path-sensitive 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 program location tuples. Accordingly, we present three orthogonal abstractions.

Data Abstraction. First, instead of tracking variables exactly, we use predicates [14], and track relationships between program variables captured by boolean formulae over the predicates. Any local variable in a predicate refers to the main thread's copy of the local.

Control Abstraction. The number of configurations of other "context" threads is exponential in the number of locations of each thread, so we represent each context thread as an abstract thread which is a state machine much smaller than the thread represented, but which overapproximates the behaviour of the represented thread. The predicates labelling the ACFA vertices are all over the globals; information pertaining to the local state of context threads is encoded in the abstract location.

Counters. To make our analysis sound in the presence of arbitrarily many other threads, we must model the location of an arbitrary number of strands. We track the *number* of strands that are at each (of finitely many) strand control location [23]. Since this representation is not finite, 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 Reachability. Given an abstraction which is a set of predicates P, an ACFA A, and a k, an abstract state is a triple (pc, φ, Γ), 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 A node n 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.

Given 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 [14, 19], and $\Gamma' = \Gamma$. If it is a context ACFA moving across an abstract edge $n \rightarrow n'$, then $pc' = pc$, φ' is the predicate abstraction (w.r.t. P) of $(\exists y_1 \cdots y_k . \varphi) \land r.n'$ where $y_1 \cdots y_n$ are havoced on edge $n \rightarrow n'$ 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 $\Gamma.n''$.

Initially, the main thread is in the initial location, Γ is ω for the inital abstract location, and 0 elsewhere, and φ is the abstraction of the state where all variables are 0. On iterating post until a fixpoint, we can build the set of reachable states and check if there are races by checking if any reachable state contains a race. If so, the reachability procedure returns an abstract error trace.

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

Minimization. The abstract reachability procedure constructs a rooted tree whose nodes are labeled with abstract states, and edges labeled with enabled operations. We construct an ACFA from this tree in two steps. First, we construct a directed graph by dropping the counter information Γ from a region, and unifying nodes with the same region. Second, we construct a bisimilarity quotient [6] of the directed graph w.r.t. predicates on the global variables (by quantifying out local variables from the regions first).

2.3 The Algorithm CIRC

The input is (a) a CFA C, the multithreaded program is arbitrarily many copies of C running concurrently, (b) a global variable `x` which we check for data races, (c) a (possibly empty) set of predicates P, and, (d) 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 a *reachability graph* (RG) which is an ACFA G overapproximating the behaviour of C in the current context (Algorithm ReachAndBuild). This is done by connecting the various reachable states by appropriate edges when there are transitions between them.

Step 2 ("Counterexample analysis") Check if the reachable states computed above contain states with races on `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 RG of which the ACFA is the minimized version. This way for every ACFA behaviour we have (a possibly infeasible) trace through the underlying CFA. I (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 [17] 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 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. Before starting, we note that there is no race on `x` as 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 access `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 RG 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 there is no context.

Step 3_1 The result of minimizing G_1 is the 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`. Since A_0 was empty, Algorithm CheckSim fails, and we rerun the loop.

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 twice to the abstract location `3'` (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: $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6$, *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 RG G_3, shown in Figure 3. 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_1, 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 $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6$ 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 $3 \rightarrow 4$. The analysis reveals the predicates $state == 0, state == 1$ rule out this behaviour 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 RG 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

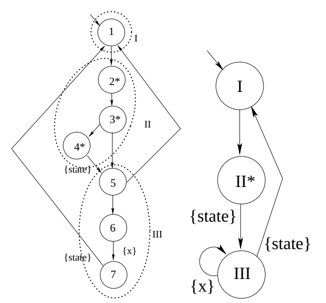

Figure 2: (a) RG (b) Min. RG

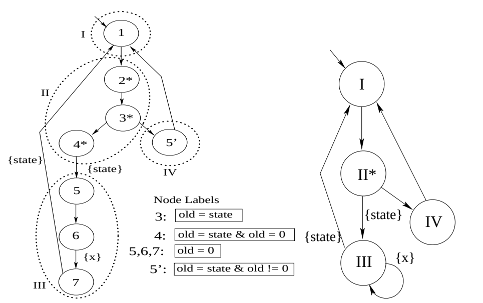

Figure 3: (a) RG (b) Min. RG

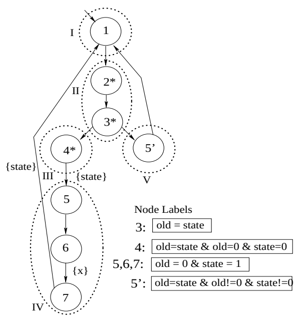
I
1
II
2*
3*
4*
5'
{state}
III
{state}
V
5
6
{x}
IV
7
Node Labels
3: old = state
4: old=state & old=0 & state=0
5,6,7: old = 0 & state = 1
5': old=state & old!=0 & state!=0

Figure 4: RG

collapsed as they differ on the values of predicate `state = 0`. Notice also, that in A_5, the various nodes are labelled 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 RG 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 $\mathtt{3} \to \mathtt{4}$ ($[\mathtt{state} = 0]$) the abstract state is empty ($state = 0 \wedge state = 1$ is unsatisfiable) meaning that edge is not behaviour is not possible. Similarly, if the main thread gets in first, when a context thread attempts to take the abstract edge $\mathtt{2'} \to \mathtt{3'}$, the abstract state is empty. The resulting RG 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 Syntax, Semantics, Abstractions

In this section we shall define our model for multithreaded programs. First we define abstractly the semantics of such programs using transition systems. Then we define the syntactic representations of threads and strands namely CFAs, 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 $\{(\leadsto_1, At_1), (\leadsto_2, At_2), \ldots\}$ where $\leadsto_i$ 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 $\leadsto_{\mathcal{P}}$ and an atomic predicate $At \subseteq \mathcal{V}_X$ defined as follows: Define the predicate $\mathsf{En.s.i}$ where s is an $X-$state and i a thread as: $\mathsf{En.s.i} \equiv ((\exists \mathsf{j} : (At_{\mathsf{j}}.\mathsf{s})) \Rightarrow At_{\mathsf{i}}.\mathsf{s})$. The atomic predicate $At.s \equiv \exists i : (\mathsf{En.s.i})$. The transition relation is defined as: $s \leadsto_{\mathcal{P}} s'$ iff (1) $\mathsf{En.s.i}$ (2) $t \leadsto_i t'$ (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 transtion relation, and all the other variables remain unchanged. The initial state of $\mathcal{P}$ is s_0 which maps every variable to 0. Let $\leadsto^*_{\mathcal{P}}$ be the reflexive transitive closure of $\leadsto_{\mathcal{P}}$. We define $[\![\mathcal{P}]\!] = \{s \mid s_0 \leadsto^*_{\mathcal{P}} s\}$ 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* $\mathtt{asm}\ [\mathtt{p}]$, 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. An edge $(q, \mathtt{op}, q') \in \rightarrow$ is also written as $q \xrightarrow{\mathtt{op}} q'$, and (5) $Q^* \subseteq Q$ is a set of "atomic" locations.

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 $X.C = X \cup \{pc\}$ [2] 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. The transition relation of a CFA is $\leadsto_C$ which is as follows: $s \leadsto_C s'$ if: (1) $s.pc \xrightarrow{\mathtt{op}} s'.pc$ and (2) if $\mathtt{op}$ is $\mathtt{asm}\ p$ then $s \models p$ and if $\mathtt{op}$ is $\mathtt{x} = \mathtt{e}$ then $s'.x = s.e$ and for all $y \notin \{x, pc\}$, $s'.y = s.y$.

A set of states π is called a *data region*, a predicate over the set of variables 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 $sp.\pi.(q, \mathtt{op}, q') = \{s\mathcal{V}_{X.C} \mid \exists s \in \pi.s \leadsto_C s'\}$.

3.3 Strands: 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) $q_0 \in Q$

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

is a *start node*, (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. An edge $(q, \mathtt{Y}, q')$ is also written as $q \xrightarrow{\mathtt{Y}} q'$, (5) $Q^* \subseteq Q$ is a set of atomic abstract locations, and (6) a node labeling function $r : V \rightarrow \mathtt{Pred}.X$ labeling each node with a an abstract data region.

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 $\{s | s.pc \in Q^*\}$ that is, a state is atomic if the abstract thread is at an atomic location. The transition relation of a ACFA is $\rightsquigarrow_A \subseteq S.A \times S.A$ which is as follows: $s \rightsquigarrow_A s'$ if: (1) $s.pc \xrightarrow{\mathtt{op}} s'.pc$ and (2) if $\mathtt{op}$ is $\mathtt{Y}$ then foreach $x \notin \{pc\} \cup \mathtt{Y}$, we have $s'.x = s.x$, and $s' \models r.(s'.pc)$ As before, we can define the *sp* operator for sets of states and ACFA operations.

3.4 Multithreaded Programs

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 (A^ω) denote the symmetric multithreaded program running an arbitrary number of copies of the CFA C (ACFA A). Formally, C^ω (A^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 (A_i) be the CFA C (ACFA A) with each local variable $\mathtt{x} \in X_L \cup \{pc\}$ renamed to x_i. Then $\rightsquigarrow_i = \rightsquigarrow_{C_i}$ ($\rightsquigarrow_{A_i}$) and $At_i = At.C_i$ ($At.A_i$).

3.5 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 [14, 2, 19, 4]. For a set of predicates $P \subseteq \mathtt{Pred}.X$ 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 $P \subseteq \mathtt{Pred}.X$ 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 exceeds the number of control locations of each thread). Thus, we approximate the behaviour of each thread using a *strand.*

Our algorithm incrementally builds strands until it has a strand that overapproximates the behaviour of the thread in the context. To know when the above happens, we require a notion of when one strand overapproximates another. We formalize this notion as *strand simulation* $\preceq$, a variant of simulation [6]. 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 threads running concurrently. Thus, we have to track for an infinite number of threads, the state for that thread, leading to an infinite number of configurations. To surmount this problem, instead of tracking the abstract control location of each strand separately, we shall *count* the number of threads at each strand 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. An *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. The set of abstract context states of a context abstraction is $S.(A, k) = A.Q \rightarrow \{0 \ldots k, \omega\}$.

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 program $\{C\} \cup A^\omega$. We now describe the transition system underlying this abstract program.

Abstract Semantics The set of abstract programs 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 $A.q_0$ the initial state 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 labelling 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 $\hat{s}$.[3]

The abstract transition relation is defined by the operator post that takes a abstract state and an operation o, and produces the successor abstract state. For the operation $\mathsf{o} = (q, \mathsf{op}, q')$ of C at the state $((q, \pi), \Gamma)$, we compute the successor state $\mathsf{post}.((q, \pi), \Gamma).(q, \mathsf{op}, q')$ as $((q', \pi'), \Gamma')$, where $\pi' = \mathsf{Abs}.P.(sp.\pi.(q, \mathsf{op}, q'))$ and $\Gamma'.q = \Gamma.q$. For the operation $\mathsf{o} = (q_1', \mathtt{Y}, q_2')$ of A, the successor state $\mathsf{post}.((q, \pi), \Gamma).(q_1', \mathtt{Y}, q_2') = ((q, \pi'), \Gamma')$ where $\pi' = \mathsf{Abs}.P.(A.r.q_2' \cap (\exists y \in \mathtt{Y}.\pi))$, and $\Gamma'.q_1' = \Gamma.q_1' - 1$, $\Gamma'.q_2' = \Gamma.q_2' + 1$, and for all $q' \in A.Q \setminus \{q_1', q_2'\}$, $\Gamma'.q' = \Gamma.q'$.

We say $\hat{s} \hat{\rightsquigarrow} \hat{s}'$ if there exists an operation o which is enabled in $\hat{s}$ such that $\mathsf{post}.\hat{s}.\mathsf{o} = \hat{s}'$. The reachable states $[\![\hat{\mathcal{P}}]\!] = \{\hat{s} \mid \hat{s}_0 \hat{\rightsquigarrow}^* \hat{s}\}$.

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.

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

A specific instance of the above is the *race detection problem.* For each global variable $x \in \mathcal{P}.X_G$, let $\mathit{Write}.i.x \subseteq \mathcal{V}_X$ (resp. $\mathit{Read}.i.x \subseteq \mathcal{V}_X$) 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 $y = e$ and x is a variable of e, or an assume `asm` p and x is a variable of p. 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.,* $\mathcal{E}_x = \cup_{i \neq j}(\mathit{Write}.i.x \cup \mathit{Read}.i.x) \cap \mathit{Write}.j.x$. 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 strand 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 $[\![C^\omega]\!] \cap \mathcal{E} = \emptyset$. There are two main steps in the checking algorithm (Algorithm Check):

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 when C is composed with a context generated by A 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 a strand G describing the behaviour of C when its context is an arbitrary number of strands A running concurrently.

2. **Guarantee** that the strand A is indeed a sound approximation of the behavior of C in this context, by checking that strand 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 [22]. We now describe ReachAndBuild and CheckSim in greater detail.

Procedure ReachAndBuild is shown in Algorithm 4.2. It is a standard worklist based reachability algorithm [6], but additionally builds an ACFA G summarizing the reachability information. The main loop of lines 3–14 runs the reachability construction, using the worklist L. At each step, the next element is chosen from the worklist, and if it has not been seen before (line 5), it is added to the set of explored states (line 6), and the current region is checked for possible errors.We check if the abstract state $((q, \pi), \Gamma)$ contains any race states, by finding the set of *enabled operations* of the abstract state (described earlier) and checking whether there exist in that set two operations belonging to different threads that access `x` (one being a write), *i.e.,* the operations have different source locations, or they have the same abstract source location, but there is more than one thread at that abstract location. If an error region has been hit (line 7) the procedure finds an (abstract) interleaved error

Algorithm 1 Algorithm ReachAndBuild

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

trace to the error region, and raises an exception containing the error trace. Otherwise, the current region is expanded. For this, we construct the successor of the current region for each operation enabled from the current region (line 12), and connect the current region and its successor as an edge in the ACFA G, using the procedure Connect described shortly. Finally, it adds the successor region to the worklist. Note that the nodes of the ACFA are abstract thread states, that is, we drop the context state information.

Procedure Connect adds the edges between the regions. It takes as argument the augmented ACFA G that is being constructed, abstract thread regions r, r' (the successor of r), and an operation o. The ACFA G is augmented with a map S: each node corresponds to a set of abstract thread states and the thread states of a node n given by $G.S.n$. It first finds nodes n, n' corresponding to r and r' respectively by invoking the procedure Find. When find is called with abstract thread state r it checks if a there exists a node n with $r \in G.S.n$; If so, it returns that node and if not, it returns a new node n where $G.S.n = \{r\}$. The node 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 $(n \xrightarrow{\mathtt{Y}} n')$ is already present in G, we replace it with $(n \xrightarrow{\mathtt{Y} \cup \{x\}} n')$. 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 $(n \xrightarrow{\mathtt{Y}} n')$. Finally, if the operation is a context edge, then the two nodes n and n' are unified by procedure Union which creates a single node n'' which is obtained by "merging" the two nodes: $G.S.n'' = G.S.n \cup G.S.n'$, $G.R.n'' = G.R.n \cup G.R.n'$ and the edges of n'' are the union of the edges of n, n'.

Procedure CheckSim The guarantee part checks that the ACFA G returned by the procedure ReachAndBuild is strand simulated by the ACFA A. The procedure CheckSim implements a variation of the standard simulation checking algorithm [6, 16]. This check ensures soundness.

Theorem 1 **[Soundness]** *If Algorithm* Check *with input thread abstraction (C, P), context abstraction (A, k), and global variable* `x` *terminates and returns "safe" then* $[\![C^\omega]\!] \cap \mathcal{E}_x = \emptyset$.

Algorithm 2 The inference algorithm CIRC

```
Require: CFA $C$, global variable x
 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.$(C, P).(A, k)$.x
 8:     until $(G \leq A)$
 9:     return SAFE
10:   with (Exception($\tau$)) $\rightarrow$
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
```

4.3 Inference

In general, the strand 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 . We start off by assuming the that each thread in the context does nothing, *i.e.,* G is set to the empty strand (line 4). We then minimize the strand G with the procedure Collapse (line 6). Collapse takes ACFA G and returns its bisimulation quotient ACFA A [6], 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 strand G (line 7). If the present approximation A simulates the new strand 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 ACFA 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 $\{C\} \cup A^\omega$, and returns either a real error REAL together with a concrete interleaved trace $\bar{s}$ in C^ω that reaches the error state, or a refinement of the abstraction (if the current error path τ does not have a concrete realization in C^ω). Accordingly, the algorithm returns UNSAFE if a real error is found, or updates the thread and context abstractions by adding new predicates and updating the maximum value of a counter in the context abstraction 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 optimizations.

Theorem 2 *If Algorithm* CIRC *on input CFA C and global variable* x *terminates and returns "safe" then C^ω is safe w.r.t. $\mathcal{E}_x$; if it returns "unsafe" then C^ω is unsafe w.r.t. $\mathcal{E}_x$.*

5 Details

Procedure Collapse The procedure Collapse takes an ACFA G and constructs a bisimulation quotient A of G with respect to the global predicates. It returns the bisimulation quotient A, along with a mapping from $G.Q$ to $A.Q$ mapping each state of G to its equivalence class in A. We do this in two steps. First, we replace, for each location $q \in G.Q$, the region $G.r.q$ with the region obtained by quantifying out all local variables from $A.r.q$ 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 bisimilarity algorithm [6] 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 $n \xrightarrow{Y} n'$, and the bisimilarity collapses n, n' to the same node n'' in A, we ensure that A has a self loop edge $n'' \xrightarrow{Y} n''$. The result is an ACFA with only global variables in its node 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. It works in two steps:

Computing an Interleaving. An abstract trace is a sequence of operations of the main thread and the context ACFAs. A scan over the entire trace suffices to check if the k parameter is large enough. If not, the only refinement is that k is incremented, and if so, 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 RG of the ACFA.

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 [17].

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 $state = 0, state = 1$. □

`T1: I → II`	`skip`	true
`T1: II → 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

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 $((C.q_0, \texttt{true}), \Gamma_0^k)$ 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$^\omega$.$(C, P).(A, k_0) \leq A$. 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 $\hat{A}^\omega$. 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 $q' \rightarrow q''$ 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 $q = q'$. A node $n \in \hat{G}.Q$ is *good* for a transition $e = q' \xrightarrow{\{x_1,\ldots,x_n\}} q''$ 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, $(\exists \mathtt{x}_1 \ldots \mathtt{x}_n.(\hat{G}.r.n)) \cap (\hat{A}.r.q'') \subseteq (\hat{G}.r.n)$. We check that all nodes $n \in \hat{G}.Q$ 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 $\hat{A}$ strand simulates a context with arbitrarily many threads. If the check fails, we increment k_0 and rerun the main loop. The check avoids analyzing C *together with* A^ω, since this takes time proportional to the product of the size of C (T1) and the size of the reachable states of A^ω (T2). Instead, we analyze A^ω alone, and use that analysis, together with C to give the soundness check. The procedure outlined above takes time T1 + T2, which is substantially less than the product.

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 region, 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.

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. Lines = Size of compiled C source

6 Experiences

nesC [12] is a programming language for networked embedded systems. It is used to implement event driven applications in the TinyOS operating system [20]. 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 unique 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 modelled 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, modelled by a special global that we added) 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. Preds is the number of predicates discovered to prove safety, ACFA is the size of 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. As far as we know this is a race though there may be some external circumstance that we cannot glean from the program that prevents this happening. 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 failure, but CIRC verified that in that context, the function always succeeded. `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 this 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 the first thread had finished writing to `tosPort`. On modelling this interrupt, the tool was able to prove safety.

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

References

[1] T. Ball, S. Chaki, and S. K. Rajamani. Parameterized verification of multithreaded software libraries. In *TACAS 01*, LNCS 2031, pp. 158–173. Springer, 2001.

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

[3] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA 02*, pp. 211–230, 2002.

[4] 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*, 2003.

[5] 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*, pp. 258–269. ACM, 2002.

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

[7] 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*, pp. 439–448, 2000.

[8] G. Delzanno, J.-F. Raskin, and L. Van Begin. Towards the automated verification of multithreaded java programs. In *TACAS 02*, pp. 173–187. Springer, 2002.

[9] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *LICS 99*, pp. 352–359. IEEE Press, 1999.

[10] C. Flanagan and S.N. Freund. Detecting race conditions in large programs. In *PASTE 01*, pp. 90–96. ACM, 2001.

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

[12] 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 03*, pp. 1–11. ACM, 2003.

[13] P. Godefroid. Model checking for programming languages using Verisoft. In *POPL 97*, pp. 174–186. ACM, 1997.

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

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

[16] M.R. Henzinger, T.A. Henzinger, and P.W. Kopke. Computing simulations on finite and infinite graphs. In *FOCS 95*, pp. 453–462. IEEE Press, 1995.

[17] T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *POPL 04*, ACM, 2004.

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

[19] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL 02*, pp. 58–70. ACM, 2002.

[20] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *ASPLOS 00*, pp. 93–104. ACM, 2000.

[21] G.J. Holzmann. Logic verification of ANSI-C code with SPIN. In *SPIN 00*, LNCS 1885, pp. 131–147. Springer, 2000.

[22] C.B. Jones. Tentative steps toward a development method for interfering programs. *ACM TOPLAS* 5(4):596–619, 1983.

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

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

[25] C. von Praun and T. Gross. Static conflict analysis for multithreaded object-oriented programs. In *PLDI 03*, pp. 115–128. ACM, 2003.