# Finding Non-terminating Executions in Distributed Asynchronous Programs

Michael Emmi[1,⋆] and Akash Lal[2]

[1] LIAFA, Université Paris Diderot
mje@liafa.univ-paris-diderot.fr
[2] Microsoft Research India
akashl@microsoft.com

**Abstract.** Programming distributed and reactive asynchronous systems is complex due to the lack of synchronization between concurrently executing tasks, and arbitrary delay of message-based communication. As even simple programming mistakes have the capability to introduce divergent behavior, a key liveness property is *eventual quiescence*: for any finite number of external stimuli (e.g., client-generated events), only a finite number of internal messages are ever created.

In this work we propose a practical three-step reduction-based approach for detecting divergent executions in asynchronous programs. As a first step, we give a code-to-code translation reducing divergence of an asynchronous program $P$ to completed state-reachability—i.e., reachability to a given state with no pending asynchronous tasks—of a polynomially-sized asynchronous program $P'$. In the second step, we give a code-to-code translation under-approximating completed state-reachability of $P'$ by state-reachability of a polynomially-sized recursive sequential program $P''(K)$, for the given analysis parameter $K \in \mathbb{N}$. Following Emmi et al. [8]'s delay-bounding approach, $P''(K)$ encodes a subset of $P'$'s, and thus of $P$'s, behaviors by limiting scheduling nondeterminism. As $K$ is increased, more possibly divergent behaviors of $P$ are considered, and in the limit as $K$ approaches infinity, our reduction is complete for programs with finite data domains. As the final step we give the resulting state-reachability query to an off-the-shelf SMT-based sequential program verification tool.

We demonstrate the feasibility of our approach by implementing a prototype analysis tool called Alive, which detects divergent executions in several hand-coded variations of textbook distributed algorithms. As far as we are aware, our easy-to-implement prototype is the first tool which automatically detects divergence for distributed and reactive asynchronous programs.

## 1 Introduction

The ever-increasing popularity of online commercial and social networks, along with proliferating mobile computing devices, promises to make distributed software an even more pervasive component of technological infrastructure. In a

⋆ Supported by a Fondation Sciences Mathématiques de Paris post-doctoral fellowship.

A. Miné and D. Schmidt (Eds.): SAS 2012, LNCS 7460, pp. 439–455, 2012.

distributed program a network of physically separated asynchronous processors coordinate by sending and asynchronously receiving messages. Such systems are challenging to implement because of several uncertainties, including processor timings, message delays, and processor failures. Although simplifying mechanisms such as *synchronizers* and shared-memory simulation do exist [16], they add significant runtime overhead which can be unacceptable in many situations.

Because of the inherit complexity in distributed asynchronous programming, even subtle design and programming mistakes have the capability to introduce erroneous or divergent behaviors, against which the usual reliability measures are much less effective. The great amount of nondeterminism in processor timings and message delays tends to make errors elusive and hard to reproduce in simulation and testing. The combinatorial explosion incurred by the vast number of processor interleavings and message-buffer contents tends to make formal verification techniques intractable. Though many distributed algorithms are proposed along with manual correctness proofs, key properties such as *eventual quiescence*—i.e., for any number of external stimuli such as client-generated events, only a finite number of internal network messages are ever created—remain difficult to ensure with automatic techniques. Practically speaking, such properties ensure the eventual construction of network spanning trees [16], the eventual election of network leaders [20], and the eventual acceptance of network peer proposals, e.g., according to the Paxos protocol [15].

In this work we develop an automatic technique to detect violations to eventual quiescence, i.e., executions of distributed systems for which a finite number of external stimuli result in an infinite number of internal messages. Our reduction-based approach works in three steps. First, we reduce the problem of finding nonterminating executions of a given (distributed) asynchronous program $P$ to the problem of computing reachability in a polynomially-sized (distributed) asynchronous program $P'$. This reduction is complete for programs with finite data domains, in the sense that an answer to the reachability query on $P'$ is a precise answer to the nontermination query on $P$. In the second step, we reduce reachability in $P'$ to reachability in a polynomially-sized recursive sequential program $P''$—without explicitly encoding the concurrent behavior of $P'$ as data in $P''$. This step is parameterized by an integer $K \in \mathbb{N}$; for small $K$, $P''$ encodes few concurrent schedules of $P'$; as $K$ is increased, $P''$ encodes and increasing number of concurrent reorderings, and in the limit as $K$ approaches infinity, $P''$ codes all possible behaviors of $P'$—and thus $P$. Finally, using existing sequential program verification tools, we check reachability in $P''$: a positive result indicates a nonterminating execution in $P$, though the lack of nonterminating executions in $P$ can only be concluded in the limit as $K$ approaches infinity. Our technique supports *fairness*, in that we may consider only infinite executions in which no message is ignored forever.

We demonstrate the feasibility of our reduction-based approach by implementing a prototype analysis tool called Alive, which detects violations to eventual quiescence in several hand-coded variations to textbook distributed algorithms [16]. Our relatively easy-to-implement prototype leverages existing

SMT-based program verification tools [14], and as far as we are aware, is the first tool which can automatically detect divergence in distributed asynchronous programs.

To begin in Section 2, we introduce a program model of distributed computation. In Section 3 we describe our reduction to sequential program analysis, and provide code-to-code translations which succinctly encode the reduction. Following in Section 4 we describe our experimental results in analyzing textbook distributed algorithms, and we conclude by discussing related work in Section 5.

## 2 Distributed Asynchronous Programs

We consider a distributed message-passing program model in which each processor is equipped with a procedure stack and an unordered buffer of pending messages. Initially all processors are idle. When an idle processor's message buffer is non-empty, some message is removed, and a message-dependent *task* is executed to completion. Each task executes essentially as a recursive sequential program, which besides accessing its own processor's global storage, can *post* messages to the buffers of any processor, including its own. When a task does complete, its processor again becomes idle, chooses a next pending message to remove, and so on. The distinction between messages and handling tasks is purely aesthetic, and we unify the two by supposing each message is a procedure-and-argument pair. Though in principle many message-passing systems, e.g., in Erlang and Scala, allow reading additional messages at any program point, we have observed that common practice is to read messages only upon completing a prior task [21].

Our choice to model message-passing programs with *unordered* buffers has two important consequences. First, although some programming models do not ensure messages are received in the order they are sent, others do; our unordered buffer model should be seen as an abstraction of a model with faithful message queues, since ignoring message order allows behaviors infeasible in the queue-ordered model. Second, when message order is ignored, distributed executions are *task-serializable*—i.e., equivalent to executions where the tasks across all processors execute serially, one after the other. Intuitively this is true because (a) tasks of different processors access disjoint memory, and (b) message posting operations commute with each other. (Message posting operations do not commute when buffers are ordered.) To simulate a distributed system with a single processor we combine each processor's global storage, and ensure each processor's tasks access only their processor-indexed storage. Since serializability implies that single processor systems precisely simulate the behavior of distributed systems, we limit our discussion, without loss of generality, to single-processor asynchronous programs [19].

### 2.1 Program Syntax

Let $\mathsf{Procs}$ be a set of procedure names, $\mathsf{Vals}$ a set of values, $\mathsf{Exprs}$ a set of expressions, $\mathsf{Pids}$ a set of processor identifiers, and let $T$ be a type. Figure 1 gives

$$\begin{array}{rcl}
P & ::= & \textbf{var}\ \texttt{g}{:}T\ (\textbf{proc}\ p\ (\textbf{var}\ \texttt{l}{:}T)\ s)^* \\
s & ::= & s\texttt{;}\ s \ \mid\ \textbf{skip} \ \mid\ x\ \texttt{:=}\ e \\
 & \mid & \textbf{assume}\ e \\
 & \mid & \textbf{if}\ e\ \textbf{then}\ s\ \textbf{else}\ s \\
 & \mid & \textbf{while}\ e\ \textbf{do}\ s \\
 & \mid & \textbf{call}\ x\ \texttt{:=}\ p\ e \\
 & \mid & \textbf{return}\ e \\
 & \mid & \textbf{post}\ p\ e \\
x & ::= & \texttt{g} \ \mid\ \texttt{l}
\end{array}$$

**Fig. 1.** The grammar of asynchronous message-passing programs $P$. Here $T$ is an unspecified type, and $e$ and $p$ range over expressions and procedure names.

$$\textsc{Dispatch}\quad \frac{}{\langle g, \varepsilon, m \cup \{f\} \rangle \rightarrow \langle g, f, m \rangle}$$

$$\textsc{Complete}\quad \frac{f = \langle \ell, \textbf{return}\ e\texttt{;}\ s \rangle}{\langle g, f, m \rangle \rightarrow \langle g, \varepsilon, m \rangle}$$

$$\textsc{Post}\quad \frac{s_1 = \textbf{post}\ p\ e\texttt{;}\ s_2 \qquad \ell_2 \in e(g, \ell_1) \qquad f = \langle \ell_2, s_p \rangle}{\langle g, \langle \ell_1, s_1 \rangle\, w, m \rangle \rightarrow \langle g, \langle \ell_1, s_2 \rangle\, w, m \cup \{f\} \rangle}$$

**Fig. 2.** The transition relation $\rightarrow$ of asynchronous message-passing programs

the grammar of *asynchronous message-passing programs.* We intentionally leave the syntax of expressions $e$ unspecified, though we do insist $\mathsf{Vals}$ contains **true** and **false**, and $\mathsf{Exprs}$ contains $\mathsf{Vals}$ and the *(nullary) choice operator* $\star$. We say a program is *finite-data* when $\mathsf{Vals}$ is finite.

Each program $P$ declares a single global variable $\mathsf{g}$ and a procedure sequence, each $p \in \mathsf{Procs}$ having a single parameter $\texttt{l}$ and top-level statement denoted $s_p$; as statements are built inductively by composition with control-flow statements, $s_p$ describes the entire body of $p$. The set of program statements $s$ is denoted $\mathsf{Stmts}$. Intuitively, a **post** $p$ $e$ statement is an asynchronous call to a procedure $p$ with argument $e$. The **assume** $e$ statement proceeds only when $e$ evaluates to **true**, and this statement plays a role in disqualifying executions in our subsequent reductions of Section 3. The programming language we consider is simple, yet very expressive, since the syntax of types and expressions is left free, and we lose no generality by considering only single global and local variables.

## 2.2 Program Semantics

A *(procedure) frame* $f = \langle \ell, s \rangle$ is a current valuation $\ell \in \mathsf{Vals}$ to the procedure-local variable $\texttt{l}$, along with a statement $s \in \mathsf{Stmts}$ to be executed. (Here $s$ describes the entire body of a procedure $p$ that remains to be executed, and is initially set to $p$'s top-level statement $s_p$; we refer to initial procedure frames $t = \langle \ell, s_p \rangle$ as *tasks*, to distinguish the frames that populate task buffers.) The set of all frames is denoted $\mathsf{Frames}$. A *configuration* $c = \langle g, w, m \rangle$ is a current valuation $g \in \mathsf{Vals}$ to the processor-global variable $\mathsf{g}$, along with a procedure-frame stack $w \in \mathsf{Frames}^*$ and a multiset $m \in \mathbb{M}[\mathsf{Frames}]$ representing the pending-tasks buffer. The configuration $c$ is called *idle* when $w = \varepsilon$, and *completed* when $w = \varepsilon$ and $m = \emptyset$. The set of configurations is denoted $\mathsf{Configs}$.

Figure 2 defines the transition relation $\rightarrow$ for the asynchronous behavior. (The transitions for the sequential statements are standard.) The $\textsc{Post}$ rule creates a new frame to execute the given procedure, and places the new frame in the pending-tasks buffer. The COMPLETE rule returns from the final frame of a task, rendering the processor idle, and the DISPATCH rule schedules a pending task on the idle processor.

An *execution* of a program $P$ (from $c_0$) is a configuration sequence $\xi = c_0 c_1 \dots$ such that $c_i \to c_{i+1}$ for $i \geq 0$; we say each configuration $c_i$ is *reachable* from $c_0$. An *initial condition* $\iota = \langle g_0, \ell_0, p_0 \rangle$ is a global-variable valuation $g_0 \in \mathsf{Vals}$, along with a local-variable valuation $\ell_0 \in \mathsf{Vals}$, and a procedure $p_0 \in \mathsf{Procs}$. A configuration $c = \langle g_0, \langle \ell_0, s_{p_0} \rangle, \emptyset \rangle$ of a program $P$ is called $\langle g_0, \ell_0, p_0 \rangle$*-initial.* An execution $\xi = c_0 c_1 \dots$ is called *infinitely-often idle* when there exists an infinite set $I \subseteq \mathbb{N}$ such that for each $i \in I$, $c_i$ is idle.

**Definition 1 (state-reachability).** *The* (completed) state-reachability problem *is to determine for an initial condition $\iota$, global valuation $g$, and program $P$, whether there exists a (completed) $g$-valued configuration reachable in $P$ from $\iota$.*

In this work we are interested in detecting non-terminating executions due to asynchrony, rather than the orthogonal problem of detecting whether each individual task may alone terminate. Our notion of non-termination thus considers only executions which return to idle configurations infinitely-often.

**Definition 2 (non-termination).** *The* non-termination problem *is to determine for an initial condition $\iota$ and a program $P$, whether there exists an infinitely-often idle execution of $P$ from some $\iota$-initial configuration.*

## 3 Detecting Non-termination

Though precise algorithms for detecting (fair) non-termination in finite-data asynchronous programs are known (see Ganty and Majumdar [10]), the fair non-termination problem is polynomial-time equivalent to reachability in Petri nets, which is an EXPSPACE-hard problem for which only non-primitive recursive algorithms are known. Though worst-case complexity is not necessarily an indication of feasibility on practically-occurring instances, here we are interested in leveraging existing tools designed for more tractable problems whose solutions can be used to incrementally under-approximate non-termination detection; i.e., where for a given analysis parameter $k \in \mathbb{N}$ we can efficiently detect non-termination from an interesting subset $B_k$ of program behaviors.

Our strategy is to reduce the problem of detecting non-terminating executions in asynchronous programs to that of completed state-reachability in asynchronous programs. We perform this step using the code-to-code translation of Section 3.1, and in Section 3.2 we consider extensions to handle fairness. Then, in the second step of Section 3.3, we apply an incrementally underapproximating reduction from state-reachability in asynchronous programs to state-reachability in sequential program [8, 4], and discharge the resulting program analysis problem using existing sequential analysis tools.

### 3.1 Reduction from Non-termination to Reachability

In the first step of our reduction, we use the fact that every infinite execution eventually passes through two configurations $c_1$, and then $c_2$, such that every possible execution from $c_1$ is also possible from $c_2$; e.g., when $c_1$ and $c_2$ are idle configurations with the same global valuation in which all tasks pending at $c_1$ are also pending at $c_2$. Formally, given two configurations $c_1 = \langle g_1, w_1, m_1 \rangle$ and $c_2 = \langle g_2, w_2, m_2 \rangle$ we define the order $c_1 \preceq c_2$ to hold when $g_1 = g_2$, $w_1 = w_2$, and $m_1 \subseteq m_2$.[1] An execution $c_0 c_1 \dots$ is called *periodic* when $c_i \preceq c_j$ for two idle configurations $c_i$ and $c_j$ such that $i < j$.[2] The following lemma essentially exploits the fact that $\preceq$ is a well-quasi-ordering on idle configurations.

**Lemma 1.** *A finite-data program $P$ has an infinitely-often idle execution from $\iota$ if and only if $P$ has a periodic execution from $\iota$.*

*Proof.* Suppose $c_0 c_1 \dots$ is the sequence of idle configurations in an infinitely-often idle execution $\xi$. As the subset order $\subseteq$ on multisets is a well-quasi order, and the domain $\mathsf{Vals}$ of global variables is finite, $\preceq$ is a well-quasi order on idle configurations. Thus there exists $i < j$ such that $c_i \preceq c_j$, so $\xi$ is also periodic.

Supposing $\xi = c_0 c_1 \dots$ is a periodic execution from $\iota$, there exists idle configurations $c_i$ and $c_j$ of $\xi$ such that $i < j$ and $c_i \preceq c_j$; let $c_i = \langle g_i, \varepsilon, m_i \rangle$ and $c_j = \langle g_j, \varepsilon, m_j \rangle$. Since $g_i = g_j$ and $m_i \subseteq m_j$, by definition of $\preceq$, the sequence of execution steps between $c_i$ and $c_j$ is also enabled from configuration $c_j$—we may simply ignore the extra tasks $m_j \setminus m_i$ pending in $c_j$. For any $k, l \in \mathbb{N}$ and task buffer $m \in \mathbb{M}[\mathsf{Frames}]$ such that $k < l < |\xi|$, let $\xi_{k,l}^m$ be the sequence of configurations $c_k c_{k+1} \dots c_{l-1}$ of $\xi$, each with additional pending tasks $m$. Furthermore, let $k \cdot m$ be the multiset union of $k$ copies of $m$. Letting $m = m_j \setminus m_i$, then $\xi_{0,i} \xi_{i,j} \xi_{i,j}^m \xi_{i,j}^{2m} \xi_{i,j}^{3m} \dots$ is an infinitely-often idle execution from $\iota$ which periodically repeats the same transitions used to construct $\xi$ between $c_i$ and $c_j$.

We reduce the detection of periodic executions to completed state reachability in asynchronous programs. Essentially, such a reduction must determine multiset inclusion between the unbounded task buffers at two idle configurations; i.e., for some idle configuration $c_i = \langle g_i, \varepsilon, m_i \rangle$ reachable in an execution $c_0 c_1 \dots$, there exists $j > i$ such that $c_j = \langle g_j, \varepsilon, m_j \rangle$ with $g_i = g_j$ and $m_i \subseteq m_j$. As the set $m_i$ of pending tasks at $c_i$ is unbounded, any reduction cannot hope to store arbitrary $m_i$ for later comparison with $m_j$ using finite-domain program variables.

Our reduction determines the correspondence between unbounded task buffers in the source program using only finite-domain program variables by leveraging the task buffers of the target program. For each instance of a task $t$ which is pending in $c_i$, we post an additional task $\mathtt{pro}(t)$ when $t$ is posted; for each task $t$ pending in $c_j$, we either post an additional task $\mathtt{anti}(t)$ instead of $t$, or we post nothing, to handle the case where $t$ is never dispatched. We then check that for each executed $\mathtt{pro}(t)$ a matching $\mathtt{anti}(t)$ is also executed, and that

[1] Here $\subseteq$ is the multiset subset relation.

[2] As our definition of $\preceq$ only relates configurations with equal global valuations, our notion of periodic is only complete for finite-data programs.

```
// translation of var g: T
var repeated: 𝔹
var turn: 𝔹
var last: Procs × Vals
var G[𝔹]: T

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

// translation of call x := p e
call x := p (e,period)
// translation of g
G[period]

// additional procedures
proc pro(var t: Procs × Vals)
    assume turn;
    last := t;
    turn := false;
    return
proc anti(var t: Procs × Vals)
    assume !turn ∧ last = t;
    turn := true;
    return
// translation of post p e
if ⋆ then
    assume !period;
    post pro (p,e);
    post p (e,true);
    repeated := true
else if ⋆ then
    assume period;
    post anti (p,e)
else if ⋆ then
    skip
else
    post p (e,period)
```

**Fig. 3.** The translation $(\!(P)\!)_{\mathrm{nt}}$ of an asynchronous program $P$

at some point no $\texttt{pro}(t)$ nor $\texttt{anti}(t)$ tasks are pending. By considering executions which alternate between tasks of $\{\texttt{pro}(t) : t \in m_i\}$ and $\{\texttt{anti}(t) : t \in m'_j\}$—where $m'_j \subseteq m_j$ such that $m_j \setminus m'_j$ correspond to the dropped tasks—we can ensure each instance of an $m_i$ task has a corresponding instance in $m_j$, storing only the last encountered $\texttt{pro}(t)$ task, for $t \in m_i$.

Figure 3 lists our code-to-code translation $(\!(P)\!)_{\mathrm{nt}}$ reducing non-termination in an asynchronous program $P$ to completed state reachability in the asynchronous program $(\!(P)\!)_{\mathrm{nt}}$. Besides the auxiliary variable `last` used to store the last encountered $\texttt{pro}(t)$ task, for $t \in m_i$, we introduce Boolean variables `repeated`, to signal whether $m_i$ is non-empty, and `turn`, to signal whether an $\texttt{anti}(t)$ task has been executed since the last executed $\texttt{pro}(t)$ task. We also divide the execution of tasks into two phases by introducing a task-local Boolean variable `period`. The first phase (`!period`) corresponds to the execution $c_0 c_1 \ldots c_i$, while the second phase (`period`) corresponds to $c_{i+1} c_{i+2} \ldots c_j$. Initially pending tasks occur in the first non-`period` phase. Then each time a new task $t$ is posted, a nondeterministic choice is made for whether $t$ will execute in the non-`period` phase, in the `period` phase, or never.

Finally, to determine which finite asynchronous executions prove the existence of infinite asynchronous executions, we define the predicate $\varphi_{\mathrm{nt}}$ over initial conditions $\iota$ and configuration $c$ as

$$\varphi_{\mathrm{nt}}(\iota, c) \overset{\mathrm{def}}{=} \begin{cases} \textbf{true} & \text{when } \neg\texttt{repeated}(\iota) \text{ and } \texttt{turn}(\iota) \\ & \text{and } \texttt{repeated}(c) \text{ and } \texttt{turn}(c) \\ & \text{and } \texttt{G[0]}(c) = \texttt{G[1]}(\iota) = \texttt{G[1]}(c) \\ \textbf{false} & \text{otherwise,} \end{cases}$$

along with the mapping $\vartheta_{\mathrm{nt}}$ which projects the initial conditions of $(\!(P)\!)_{\mathrm{nt}}$ to those of $P$, as $\vartheta_{\mathrm{nt}}(\langle g, \ell, p\rangle) \overset{\mathrm{def}}{=} \langle g', \ell', p'\rangle$ when $\texttt{g}(g') = \texttt{G[0]}(g)$, $\texttt{l}(\ell') = \texttt{l}(\ell)$, and $p' = p$. Essentially, in any completed configuration $c$ reachable from $\iota$ satisfying $\varphi_{\mathrm{nt}}(\iota, c)$, we know that some task has executed during the period (since `repeated` evaluates to **true**), and that for each task pending at the beginning of the period, an identical task is pending at the end of the period (since `turn` evaluates to true, and there are no pending tasks in $c$). Finally, the conditions on the global variable `G` ensure that the beginning and end of each period reach the same global valuation.

**Lemma 2.** *A finite-data program $P$ has an infinitely-often idle execution from $\iota_0$ if and only if a completed configuration $c$ is reachable in $(\!(P)\!)_{nt}$ from some $\iota$ such that $\varphi_{\mathrm{nt}}(\iota, c) = \mathbf{true}$ and $\vartheta_{\mathrm{nt}}(\iota) = \iota_0$.*

*Proof.* For the forward direction, by Lemma 1, $P$ also has a periodic execution $\xi = \xi_{0,i}\xi_{i,j}\xi_{j,\omega}$ from $\iota_0$—where $\xi_{k,l} \overset{\mathrm{def}}{=} c_k c_{k+1} \ldots c_{l-1}$ for $k < l < |\xi|$—and $c_i \preceq c_j$ for idle configurations $c_i = \langle g, \varepsilon, m_1 \rangle$ and $c_j = \langle g, \varepsilon, m_2 \rangle$. We build an execution $\xi' = \xi'_{0,i}\xi'_{i,j}\xi_{\mathrm{match}}$ of $(\!(P)\!)_{\mathrm{nt}}$ such that

- the configurations $c'_k$ of $\xi'_{0,i}$ correspond to configurations $c_k$ of $\xi_{0,i}$, with $\mathtt{g}(c_k) = \mathtt{G[0]}(c'_k)$, $\mathtt{G[1]}(c'_k) = g$,
- the configurations $c'_k$ of $\xi'_{i,j}$ correspond to configurations $c_k$ of $\xi_{i,j}$, with $\mathtt{g}(c_k) = \mathtt{G[1]}(c'_k)$ and $\mathtt{G[0]}(c'_k) = g$,
- the pending tasks of each configuration $c'_k$ of $\xi'_{0,j}$, excluding `pro` and `anti` tasks, are contained within those of $c_k$,
- the local valuations of each configuration $c'_k$ of $\xi'_{0,i}$ (resp., of $\xi'_{i,j}$) match those of $c_k$, except `period` evaluates to 0 (resp., to 1) in every frame of $c'_k$, and
- the sequence $\xi_{\mathrm{match}}$ alternately executes `pro` and `anti` tasks such that each `pro`$(t)$ task is followed by a matching `anti`$(t)$ task.

It follows that we can construct such a $\xi'$ which reaches a completed configuration $c$ from some $\iota$ such that $\varphi_{\mathrm{nt}}(\iota, c)$, $\vartheta_{\mathrm{nt}}(\iota) = \iota_0$, and $\mathtt{G[0]}(c) = \mathtt{G[1]}(c) = g$.

For the backward direction, the reachability of a completed configuration $c$ of $(\!(P)\!)_{\mathrm{nt}}$ from $\iota$ such that $\varphi_{\mathrm{nt}}(\iota, c)$ implies that there exists a periodic execution $\xi = c_0 c_1 \ldots$ of $P$; in particular, there exist configurations $c_i \preceq c_j$ of $\xi$ with $i < j$, and which have the global valuations $\mathtt{g}(c_i) = \mathtt{g}(c_j) = \mathtt{G[0]}(c) = \mathtt{G[1]}(c)$ reached at the end of each period of $(\!(P)\!)_{\mathrm{nt}}$'s execution, and the set of pending tasks $m$ in $c_i$ are those second-period tasks posted by $(\!(P)\!)_{\mathrm{nt}}$ from first-period tasks. Since the set of tasks posted and pending by the end of the second period must contain $m$—otherwise unexecutable `pro` tasks would remain pending—we can construct a run where the pending tasks of $c_j$ contain the pending tasks of $c_i$, and so $P$ has a periodic execution. By Lemma 1 we conclude that $P$ also has an infinitely-often idle execution.

### 3.2 Ensuring Scheduling Fairness

In many classes of asynchronous systems there are (at least) two sensible notions of scheduling fairness against which to determine liveness properties: an infinite execution is called *strongly-fair* if every infinitely-often enabled transition is fired infinitely often, and *weakly-fair* if every infinitely-often *continuously* enabled transition is fired infinitely often. In our setting where asynchronous tasks execute serially from a task buffer, weak fairness becomes irrelevant; while one task executes no other transitions are enabled, and when idle (i.e., while no tasks are executing), all pending tasks become enabled. Furthermore once a task is posted, it becomes pending, and it is thus enabled in all subsequent idle configurations until dispatched. We thus define fairness according to what is normally referred to as strong fairness: an execution is *fair* when each infinitely-often posted task is infinitely-often dispatched.

To extend our reduction so that only fair infinite executions are considered we make two alterations to the translation of Figure 3. First, on Line 36 we replace **skip** with **assume** `period`; this ensures participation of all tasks pending at the beginning of each period. Second, we add auxiliary state to ensure at least one instance of each task posted during the period is dispatched. This can be encoded in various ways; for instance, we can add two arrays `dropped` and `dispatched` of index type $\mathsf{Procs} \times \mathsf{Vals}$ and element type $\mathbb{B}$ that indicate whether each task has been dropped/dispatched during the period phase (i.e., where the local variable `period` evaluates to **true**). Initially `dropped[`$t$`] = dispatched[`$t$`] =` **false** for all $t \in \mathsf{Procs} \times \mathsf{Vals}$. Each time a post to task $t$ is dropped during the period phase (i.e., Line 36) we set `dropped[`$t$`]` to **true**, and each time task $t$ is executed during the period phase (i.e., Line 38 when `period` is **true**) we set `dispatched[`$t$`]` to **true**. (Note that we need not consider the non-post of $t$ on Line 34 as dropped, since $t$ is necessarily dispatched during the period phase; otherwise there would remain a pending `anti(`$t$`)` task.) Finally, we add to our reachability query the predicate $\forall t.$`dropped[`$t$`]`$\Rightarrow$`dispatched[`$t$`]`, thus ensuring that when all asynchronous tasks have completed the only dropped tasks have been dispatched during the period.

Alternatively, we may also encode this fairness check by posting auxiliary `dropped` and `dispatched` tasks to the task buffer, in place of using the `dropped` and `dispatched` arrays. Essentially for each task $t$ dropped during the period phase on Line 36 we add **post** `dropped(`$t$`)`, and for each task $t$ posted into the period phase we add **post** `dispatched(`$t$`)`. Then, using a single additional variable of type $\mathsf{Procs} \times \mathsf{Vals}$ we ensure that for every executed `dropped(`$t$`)` task some `dispatched(`$t$`)` task also executes; a single variable suffices for this check because we may consider only schedules where all `dropped(`$t$`)` and `dispatch(t)` tasks execute contiguously for each $t$.

### 3.3 Delay-Bounded Reachability

Following the reduction from (fair) nontermination, we are faced with a highly-complex problem: determining completed state-reachability in finite-data programs is polynomial-time equivalent to computing exact reachability in Petri nets (i.e., such that all places representing pending tasks are empty), or alternatively in vector addition systems (i.e., such that all vector components counting pending tasks are zero). Though these problems are known to be decidable, there is no known primitive-recursive upper complexity bound.

Rather than dealing with such difficult problems, our strategy is to consider only a restricted yet interesting set of actual program behaviors. Following Emmi et al. [8]'s delay-bounding scheme, we equip some deterministic task scheduler with the ability to deviate from its deterministic schedule only a bounded number of times (per task). As this development is very similar to Emmi et al. [8]'s,

```
// translation of var g: T
var g: T
var G[K]: T

// translation of
// proc p (var l: T) s
proc p (var l: T, k: K) s

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

// translation of post p e
let temp: T = g
and guess: T
and k': K in
    assume k ≤ k' < K;
    g := G[k'];
    G[k'] := guess;
    call p (e,k');
    assume g = guess;
    g := temp;
```

**Fig. 4.** The $K$ delay sequential translation $(\!(P)\!)_{\mathrm{db}}^K$ of an asynchronous program $P$

we refer the interested reader there. We recall in Figure 4 the essential delay-bounded asynchronous to sequential translation.

To determine which executions of the sequential program $(\!(P)\!)_{\mathrm{db}}^K$ prove the existence of a valid asynchronous execution, we define the predicate $\varphi_{\mathrm{db}}$ over initial conditions $\iota$ and configuration $c$ as

$$\varphi_{\mathrm{db}}(\iota, c) = \begin{cases} \textbf{true} & \text{when } \mathtt{G[0]}(\iota) = \mathtt{g}(c) \\ & \text{and } \forall i \in \mathbb{N}. 0 < i < K \Rightarrow \mathtt{G[}i\mathtt{]}(\iota) = \mathtt{G[}i-1\mathtt{]}(c) \\ \textbf{false} & \text{otherwise,} \end{cases}$$

along with the mapping $\vartheta_{\mathrm{db}}$ from initial conditions of $(\!(P)\!)_{\mathrm{db}}^K$ to those of $P$ as $\vartheta_{\mathrm{db}}(\langle g, \ell, p \rangle) \stackrel{\mathrm{def}}{=} \langle g', \ell', p' \rangle$ when $\mathtt{g}(g') = \mathtt{g}(g)$, $\mathtt{l}(\ell') = \mathtt{l}(\ell)$, and $p' = p$. Essentially, in any completed configuration $c$ reachable from $\iota$ satisfying $\varphi_{\mathrm{db}}(\iota, c)$, we know that the initially pending task returned with the shared global valuation $\mathtt{G[0]}(\iota)$ resumed by the first-round tasks, and that the last $(i-1)$ round task, for $0 < i < K$, returned with the shared global valuation $\mathtt{G[}i\mathtt{]}(c)$ resumed by the first $i$ round task. The following lemma follows from Emmi et al. [8].

**Lemma 3.** *A valuation $g$ is reachable in some completed configuration of a program $P$ from $\iota_0$ if some $g$-valued completed configuration $c$ is reachable in $(\!(P)\!)_{db}^K$ from some $\iota$, such that $\varphi_{\mathrm{db}}(\iota, c) = \textbf{true}$ and $\vartheta_{\mathrm{db}}(\iota) = \iota_0$, for some $K \in \mathbb{N}$.*

## 4 Experience

We have implemented a prototype analysis tool called Alive. Our tool takes as input distributed asynchronous programs written in a variation of the Boogie language [2] in which message posting is encoded with specially-annotated procedure calls. Given a possibly non-terminating input program $P$, Alive translates $P$ into another asynchronous program $P'$ (according to the translation of Sections 3.1 and 3.2) that may violate a particular assertion if and only if $P$ has a (fair) non-terminating execution. Then Alive passes $P'$ and a bounding parameter $K \in \mathbb{N}$ to our AsyncChecker delay-bounded asynchronous program analysis tool [9] which attempts to determine whether the assertion can be

| Example | bug? | $K$ | $N$ | time (s) |
|---|---|---|---|---|
| PingPong | √ | 1 | 5 | 5.32 |
| PingPong-mod2 | √ | 2 | 5 | 19.01 |
| PingPong-mod2-1md | × | 1 | 5 | 4.94 |
| PingPong-mod3 | √ | 3 | 5 | 86.61 |
| PingPong-mod3-1md | × | 2 | 5 | 23.53 |
| PingPong-mod3-2md | × | 1 | 5 | 4.66 |
| PingPongPung | √ | 2 | 5 | 111.92 |
| PingPongPung-1md | × | 1 | 5 | 19.87 |
| SpanningTree-bug | √ | 1 | 5 | 165.19 |
| SpanningTree-correct | × | 2 | 3 | 28.80 |
| Bfs-bug | √ | 1 | 5 | 286.95 |
| Bfs-correct | × | 2 | 3 | 32.15 |
| BellmanFord-bug | √ | 1 | 5 | 303.98 |
| BellmanFord-correct | × | 2 | 3 | 33.74 |

**Fig. 5.** Experimental results with Alive. Here $K$ indicates the delay-bound, and $N$ the recursion-depth bound.

```
// program PingPong
var x: bool;

proc Ping ()
    if ¬x then
        post Ping ();
        x := true;
    return

proc Pong ()
    if x then
        post Pong ();
        x := false;
    return

proc Main ()
    x := false;
    post Ping ();
    post Pong ();
    return
```

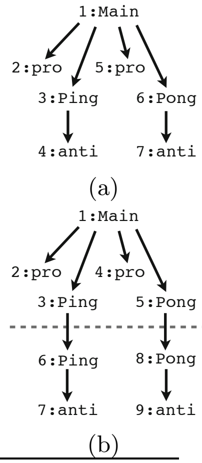

**Fig. 6.** The PingPong program, along with asynchronous executions of the translations $(\!(\texttt{PingPong})\!)_{\mathrm{nt}}$ (a) and $(\!(\texttt{PingPong-mod2})\!)_{\mathrm{nt}}$ (b). Task order is indicated by numeric prefixes; the dotted line indicates delaying.

violated (in an execution using at most $K$ delay operations, per task). Async-Checker essentially performs a variation of our delay-bounded translation of Section 3.3—which results in a sequential Boogie program—and hands the resulting program $P''$ to the Corral SMT-based bounded model checker [14] to detect assertion violations.

Our implementation is able to find (fair) non-terminating executions in several toy examples, and handed-coded implementations of textbook distributed algorithms [16]; the source code of our examples can be found online [7]. Figure 5 summarizes our experiments on two families of examples which we discuss below: the PingPong family of toy examples, and the SpanningTree family of textbook examples. For each family, Figure 5 lists both "buggy" variations (i.e., those with infinite executions) and "correct" variations (those without infinite executions—at least up to the given delay bound). In each case the delay bound is given by $K$, and a recursion bound is given by $N$; our back-end bounded model checker Corral only explores executions in which the procedure stack never contains more than $N$ frames of any procedure, for a given recursion bound $N \in \mathbb{N}$. Note that our implementation is a simple unoptimized prototype; the running times are simply listed as a validation that our reduction is feasible.

## 4.1 PingPong

As a simple example of a non-terminating asynchronous program, consider the `PingPong` program of Figure 6. Initially the `Main` procedure initializes the Boolean variable x to **false** and posts asynchronous calls to Ping and Pong. When Ping executes and x is **false**, then Ping posts a subsequent call to Ping, and sets x to **true**; otherwise Ping simply returns. Similarly, when Pong executes and x is **true**, then Pong posts a subsequent call to Pong, and sets x to **false**; otherwise Pong simply returns. This program has exactly one non-terminating execution: that where the pending instances to Ping and Pong execute in alternation. This execution is periodic, as the configuration where x=**false**, and both Ping and Pong have a single pending instance, is encountered infinitely often.

Figure 6a depicts an execution of the program resulting from our translation (Section 3.1) of the PingPong program. Following our translation, the Main procedure takes the branch of Line 28 in Figure 3, posting both pro(Ping) and Ping, then both pro(Pong) and Pong. Without using any delay operations, the scheduler encoded by AsyncChecker executes the posted tasks in depth-first order over the task-creation tree [8, 9]. Thus following Main, pro(Ping) executes, then Ping, followed by anti(Ping). Subsequently, pro(Pong), Pong, and anti(Pong) execute, in that order. Luckily this execution provides a witness to nontermination without spending a single delay.

Our experiments include several variations of this example. The -mod2 and -mod3 variations add an integer variable i which is incremented (modulo 2, resp., 3) by each call of Ping. The addition of this counter complicates the search for a repeated configuration, since besides the global variable x and pending tasks Ping and Pong, the value of i must also match in the repeating configuration. This addition also increases the number of delay operations required to discover an infinite execution, as the depth-first task scheduler without delaying considers only executions where all Ping tasks execute before all Pong tasks—see Figure 6b; since, for instance, modulo 2 incrementation requires two of each Ping and Pong tasks to return to a repeating configuration (i.e., with i=0), the second Ping task must delay in order to occur after the first Pong task. In the -1md and -2md variations, we reduce the budget of task delaying, and observe that indeed the additional delay budgets are required to witness nonterminating executions. The PingPongPung variation is an even more intricate variation in which each task (i.e., Ping, Pong, or Pung) posts a different task.

## 4.2 SpanningTree

In Figure 7 we consider two examples of distributed algorithms taken from the textbook of Lynch [16], and modified to introduce nonterminating executions. Essentially, SpanningTree attempts to compute a spanning tree for an arbitrary network by building a parent relation from message broadcasts. When the parent link is established asynchronously there exist (unfair) executions in which nodes cyclically propagate their search messages without ever establishing the parent relation. The BellmanFord algorithm is a generalization of SpanningTree in which links between nodes have weights; the algorithm attempts to establish a spanning tree in which each node is connected by a minimal-weight path. Our injection of a bug demonstrates that even the most trivial of

```
// program SpanningTree
type Pid;
var parent[Pid]: Pid;
var reported[Pid]: bool;

proc Main ()
    var root: Pid;
    assume ∀p: Pid. reported[p] = false;
    post search (root, root);
    return

proc search (var this: Pid, sender: Pid)
    var neighbor: Pid;

    if ¬reported[this] then

        // BUG: should be done synchronously!
        post parent (this, sender);

        while ⋆ do
            let neighbor: Pid in
            assume neighbor ≠ this;
            assume neighbor ≠ sender;
            post search (neighbor, this);

    return

proc parent (var this: Pid, p: Pid)
    parent[this] := p;
    reported[this] := true;
    return
```

```
// program BellmanFord
type Pid;
type Val;
var dist [Pid]: int;
var parent [Pid]: Pid;
const weight [Pid, Pid]: int;

proc Main ()
    var root: Pid;
    assume ∀p: Pid. dist[p] = INF;
    post bellmanFord (root, 0, root);
    return

proc bellmanFord (var this: Pid, w: int,
                      sender: Pid)
    var neighbor: Pid;

    // BUG: should check <, not ≤
    if w + weight[this,sender] ≤ dist[this]
    then
        dist[this] := w + weight[this,sender];
        parent[this] := sender;

        while ⋆ do
            let neighbor: Pid in
            assume neighbor ≠ this;
            assume neighbor ≠ sender;
            post bellmanFord
                (neighbor, dist[this], this);
    return
```

**Fig. 7.** Two distributed asynchronous programs with divergent infinite executions

programming errors (e.g., typing ≤ rather than <) can introduce fair nonterminating executions. Alive automatically discovers these nonterminating executions for an arbitrary, unspecified network.

## 4.3 Paxos

Lamport's Paxos algorithm [15] provides a two-phase protocol for collaboratively choosing a (numeric) value from a set of values proposed by various nodes in a network; Figure 8 lists a basic variation of the algorithm. Initially a set of *proposers* choose a unique value to propose, and broadcast their intention to the set of *acceptors* via the `prepare` message. Each acceptor then decides whether to support the proposed value, depending on whether or not a higher proposal has already been seen. When a `proposal_OK` message is received, the proposer checks whether a majority has been achieved, and if so broadcasts an `accept` message. If in the meantime the acceptors have not encountered a higher proposal, they agree on the given proposal by setting `accepted` on Line 46.

Even in fair executions, divergent behavior can arise from several places. As in the program of Figure 8, the proposers may periodically post higher proposals in case their initial proposal is not answered within a timeout (Line 12), when `NOTIFY_DECLINED` is false. Then even an individual proposer may repeatedly `propose` new values just before receiving the acceptors' `proposal_OK` messages.

The acceptors, in turn, may continue to increment their `prepared` values, such that previously agreed proposals will no longer be accepted (see the condition on Line 40). Even preventing such behavior by assuming the proposers only submit new proposals upon the reception of `declined` messages (i.e., suppose `NOTIFY_DECLINED` is true), fair nonterminating executions may still arise by competition between two or more proposers; for instance where two proposers continuously outbid the other before either's proposal has been accepted.

Since each subsequent proposal in the Paxos algorithm proposed an increasingly large number, strictly speaking our detection algorithm will not discover such nonterminating executions, since the same values of `proposal` and `prepared` will not be encountered twice. Essentially we must extend our well-quasi-ordering of Section 3.1 by relaxing the equality on global state valuations to a well-quasi-ordering which is compatible with the program's transition relation. For the purpose of our experiments, we have encoded manually such an order $\preceq'$ for our variations on the Paxos algorithm; the order relates global valuations $g_1 \preceq' g_2$ when there exists some $\delta \in \mathbb{N}$ such that the values of `proposal` for proposing processes, and `prepared` for accepting processes, in $g_1$ and $g_2$ uniformly increase by $\delta$, and all other variables in $g_1$ and $g_2$ are equal. With this small manual effort, Alive is able to discover the "individual" nonterminating execution described above, and while Alive can also detect the "competing" nonterminating execution in theory, AsyncChecker times out on the reachability check after 30 minutes.

## 5 Related Work

Contrary to much work on sequential program (non)termination detection [5, 11], less attention has been paid to concurrent programs, where nontermination can arise from asynchronous interaction rather than diverging data values. Though both Cook et al. [6] and Popeea and Rybalchenko [17] have proposed techniques to prove termination in multithreaded programs, failure to prove termination does not generally indicate the existence of nonterminating executions. In very recent work, Atig et al. [1] suggest compositional nontermination detection for multithreaded programs based on bounded context-switch; their technique detects infinite executions between a group of interfering, and each non-terminating, threads. Our approach is orthogonal, as we detect infinite executions in which every task terminates; nontermination arises from the never-ending creation of new tasks. Technically, while Atig et al. [1] explore the behaviors between statically-known threads, our problem is to detect the repetition of an unbounded set of dynamically-created tasks.

Our reduction-based technique follows a recent trend of compositional translations to sequential program analysis by considering bounded program behaviors. Based on the notion of bounded context-switch [18], Lal and Reps [13] proposed a reduction from detecting safety violations in multithreaded programs (with a finite number of statically-known threads) to detecting safety violations in sequential programs; shortly after La Torre et al. [12] extended this result to handle an arbitrary number of parametric threads, which was further extended by Emmi et al. [8]

```
// The Proposers
var proposal[Pid]: int;
var agreed[Pid]: int;

proc propose (var p: Pid)
    let n: int = gen_proposal_number () in
    proposal[p] := n;
    agreed[p] := 0;
    post prepare (ACCEPTOR, p, n);

    if ¬NOTIFY_DECLINED then
        post propose(p);
    return

proc proposal_OK (var p: Pid, n: int)
    agreed[p] := agreed[p] + 1;
    if agreed[p] ≥ MAJORITY then
        post accept (ACCEPTOR, p,
                     proposal[p]);
    return

proc declined (var p: Pid, n: int)
    call propose (p);
    return
```

```
// The Accepters
var prepared[Pid]: int;
var accepted[Pid]: int;

proc prepare (var p: Pid, sender: Pid, n: int
      )
    if prepared[p] ≥ n then
        if NOTIFY_DECLINED then
            post declined(sender, n)
    else
        prepared[p] := n;
        post proposal_OK(sender, accepted[p])
    return

proc accept (var p: Pid, sender: Pid, n: int)
    if prepared[p] > n then
        if NOTIFY_DECLINED then
            post declined(sender, n)
    else
        // do there exists infinite runs
        // which never accept any proposal?
        accepted[p] := n
    return
```

**Fig. 8.** A basic variation of the Paxos distributed algorithm; for simplicity we suppose there is only a single accepting process named ACCEPTOR

to handle dynamic thread creation—including the case of task-buffer based "asynchronous programs" [19]. More recently Bouajjani and Emmi [3] proposed a reduction from safety violations in distributed asynchronous programs with ordered message queues. Thus far, only the recent (yet orthogonal—see above) work of Atig et al. [1] considers liveness properties such as nontermination.

Finally, although reductions from fair nontermination of task-buffer based finite-data asynchronous programs (alternatively, Petri nets) are known—e.g., by encoding into Petri net path logic formalæ [10]—our encoding *into* asynchronous programs is original, and takes advantage of existing program analysis tools with efficient under-approximating exploration strategies. Technically, Ganty and Majumdar [10]'s encoding uses constraints on marking-valued variables to ensure that each task pending at the beginning of a repeating period is re-posted and pending at the period's end; a path-logic solver must then determine satisfiability under those constraints. Our encoding handles the matching of pre- and post-period pending tasks directly; we pose an asynchronous program reachability query on a program whose additional tasks block executions in which pre- and post-period tasks cannot be matched.

## 6 Conclusion

We have proposed a practical reduction-based algorithm for detecting divergent executions in distributed asynchronous programs. By incrementally increasing possible task reordering, our approach explores an increasing number of possibly-divergent behaviors with increasing analysis cost, and any possibly-divergent behavior is considered at some cost. By reducing divergence of distributed asynchronous programs to assertion violation in sequential programs, our approach leverages efficient off-the-shelf sequential program analysis tools. Using our prototype tool, Alive, we demonstrate that the approach is able to find divergent executions in modified versions of typical textbook distributed algorithms.

## References

[1] Atig, M.F., Bouajjani, A., Emmi, M., Lal, A.: Detecting Fair Non-termination in Multithreaded Programs. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 210–226. Springer, Heidelberg (2012)

[2] Barnett, M., Leino, K.R.M., Moskal, M., Schulte, W.: Boogie: An intermediate verification language, http://research.microsoft.com/en-us/projects/boogie/

[3] Bouajjani, A., Emmi, M.: Bounded Phase Analysis of Message-Passing Programs. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 451–465. Springer, Heidelberg (2012)

[4] 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)

[5] Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI 2006: Proc. ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, pp. 415–426. ACM (2006)

[6] Cook, B., Podelski, A., Rybalchenko, A.: Proving thread termination. In: PLDI 2007: Proc. ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, pp. 320–330. ACM (2007)

[7] Emmi, M., Lal, A.: Finding non-terminating executions in distributed asynchronous programs (May 2012), http://hal.archives-ouvertes.fr/hal-00702306/

[8] Emmi, M., Qadeer, S., Rakamarić, Z.: Delay-bounded scheduling. In: POPL 2011: Proc. 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 411–422. ACM (2011)

[9] Emmi, M., Lal, A., Qadeer, S.: Asynchronous programs with prioritized task-buffers. Technical Report MSR-TR-2012-1, Microsoft Research (2012)

[10] Ganty, P., Majumdar, R.: Algorithmic verification of asynchronous programs. CoRR, abs/1011.0551 (2010), http://arxiv.org/abs/1011.0551

[11] Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.-G.: Proving non-termination. In: POPL 2008: Proc. 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 147–158. ACM (2008)

[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] 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)

[14] Lal, A., Qadeer, S., Lahiri, S.K.: 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] Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. 16(2), 133–169 (1998)

[16] Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann (1996) ISBN 1-55860-348-4

[17] Popeea, C., Rybalchenko, A.: Compositional Termination Proofs for Multithreaded Programs. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 237–251. Springer, Heidelberg (2012)

[18] 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)

[19] Sen, K., Viswanathan, M.: Model Checking Multithreaded Programs with Asynchronous Atomic Methods. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 300–314. Springer, Heidelberg (2006)

[20] Svensson, H., Arts, T.: A new leader election implementation. In: Erlang 2005: Proc. 2005 ACM SIGPLAN Workshop on Erlang, pp. 35–39. ACM (2005)

[21] Trottier-Hebert, F.: Learn you some Erlang for great good!, `http://learnyousomeerlang.com/`