Detecting Fair Non-termination in Multithreaded Programs*

Mohamed Faouzi Atig[1], Ahmed Bouajjani[2], Michael Emmi[2,**], and Akash Lal[3]

[1] Uppsala University, Sweden
mohamed_faouzi.atig@it.uu.se
[2] LIAFA, Université Paris Diderot, France
{abou,mje}@liafa.jussieu.fr
[3] Microsoft Research, Bangalore, India
akashl@microsoft.com

Abstract. We develop compositional analysis algorithms for detecting non-termination in multithreaded programs. Our analysis explores fair and ultimately-periodic executions—i.e., those in which the infinitely-often enabled threads repeatedly execute the same sequences of actions over and over. By limiting the number of context-switches each thread is allowed along any repeating action sequence, our algorithm quickly discovers practically-arising non-terminating executions. Limiting the number of context-switches in each period leads to a compositional analysis in which we consider each thread separately, in isolation, and reduces the search for fair ultimately-periodic executions in multithreaded programs to state-reachability in sequential programs. We implement our analysis by a systematic code-to-code translation from multithreaded programs to sequential programs. By leveraging standard sequential analysis tools, our prototype tool MUTANT is able to discover fair non-terminating executions in typical mutual exclusion protocols and concurrent data-structure algorithms.

1 Introduction

Multithreaded programming is the predominant style for implementing parallel and reactive single-processor software. A multithreaded program is composed of several sequentially-executing *threads* who share the same memory address space. As a thread's operations on shared memory generally do not commute with the operations of others, each *schedule*—i.e., each distinct order on the actions of different threads—leads to distinct program behavior. Generally speaking, the schedule of inter-thread execution relies on factors external to the program, such as processor utilization and I/O activity. Though some programming errors are witnessed in many different schedules, and are thus likely to be discovered by testing, others manifest only in a small number of rarely-encountered schedules; these *Heisenbugs* are notoriously difficult to debug.

The correctness criteria for multithreaded programs generally include both safety and liveness conditions, and ensuring safety can threaten liveness. For instance, to ensure linearizability—i.e., the result of concurrently executing operations is equivalent to

* Partially supported by the project ANR-09-SEGI-016 Veridyc.
** Supported by a Fondation Sciences Mathématiques de Paris post-doctoral fellowship.

P. Madhusudan and S.A. Seshia (Eds.): CAV 2012, LNCS 7358, pp. 210–226, 2012.

some sequential execution of the same operations—concurrent data structure implementations often employ a *retrying* mechanism [9] (see Figure 1c for a simple instance): a validation phase before the effectuation of each operation ensures concurrent modifications have not interfered; when validation fails, the operation is simply attempted again. A priori nothing prevents an operation from being retried forever. Retry is also a mechanism used in mutual exclusion protocols. For instance, a common solution to the dining philosophers problem proposes that philosophers drop the fork they first picked up when they cannot obtain the second fork—presumably because a neighboring philosopher already holds the second. Though this scheme avoids deadlock, it also leads to non-terminating executions in which no philosophers ever eat; particularly when each philosopher picks up his first fork, finds his neighbor has the other, and then all release their first fork, repeatedly; Figure 1b illustrates a simplification of this pattern. As such retrying raises the possibility that some or all interfering operations are never completed even under *fair* schedules—repeatedly failing operations already execute infinitely often—one does want to ensure that concurrent operations do always terminate. Note that unlike in sequential programs, where interesting non-terminating executions involve ever diverging data values, non-terminating executions in multithreaded programs also involve repeated inter-thread interference, even over small finite data domains (see Figure 1).

Proving the absence of programming errors such as assertion violations, and unintentional non-termination due to inter-thread interference, in multithreaded programs is difficult precisely because of the enormous number of possible schedules which need be considered. Automated approaches based on model checking are highly complex—e.g., computing state-reachability is PSPACE-complete when threads are finite state [10], and undecidable when threads are recursive [23]—and are susceptible to state-explosion; naïve approaches are unlikely to scale to realistic programs. Otherwise, modular deductive verification techniques may apply, though they require programmer-supplied invariants, which for multithreaded programs are regarded as difficult to divine. Furthermore, a failed verification attempt may only prove that the supplied invariants are insufficient, rather than the existence of a programming error.

Instead of exhaustive program exploration, recent approaches to detecting safety violations (e.g., assertion violations) have focused on exploring only a representative subset of program behaviors by limiting inter-thread interaction [22, 21, 17, 15, 3]; for instance, Qadeer and Rehof [21] consider only executions with a given number $k \in \mathbb{N}$ of *context switches* between threads. Though techniques like *context-bounding* are clearly incomplete for any given $k \in \mathbb{N}$, every execution is considered in the limit as k approaches infinity, and small values of k have proved to provide great coverage [17] and uncover subtle bugs [13] in practice. The bounded analysis approach is particularly attractive since it enables compositional reasoning: each thread can be considered separately, in isolation, once the number of environmental interactions is fixed. This fact has been exploited by the so-called "sequentializations" which reduce multithreaded state-reachability under an interaction bound to state-reachability in a polynomially-sized sequential program [15, 11, 7, 3], leading to efficient analyses. Conveniently these reductions allow leveraging highly-developed sequential program analysis tools for multithreaded program analysis.

```
1 // One thread
2 // forever
3 // spins
4 var g: 𝔹
5
6 proc Thread1 ()
7    g := false;
8    while !g do
9       skip;
10   return
11
12 proc Thread2 ()
13   g := true;
14   return
```

(a)

```
1 // Both threads
2 // can retry
3 // forever
4 var g: 𝔹
5
6 proc Thread1 ()
7    while g do
8       g := false;
9    return
10
11 proc Thread2 ()
12   while !g do
13      g := true;
14   return
```

(b)

```
// The second thread can forever retry
1 var g: T
2 var x: 𝔹
3
4 proc Thread1 ()
5    while ⋆ do
6       acquire x;
7       g := ⋆;
8       release x;
9    return
```

```
10 proc Thread2 ()
11    var gi, gf: T
12    while true do
13       gi := g;
14       gf := ...;
15       acquire x;
16       if g = gi then
17          g := gf;
18          release x;
19          return
20       else
21          release x
22    return
```

(c)

Fig. 1. Three programs with non-terminating executions. (a) Though the first thread may execute forever if the second never sets g to `true`, no such execution is fair. (b) Two threads repeatedly trying to validate their set values of g will keep retrying forever under a schedule which schedules each loop head just after the opposing thread's assignment. (c) As long as the first thread executes an iteration between each of the second thread's reads and validations of g, the second thread is never able to finish its operation.

Though these techniques seem promising for the detection of safety violations, they have been deemed inapplicable for detecting *liveness* violations, since, for instance, in any context-bounded execution, only one thread can execute infinitely often; interesting concurrency bugs such as unintentional yet coordinated non-termination require the participation of multiple infinitely-often executing threads. This limitation has effectively prevented the application of compositional bounded analyses to detecting liveness violations in multithreaded programs.

In this work we demonstrate that restricting thread interaction also leads to an effective technique for detecting liveness violations in recursive multithreaded programs—in particular we detect the presence of fair non-terminating executions. Though in general the problem of detecting non-terminating executions is very difficult, we restrict our attention to the simpler (recursively-enumerable yet still undecidable) case of fair *ultimately periodic* executions, which after a finite execution prefix (called the *stem*) ultimately repeat the same sequence of actions (the *lasso*) over and over again. Many interesting non-terminating executions occurring in practice are ultimately periodic. For instance, in the program of Figure 1b, every non-terminating execution must repeat the same sequence of statements on Lines 7, 8, 12, and 13. Similarly, every fair non-terminating execution of the program in Figure 1c must repeat the statements of Lines 5–8, 12–15, and 20–21. Thus focusing on periodically repeating executions is already quite interesting. Furthermore, every ultimately periodic execution is described with a finite number of thread contexts: those occurring during the stem, and those occurring during each iteration of the lasso; e.g., the non-terminating executions of each program in Figure 1 require just two contexts per thread: one per stem, and one per lasso.

By bounding the number of thread contexts we detect ultimately periodic executions compositionally, without exposing the local configurations of each thread to one another. We actually detect ultimately periodic executions that repeatedly encounter, along some lasso, the same sequence of shared global state valuations at thread context-switch points. Clearly ultimate state-repeatability is a sufficient condition for ultimate periodicity. We prove that this condition is necessary when the domain of shared global state valuations is finite. (This is not trivial in the presence of recursion, where threads access unbounded procedure stacks). Then, supposing each thread executes within k_1 contexts during the stem, and within k_2 contexts during each iteration of this lasso, its execution is summarized by an *interface* of $k = 2(k_1 + k_2)$ valuations $g_1 g'_1 \dots g_k g'_k$: the shared global state valuations g_i and g'_i, resp., encountered at the beginning and end of each execution context during the stem and lasso. Given the possible bounded interfaces of each thread, we infer the existence of ultimately periodic executions by composing thread interfaces. Essentially, two context summaries $g_1 g'_1$ and $g_2 g'_2$ compose when $g'_1 = g_2$; by composing interfaces so that the valuation reached in the last context of the lasso match both the valuation reached in the last context of the stem, and the starting valuation of the first context of the lasso, we deduce the existence of a periodic computation.

We thus reduce the problem of detecting ultimately periodic computations to that of computing thread interfaces. Essentially, we must establish two conditions on an interface $g_1 g'_1 \dots g_k g'_k$ of a thread t: first, the interface describes a valid thread computation, i.e., beginning from g_1, t executing alone reaches g'_1, and when resumed from the valuation g_2, t executing alone reaches g'_2, etc. Second, the interface is *repeatable*, i.e., each time t returns to its first lasso context i, t can again repeat the same sequence of global valuations $g_i g'_i \dots g_k g'_k$. Though both conditions reduce to (repeated) state-reachability for non-recursive programs, ensuring repeatability in recursive programs requires establishing equivalence of an unbounded number of procedure frames visited along each period of the lasso. An execution in which the procedure stack incurs a net decrease, for instance, along the lasso is not repeatable. We avoid explicitly comparing stack frames simply by noticing that along each period of any repeating execution there exists a procedure *keyframe* which is never returned from. By checking whether one keyframe can reach the same keyframe—perhaps with the first keyframe below on the procedure stack—in the same context number one period later, we ensure repeatability.

Finally, to ensure that the detected non-terminating executions are fair, we expose a bounded amount of additional information across thread interfaces. For the case of strong fairness, we observe that any thread t which does not execute during the lasso must be *blocked*, i.e., waiting on a synchronization object x which has not been signaled. Furthermore, in any fair execution, no concurrently executing thread may signal x, since otherwise t would become temporarily enabled—thus a violation of strong fairness. In this way, by ensuring thread interfaces agree on the set X of indefinitely waited-on synchronization objects, each thread can locally ensure no $x \in X$ is signaled during the lasso, and only threads waiting on some $x \in X$ are exempt from participating in the lasso.

As is the case for finding safety violations, the compositional fair non-termination analysis we describe in Section 3 has a convenient encoding as sequential program

analysis. In Section 4 we describe a code-to-code translation from multithreaded programs to sequential programs which violate an assertion exactly when the source program has a fair ultimately periodic execution with given bounds k_1 and k_2 on the number of stem and lasso contexts.[1] In Section 5 we discuss our implementation MUTANT, which systematically detects fair non-terminating executions in typical concurrent data structure and mutual exclusion algorithms.

2 Recursive Multithreaded Programs

We consider a simple but general multithreaded program model in which each of a statically-determined collection Tids of threads concurrently execute as recursive sequential programs which access a shared global state. For simplicity we suppose each program declares a single shared global variable $\mathtt{g}$ with domain Vals, and each procedure from a finite set Procs declares only a single parameter $\mathtt{l}$, also of domain Vals; furthermore each program statement is uniquely labeled from a set Locs of program locations. A *(procedure) frame* $f = \langle \ell, v \rangle$ is a program location $\ell \in \mathsf{Locs}$ along with a local variable valuation $v \in \mathsf{Vals}$, and a *configuration* $c = \langle g, \sigma \rangle$ is a shared global state valuation $g \in \mathsf{Vals}$ along with a *local state map* $\sigma : \mathsf{Tids} \to (\mathsf{Locs} \times \mathsf{Vals})^+$ mapping each thread t to a procedure frame stack $\sigma(t)$. The transition relation $\overset{t,\ell}{\Longrightarrow}$ between configurations is labelled by the active program location $\ell \in \mathsf{Locs}$ and acting thread $t \in \mathsf{Tids}$. We suppose a standard set of inter-procedural program statements, including assignment $x \ \mathtt{:=}\ e$, branching **if** e **then** s_1 **else** s_2, and looping **while** e **do** s statements, lock **acquire** e and **release** e, and procedure **call** $x \ \mathtt{:=}\ p\ e$ and **return** e, where e are expressions from an unspecified grammar, s are labeled sub-statements, and $p \in \mathsf{Procs}$. The definition of the transition relation is standard, as are the following:

Trace, Reachable: A *trace* π of a program P from a configuration c is a possibly empty transition-label sequence $a_0 a_1 a_2 \ldots$ for which there exists a configuration sequence $c_0 c_1 c_2 \ldots$ such that $c_0 = c$ and $c_j \overset{a_j}{\Longrightarrow}_P c_{j+1}$ for all $0 \leq j < |\pi|$; each configuration $c_j = \langle g, \sigma \rangle$ (alternatively, the shared global valuation g) is said to be *reachable* from c by the finite trace $\pi_j = a_0 a_1 \ldots a_{j-1}$.

Context: A *context of thread* t is a trace $\pi = a_0 a_1 \ldots$ in which for all $0 \leq j < |\pi|$ there exists $\ell \in \mathsf{Locs}$ such that $a_j = \langle t, \ell \rangle$; every trace is a context-sequence concatenation.

Enabled, Blocked, Fair: A thread $t \in \mathsf{Tids}$ is *enabled* after a finite trace π if and only if there exists an a labeling a t-transition such that $\pi \cdot a$ is also a trace; otherwise t is *blocked*. An infinite trace is *strongly fair* (resp., *weakly fair*) if each infinitely-often (resp., continuously) enabled thread makes a transition infinitely often.

Checking typical safety and liveness specifications often reduces to finding whether certain program configurations are reachable, or determining whether fair infinite traces are possible. The following two problems are thus fundamental.

Problem 1 (State-Reachability). Given a configuration c of a program P, and a shared global state valuation g, is g reachable from c in P?

[1] Technically, our reduction considers round-robin schedules of thread contexts.

Problem 2 (Fair Non-Termination). Given a configuration c of a program P, does there exist an infinite strongly (resp., weakly) fair trace of P from c?

Even for recursive multithreaded programs accessing finite data, both problems are undecidable [23]. However, while state-reachability is recursively enumerable by examining all possible concurrent traces in increasing length, detecting non-terminating traces is more complex; from Yen [24] one deduces that the problem is not even semi-decidable. A simpler problem is to detect non-terminating traces which eventually repeat the same sequence of actions indefinitely. Formally, an infinite trace π is *ultimately periodic* when there exists two finite traces μ and ν, called resp., the *stem* and *lasso*, such that $\pi = \mu \cdot \nu^\omega$. Then a key question is the detection of ultimately periodic traces.

Problem 3 (Fair Periodic Non-Termination). Given a configuration c of a program P, does there exist an ultimately periodic strongly (resp., weakly) fair trace of P from c?

Periodic non-termination is also undecidable, yet still recursively enumerable—by examining all possible stems and lassos in increasing length. This implies that not all non-terminating executions are ultimately periodic. In principle, coordinating threads can construct phased executions in which each phase consists of an increasingly-longer sequence of actions, using their unbounded procedure stacks to simulate unbounded integer counters. Still, it is unclear whether non-periodic executions arise in practice. Our goal is to efficiently detect ultimately periodic fair traces where they exist.

3 Bounded Compositional Non-termination Analysis

Rather than incrementally searching for non-terminating executions by bounding the length of the considered stems and lassos, our discovery strategy bounds the number of thread contexts in the considered stems and lassos; this strategy is justified by the hypothesis that many interesting bugs are likely to occur within few contexts per thread [21, 17]. Notice, for instance, that the non-terminating executions of each of the programs in Figure 1 require only one context-switch per thread during their repeating sequences of actions. Formally for $k \in \mathbb{N}$, we say a trace $\pi = a_0 a_1 \ldots$ is *k context-bounded* when there exist $j_1, j_2, \ldots, j_k \in \mathbb{N}$ and $j_{k+1} = |\pi|$ such that $\pi = \pi_1 \pi_2 \ldots \pi_k$ is a sequence of k thread contexts $\pi_i = a_{j_i} \ldots a_{j_{i+1}-1}$; we refer to each j_i as a *context-switch point*. Though we expect many ultimately periodic traces to exhibit few context switches per period, context-bounding is anyhow complete in the limit as context-bounds approach infinity.

Remark 1. For every ultimately periodic trace $\mu \cdot \nu^\omega$ there exists $k_1, k_2 \in \mathbb{N}$ such that μ and ν are, resp., k_1 and k_2 context-bounded.

In what follows, we show that for given stem and lasso context-bounds, resp., $k_1 \in \mathbb{N}$ and $k_2 \in \mathbb{N}$, the fair periodic non-termination problem reduces to the state-reachability problem in sequential programs; the salient feature of this reduction is compositionality: the resulting sequential program considers each thread independently, without explicitly representing thread-product states. The general idea is to show that all ultimately periodic executions can be decomposed into stem and lasso such that during each period of

the given lasso, each thread reencounters the same global valuations at context-switch points, and reencounters the same topmost stack-frame valuation; since each procedure stack must be non-decreasing over each lasso iteration, there must be some frame during each period which is never returned from. We show that detecting these repeated global valuations and topmost stack frames over a single lasso iteration implies periodicity.

To begin, we show that the existence of an ultimately periodic trace $\mu \cdot \nu^\omega$ implies the existence of an ultimately periodic trace $\mu' \cdot \nu'^\omega$ in which the sequence of global valuations (and topmost procedure-stack frames) of each thread at context-switch points repeat in each iteration of the lasso ν'.

3.1 Annotated Traces

For a configuration $c = \langle g, \sigma \rangle$ of a program P, we write $c[\mathsf{g} := g']$ to denote the configuration $c = \langle g', \sigma \rangle$. An *annotated trace* $\bar{\pi}$ of a program P is a sequence $\bar{\pi} = \langle g_i, \tau_i, \pi_i, g'_i, \tau'_i \rangle_{i=1\ldots k}$—where each $g_i, g'_i \in \mathsf{Vals}$ are global valuations, $\tau_i, \tau'_i \in \mathsf{Tids} \to (\mathsf{Locs} \times \mathsf{Vals})$ are thread-to-frame mappings, and π_i is a thread context—for which there exist local-state maps $\sigma_1, \sigma'_1, \ldots, \sigma_k, \sigma'_k : \mathsf{Tids} \to (\mathsf{Locs} \times \mathsf{Vals})^+$ of configurations $c_1, c'_1, \ldots, c_k, c'_k$ where for each $1 \leq i \leq k$:

- $c_i = \langle g_i, \sigma_i \rangle$ and $c'_i = \langle g'_i, \sigma'_i \rangle$,
- $\sigma_i(t) = \tau_i(t) \cdot w_i$ and $\sigma'_i(t) = \tau'_i(t) \cdot w'_i$ for each thread $t \in \mathsf{Tids}$, for some w_i, w'_i,
- each c'_i is reachable from c_i via the trace π_i, and
- $c_{i+1} = c'_i[\mathsf{g} := g_{i+1}]$ for $i < k$.

We say the annotated trace $\bar{\pi}$ is *valid* when $c_{i+1} = c'_i$ for $1 \leq i < k$. The definitions applying to traces are lifted naturally to annotated traces.

Lemma 1. *There exists an ultimately periodic trace $\mu \cdot \nu^\omega$ from a configuration c in a program P iff there exists a valid annotated ultimately periodic trace $\bar{\mu} \cdot \bar{\nu}^\omega$ from c in P.*

As we are mainly concerned with annotated traces, we usually drop the bar-notation, writing, e.g., π to denote an annotated trace $\bar{\pi}$, and use "trace" to mean "annotated trace."

3.2 Compositional Detection of Periodic Traces

In the following we reduce the detection of valid ultimately periodic traces to the detection of ultimately periodic traces for each individual thread $t \in \mathsf{Tids}$. Let $\pi = \mu \cdot \nu^\omega$ be a valid ultimately periodic trace which divides μ and ν, resp., into $k_1 \in \mathbb{N}$ and $k_2 \in \mathbb{N}$ contexts, indexed by $I^\mu \subseteq \mathbb{N}$ and $I^\nu \subseteq \mathbb{N}$, as $\mu = \langle g_i, \tau_i, \mu_i, g'_i, \tau'_i \rangle_{i \in I^\mu}$ and $\nu = \langle g_i, \tau_i, \nu_i, g'_i, \tau'_i \rangle_{i \in I^\nu}$. We construct an ultimately periodic trace π_t in which only t is active. Roughly speaking, the constructed trace π_t corresponds to the projection of π on the set of t-labeled transitions. Given the global values g_i and g'_i seen at the beginning and end of each context i of thread t, the trace π_t can be computed in complete isolation: we simply resume the ith context of thread t with the global value g_i, and ensure g'_i is encountered at the end of the ith context. Supposing thread t executes in the contexts

indexed by $I_t^\mu \subseteq I^\mu$, along the stem μ, and in the contexts indexed by $I_t^\nu \subseteq I^\nu$ along the lasso ν, we define the *thread-periodic trace for* t as $\pi_t \stackrel{\text{def}}{=} \mu_t \cdot \nu_t^\omega$, where

$$\mu_t = \left\langle g_i, \tau_i(t), \mu_i, g_i', \tau_i'(t) \right\rangle_{i \in I_t^\mu} \qquad \nu_t = \left\langle g_i, \tau_i(t), \nu_i, g_i', \tau_i'(t) \right\rangle_{i \in I_t^\nu}$$

For the thread-periodic trace π_t of t, we associate two sequences $SI(\pi_t) = \langle g_i, g_i' \rangle_{i \in I_t^\mu}$ and $LI(\pi_t) = \langle g_i, g_i' \rangle_{i \in I_t^\nu}$ of global valuation pairs encountered at the beginning and end of each context, called, resp., the *stem and lasso interfaces*; the sizes of interfaces are bounded by the number of contexts: $|SI(\pi_t)| \leq k_1$ and $|LI(\pi_t)| \leq k_2$.

We define the *shuffle* of a sequence set S inductively as $\text{shuffle}(\{\varepsilon\}) = \{\varepsilon\}$, and $\text{shuffle}(S) = \bigcup \{s_1 \cdot \text{shuffle}(S') : s_1 s_2 \ldots s_j \in S \text{ and } S \setminus \{s_1 \ldots s_j\} \cup \{s_2 \ldots s_j\} = S'\}$; for instance, $\text{shuffle}(\{s_1 s_2, s_3\}) = \{s_1 s_2 s_3, s_1 s_3 s_2, s_3 s_1 s_2\}$. We say the thread interface sets S and L are *compatible* when there exists $s_1 \ldots s_{k_1} \in \text{shuffle}(S)$ and $s_{k_1+1} \ldots s_{k_1+k_2} \in \text{shuffle}(L)$ where each $s_i = \langle g_i, g_i' \rangle$, and $g_i' = g_{i+1}$ for $0 < i < k_1 + k_2$, and $g_{k_1+k_2}' = g_{k_1+1}$. Extending this definition, we say a set $\{\pi_t : t \in \mathsf{Tids}\}$ of thread-periodic traces is *compatible* if and only if $\{SI(\pi_t) : t \in \mathsf{Tids}\}$ and $\{LI(\pi_t) : t \in \mathsf{Tids}\}$ are compatible.

Lemma 2. *If there exists a compatible set of thread-periodic traces* $\{\pi_t : t \in \mathsf{Tids}\}$ *of a program P, then there exists a valid ultimately periodic trace* $\pi = \mu \cdot \nu^\omega$ *of P. Moreover,* μ *and* ν *are, resp.,* $\sum_{t \in \mathsf{Tids}} |SI(\pi_t)|$ *and* $\sum_{t \in \mathsf{Tids}} |LI(\pi_t)|$ *context-bounded.*

Lemma 2 suggests a compositional algorithm to detect ultimately periodic valid traces. As each trace is constructed from a straight-forward composition of thread-periodic traces, we need simply to compute a compatible set of thread-periodic traces. We thus reduce the detection of valid ultimately periodic traces to computing (finite) compatible thread interface sets $\{S_t : t \in \mathsf{Tids} \text{ and } |S_t| \leq k_1\}$ and $\{L_t : t \in \mathsf{Tids} \text{ and } |L_t| \leq k_2\}$, and ensure the existence of, for each thread $t \in \mathsf{Tids}$, a thread-periodic trace π_t such that $SI(\pi_t) = S_t$ and $LI(\pi_t) = L_t$.

3.3 From Thread-Periodic Traces to Sequential Reachability

Section 3.2 reduced the problem of finding periodic executions to that of computing thread interfaces. Now we demonstrate that thread interfaces can be computed by state-reachability in sequential programs. For the remainder of this section we fix an initial configuration c_0 of a program P, and a thread-periodic trace $\pi_t = \mu_t \cdot \nu_t^\omega$ of a thread t.

We know that the thread period trace π_t repeats the same sequence of actions per period over and over indefinitely. It follows that although during each period the size of t's frame stack may increase and decrease due to procedure calling and returning, the *net size* of t's frame stack must not be decreasing—otherwise t cannot repeat ν_t indefinitely. This implies that there exists a sequence $f_1 f_2 \ldots$ of t's procedure frames—each $f_i \in (\mathsf{Locs} \times \mathsf{Vals})$ encountered in the ith period—which are never returned from; we call these frames the *keyframes* of t. Since we repeat the same sequence of calls and returns along each period, we can assume w.l.o.g. that each keyframe f_i is the procedure frame encountered at the beginning of the same context `shift` in ν_t with $0 \leq \texttt{shift} < |LI(\pi_t)|$. Furthermore, we know that these keyframes correspond to the same procedure frame f (from definition of value annotated traces).

In order to check that f is a keyframe (i.e., never removed from the stack), we check that from a configuration where the stack contains only the frame f, we can reach a configuration with topmost frame f after executing the trace ν_t (modulo rotation). We know also that executing the trace μ_t followed by the first `shift` contexts in ν_t will result in a configuration with topmost frame f. This is exactly what is defined below:

Feasibility: Let $SI(\pi_t) = \langle g_i, g'_i \rangle_{i \in I_t^\mu}$ and $LI(\pi_t) = \langle g_i, g'_i \rangle_{i \in I_t^\nu}$ be the given stem and lasso interfaces. We say that $\langle SI(\pi_t), LI(\pi_t) \rangle$ is *feasible* if there are a frame f, a natural number `shift` with $0 \leq \texttt{shift} < k_1$, and a sequence of configurations $c_1, c'_1, \ldots c_m, c'_m$ with $m = (k_1 + k_2 + \texttt{shift})$ such that, for every $1 \leq j \leq m$:

- c'_j is reachable from c_j via a trace of the thread t.
- The global valuation in c_j and c'_j are g_i and g'_i with $i = j(\text{mod } k_1 + k_2) + k_2 + 1$.
- The stack in c'_{j-1} and c_j are the same when $j \neq (k_2 + \texttt{shift} + 1)$ with $c'_0 = c_0$.
- The stack in $c_{k_2+\texttt{shift}+1}$ contains only the frame f. Moreover, the topmost frame in $c'_{k_2+\texttt{shift}}$ and c'_m is precisely f.

Lemma 3. *If the thread trace π_t is periodic, then $\langle SI(\pi_t), LI(\pi_t) \rangle$ is feasible.*

Now, we can show if there is a thread trace π'_t of t from a configuration containing the keyframe f, satisfying the interface L_t, and reaching a configuration whose topmost frame is precisely f, then this thread trace can be executed infinity often. This means that π'_t can be considered as a lasso trace of t whose lasso interface is precisely L_t. On the other hand, if there is a thread trace π''_t of t from the initial configuration to a configuration whose topmost keyframe is precisely f while respecting the stem interface S'_t (which is the concatenation of S_t and the first $(\texttt{shift})$-elements of L_t) then $\pi''_t \cdot \pi'_t$ can be considered as a stem trace for the lasso trace π'_t.

Lemma 4. *Given a compatible interface sets $\{S_t : t \in \mathsf{Tids}\}$ and $\{L_t : t \in \mathsf{Tids}\}$ such that $\langle S_t, L_t \rangle$ is feasible for each $t \in \mathsf{Tids}$, we can construct compatible thread-periodic traces $\{\pi_t : t \in \mathsf{Tids}\}$ such that $|SI(\pi_t)| = |S_t| + |L_t|$ and $|LI(\pi_t)| = |L_t|$ for each $t \in \mathsf{Tids}$.*

The lemmata above suggest the following procedure: first, guess compatible interfaces $\{S_t, L_t : t \in \mathsf{Tids}\}$, then check feasibility of each $\langle S_t, L_t \rangle$. Observe that checking the feasibility of each given pair $\langle S_t, L_t \rangle$ boilds down to solving reachability problems in the sequential program describing the behavior of the thread t. Section 4 concretizes this algorithm in a code-to-code reduction to sequential program analysis.

3.4 Encoding Fairness

By our definitions in Section 2 any blocked thread must be waiting to acquire a held lock. This leads to the following characterization of strongly fair ultimately periodic traces: for each thread $t \in \mathsf{Tids}$, either

Case 1. The lasso contains at least one transition of t, or

Case 2. The thread t is blocked throughout the lasso, waiting to acquire some lock $x \in \mathsf{Locks}$; this further implies that

Cond. 1 the lock x may not be released during the lasso by any thread,

Cond. 2 the lock x must be held by another thread at the beginning of the lasso, and

Cond. 3 t remains at the control location of the acquire of x throughout the lasso.

These conditions characterize strongly fair ultimately periodic computations. We ensure these conditions are met by extending the notion of interfaces to include the set $X \subseteq \mathsf{Locks}$ of locks which are held throughout the lasso. Then, we must ensure locally per thread that any $x \in X$ is not released during the lasso (Cond. 1), and that some thread holds x when entering the lasso (Cond. 2); additionally, we allow any thread attempting to acquire some $x \in X$ to execute no further action. (Observe that if the lock x is released during the lasso by any thread (see Cond. 1 of Case 2) then the resulting ultimately periodic computation is not strongly-fair since the thread t is infinitely often enabled and does not infinity often fire a transition.) Weak fairness can be similarly characterized using a set $Y \subseteq \mathsf{Locks}$ of locks which are held at some point during the lasso; we then ensure that each $y \in Y$ is either held at the beginning of the lasso, or acquired at some point during the lasso.

4 Reduction to Sequential Program Analysis

The compositional analysis outlined in Section 3 reduces (context-bounded) fair periodic non-termination to state-reachability in sequential programs. Given thread stem and lasso interfaces, and the set of locks held throughout the lasso, the feasibility of each interface is computed separately, per thread. In this section we describe how to implement this reduction by a code-to-code translation to sequential programs with an assertion which fails exactly when the source program has a strongly fair ultimately periodic execution whose stem and lasso satisfy a given context bound.[2] Figure 2 lists our translation in full.

Essentially, we introduce a `Main` procedure for the target program which executes each thread one-by-one using an initially-guessed sequence of global valuations stored in $\texttt{Stem}_0$ and $\texttt{Lasso}_0$. For each thread t, we guess the number—stored in `shift`—of contexts following the k_1st context until t's keyframe is encountered on Line 18, and begin executing t's main procedure `Main[`t`]` on Line 21. Initially, the values stored in `Stem` and `Lasso` are the values seen at the beginning of each context of the first thread during, resp., the stem and repeating lasso. After execution of the ith thread, the values of `Stem` and `Lasso` are the values seen at the end of each context of the ith thread, and at the beginning of each context of the $(i+1)$st thread. Accordingly, after the execution of the final thread, the values seen at the end of each context must match the values guessed at the beginning of the following contexts of the first thread, according to the round-robin order; the assumptions on Lines 22–27 ensure these values match.

The execution of each thread thus acts simply to compute its interface. As the keyframes of different threads may be encountered at different points along the lasso,

[2] Technically we consider bounded round-robin thread schedules rather than bounded context switch. Though in principle the two notions are equivalent for a fixed number of threads—i.e., any k-context execution takes place within k rounds, and any k-round n-thread execution takes place in kn contexts [15]—ensuring interface compatibility is simpler assuming round-robin.

the length of each thread's stem varies. Our translation computes for each thread a stem long enough (at most $k_1 + k_2 - 1$ contexts) to cover the stem of any thread. Since each thread's repeating sequence may begin as soon as the k_1st context, the stem and lasso computation may overlap. Our translation maintains the invariant that the `Stem` (resp., `Lasso`) values are active exactly when `k1` $\neq \perp$ (resp., `k2` $\neq \perp$). Reads and writes to shared variables (on Lines 47–57) read and write to both `Stem` and `Lasso` as they are active.

Our translation also adds code at every potential context switch point (Lines 71–96). Initially, the context counters `k1` and `k2` are incremented nondeterministically and synchronously (the block starting at Line 74). Then, at Line 78, we check whether the thread's keyframe has been encountered for the first time, and if so make a snapshot of the local valuation and program location, and activate the lasso; later along, at Line 90, we validate the snapshot when returning to the same keyframe (perhaps with a larger procedure stack). At some point in between, at Line 86, the stem becomes inactive. We ensure using the local variable `bottom` that the keyframe in which a thread begins repeating is never returned from.

We ensure strong fairness using an auxiliary vector of Boolean constants `waited`, one per lock $x \in$ `Locks`, indicating the set of locks which are held throughout the lasso. According to Section 3.4, we ensure each waited lock is held at the beginning of the lasso (Lines 28–30) and not released during the lasso (Line 68), and allow attempted acquires to abort (Line 63).

Lemma 5. *The program* $((P))^{k_1,k_2}$ *violates its assertion if P has a strongly-fair ultimately periodic round-robin execution with* $k_1 \in \mathbb{N}$ *and* $k_2 \in \mathbb{N}$*, resp., stem and lasso rounds; if* $((P))^{k_1,k_2}$ *violates its assertion then P has a strongly-fair ultimately periodic round-robin execution with* $k_1 + k_2$ *and* k_2*, resp., stem and lasso rounds.*

5 Experimental Evaluation

We have implemented our analysis, based on the code-to-code translation presented in Section 4. Our prototype tool, called MUTANT[3], takes as input a program written in the BOOGIE intermediate verification language [2]. Though normally a rich sequential language with recursive procedures, integers, maps, and algebraic datatypes, we have extended BOOGIE with thread-creation and atomic blocks, which we use to model shared-memory multithreaded programs with synchronization operations. Given a bound $K \in \mathbb{N}$ (where $K = k_1 + k_2$), MUTANT outputs an assertion-annotated sequential BOOGIE program. We feed the resulting program to our SMT-based bounded model checker CORRAL [14]. MUTANT has support for strong fairness, and does not falsely detect nonterminating executions in the program of Figure 1a, for instance.

As an initial example to demonstrate MUTANT's effectiveness, we consider a *try-lock* based algorithm for the *dining philosophers* problem. This program involves N locks and N threads, each of which executes the code shown in Figure 3a. Each philosopher tries to acquire two locks. `TryLock` is a non-blocking synchronization operation that returns **true** when the lock is successfully acquired, otherwise it returns **false**. If

[3] MUTANT stands for MUltiThreAded Non Termination.

```
  // translation of
  // var g: T
  var Stem[k1+k2-1]: T
  var Lasso[k2]: T
  var Local: T
  var Location: Locs
  var shift: ℕ^<k2 ∪ {⊥}
  const waited[Locks]: 𝔹
  var k1: ℕ ∪ {⊥}
  var k2: ℕ ∪ {⊥}

  proc Main ()
    const Stem0 := ⋆;
    const Lasso0 := ⋆;
    Stem := Stem0;
    Lasso := Lasso0;
    foreach t in Tids do
      shift := ⋆;
      k1 := 0;
      k2 := ⊥;
      call Main[t] ();
    assume
      Stem[0..k1+k2-3]
      = Stem0[1..k1+k2-2];
    assume
      Lasso[0..k2-2]
      = Lasso0[1..k2-1];
    assume ∀x ∈ Locks·
      x(Lasso[0])
      ⇔ waited[x];
    assert false;
    return

  // translation of
  // proc p (var l: T) s
  proc p (var l: T,
          bottom: 𝔹) s

  // translation of
  // call x := p e
  call x := p (e,⋆)

  // translation of
  // return e
  assume !bottom;
  return e

  // translation of shared
  // variable read x := g
  assume Stem[k1]
       = Lasso[k2];
  x := Stem[k1]
  x := Lasso[k2];

  // translation of shared
  // variable write g := e
  Stem[k1] := e;
  Lasso[k2] := e

  // translation of
  // acquire x
  if shift = ⊥ ∧ ⋆
     ∧ waited[x] then
     abort;
  acquire x

  // translation of
  // release x
  assume k2 ⇒ !waited[x];
  release x

  // translation of
  // (implicit) yield
  // at location 'loc'
  while ⋆ do
    k1 := k1 + 1;
    k2 := (k2+1) mod k2;

  if k1 = k1+shift
     ∧ k2 = ⊥ then
    // begin the lasso
    assume bottom;
    k2 := shift;
    Local := l;
    Location := loc;

  if k1 ≥ k1+k2-1 then
    // end the stem
    k1 := ⊥;

  if k2 = shift
     ∧ k1 = ⊥ then
    // end the lasso
    assume Local = l;
    assume Location = loc;
    // exit to main
    abort;
```

Fig. 2. The sequential translation $((P))^{k_1,k_2}$ of a multithreaded program P. We assume that statements which evaluate undefined expressions (i.e., using $\perp$ in arithmetic or array indexing) are simply skipped, and that no statement both reads and writes to g. The expression $\star$ nondeterministically evaluates to any well-typed value, and the **assume** e statement proceeds only when e evaluates to **true**. The **abort** statement discards the procedure stack and returns control to `Main`.

a philosopher acquires the left lock but is not able to acquire the right lock, then he releases the left lock and tries again. A philosopher terminates when he is able to acquire both locks (Line 10). This program has a fair non-terminating execution for each $N \geq 2$, namely where each philosopher first acquires their left lock, then upon seeing their right lock unavailable, they release their left lock. MUTANT is able to automatically detect this execution for each value of N with $K = 2$; we report running times in Figure 3d. Note that while this execution requires all N threads to participate, each thread only uses a fixed number of context switches in each period of the lasso. Though the state-space of the program grows exponentially with N, Figure 3d demonstrates that MUTANT scales sub-exponentially. Though the program has unfair non-terminating executions—e.g., where one philosopher acquires a lock and ceases to participate further, while the others continuously spin waiting to acquire both their locks—MUTANT correctly does not report any such unfair non-terminating executions.

```
1 // An array of N locks
2 var Lock[N]: mutex
3
4 proc Philosopher(n: int)
5    var left := Lock[n];
6    var right := Lock[(n+1)%N];
7    while true do
8       if TryLock(left)
9          if TryLock(right)
10            break
11         else
12            ReleaseLock(left);
13   ReleaseLock(right);
14   ReleaseLock(left);
15   return
```

(a)

```
1 proc Thread1()
2    var v1 := ⋆;
3    add(v1);
4    flag := false;
5    return
6
7 proc Thread2()
8    while flag do
9       var v2 := ⋆;
10      if ⋆ then
11         add(v2)
12      else
13         remove(v2);
14   return
```

(b)

```
1 while e1 do
2    timeout := false;
3    if ⋆ and e2 then
4       timeout := true;
5       break
```

(c)

N=2	3	4	5	6
2.1s	3.43s	6.13s	8.94s	21.91s
N=7	8	9	10	
15.79s	30.77s	31.66s	43.54s	

(d)

Fig. 3. (a) TryLock based dining phisophers. (b) A concurrent client operating on an OptimisticList. (c) Modeling timeout. (d) Running time of MUTANT on the dining philosophers example. As our verifier is based on the Z3 SMT solver, running times may increase non-uniformly with N due to Z3's internal heuristics, which may vary widely across different instances.

As a second example we consider the concurrent `OptimisticList` algorithm from Section 9.6 of Herlihy and Shavit [9], supporting concurrent insertions and deletions on sorted lists using optimistic concurrency control. Our BOOGIE encoding spans roughly 250 lines. In order to determine whether each operation is guaranteed to terminate in the presence of an environment performing arbitrary list operations, we wrote the two-thread driver of Figure 3b. While the first thread tries to insert an element, the second thread continuously fires `add` and `remove` operations with arbitrary arguments. The shared variable `flag` ensures that the second thread terminates when the first thread does. Though not shown, the driver also initializes the list with a few arbitrary elements.

This program has the following fair non-terminating execution, similar in spirit to that in Figure 1c: first, the `add` operation of `Thread1` selects a position in the (sorted) list where to insert a value v_1, say between consecutive nodes with values a and b (i.e., such that $a < v_1 < b$). Then the second thread picks a value v_2, such that $a < v_2 < b$, and inserts. When the first thread then sees that list has been modified at the position it was about to insert, it retries the `add` operation. Meanwhile, the second thread fires a `remove` operation and deletes v_2. This program then reencounters the initial configuration, and the `add` operation has not succeeded. MUTANT finds this execution with three contexts per thread in 44 seconds. Interesting to note is that even though this program may use infinite-domain data values, there remains nevertheless an execution that loops back exactly to the configuration. One slightly tricky aspect of this example is modeling memory allocation: because the second thread allocates and removes a list node in each period, we must explicitly free the removed node in order to reencounter the same configuration at the end of the lasso. As future work, using a more abstract notion of heap equality could simplify this aspect.

As a third example we consider a algorithm developed by our colleagues [20] that enables programmers to write assertions which are checked *continuously and concurrently* with the actual program, in similar spirit to *asynchronous assertions* [1]. One salient

feature of this algorithm is that it is non-blocking, i.e., the evaluation of the asserted expressions does not block other threads from making progress. We coded the algorithm, and two variations with possible non-termination bugs, in roughly 230 lines of BOOGIE code. In each of the potentially-buggy variations we found a non-terminating execution where incorrect assertion evaluations led to livelock. To our surprise, we also found a non-terminating execution in our supposedly-correct variation. After consulting with the developers, the problem turned out to be in our modeling. To understand the problem, consider the code in Fig. 3c. MUTANT detected non-termination by skipping the **then**-branch in each iteration of the lasso. (The actual non-termination found by MUTANT required concurrent reasoning, even though the lasso only involved one thread.) However, the intention of the designers was that this branch represents an actual time out reflecting a timer running down to zero. We corrected this modeling by ensuring that the above choice must evaluate to **true** at least once within the lasso. This is similar to enforcing Condition 1, Case 2 of strong fairness in Section 3. Nonetheless, MUTANT's output is still valuable: it says that if the time out is not implemented correctly, then the program may enter a livelock.

MUTANT is able to determine the absence of periodic nontermination bugs in the corrected variation with up to 3 contexts per thread in 402 seconds. MUTANT also detects nonterminating executions in the three buggy variations in 11, 21, and 36 seconds. These experiments demonstrate that MUTANT is effective on real-world algorithms.

6 Related Work

Our work follows the line of research on compositional reductions from concurrent to sequential programs. The initial so-called "sequentialization" [22] explored multithreaded programs up to one context-switch between threads. Following Qadeer and Rehof [21]'s generalization of context-bounding to an arbitrary number of context switches, Lal and Reps [15] later proposed a sequentialization to handle a parameterized amount of context-switches between a statically-determined set of threads executing in round-robin order. La Torre et al. [12] extended the approach to handle programs parameterized by an unbounded number of statically-determined threads, and shortly after, Emmi et al. [6] further extended these results to handle an unbounded amount of dynamically-created tasks. Bouajjani et al. [3] pushed these results even further to a sequentialization which attempts to explore as many behaviors as possible within a given analysis budget. The compositional analyses resulting from each of these sequentializations however only consider finite executions, and are thus incapable of establishing liveness properties.

Although much previous work has been done for proving termination and detecting non-termination in sequential programs—for instance, Cook et al. [4] discover ranking functions to prove termination of sequential programs, and Gupta et al. [8] use concolic execution to detect non-terminating executions in sequential programs—relatively little attention has been paid to multithreaded programs, where interesting non-terminating executions often have little to do with possible divergence of data values. Though Cook et al. [5] have extended TERMINATOR to multithreaded programs, their analysis is oriented to proving termination; failure to prove termination does not generally indicate the

existence of a non-terminating execution. More recently Popeea and Rybalchenko [19] have developed compositional techniques to prove termination in multithreaded programs, though again, their approach does not certify the existence of non-terminating executions. Because both of these techniques focus on establishing a proof of termination, they necessarily consider over-approximations of concurrent programs, whereas our technique looks at an under-approximation to find counterexamples faster.

Musuvathi and Qadeer [18] consider liveness properties in multithreaded programs, but their approach is based on systematic testing, and thus behavioral coverage is limited by test harnesses and concrete input values. Moreover, their approach is stateless (i.e., they never store states during the execution of the program), hence they can only detect possible non-termination by identifying lengthy executions.

In the most closely related work of which we are aware, Morse et al. [16] propose a compositional LTL model checking technique for multithreaded programs based on context-bounding. As far as we can tell, their technique (a) does not ensure non-terminating executions are fair, (b) does not consider lassos in which multiple recursive threads interfere, and (c) requires very high context-bounds to capture synchronized interaction between the program and a monitor Büchi automaton.

7 Conclusion

We have developed a compositional algorithm for detecting fair ultimately periodic executions in recursive multithreaded programs by bounding the number of context-switches in each repeating period. Our approach reveals a simple-to-implement code-to-code translation, which reduces the problem to finding assertion violations in recursive sequential programs; consequently we leverage existing sequential analysis algorithms.

Our approach can be used to encode other linear temporal logic conditions besides non-termination, e.g., response properties. Though for specific classes of formulae/properties efficient encodings are possible, a sequentialization parameterized by arbitrary linear temporal logic formulae must essentially construct the product of the input program with an arbitrary Büchi automaton; the encoding of this (synchronous) product as a sequential program may not be as succinct.

In this work, discovering ultimately periodic executions is done by detecting repeated state valuations. This notion of repeatability is complete for programs manipulating finite data, but is not complete in general. Still, this notion is actually relevant in many practical cases, since non-termination bugs in concurrent programs are often due to non-state-changing retry mechanisms. In the case of infinite data domains periodic executions may exhibit, for instance, ever increasing counter values; there a notion of repeatability more relaxed than state-equality may be necessary. This notion however, contrary to the one we consider here, would have to account for the actions encountered during the lasso. Ensuring repeatability may be complex to define and check, depending on the data domains and the nature of program operations.

References

[1] Aftandilian, E., Guyer, S.Z., Vechev, M.T., Yahav, E.: Asynchronous assertions. In: OOPSLA 2011: Proc. 26th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications, pp. 275–288. ACM (2011)

[2] Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
[3] Bouajjani, A., Emmi, M., Parlato, G.: On Sequentializing Concurrent Programs. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 129–145. Springer, Heidelberg (2011)
[4] Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI 2006: Proc. ACM SIGPLAN 2006 Conf. on Programming Language Design and Implementation, pp. 415–426. ACM (2006)
[5] Cook, B., Podelski, A., Rybalchenko, A.: Proving thread termination. In: PLDI 2007: Proc. ACM SIGPLAN 2007 Conf. on Programming Language Design and Implementation, pp. 320–330. ACM (2007)
[6] Emmi, M., Qadeer, S., Rakamarić, Z.: Delay-bounded scheduling. In: POPL 2011: Proc. 38th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, pp. 411–422. ACM (2011)
[7] Garg, P., Madhusudan, P.: Compositionality Entails Sequentializability. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 26–40. Springer, Heidelberg (2011)
[8] Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.G.: Proving non-termination. In: POPL 2008: Proc. 35th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, pp. 147–158. ACM (2008)
[9] Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (2008)
[10] Kozen, D.: Lower bounds for natural proof systems. In: FOCS 1977: Proc. 18th Annual Symp. on Foundations of Computer Science, pp. 254–266. IEEE Computer Society (1977)
[11] La Torre, S., Madhusudan, P., Parlato, G.: Reducing Context-Bounded Concurrent Reachability to Sequential Reachability. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 477–492. Springer, Heidelberg (2009)
[12] La Torre, S., Madhusudan, P., Parlato, G.: Model-Checking Parameterized Concurrent Programs Using Linear Interfaces. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 629–644. Springer, Heidelberg (2010)
[13] Lahiri, S.K., Qadeer, S., Rakamarić, Z.: Static and Precise Detection of Concurrency Errors in Systems Code Using SMT Solvers. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 509–524. Springer, Heidelberg (2009)
[14] Lal, A., Qadeer, S., Lahiri, S.: Corral: A Solver for Reachability Modulo Theories. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 427–443. Springer, Heidelberg (2012)
[15] Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. Formal Methods in System Design 35(1), 73–97 (2009)
[16] Morse, J., Cordeiro, L., Nicole, D., Fischer, B.: Context-Bounded Model Checking of LTL Properties for ANSI-C Software. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 302–317. Springer, Heidelberg (2011)
[17] Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI 2007: Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, pp. 446–455. ACM (2007)
[18] Musuvathi, M., Qadeer, S.: Fair stateless model checking. In: PLDI 2008: Proc. ACM SIGPLAN 2008 Conf. on Programming Language Design and Implementation, pp. 362–371. ACM (2008)
[19] Popeea, C., Rybalchenko, A.: Compositional Termination Proofs for Multi-threaded Programs. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 237–251. Springer, Heidelberg (2012)

[20] Qadeer, S., Musuvathi, M., Burnim, J.: Personal communication (January 2012)
[21] Qadeer, S., Rehof, J.: Context-Bounded Model Checking of Concurrent Software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
[22] Qadeer, S., Wu, D.: KISS: Keep it simple and sequential. In: PLDI 2004: Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, pp. 14–24. ACM (2004)
[23] Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM Trans. Program. Lang. Syst. 22(2), 416–430 (2000)
[24] Yen, H.C.: Communicating processes, scheduling, and the complexity of nondeterminism. Mathematical Systems Theory 23(1), 33–59 (1990)