Int J Softw Tools Technol Transfer (2014) 16:127–146
DOI 10.1007/s10009-013-0276-z

TACAS 2012

Bounded phase analysis of message-passing programs

Ahmed Bouajjani · Michael Emmi

Published online: 4 May 2013

Abstract We describe a novel technique for bounded analysis of asynchronous message-passing programs with ordered message queues. Our bounding parameter does not limit the number of pending messages, nor the number of "context-switches" between processes. Instead, we limit the number of process communication cycles, in which an unbounded number of messages are sent to an unbounded number of processes across an unbounded number of contexts. We show that remarkably, despite the potential for such vast exploration, our bounding scheme gives rise to a simple and efficient program analysis by reduction to sequential programs. As our reduction avoids explicitly representing message queues, our analysis scales irrespectively of queue content and variation.

Keywords Concurrency · Verification · Analysis · Message-passing · Distributed

1 Introduction

Software is becoming increasingly concurrent: reactivity (e.g., for user interfaces, web servers), parallelization (e.g., in scientific computations), and decentralization (e.g., in web applications) necessitate asynchronous computation. Although shared-memory implementations are often possible, the burden of preventing unwanted thread interleavings without crippling performance is onerous. Many have instead adopted asynchronous programming models in which processes communicate by posting messages/tasks to others' message/task queues—Miller et al. [34] discuss why such models provide good programming abstractions. Single-process systems such as the JavaScript page-loading engine of modern web browsers [22], and the highly scalable Node.js asynchronous web server [13], execute a series of short-lived tasks one-by-one, each task potentially queueing additional tasks to be executed later. This programming style ensures that the overall system responds quickly to incoming events (e.g., user input, connection requests). In the multi-process setting, languages such as Erlang and Scala have adopted message-passing as a fundamental construct with which highlyscalable and highly reliable distributed systems are built.

Despite the increasing popularity of such programming models, little is known about precise algorithmic reasoning. This is perhaps not without good reason: decision problems such as state-reachability for programs communicating with unbounded reliable queues are undecidable [11], even when there is only a single finite-state process (posting messages to itself). Furthermore, the known algorithms for decidable under-approximations (e.g., bounding the size of queues) represent queues explicitly, and are thus doomed to combinatorial explosion as the size and variability of queue content increases.

Some have proposed analyses which abstract message arrival order [18,23,39], or assume messages can be arbitrarily lost [1,2]. Such analyses do not suffice when correctness arguments rely on reliable messaging—we discuss in Sect. 6.2 a fairly realistic case study in which correctness arguments can rely on reliable messaging—and several systems specifically do ensure the ordered delivery of messages, including Scala's runtime system [20], and recent web-browser specifications [22]. Others have proposed analyses which compute

Michael was supported by a post-doctoral fellowship from la Fondation Sciences Mathématiques de Paris.

A. Bouajjani · M. Emmi (✉)
LIAFA, Université Paris Diderot, Paris, France
e-mail: mje@liafa.univ-paris-diderot.fr

A. Bouajjani
e-mail: abou@liafa.univ-paris-diderot.fr

finite symbolic representations of queue contents [6,9]. Known bounded analyses which model queues precisely either bound the maximum capacity of message-queues, ignoring executions which exceed the bound, or bound the total number of process "contexts" [21,27,36], where each context involves a single process sending and receiving messages. For each of these bounding schemes there are trivial systems which cannot be adequately explored, e.g., by sending more messages than the allowed queue-capacity, having more processes than contexts, or by alternating message-sends to two processes—we discuss such examples in Sect. 3. All of the above techniques explicitly maintain some (perhaps symbolic) representation of queue contents, and face combinatorial explosion as queue content and variation increase.

In this work we propose a novel technique for bounded analysis of asynchronous message-passing programs with reliable, ordered message queues. Our bounding parameter, introduced in Sect. 3, is not sensitive to the capacity nor content of message queues, nor the number of process contexts. Instead, we bound the number of process communication cycles by labeling each message with a monotonically increasing phase number. Each time a message chain visits the same process, the phase number must increase. For a given parameter k, we only explore behaviors of up to k phases—though k phases can go a long way. In the leader election distributed protocol [41] for example, each election round occurs in 2 phases: in the first phase each process sends *capture* messages to the others; in the second phase some processes receive *accept* messages, and those that find themselves majority-winners broadcast *elected* messages. In these two phases an unbounded number of messages are sent to an unbounded number of processes across an unbounded number of process contexts.

We demonstrate the strength of phase-bounding by showing in Sects. 4 and 5 that the bounded phase executions of a message-passing program can be concisely encoded as a nondeterministic sequential program, in which message-queues are not explicitly represented. Our so-called "sequentialization" sheds hope for scalable analyses of message-passing programs. In a small set of simple experiments (Sect. 4), we demonstrate that our phase-bounded encoding scales far beyond known explicit-queue encodings as queue-content increases, and even remains competitive as queue-content is fixed while the number of phases grows. In Sect. 6 we present a case study using a prototype implementation of our sequentialization to quickly discover bugs in typical textbook asynchronous network algorithms. By reducing to sequential programs, we leverage highly developed sequential program analysis tools for the algorithmic analysis of message-passing programs.

A shorter version of this paper appears in the LNCS proceedings of TACAS 2012 [7]. The current version contains additional complexity results, complete proofs to theorems and lemmas, and the experiment case study of Sect. 6.

2 Asynchronous message-passing programs

We consider a simple multi-processor programming model in which each processor is equipped with a procedure stack and a queue of pending tasks. Initially all processors are idle. When an idle processor's queue is non-empty, the oldest task in its queue is removed and executed to completion. Each task executes essentially a recursive sequential program, which besides accessing its own processor's global storage, can *post* tasks to the queues of any processor, including its own. When a task does complete, its processor again becomes idle, chooses the next pending task to execute to completion, and so on. The distinction between queues containing messages and queues containing tasks is mostly aesthetic, but in our task-based treatment queues are only read by idle processors; reading additional messages during a task's execution is prohibited. While 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 task [42]. We thus make the *well-queueing* assumption [27] that only idle processors (i.e., those not currently executing a task, or in other words, those whose current procedure activation stacks are empty) can take new tasks from their task queues.

Though similar to Sen and Vishwanathan [39]'s model of asynchronous programs, the model we consider has two important distinctions. First, tasks execute across potentially several processors, rather than only one, each processor having its own global state and pending tasks. Second, the tasks of each processor are executed in exactly the order they are posted. In the case of single-processor programs, Sen and Vishwanathan [39]'s model can be seen as an abstraction of the model we consider, since there the task chosen to execute next when a processor is idle is chosen non-deterministically among all pending tasks.

2.1 Program syntax

Let `Procs` be a set of procedure names, `Vals` a set of values, `Exprs` a set of expressions, `Pids` a set of processor identifiers, and let T be a type. Figure 1 gives the grammar of *asynchronous message-passing programs*. We intentionally leave the syntax of expressions e unspecified, though we do insist `Vals` contains **true** and **false**, and `Exprs` contains `Vals` and the (*nullary*) *choice operator* $\star$.

Each program P declares a single global variable `g` and a procedure sequence, each $p \in$ `Procs` having a single parameter `l` and top-level statement denoted s_p; as statements are built inductively by composition with control-flow state-

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

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

ments, s_p describes the entire body of p. The set of program statements s is denoted $\mathtt{Stmts}$. Intuitively, a **post** $\rho\ p\ e$ statement is an asynchronous call to a procedure p with argument e to be executed on the processor identified by ρ; a *self-post* to one's own processor is made by setting ρ to _. A program in which all **post** statements are self-posts is called a *single-processor program*, and a program without **post** statements is called a *sequential program*. The **assume** e statement proceeds only when e evaluates to **true**; we use this statement to prevent undesired executions in later sections.

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. Appendix A lists several syntactic extensions which we use in the source-to-source translations of the subsequent sections, e.g., multiple variables, and which easily reduce to the syntax of our grammar.

2.2 Single-processor semantics

A (*procedure*) *frame* $f = \langle \ell, s \rangle$ is a current valuation $\ell \in \mathtt{Vals}$ to the procedure-local variable $\mathtt{l}$, along with a statement $s \in \mathtt{Stmts}$ 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 processor queues.) The set of all frames is denoted $\mathtt{Frames}$.

A *processor configuration* $\kappa = \langle g, w, q \rangle$ is a current valuation $g \in \mathtt{Vals}$ to the processor-global variable $\mathtt{g}$, along with a procedure-frame stack $w \in \mathtt{Frames}^*$ and a pending-tasks queue $q \in \mathtt{Frames}^*$. A processor is idle when $w = \varepsilon$. The set of all processor configurations is denoted $\mathtt{Pconfigs}$. A processor configuration map $\xi : \mathtt{Pids} \to \mathtt{Pconfigs}$ maps each processor $\rho \in \mathtt{Pids}$ to a processor configuration $\xi(\rho)$. We write $\xi(\rho \mapsto \kappa)$ to denote the configuration ξ updated with the mapping $(\rho \mapsto \kappa)$, i.e., the configuration ξ' such that $\xi'(\rho) = \kappa$, and $\xi'(\rho') = \xi(\rho')$ for all $\rho' \in \mathtt{Pids} \backslash \{\rho\}$.

For expressions without program variables, we assume the existence of an evaluation function $[\![\cdot]\!]_{\mathrm{e}} : \mathtt{Exprs} \to \wp(\mathtt{Vals})$ such that $[\![\star]\!]_{\mathrm{e}} = \mathtt{Vals}$. For convenience, given a processor configuration $\kappa = \langle g, w, q \rangle$ and $w = \langle \ell, s \rangle\, w'$, we define

$$e(\kappa) \stackrel{\text{def}}{=} e(g, \ell) \stackrel{\text{def}}{=} [\![e[g/\mathtt{g}, \ell/\mathtt{l}]]\!]_{\mathrm{e}}$$

to evaluate the expression e in a processor configuration κ (alternatively, in a global valuation g and local valuation ℓ) by substituting the current values for variables $\mathtt{g}$ and $\mathtt{l}$. As these are the only program variables, the substituted expression $e[g/\mathtt{g}, \ell/\mathtt{l}]$ has no free variables. Additionally we define

$$
\begin{array}{ll}
\kappa(\mathtt{g} \leftarrow g') \stackrel{\text{def}}{=} \langle g', w, q \rangle & \text{global assignment,} \\
\kappa(\mathtt{l} \leftarrow \ell') \stackrel{\text{def}}{=} \langle g, \langle \ell', s \rangle\, w', q \rangle & \text{local assignment,} \\
\kappa \cdot f \stackrel{\text{def}}{=} \langle g, f \cdot w, q \rangle & \text{append stack frame.}
\end{array}
$$

To further reduce clutter in the operational program semantics, we introduce a notion of context. A *statement context* S is a term derived from the grammar $S ::= \diamond \mid S; s$, where $s \in \mathtt{Stmts}$. We write $S[s]$ for the statement obtained by substituting a statement s for the unique occurrence of $\diamond$ in S. Intuitively, a context filled with s, e.g., $S[s]$, indicates that s is the next statement to execute in the statement sequence $S[s]$. Similarly, a *processor configuration context* $C = \langle g, \langle \ell, S \rangle\, w, q \rangle$ is a processor configuration whose topmost frame's statement is replaced with a statement context, and we write $C[s]$ to denote the processor configuration $\langle g, \langle \ell, S[s] \rangle\, w, q \rangle$. When e is an expression, we abbreviate $e(C[\textbf{skip}])$ by $e(C)$.

$$
\textsc{Skip}\ \frac{}{C[\textbf{skip};\ s] \to C[s]}
\qquad
\textsc{If-Then}\ \frac{\mathtt{true} \in e(C)}{C[\textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2] \to C[s_1]}
\qquad
\textsc{If-Else}\ \frac{\mathtt{false} \in e(C)}{C[\textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2] \to C[s_2]}
$$

$$
\textsc{Assume}\ \frac{\mathtt{true} \in e(C)}{C[\textbf{assume}\ e] \to C[\textbf{skip}]}
\qquad
\textsc{Loop-Do}\ \frac{\mathtt{true} \in e(C)}{C[\textbf{while}\ e\ \textbf{do}\ s] \to C[s; \textbf{while}\ e\ \textbf{do}\ s]}
\qquad
\textsc{Loop-End}\ \frac{\mathtt{false} \in e(C)}{C[\textbf{while}\ e\ \textbf{do}\ s] \to C[\textbf{skip}]}
$$

$$
\textsc{Assign}\ \frac{v \in e(C)}{C[x\ :=\ e] \to C[\textbf{skip}]\,(x \leftarrow v)}
\qquad
\textsc{Call}\ \frac{v \in e(C) \qquad f = \langle v, s_p \rangle}{C[\textbf{call}\ x\ :=\ p\ e] \to C[x\ :=\ \star] \cdot f}
\qquad
\textsc{Return}\ \frac{f = \langle \ell, S[\textbf{return}\ e] \rangle \qquad v \in e(C \cdot f)}{C[x\ :=\ \star] \cdot f \to C[x\ :=\ v]}
$$

Fig. 2 The single-processor transitions relation $\to$ for the standard sequential program statements

$$\textsc{Dispatch}\ \frac{}{\langle g, \varepsilon, f \cdot q\rangle \to \langle g, f, q\rangle} \qquad \textsc{Complete}\ \frac{f = \langle \ell, S[\mathbf{return}\ e]\rangle}{\langle g, f, q\rangle \to \langle g, \varepsilon, q\rangle}$$

$$\textsc{Self-Post}\ \frac{\ell_2 \in e(g, \ell_1) \qquad f = \langle \ell_2, s_p\rangle}{\langle g, \langle \ell_1, S[\mathbf{post}\ \_\ p\ e]\rangle\, w, q\rangle \to \langle g, \langle \ell_1, S[\mathbf{skip}]\rangle\, w, q \cdot f\rangle}$$

Fig. 3 The single-processor transition rules $\to$

Figure 2 defines the transition relation $\to$ for the standard sequential program statements. The SKIP rule simply steps past the `skip` statement. The ASSUME rule proceeds only when the given expression e evaluates to **true**. The ASSIGN statement stores the value of a given expression in either the local variable `l` or the global variable `g`. The IF- THEN and IF- ELSE rules proceed to either the **then** or `else` branch, depending on the current valuation of the given expression e. Similarly, the LOOP- DO and LOOP- END rules proceed to (re-)enter the loop when the given expression e evaluates to **true**, and step past the loop when e evaluates to false. More interestingly, the CALL rule creates a new procedure frame f by evaluating the given argument e, and places f at the top of the procedure-frame stack. The RETURN rule removes the top-most procedure frame from the stack, and substitutes the valuation of the return expression e into the assignment $x := \star$ left below by the matching **call** statement. Note that the transition relation $\to$ is non-deterministic, since the evaluation of an expression e can result in an arbitrary set of possible values.

Figure 3 defines the transition relation $\to$ for the asynchronous behavior of each processor. The SELF- POST rule creates a new frame to execute the given procedure, and places the new frame in the current processor's pending-tasks queue. The COMPLETE rule returns from the final frame of a task, rendering the processor idle, and the DISPATCH rule schedules the least-recently posted task on a idle processor.

2.3 Multi-processor semantics

In reality the processors of multi-processor systems execute independently in parallel. However, as long as they either do not share memory, or access a sequentially consistent shared memory, it is equivalent, w.r.t. the observations of any single processor, to consider an *interleaving semantics*: at any moment only one processor executes. Intuitively, this is true since we consider that each message is *received* atomically. In order to later restrict processor interleaving, we make explicit the *scheduler* which arbitrates the possible interleavings. Formally, a *scheduler*

$$M = \langle D, \texttt{empty}, \texttt{enabled}, \texttt{step}\rangle$$

consists of a data type D of scheduler objects $m \in D$, a scheduler constructor $\texttt{empty} \in D$, a scheduler decision function

$$\textsc{Switch}\ \frac{\rho_2 \in \mathsf{enabled}(m, \xi)}{\langle \rho_1, \xi, m\rangle \to \langle \rho_2, \xi, m\rangle}$$

$$\textsc{Step}\ \frac{\xi_1(\rho) \to \kappa \qquad \xi_2 = \xi_1\,(\rho \mapsto \kappa) \qquad \rho \in \mathsf{enabled}(m_1, \xi_1) \qquad m_2 = \mathsf{step}(m_1, \xi_1, \xi_2)}{\langle \rho, \xi_1, m_1\rangle \to \langle \rho, \xi_2, m_2\rangle}$$

$$\textsc{Post}\ \frac{\begin{array}{c}\xi_1(\rho_1) = \langle g_1, \langle \ell_1, S[\mathbf{post}\ \rho_2\ p\ e]\rangle\, w_1, q_1\rangle \\ \xi_1(\rho_2) = \langle g_2, w_2, q_2\rangle \\ \rho_1 \neq \rho_2 \qquad \ell_2 \in e(g_1, \ell_1) \qquad f = \langle \ell_2, s_p\rangle \\ \rho_1 \in \mathsf{enabled}(m_1, \xi_1) \qquad m_2 = \mathsf{step}(m_1, \xi_1, \xi_3) \\ \xi_2 = \xi_1\,(\rho_1 \mapsto \langle g_1, \langle \ell_1, S[\mathbf{skip}]\rangle\, w_1, q_1\rangle) \\ \xi_3 = \xi_2\,(\rho_2 \mapsto \langle g_2, w_2, q_2 \cdot f\rangle)\end{array}}{\langle \rho_1, \xi_1, m_1\rangle \to \langle \rho_1, \xi_3, m_2\rangle}$$

Fig. 4 The multi-processor transition relation $\to$ parameterized by a scheduler $M = \langle D, \texttt{empty}, \texttt{enabled}, \texttt{step}\rangle$

$\texttt{enabled} : (D \times (\texttt{Pids} \to \texttt{Pconfigs})) \to \wp(\texttt{Pids})$, and a scheduler update function $\texttt{step} : (D \times (\texttt{Pids} \to \texttt{Pconfigs}) \times (\texttt{Pids} \to \texttt{Pconfigs})) \to D$. The arguments to `enabled` allow a scheduler to decide which processors are enabled depending on the execution history. A scheduler is *deterministic* when $|\texttt{enabled}(m, \xi)| \leq 1$ for all $m \in D$ and $\xi : \texttt{Pids} \to \texttt{Pconfigs}$, and is *non-blocking* when for all m and ξ, if there is some $\rho \in \texttt{Pids}$ such that $\xi(\rho)$ is either non-idle or has pending tasks, then there exists $\rho' \in \texttt{Pids}$ such that $\rho' \in \texttt{enabled}(m, \xi)$ and $\xi(\rho')$ is either non-idle or has pending tasks. A *configuration* $c = \langle \rho, \xi, m\rangle$ is a currently executing processor $\rho \in \texttt{Pids}$, along with a processor configuration map ξ, and a scheduler object m.

Figure 4 defines the multi-processor transition relation $\to$, parameterized by a scheduler M. The *Switch* rule nondeterministically schedules any enabled processor, while the STEP rule executes one single-processor program step on the currently scheduled processor, and updates the scheduler object. Finally, the POST rule creates a new frame to execute the given procedure, and places the new frame on the target processor's pending-tasks queue.

Until further notice, we assume M is a completely non-deterministic scheduler; i.e., all processors are always enabled. In Sect. 5 we discuss alternatives.

An *M-execution of a program P (from c_0 to c_j)* is a configuration sequence $c_0 c_1 \ldots c_j$ such that $c_i \to c_{i+1}$ for $0 \leq i < j$. An *initial condition* $\iota = \langle \rho_0, g_0, \ell_0, p_0\rangle$ is a processor identifier ρ_0, along with a global-variable valuation $g_0 \in \texttt{Vals}$, a local-variable valuation $\ell_0 \in \texttt{Vals}$, and a procedure $p_0 \in \texttt{Procs}$. A configuration $c = \langle \rho_0, \xi, m\rangle$ of a program P is $\langle \rho_0, g_0, \ell_0, p_0\rangle$-*initial* when $m = \texttt{empty}$, $\xi(\rho_0) = \langle g_0, \varepsilon, \langle \ell_0, s_{p_0}\rangle\rangle$ and $\xi(\rho) = \langle g_0, \varepsilon, \varepsilon\rangle$ for all $\rho \neq \rho_0$. A configuration $\langle \rho_0, \xi, m\rangle$ is *final* when $\xi(\rho) = \langle g, \varepsilon, \varepsilon\rangle$ for some $g \in \texttt{Vals}$ for all $\rho \in \texttt{Pids}$, and an execution to a final configuration is called *completed*. We say a global

valuation g is *M-reachable* in P from ι when there exists an M-execution[1] of P from some c_0 to some $\langle \rho, \xi, m \rangle$ such that c_0 is ι-initial and $\xi(\rho') = \langle g, w, q \rangle$ for some $\rho' \in \texttt{Pids}$.

Definition 1 The *state-reachability problem* is to determine for an initial condition ι, valuation g, and program P, whether g is reachable in P from ι.

3 Phase-bounded execution

Because processors execute tasks precisely in the order which they are posted to their unbounded task-queues, our state-reachability problem is undecidable, even with only a single processor accessing finite-state data [11]. Since it is not algorithmically possible to consider every execution precisely, in what follows we present an incremental under-approximation. For a given bounding parameter k, we consider a subset of execution (prefixes) precisely; as k increases, the set of considered executions increases, and in the limit as k approaches infinity, every execution of any program is considered—though for many programs, every execution is considered with a finite value of k.

In a given execution, a *task-chain* $t_1 t_2 \ldots t_i$ from t_1 to t_i is a sequence of tasks [2] such that the execution of each t_j posts t_{j+1}, for $0 < j < i$, and we say that t_1 is an *ancestor* of t_i. We characterize execution prefixes by labeling each task t posted in an execution with a *phase number* $\varphi(t) \in \mathbb{N}$:

$$\varphi(t) = \begin{cases} 0 & \text{if } t \text{ is initially pending} \\ \varphi(t') & \text{if } t \text{ is posted to processor} \rho \text{ by } t', \text{ and } t \\ & \text{has no phase-}\varphi(t') \text{ ancestor on processor } \rho \\ \varphi(t') + 1 & \text{if } t \text{ is posted by } t', \text{ otherwise.} \end{cases}$$

For instance, considering Fig. 5a, supposing all on a single processor, an initial task A_1 posts A_2, A_3, and A_4, then A_2 posts A_5 and A_6, and then A_3 posts A_7, which in turn posts A_8 and A_9. Task A_1 has phase 0. Since each post is made to the same processor, the phase number is incremented for each posted task. Thus the phase 1 tasks are $\{A_2, A_3, A_4\}$, the phase 2 tasks are $\{A_5, A_6, A_7\}$, and the phase 3 tasks are $\{A_8, A_9\}$. Notice that tasks of a given phase only execute after all tasks of the previous phase have completed, i.e., execution order is in phase order; only executing tasks up to a given phase does correspond to a valid execution prefix.

Definition 2 An execution is *k-phase* when $\varphi(t) < k$ for each executed task t.

[1] Since the **assume** statement may block executions depending on earlier nondeterministic choices, we consider only the values reached in completed executions as reachable; this avoids the problematic situation where a valuation is reached in an execution which cannot later complete because of finally non-realizable nondeterministic choices.

[2] We assume each task in a given execution has implicitly a unique task-identifier.

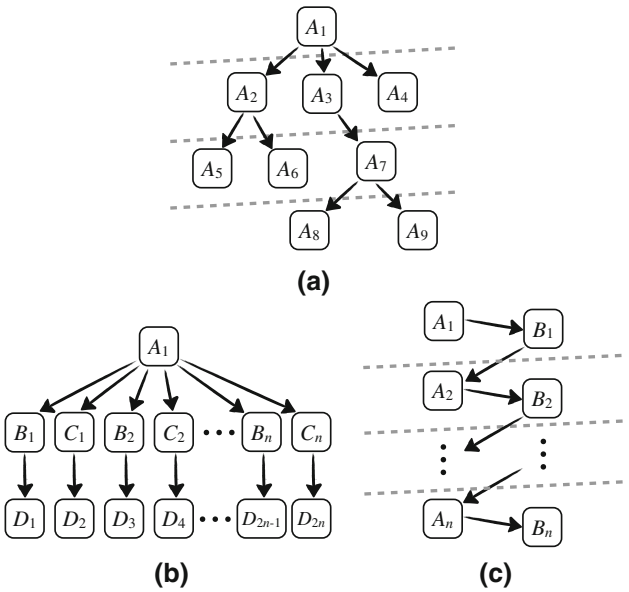


Fig. 5 Phase-bounded executions with processors A, B, C, and D; each task's *label* (e.g., A_i) indicates the processor it executes on (e.g., A). *Arrows* indicate the posting relation, *indices* indicate execution order on a given processor, and *dotted lines* indicate phase boundaries

The execution in Fig. 5 a is a 4-phase execution, since all tasks have phase <4. Despite there being an arbitrary number $4n + 1$ of posted tasks, the execution in Fig. 5b is 1-phase, since there are no task-chains between same-processor tasks. Contrarily, the execution in Fig. 5c requires n phases to execute all $2n$ tasks, since every other occurrence of an A_i task creates a task-chain between A-tasks.

Note that bounding the number of execution phases does not necessarily bound the total number of tasks executed, nor the maximum size of task queues, nor the amount of switching between processors. Instead, a bound k restricts the maximum length of task chains to $k \cdot |\texttt{Pids}|$. In fact, phase-bounding is incomparable to bounding the maximum size of task queues. On the one hand, every execution of a program in which one root task posts an arbitrary, unbounded number of tasks to other processors (e.g., in Fig. 5b) are explored with 1 phase, though no bound on the size of queues will capture all executions. On the other hand, all executions with a single arbitrarily long chain of tasks (e.g., in Fig. 5c) are explored with size 1 task queues, though no limited number of phases captures all executions. In the limit as the bounding parameter increases, either scheme does capture all executions.

In comparison to the context-bounded approach [21,27], where each "context" involves a single process reading from its queue, and posting to the queues of other processes, each k-context execution is a k-phase execution, since the ith context may create messages of phase at most $i+1$. However, as Fig. 5b illustrates, there are programs whose fixed-phase executions require an unbounded number of contexts to capture:

a single phase captures the execution order $D_1 D_2 D_3 \ldots D_{2n}$ on processor D, which requires $n+2$ contexts (specifically, of processors $A(BC)^n D$) to capture. We provide a more thorough comparison between phase-bounding and context-bounding in Sect. 7.

Lemma 1 (Completeness) *For every execution h of a program P, there exists $k \in \mathbb{N}$ such that h is a k-phase execution.*

Proof This follows from the inductive definition of φ. □

4 Phase-bounding for single-processor programs

Characterizing executions by their phase-bound reveals a simple and efficient technique for bounded exploration. This seems remarkable, given that phase-bounding explores executions in which arbitrarily many tasks execute, making the task queue arbitrarily large. We demonstrate in Sect. 4.1 a succinct encoding of phase-bounded state-reachability into state-reachability in sequential programs. This reduction leads to an asymptotically optimal algorithm, in a certain sense, since we show in Sect. 4.2 that our EXPTIME reduction-based algorithm for finite-data programs decides an EXPTIME-hard problem. In Sect. 4.3 we demonstrate that our encoding can indeed be implemented efficiently.

4.1 A sequential encoding of phase-bounded exploration

Towards an encoding of phase-bounded executions into executions of sequential programs, the first key ingredient is that once the number of phases is bounded, each phase can be executed in isolation. For instance, consider again the execution of Fig. 5a. In phase 1, the tasks A_2, A_3, and A_4 pick up execution from the global valuation g_1 which A_1 left off at, and leave behind a global valuation g_2 for the phase 2 tasks. In fact, given the sequence of tasks in each phase, the only other "communication" between phases is a single passed global valuation; executing that sequence of tasks on that global valuation is a faithful simulation of that phase.

The second key ingredient is that the ordered sequence of tasks executed in a given phase is exactly the ordered sequence of tasks posted in the previous phase. This is obvious, since tasks are executed in the order they are posted. However, combined with the first ingredient we have quite a powerful recipe. Supposing the global state g_i at the beginning of each phase i is known initially, we can simulate a k-phase execution by executing each task posted to phase i as soon as it is posted, with an independent virtual copy of the global state, initially set to g_i. That is, our simulation will store a vector of k global valuations, one for each phase. Initially, the ith global valuation is set to the state g_i in which phase i begins; tasks of phase i then read from and write to the ith global valuation. It then only remains to ensure that the global valuations g_i used at the beginning of each phase $i : 0 < i < k$ match the valuations reached at the end of phase $i-1$.

```
1  // translation of decl. var g: T
2  var G[k]: T
3
4  // translation of declaration
5  // proc p (var l: T) s
6  proc p (var l: T, phase: k) s
7
8  // translation of lvalue g
9  G[phase]
10
11 // translation of call x := p e
12 call x := p (e,phase)
13
14 // translation of post _ p e
15 if phase+1 < k then
16     call p (e,phase+1)
17
18 // the entry procedure for ((P))_k
19 proc Top (var l: T)
20     let G0[k]: T in
21     G := G0;
22     call p0(l,0);
23     assume ∀i.0<i<k ⇒ G[i-1] = G0[i];
24     // executions to this point are
25     // known to be valid
26     return
```

Fig. 6 The k-phase sequential translation $((P))_k$ of a single-processor asynchronous message-passing program P with entry procedure p_0

This simulation is easily encoded into a non-deterministic sequential program with k copies of global storage. The program begins by non-deterministically setting each copy to an arbitrary value. Each task maintains their current phase number i, and accesses the ith copy of global storage. Each posted task is simply called instead of posted, its phase number set to one greater than its parent—posts to tasks with phase number k are ignored. At the end of execution, the program ensures that the ith global valuation matches the initially used valuation for phase $i+1 : 0 \leq i < k-1$. When this condition holds, any global valuation observed along the execution is reachable within k phases in the original program. Figure 6 lists a code-to-code translation which implements this simulation.

Lemma 2 *A global-valuation g is reachable in a k-phase execution of a single-processor program P if and only if g is reachable in a completed execution of $((P))_k$—the k-phase sequential translation of P.*

Proof This follows easily from our preceding development. Any g reachable in a k-phase execution is reachable by executing a sequence of task sequences $\tau_0 \tau_1 \ldots$, each sequence τ_i containing tasks with phase number i, which encounter the global valuations $g_0 g_1 \ldots$, each g_i encountered at the

```
var state: Q
const stt[0...k]: Q
const fin[0...k]: Q
const src[δ]: Q
const ltr[δ]: Σ
const tgt[δ]: Q
const top[δ]: Γ
const fst[δ]: Γ
const snd[δ]: Γ

proc main()
  call init 0;
  call pda $;
  call last 0;
  return
```

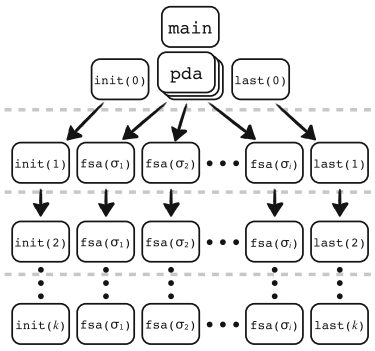
```
proc pda(var ss: Γ)
  while true do
    let tx: δ in
    assume state = src[tx];
    assume ss = top[tx];
    state := tgt[tx];
    if snd[tx] ≠ ⊥ then
      // push
      ss := fst[tx];
      call pda snd[tx]
    else if fst[tx] = ⊥
      // pop
      return
    else
      // internal
      ss := fst[tx];
    post fsa ltr[tx];
  return
```

```
proc init(var i: 0...k)
  state := stt[i];
  if i < k then
    post init (i+1);
  return

proc last(var i: 0...k)
  assume state = fin[i];
  if i < k then
    post last (i+1);
  return

proc fsa(var sym: Σ)
  let tx: δ in
  assume state = src[tx];
  assume sym = ltr[tx];
  state := tgt[tx];
  post fsa sym;
  return
```

(a)



(b)

Fig. 7 Encoding emptiness of the language intersection of a pushdown automaton and k finite state automata, as $(k+1)$-phase state-reachability for finite-data single-processor programs. **a** The program P constructed from a given set of automata with states Q, alphabet Σ, stack alphabet Γ, and transitions δ; `$` denotes the *bottom*-of-the-stack *symbol*. **b** A $(k+1)$-phase execution of P, where *arrows* indicate task posting, and *dashed lines* indicate phase boundaries

beginning of the ith sequence τ_i. Given correctly guessed values of $g_0 g_1 \ldots$, the sequential program $((P))_k$ executes each task of phase $i : 0 \leq i < k$ in the order of τ_i, posting, in order, the tasks τ_{i+1} to phase $i+1$. As the **assume** statement of Line 23 ensures that each $g_i : 0 < i < k$ matches the value reached at the end of the phase $i-1$ tasks τ_{i-1}, the completed executions of $((P))_k$ correspond exactly to the k-phase executions of P. □

Given any underlying sequential program model, e.g., programs with integer variables, our translation makes applicable any analysis tool for the said model to message-passing programs, since the values of the additional variables are either from the finite domain $\{0, \ldots, k-1\}$, or in the domain of the original program variables. When the underlying sequential program model has a decidable state-reachability problem, Lemma 2 yields a decision procedure for the phase-bounded state-reachability problem, by applying the decision procedure for the underlying model to the translated program. This allows us for instance to derive a decidability result for programs with finite data domains.

4.2 Complexity of phase-bounded exploration

Here we demonstrate that state-reachability of single-procesor programs is EXPTIME-complete for programs with finite data domains. We show EXPTIME-hardness by reduction from the language emptiness problem of the intersection of a pushdown automaton and set of regular automata, and EXPTIME-membership by leveraging state-reachability algorithms for the polynomial-sized sequential programs resulting from our phase-bounded sequential translation.

Lemma 3 *The language emptiness problem for the intersection of a pushdown automaton and $k \in \mathbb{N}$ finite state automata is polynomial-time reducible to the $(k+1)$-phase state-reachability problem for finite-data programs.*

Proof Let $\mathcal{A}_0$ be a pushdown automaton, and let $\mathcal{A}_1, \ldots, \mathcal{A}_k$ be finite state automata. For simplicity suppose that the sets $Q_i : 0 \leq i \leq k$ of $\mathcal{A}_i$ states are disjoint. Let $Q = \bigcup_i Q_i$, and let Σ and δ be the sets of symbols and transitions of $\bigcup_i \mathcal{A}_i$. Further suppose that each $\mathcal{A}_i$ has a single initial state, and a single final state without outgoing transitions; this is without loss of generality since $\mathcal{A}_i$ may be nondeterministic. Finally, we suppose $\mathcal{A}_0$ accepts upon reaching its final state with an empty stack, and that each transition of $\mathcal{A}_0$ changes the stack size by at most one.

We define a finite-data single-processor program P with a single global variable `state`, along with five procedures, in Fig. 7. Initially, the `main` procedure calls `init(0)`, then `pda($)`—we use `$` to denote the empty stack symbol—then `last(0)`. Each invocation of `init(i)` (resp., `last(i)`) checks that the current state is equal to the initial (resp., final) state of automaton $\mathcal{A}_i$, and posts the next instance of `init(i+1)` (resp., `last(i+1)`). The `pda` procedure simulates the pushdown automaton $\mathcal{A}_0$, repeatedly and nondeterministically choosing some transition `tx` enabled given the current state and stack symbol (i.e., when `state` and `ss` are equal to `src[tx]` and `top[tx]`), updating the state to the transition's target `tgt[tx]`, and either simulating a push transition by calling `pda` recursively, a pop transition by returning from the topmost instance of `pda`, or an internal transition, by simply updating the top stack symbol `ss`; the stack symbols `fst[tx]` and `snd[tx]` are the first and second symbols transition `tx`puts on the stack, and $\perp$ denotes no symbol. For each transition `tx`, `pda` also posts an instance of `fsa` with the symbol `ltl[tx]` it has read as an argument.

Subsequently, each automata $\mathcal{A}_i : 0 < i \leq k$ is simulated in each phase i by the sequence of pending tasks `init(i)` `fsa`(σ_1) ... `fsa`(σ_j) `last(i)`, according to the word $w = \sigma_1 \ldots \sigma_j$ initially read by $\mathcal{A}_0$. Similarly to the simulation of $\mathcal{A}_0$, each $\mathcal{A}_i$ is simulated by choosing nondeterministically a sequence of transitions, at each step ensuring that one transition follows from the state reached by the previous; the `init(i)` and `last(i)` procedures ensure $\mathcal{A}_i$ has started in its initial state, and ends in its final state after reading w. In this way, the valuation `state` $=q_f$, where q_f is the final state of $\mathcal{A}_k$, is reachable in a $k+1$ phase execution of P if and only if w is accepted by each $\mathcal{A}_i : 0 \leq i \leq k$. □

Theorem 1 *The phase-bounded state-reachability problem for finite-data single-processor programs is EXPTIME-complete.*

Proof As the language emptiness problem for the intersection of a pushdown automaton and $k \in \mathbb{N}$ finite state automata is EXPTIME-complete [17], the reduction of Lemma 3 proves EXPTIME-hardness. An EXPTIME algorithm is obtained by applying a polynomial-time state-reachability algorithm for pushdown systems [12,38,40] to the exponentially sized system obtained by representing explicitly the valuations of program variables in the k-phase translation $((P))_k$ of a given finite-data program P. □

4.3 Feasibility of phase-bounded exploration

Note that our simulation of a k-phase execution does not explicitly store the unbounded task queue. Instead of storing a multitude of possible unbounded task sequences, our simulation stores exactly k global state valuations. Accordingly, our simulation is not doomed to the unavoidable combinatorial explosion encountered by storing (even bounded-size) task queues explicitly. To demonstrate the capability of our advantage, we measure the time to verify two fabricated yet illustrative examples, comparing our bounded-phase encoding with a bounded task-queue encoding. In the bounded task-queue encoding, we represent the task-queue explicitly by an array of integers, which stores the identifiers of posted procedures.[3] When the control of the initial task completes, the program enters a loop which takes a procedure identifier from the head of the queue, and calls the associated procedure. When the queue reaches a given bound, any further posted tasks are ignored.

The first program $P_1(i)$ of Fig. 8, parameterized by $i \in \mathbb{N}$, has a single Boolean global variable `b`, i procedures named $p_1, \ldots, p_i$, which assert `b` to be **false** and set `b` to **true**, and

```
var b: bool

// for j = 1,...,i
proc p_j ()
    assert !b;
    b := true;
    return

// for j = 1,...,i
proc q_j ()
    b := false;
    return
```

```
proc main ()
    b := false;
    while ⋆ do
        if ⋆ then post p_1 ()
        else if ⋆ then post p_2 ()
        ..
        else post p_i ();

        if ⋆ then post q_1 ()
        else if ⋆ then post q_2 ()
        ..
        else post q_i ()
    return
```

Fig. 8 The program $P_1(i)$

i procedures named $q_1, \ldots, q_i$ which set `b` to **false**. Initially, $P_1(i)$ sets `b` to **false**, and enters a loop in which each iteration posts some p_j followed by some q_j. Since a q_j task must be executed between each p_j task, each of the assertions are guaranteed to hold.

Figure 9a compares the time required to verify $P_1(i)$ (using the BOOGIE verification engine [5]) for various values of i, and various bounds n on loop unrolling. Note that although every execution of $P_1(i)$ has only two phases, to explore all n loop iterations in any given execution, the size of queues must be at least $2n$, since two tasks are posted per iteration. Even for this very simple program, representing (even bounded) task-queues explicitly does not scale, since the number of possible task-queues grows astronomically as the size of task-queues grow. This ultimately prohibits the bounded task-queues encodings from exploring executions in which more than a mere few simple tasks execute. On the contrary, our bounded-phase simulation easily explores every execution up to the loop-unrolling bound in a few seconds.

To be fair, our second program P_2 of Fig. 10 is biased to support the bounded task-queue encoding. Following the example of Fig. 5c, P_2 again has a single Boolean global variable `b`, and two procedures: p_1 asserts `b` to be **false**, sets `b` to **true**, and posts p_2, while p_2 sets `b` to **false** and posts p_1. Initially, the program P_2 sets `b` to **false** and posts a single p_1 task. Again here, since a p_2 task must execute between each p_1 task, each of the assertions are guaranteed to hold.

Figure 9b compares the time required to verify P_2 for various bounds n on the number of tasks explored.[4] Note that although every execution of P_2 uses only size 1 task-queues, to explore all n tasks in any given execution, the number of phases must be at least n, since each task must execute in its own phase. Although verification time for the bounded-phase encoding does increase with n faster than the bounded task-queue encoding—as expected—due to additional copies of the global valuation, and more deeply

[3] For simplicity our examples do not pass arguments to tasks; in general, one should also store in the task-queue array the values of arguments passed to each posted procedure.

[4] The number n of explored tasks is controlled by limiting the number of loop unrollings in the bounded task-queue encoding, and limiting the recursion depth, and phase-bound, in the bounded-phase encoding.

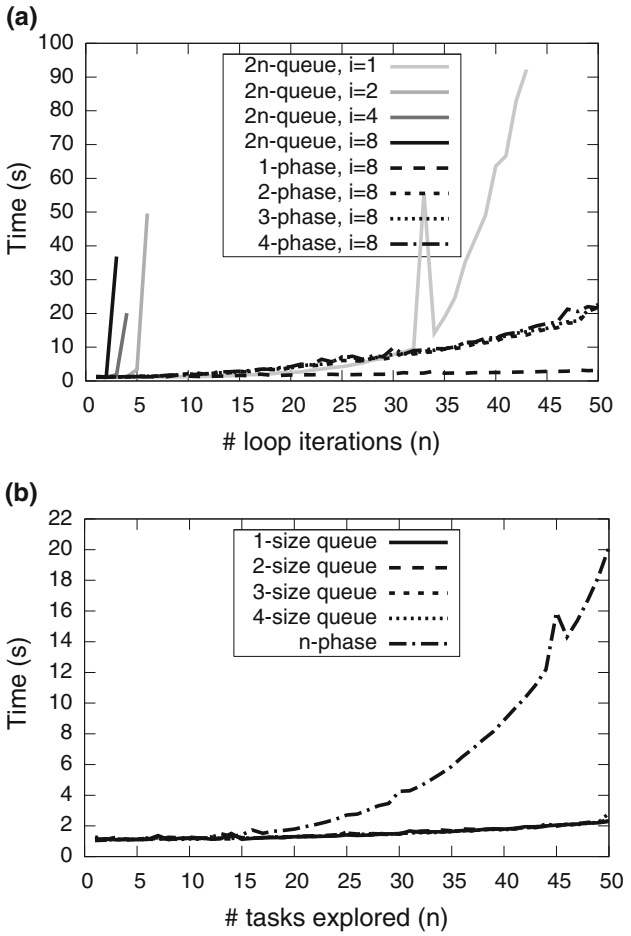


Fig. 9 Time required to verify **a** the program $P_1(i)$, and **b** the program P_2 with the BOOGIE verification engine using various encodings (bounded queues, bounded phase), and various loop unrolling bounds, where all procedures are *in-lined*. Time-out is set to 100 s

```
var b: bool

// for j = 1,...,i
proc p_j ()
    assert !b;
    b := true;
    return

// for j = 1,...,i
proc q_j ()
    b := false;
    return

proc main ()
    b := false;
    while ⋆ do
        if ⋆ then post p_1 ()
        else if ⋆ then post p_2 ()
        ..
        else post p_i ();

        if ⋆ then post q_1 ()
        else if ⋆ then post q_2 ()
        ..
        else post q_i ()
    return
```

```
var b: bool

proc main ()
    b := false;
    post p_1 ()
    return

proc p_1 ()
    assert !b;
    b := true;
    post p_2 ();
    return

proc p_2 ()
    b := false;
    post p_1 ()
    return
```

Fig. 10 The program $P_2(i)$

in-lined procedures, the verification time remains manageable. In particular, the time does not explode uncontrollably: even 50 tasks are explored in under 20 s.

5 Phase-bounding for multi-processor programs

Though state-reachability under a phase bound is immediately and succinctly reducible to sequential program analysis for single-processor programs, the multi-processor case is more complicated. The added complexity arises due to the many orders in which tasks on separate processors can contribute to others' task-queues. As a simple example, consider the possible bounded-phase executions of Fig. 5b with four processors, A, B, C, and D. Though B's tasks $B_1, \ldots, B_n$ must be executed in order, and C's tasks $C_1, \ldots, C_n$ must also be executed in order, the order of D's tasks are not predetermined: the arrival order of D's tasks depends on how B's and C's tasks *interleave*. Suppose for instance B_1 executes to completion before C_1, which executes to completion before B_2, and so on. In this case D's tasks arrive to D's queue, and ultimately execute, in the index order $D_1, D_2, \ldots$ as depicted. However, there exist executions for every possible order of D's tasks respecting $D_1 < D_3 < \ldots$ and $D_2 < D_4 < \ldots$ (where $<$ denotes an ordering constraint)—many possible orders indeed!

Here we show, in Sect. 5.1, that due to the capability of such unbounded interleaving, the problem of state-reachability under a phase-bound is undecidable for even finite-data multi-processor programs. Wishing still for algorithmic analyses, we attempt to leverage existing strategies for dealing with the complexity arising from processor interleaving. For this we recall delay-bounded scheduling [15] in Sect. 5.2, following which in Sects. 5.3 and 5.4 we demonstrate an effective "depth-first" delay-bounded scheduler which gives rise to an efficient encoding, again into sequential program analysis.

5.1 Undecidability of multi-processor phase-bounding

Theorem 2 *The phase-bounded state-reachability problem for finite-data multi-processor programs is undecidable.*

Our reduction-based proof makes use of the syntactic extensions of Appendix A.

Proof We proceed by reduction from Post's correspondence problem [35]: given words $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_n \in \Sigma^*$ of a finite alphabet Σ such that $|\Sigma| \geq 2$, find a sequence $i_1 \ldots i_k \in \{1 \ldots n\}^*$ such that $\alpha_{i_1}\alpha_{i_2} \ldots \alpha_{i_k} = \beta_{i_1}\beta_{i_2} \ldots \beta_{i_k}$. For this problem instance we build in Fig. 11 the finite-data asynchronous message-passing program with four processors ρ_0, ρ_1, ρ_2, and ρ_3.

Initially, the `main` procedure is pending on processor ρ_0. In each loop iteration, `main` chooses a branch corresponding to an index $i \in \{1 \ldots n\}$ and posts each symbol of α_i individually and in order to ρ_1, and each symbol of β_i individually and in order to ρ_2. In this way, `main` sends to ρ_1 the sequence

```
var turn: {L,R}
var prev: Σ
var done[{L,R}]: 𝔹
var empty: 𝔹

// initially on each processor
turn = L
  and done[L] = done[R] = false
  and empty = true

// check reachability to
done[L] = done[R] = true and empty
     = false

proc hold (var side: {L,R}, x: Σ)
  post ρ3 check (side, x);
  return
```

```
proc check (var side: {L,R}, x: Σ)
  assume side = turn;
  assume !done[side];
  assume turn = R => prev = x;
  prev := x;
  empty := false;
  if turn = L then
    turn := R
  else
    turn := L;
  return

proc last (var side: {L,R})
  post ρ3 last' (side);
  return

proc last' (var side: {L,R})
  assume turn = L;
  done[side] := true;
  return
```

```
proc main ()
  while ⋆ do
    if ⋆ then
      post ρ1 hold (L, α1(1));
      .. ;
      post ρ1 hold (L, α1(|α1|));
      post ρ2 hold (R, β1(1));
      .. ;
      post ρ2 hold (R, β1(|β1|));

    ... else
      post ρ1 hold (L, αn(1));
      .. ;
      post ρ1 hold (L, αn(|αn|));
      post ρ2 hold (R, βn(1));
      .. ;
      post ρ2 hold (R, βn(|βn|));

  post ρ1 last (L);
  post ρ2 last (R)
  return
```

Fig. 11 Encoding Post's correspondence problem into the single-phase bounded executions of a finite-data asynchronous message-passing program

$\alpha_{i_1} \dots \alpha_{i_k}$, and to ρ_2 the sequence $\beta_{i_1} \dots \beta_{i_k}$ in k loop iterations, each terminated by a `last` message. Each instance of the `hold` tasks which execute on ρ_1 and ρ_2 simply propagate their symbol to ρ_3. Using the global variable `turn`, ρ_3 ensures that he only sees symbols sent from ρ_1 and ρ_2 in alternating order, starting with ρ_1. Using the global variable `prev`, ρ_3 ensures that each symbol of $\beta_{i_1} \dots \beta_{i_k}$ sent from ρ_2 matches the previous symbol of $\alpha_{i_1} \dots \alpha_{i_k}$ sent from ρ_1. Finally, if ρ_3 receives both terminating `last'` messages from ρ_1 and ρ_2 before another `L`-turn, then he has successfully checked the equality of sequences $\alpha_{i_1} \dots \alpha_{i_k} = \beta_{i_1} \dots \beta_{i_k}$. Thus, when `down[L]` and `done[R]` are both set to true, this means `main` was able to guess a solution $i_1 \dots i_k$ to the correspondence problem. □

Note that Theorem 2 holds independently of whether memory is shared between processors: the fact that a task-queue can store any possible (unbounded) shuffling of tasks posted by the two processors ρ_1 and ρ_2 lends the power to simulate Post's correspondence problem [35]. Furthermore, while our proof used four processors $\rho_0 \dots \rho_3$ for clarity, by merging functionality, e.g., of ρ_0 with ρ_1 and ρ_2 with ρ_3, a two-processor reduction is also possible; the universal computing power arises from the order-preserving interleaving of as few as two processors.

5.2 Delay-bounded scheduling

Theorem 2 insists that phase-bounding alone will not lead to the elegant encoding to sequential programs which was possible for single-processor programs. If that were possible, then the translation from a finite-data program would lead to a finite-data sequential program, and thus a decidable state-reachability problem. Since a precise algorithmic solution to bounded-phase state-reachability is impossible for multi-processor programs, we resort to a further incremental yet orthogonal under-approximation, which limits the number of considered processor interleavings. The following development is based on delay-bounded scheduling [15].

Roughly speaking, the semantics of an asynchronous concurrent program is dependent on the timing with which processors take their respective steps. This concept of timing is often captured by considering a *scheduler* that decides which processor's turn it is at any moment. Since the factors which determine the scheduler are often deemed too complex to model, many approaches to program analysis consider an abstract, nondeterministic scheduler, which allows *any* processor to take a step at any moment. Concordantly, the very large number of possible process/thread interleavings such analyses must consider is a key source of analysis complexity.

Delay-bounding [15] attempts to dodge much of the interleaving complexity by considering much fewer interleavings, by systematically limiting the amount of nondeterminism added to a given deterministic scheduler M. In the simplest case, when the bound on delays is zero, only a single schedule is explored, as M is deterministic.[5] However, when M is augmented with $k > 0$ delays, M gets to choose at up to k arbitrary scheduling points throughout each execution to postpone its next-scheduled task until some later time, picking the following task instead. So long as the base deterministic scheduler M and delaying policy are chosen well, any execution possible with a completely nondeterministic scheduler will be possible with M using some number of delay operations.

We formalize this idea by defining a *delaying scheduler*

$$M = \langle D, \texttt{empty}, \texttt{enabled}, \texttt{step}, \texttt{delay} \rangle ,$$

[5] For simplicity, this description supposes the scheduler can be the only source of program nondeterminism.

$$\textsc{Delay}\quad \frac{m_2 = \mathsf{delay}(m_1, \rho, \xi)}{\langle \rho, \xi, m_1 \rangle \to \langle \rho, \xi, m_2 \rangle}$$

Fig. 12 The delay operation, which updates the current scheduler state from m_1 to m_2 as a function of the processor configuration map ξ and currently executing processor ρ

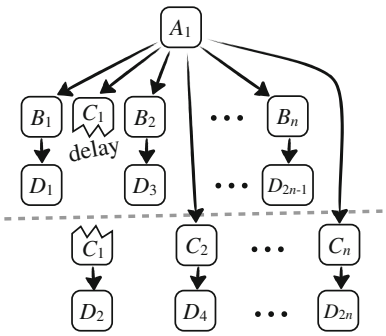


Fig. 13 A 2-phase delaying execution varying the 1-phase execution of Fig. 5b

as a scheduler $\langle D, \texttt{empty}, \texttt{enabled}, \texttt{step} \rangle$, along with a function $\texttt{delay} : (D \times \texttt{Pids} \times (\texttt{Pids} \to \texttt{Pconfigs})) \to D$. Furthermore, we extend the transition relation of Fig. 4 with a postponing rule of Fig. 12 which we henceforth refer to as a *delay (operation)*, saying that processor ρ is delayed. Note that a delay operation may or may not change the set of enabled processors in any given step, depending on the scheduler. Thinking of `delay` operations as invoked outside of the scheduler, we say a delaying scheduler M is *deterministic* when M's base scheduler (i.e., without the `delay` operation) is deterministic. A delaying scheduler is *delay-accessible* when for every configuration c_1 and non-idle or task-pending processor ρ, there exists a sequence $c_1 \to \ldots \to c_j$ of DELAY-steps such that ρ is enabled in c_j. Given executions h_1 and h_2 of (delaying) schedulers M_1 and M_2 resp., we write $h_1 \sim h_2$ when h_1 and h_2 are identical after projecting away delay operations.

Definition 3 An execution with at most k delay operators is called *k-delay*.

Consider again the possible executions of Fig. 5b, but suppose we fix a deterministic scheduler M which without delaying would execute D's tasks in index order: $D_1, D_2, \ldots$; furthermore suppose that delaying a processor ρ in phase i causes M to execute the remaining phase i tasks of ρ in phase $i+1$, while keeping the tasks of other processors in their current phase. Without using any delays, the execution of Fig. 5b is unique, since M is deterministic. However, as Fig. 13 illustrates, using a single delay, it is possible to also derive the order $D_1, D_3, \ldots, D_{2n-1}, D_2, D_4, \ldots, D_{2n}$ (among others): simply delay processor C once before C_1 posts D_2. Since this forces the D_{2i} tasks posted by each C_i to occur in the second phase, it follows they must all happen after the D_{2i-1} tasks posted by each B_i.

Lemma 4 (Completeness) *Let M be any delay-accessible scheduler. For every execution h of a program P, there exists an M-execution h' and $k \in \mathbb{N}$ such that h' is a k-delay execution and $h' \sim h$.*

Note that Lemma 4 holds for *any* delay-accessible scheduler M—even deterministic schedulers.

5.3 The multi-processor depth-first scheduler

As it turns out there is one particular scheduler M_{dfs} for which we know a convenient sequential encoding. Here we define a deterministic, non-blocking, delay-accessible delaying scheduler M_{dfs} which though perhaps odd from an operational point of view, has a very useful application: given a multi-processor message-passing program P, the phase- and delay-bounded executions of P according to M_{dfs} are simulated by executions of a sequential program P'; furthermore, P' is obtained by a simple code-to-code translation of P which does not explicitly represent pending-task queues.

Let U be a set of identifiers uniquely identifying each task along an execution with a single initially pending task $u_0 \in U$. Ourscheduler keeps a monotonically increasing phase number $i \in \mathbb{N}$, along with an ordered task-posting tree T over nodes U, a completion-labeling $\surd : U \to \mathbb{B}$, and a phase-labeling $\Phi : U \to \mathbb{N}$. Initially the tree contains a single node u_0, with $\Phi(u_0) = 0$ and $\surd(u_0) = \texttt{false}$. As additional tasks are posted, we add them as children of the posting task, in the order they are posted. Normally, the scheduler allows tasks to execute to completion; when a task does complete, the scheduler marks it as completed. When choosing the next task to execute, our scheduler selects the smallest—in depth-first order over the task-posting tree—unexecuted task in the current phase; if there are no non-completed tasks in the current phase, the scheduler moves to the next phase. In this way, the scheduler executes all tasks in phase order, and same-phase tasks in depth-first order of the task-posting tree.

To implement delaying, our scheduler also keeps a phase-delay counter $\Delta(\rho) : \mathbb{N}$ for each processor ρ. Supposing an executing task u has phase-i on a processor whose phase-delay counter has current value j, the task u is treated as though it is in phase $i + j$. When a processor is delayed, its phase-delay counter is simply incremented; the effect is to shift all following tasks on the given processor one additional phase later. Delaying causes the currently executing task to be interrupted and resumed in the following phase.

Formally the *depth-first scheduler*

$$M_{\text{dfs}} = \langle D, \texttt{empty}, \texttt{enabled}, \texttt{step}, \texttt{delay} \rangle$$

is defined over scheduler objects $m = \langle i, T, \surd, \Phi, \Delta \rangle \in D$ as described above; the initial object is defined as $\texttt{empty} = \langle 0, T_0, \surd_0, \Phi_0, \Delta_0 \rangle$, where T_0 is the single-node tree with root u_0, $\surd_0(u) = \texttt{false}$ and $\Phi_0(u) = 0$ for all $u \in U$, and $\Delta_0(\rho) = 0$ for all $\rho \in \texttt{Pids}$. The $\texttt{enabled}(\langle i, T, \surd, \Phi, \Delta \rangle, \xi)$ operation uniquely returns the processor identifier ρ of the smallest task u—according to the depth-first order of T—such that $\Phi(u) + \Delta(\rho) = i$. The $\texttt{step}(\langle i_1, T_1, \surd_1, \Phi_1, \Delta \rangle, \xi_1, \xi_2)$ operation for a transition $\tau = \xi_1(\rho) \to S\xi_2(\rho)$ returns the object $\langle i_2, T_2, \surd_2, \Phi_2, \Delta \rangle$ such that

- If τ is a COMPLETE-step of task u, then $\surd_2 = \surd_1(u| \to \texttt{true})$; otherwise $\surd_2 = \surd_1$.
- If τ is a POST- or SELF- POST-step of task u posting task u', then T_2 is obtained from T_1 by adding to u new a rightmost child u', and $\Phi_2 = \Phi_1(u' \mapsto \Phi_1(u) + \Delta(\rho))$; otherwise, $T_2 = T_1$ and $\Phi_2 = \Phi_1$.
- If there no longer exists a non-completed task u on some processor ρ' such that $\Phi_2(u) + \Delta(\rho') = i_1$ then $i_2 = i_1 + 1$; otherwise $i_2 = i_1$.

The $\texttt{delay}(\langle i_1, T, \surd, \Phi, \Delta_1 \rangle, \rho, \xi)$ operation returns $\langle i_2, T, \surd, \Phi, \Delta_2 \rangle$ such that

- $\Delta_2 = \Delta_1(\rho \mapsto \Delta_1(\rho) + 1)$ increments Δ_1's mapping for processor ρ.
- If there no longer exists a non-completed task u on some processor ρ' such that $\Phi(u) + \Delta_2(\rho') = i_1$, then $i_2 = i_1 + 1$; otherwise $i_2 = i_1$.

According to our definition, M_{dfs} repeatedly picks a unique processor ρ to execute such that ρ is non-idle or has pending tasks, and ρ's first non-idle or pending task u has the lowest offsetted phase $\Phi(u) + \Delta(\rho)$ of any task on any processor.

Note that M_{dfs} is a deterministic delaying scheduler which executes all tasks of a given phase before any task of a subsequent phase. Since M_{dfs} must pick an enabled task so long as there are pending tasks on some processor, M_{dfs} is non-blocking. Finally, since for any $i = \Phi(u) + \Delta(\rho)$, repeatedly delaying every other processor $\rho' \neq \rho$ eventually increments $\Delta(\rho')$ such that for any pending u' on ρ', $\Phi(u') + \Delta(\rho') > i$, M_{dfs} is delay accessible.

On non-delaying executions, M_{dfs} essentially performs a phase-by-phase depth-first traversal of the task-posting tree T—a tree which includes tasks across all processors.

5.4 A sequential encoding of the depth-first scheduler

Note that M_{dfs} is deterministic, non-blocking, and delay-accessible. Essentially, determinism allows us to encode the scheduler succinctly in a sequential program; the non-blocking property ensures this scheduler does explore some execution, rather than needlessly ceasing to continue; delay-accessibility combined with Theorem 4 ensure the scheduler is complete in the limit. Figure 14 lists a code-to-code translation which encodes bounded-phase and bounded-delay exploration of a given program according to the M_{dfs} scheduler as a sequential program.

```
1  // translation of decl. var g: T
2  var G[k+d]: T
3  var shift[Pids][k], delay: d
4  var ancestors[Pids][k+d]: 𝔹
5
6  // translation of declaration
7  // proc p (var l: T) s
8  proc p (var l: T, pid: Pids, phase: k) s
9
10 // translation of lvalue g
11 G[ phase + shift[pid][phase] ]
12
13 // code to be sprinkled throughout
14 while ⋆ and delay < d do
15    shift[pid][phase]++; delay++
16
17 // translation of call x := p e
18 call x := p (e,pid,phase)
19
20 // translation of post ρ p e
21 let p = phase + shift[pid][phase] in
22 let p'=p+(if ancestors[ρ][p] then 1 else 0) in
23 if p' < k then
24    ancestors[ρ][p'+shift[ρ][p']] := true;
25    call p (e, ρ, p');
26    ancestors[ρ][p'+shift[ρ][p']] := false
27
28 // the entry procedure for ((P))^dfs_{k,d}
29 proc Top ()
30    let G0[k+d]: T in
31    let shift0[Pids][k]: d in
32    G := G0;
33    shift := shift0;
34    delay := 0;
35    call p0(⋆,0);
36    assume ∀i.0<i<k+d ⇒ G[i-1] = G0[i];
37    assume ∀i.0<i<k ⇒ shift[i-1]=shift0[i];
38    // executions to this point are
39    // known to be valid
40    return
```

Fig. 14 The k-phase d-delay translation $((P))^{\text{def}}_{k,d}$ of a multi-processor message-passing asynchronous program P with entry procedure p_0. Here we freely make use of the syntactic extensions of Appendix A

Our translation closely follows the single-processor translation of Sect. 4, the key differences being:

- the `phase` of a posted task is not necessarily incremented, since posted tasks may not have same-processor ancestors in the current phase; the `ancestors` array records for each processor ρ and phase i, whether the

current task was transitively posted by a phase i task on processor ρ;
- at any point, the currently executing task may increment a `delay` counter, causing all following tasks on the same processor to shift forward one additional phase.

As the global values reached by each processor at the end of each phase $i-1$ must be ensured to match the initial values of phase i, for $0 < i < k+d$, so must the values for the phase-delay counter (the Δ of Sect. 5.3, named `shift` in Fig. 14): an execution is only valid when for each processor $\rho \in$ `Pids` and each phase $0 < i < k$, $\texttt{shift}[\rho][i-1]$ matches the initial value of $\texttt{shift}[\rho][i]$.

Note that even though our sequential translation for the (zero-delay) depth-first scheduler executes same-phase tasks according to depth-first traversal of the task-posting tree, the resulting task execution order always corresponds to *some* valid order. Taking a slightly counterintuitive example, consider a 1-phase computation with processors A, B, and C starting with task A_1 which posts B_1 and C_1, where B_1 posts C_2. The sequentialization $((P))^{\text{def}}_{k,0}$ without delays simulates an execution where C_2 executes before C_1. Though at first sight this may seem like an invalid interleaving, it is in fact valid, since B_1, running on a different processor than A_1, once posted, can post C_2 before A_1 has the chance to post C_1, thus C's task-queue can legitimately contain C_2 before C_1.

Lemma 5 *A global valuation g is reachable in a k-phase d-delay M_{dfs}-execution of a multi-processor program P if and only if g is reachable in a completed execution of $((P))^{\text{def}}_{k,d}$.*

Proof By extending Lemma 2 from the single-processor case, the case where $d = 0$ follows easily, since $\texttt{shift}[\rho][i]$ is always zero, and the scheduler M_{dfs} executes tasks of the same phase in depth-first order—the same order in which same-phase tasks execute in the sequential program $((P))^{\text{def}}_{k,d}$. In the presence of delays, our translation maintains the order of M_{dfs} by incrementing the `shift` variable of a given processor, thus postponing to a later phase all subsequently posted tasks by said processor. As is done with the global values guessed at the beginning of each phase, the guessed total number of delays per-processor used during all previous phases is validated on Line 37. □

As is the case for our single-processor translation, our simulation does not explicitly store the unbounded tasks queue, and is not doomed to combinatorial explosion faced by storing task-queues explicitly.

Theorem 3 *The phase and delay-bounded state-reachability problem for finite-data programs is EXPTIME-complete.*

Proof As multi-processor programs subsume single-processor programs, the lower bound of Theorem 1 yields EXPTIME-hardness. Similarly, by the same reasoning used in the proof of Theorem 1, our translation $((P))^{\text{def}}_{k,d}$ reduces program state-reachability to pushdown reachability in exponentially sized pushdown systems. □

6 An implementation of phase-bounded exploration

To demonstrate the effectiveness of our phase-bounded sequentialization, we have implemented a code-to-code translation following the translation of Sect. 5.4, using our translation as a key step in a bug-detection algorithm for asynchronous message-passing programs.

6.1 A bug-detection algorithm

The input to our bug-detection algorithm is a multiprocessor asynchronous message-passing program written in the Boogie intermediate verification language [5], extended with an asynchronous call statement to model procedure **post**ing. Each input program declares an uninterpreted **pid** type to encode processor identifiers, and each processor global state variable of type T becomes a map in the input Boogie program from **pid** to T. Each procedure in the input program takes a `self` parameter of type **pid**, and we ensure that all global variable accesses are indexed by `self`; this ensures that the tasks of each processor only access its own processor's global storage. Though it is possible to encode a statically configured network in the input message-passing program, our input programs each assume an arbitrary number of processors which send messages following an arbitrary relation between neighboring processors.

Following the translation of Sect. 5.4, our algorithm first constructs a sequential Boogie program from the given asynchronous program. For simplicity, our prototype implementation does not maintain the `ancestors` map of Fig. 14; instead we conservatively increment the phase of every single message sent in a message chain, regardless of whether or not there existed a same-phase ancestor of the given target processor. Thus, in principle our implementation will encode fewer behaviors than the translation of Fig. 14 for the same phase and delay bounds. Our translation ensures that the target sequential Boogie program can violate an assertion only if a state which violates an assertion in the source asynchronous Boogie program is reachable, according to the depth-first scheduler with given phase and delay bounds.

To search for assertion-violating executions in the resulting sequential Boogie program, we leverage the Corral reachability engine [32].

6.2 Case study: bug detection in network algorithms

The code of Fig. 15 lists a canonical implementation of a textbook asynchronous algorithm to calculate the spanning tree

```
type pid
const root: pid
var parent[pid]: pid
var reported[pid]: bool

proc Main ()
    assume (∀p: pid. ¬reported[p]);
    post root Search (root, root);
    return

proc Search (self: pid, sender: pid)
    var neighbor: pid
    var sent[pid]: bool

    if ¬reported[self] then

        if self ≠ sender then
            call Parent (self, sender);

        // broadcast the Search message
        while ⋆ do
            let neighbor: pid in
            assume neighbor ≠ self;
            assume neighbor ≠ sender;
            assume neighbor ≠ root;
            assume ¬sent[neighbor];
            sent[neighbor] := true;
            post neighbor Search
                        (neighbor, self);
    return

// set the parent link
proc Parent (self: pid, p: pid)
    parent[self] := p;
    reported[self] := true;
    assert ¬cycle(parent, reported);
    return
```

Fig. 15 A possible implementation of the spanning tree calculation. The predicate `cycle` holds when there exists a sequence $p_1, p_2, \ldots, p_i$ of `pids` such that `reported`$[p_j]$ and `parent`$[p_j] = p_{j\%i+1}$ for $1 \leq j \leq i$

of an arbitrary network [33]. Starting from some `root` node, `Search` messages are broadcast to all nodes connected to the root. Each time a process node receives a `Search` message for the first time, the process makes a synchronous procedure call to `Parent`, which sets its `parent` link to the process identifier of the sender of the `Search` message, and sets `reported` to **true**. Then, the process node propagates the `Search` message, sending it to all of its neighbors; we have modeled this broadcast with a nondeterministic loop which chooses arbitrary nodes, other than the current process node, its sender, and the root, such that each broadcast posts at most one message to each other process. The `Parent` procedure contains an assertion which is violated if the `parent` relation ever becomes cyclic. Note that the canonical correctness criterion for the network spanning tree calculation is precisely that a tree is constructed, and thus there are no cycles.

```
proc Search (self: pid, sender: pid)
    ...
    if ¬reported[self] then
        post self Parent (self, sender);

        // broadcast the Search message
        ...

    return
```

Fig. 16 A second possible implementation of the spanning tree calculation, in which the `Parent` procedure is called asynchronously

The implementation of Fig. 15 cannot violate its assertion of `parent`-acyclicity. The essential argument supporting this claim is that each processor ignores (i.e., does not follow the **if** branch) every `Search` message which it receives, following the first one, in which the `parent` link and `reported` bit are set synchronously. Crucial to this argument is that the `Parent` procedure is called synchronously, and it is not hard to imagine alternative implementations whose correctness is not so evident. Consider for example the implementation of the `Search` procedure of Fig. 16, in which the `Parent` procedure is called asynchronously.

Although the implementation of Fig. 16 also cannot violate the assertion when messages are processed in FIFO order, the correctness argument is more subtle. Note that if FIFO message order is not necessarily preserved, our model, like that of asynchronous programs [39], would consider that pending messages may be handled in any possible order; in this unordered setting, an assertion violation is indeed possible, since a pending `Parent` call on some processor p_1 may remain pending, while in the meantime a descendent p_2 of p_1's `Search` broadcast may send, cyclicly, another `Search` message to p_1; since p_1's initial asynchronous `Parent` call may remain pending, while the asynchronous `Parent` call initiated by p_2's `Search` executes, setting p_1's `parent` link to p_2, thus introducing the capability of cycle formation on the `parent` link between p_1 and p_2.

However, while this behavior is possible under the premise that message buffers are unordered, it is prohibited when message buffers are ordered. The reason is quite simple: at any moment when processor p_1 processes the `Search` message broadcast from some descendant p_2, p_1 is guaranteed to have already executed its initially pending `Parent` procedure, since it was already pending at the time that p_1 broadcast its `Search` messages to its neighbors. In fact, one can prove that ordered message buffers rule out not only the previously mentioned behavior, but any behavior leading to an assertion violation in the implementation of Fig. 16.

Still, it is not difficult to imagine yet other alternative implementations which can violate the assertion even when message buffers are assumed FIFO. Figure 17 lists such an implementation of `Search`. In this version, the sender of the `Search` message is responsible for asynchronously

```
proc Search (self: pid, sender: pid)
    ...
    if ¬reported[self] then

        // broadcast the Search message
        while ⋆ do
            let neighbor: pid in
            ...

            post neighbor Search (neighbor,
                self);
            post neighbor Parent (neighbor,
                self);

    return
```

Fig. 17 A third possible implementation of the spanning tree calculation, in which the `Parent` procedure is called asynchronously by the *sender* of the `Search` message

calling the `Parent` procedure of the target processor. In this case, assertion-violating executions are possible in which a descendent's `Search` and `Parent` messages both arrive in a processors message buffer, positioned after the original `Search` and corresponding `Parent` messages. Then the `Parent` message originating from the descendent can erroneously update the given processors `parent` link to point to said descendent. Note that if the `Search` and `Parent` messages were posted in the reverse order, i.e., `Parent` then `Search`, this error is not possible.

Important to note here is the subtlety of asynchronous programming; even the simplest asynchronous network algorithms such as spanning tree calculation can be made correct or incorrect according to slight reordering in message sending and receiving.

6.3 Experimentation

We have applied the bug detection algorithm outlined in Sect. 6.1 to the erroneous spanning tree algorithm variations just described, as well as two more advanced algorithms: the Bellman–Ford algorithm for computing the distance of each node in a network from a given root, and the breadth-first spanning tree algorithm for computing a spanning tree in which each node is connected by a minimal-distance path from a given root [33]—each with a subtle, injected bug. The Boogie source of these examples are listed in full in Appendix 1. In each case we invoked the Corral reachability engine [32] on a 5-phase sequential program translation with a recursion-depth bound of 5. All of our experiments used a delay bound of 0. Our algorithm discovers the assertion violations in 20, 10, and 5 s, resp., for the buggy spanning tree, Bellman–Ford, and breadth-first spanning tree algorithms. Additionally, after exploring all possible 5-phase executions of the (correct for FIFO buffers) spanning tree variation of Fig. 16 up to the recursion depth of 5—of an arbitrarily connected network of arbitrarily many processors—our algorithm concludes that the assertion violation is not possible (up to recursion depth 5, and in 5 phases) after 199 s. Though not as effective at determining the absence of assertion violations, our prototype demonstrates that our phase-bounded sequentialization is a viable approach for discovering subtle programming bugs in asynchronous message-passing systems with ordered message buffers.

7 Related work

Our work follows the line of research on compositional reductions from concurrent to sequential programs. The initial so-called "sequentialization" [37] explored multi-threaded programs up to one context-switch between threads, and was later expanded to handle a parameterized amount of context-switches between a statically determined set of threads executing in round-robin order [31,36]. Contrary to these approaches which rely on nondeterministically guessing the states reached by other threads, La Torre et al. [28] demonstrated a reduction from bounded-context round-robin to sequential program analysis which avoids guessing, and thus the exploration of unreachable states. La Torre et al. [30] later extended the approach to handle programs parameterized by an unbounded number of statically determined threads by the computation of linear interfaces [29], and shortly after, Emmi et al. [15] further extended these results to handle an unbounded amount of dynamically created tasks, which besides applying to multi-threaded programs, naturally handles asynchronous event-driven programs [39]. Bouajjani et al. [10] pushed these results even further to a sequentialization which attempts to explore as many behaviors as possible within a given analysis budget, and La Torre and Parlato [25] demonstrate a sequential reduction by *scope bounding*, which captures more concurrent behaviors than context bounding.

While the previously mentioned sequentializations provide reductions only from state-reachability problems (e.g., including violations to safety properties), Atig et al. [4] and Emmi and Lal [14] have recently demonstrated sequentializations which even reduce the detection of liveness property violations (e.g., non-termination) in multi-threaded and asynchronous programs to sequential program state-reachability. All of these sequentializations necessarily do provide a bounding parameter which limits the amount of interleaving between threads or tasks, but none are capable of precisely exploring tasks in creation order, which is abstracted away from their program models [39]. While Kidd et al. [24] and Emmi et al. [16] have developed sequentializations which are sensitive to task priorities, their reductions assume a finite number of priorities, and thus cannot capture unbounded queues.

In closely related works, La Torre et al. [27] and Heußner et al. [21] develop decidable "context-bounded" analyses of shared-memory multi-pushdown systems communicating with message-queues. According to this approach, one "context" involves a single process reading from its queue, and posting to the queues of other processes, and the number of contexts per execution is bounded. Our approach differers along several points.

1. There are programs whose fixed-phase executions require an unbounded number of contexts to capture; Fig. 5b serves as an example: a single phase captures the execution order $D_1 D_2 D_3 \ldots D_{2n}$ on processor D, which requires $n + 2$ contexts (specifically, of processors $A(BC)^n D$) to capture.
2. While each k-context execution is a k-phase execution, since the ith context creates messages of phase at most $i + 1$, not all k-phase executions occur in k contexts, since, according to Theorem 2, k-phase bounded state-reachability is undecidable even for finite-data programs.
3. Any k-context execution of a program with n processors is a $(nk)^k$ delay execution according to our depth-first scheduler, since each context may occur across k phases, forcing each processor to delay its tasks at most k times, per context.
4. While the context-bounded state-reachability problem for finite-data programs is 2EXPTIME-complete [3,26], Theorem 3 demonstrates that phase and delay-bounded state-reachability, with the depth-first scheduler, is only EXPTIME-complete. Note that since every k-context execution (and many more, following our first point) is captured by a k-phase $\mathcal{O}(k^k)$-delay execution, phase and delay-bounding subsume context-bounding, capturing a superset of behaviors, in the same 2EXPTIME worst-case complexity.
5. Since La Torre et al. [27] and Heußner et al. [21]'s setting does not allow a process to post messages to its own queue, simulating k-phase executions of single-processor programs requires at least $2k$ contexts, i.e., of two separate processors emulating a single processor.
6. Though phase-bounding leads to a convenient sequential encoding with easily implementable analysis algorithms, we are unaware whether a similar encoding is possible for context-bounding systems communicating with message queues.

Otherwise, Boigelot and Godefroid [6] and Bouajjani and Habermehl [8] have proposed analyses of message-passing programs by computing explicit finite symbolic representations of message-queues, which generally represent a superset of possible queue contents, leading to overapproximation-based analyses. Unlike these approaches, our sequentialization does not represent queues explicitly, and we do not restrict the content of queues to conveniently representable descriptions. Furthermore, reduction to sequential program analyses is easily implementable, and allows us to leverage highly developed and optimized program analysis tools. Finally, any analysis of *asynchronous programs* with dynamic task-creation [14–16,18,19,23,39] can be seen as overapproximating fifo task-buffer semantics, since no order between pending tasks is enforced.

8 Conclusion

By introducing a novel phase-based characterization of message-passing program executions, we enable bounded program exploration which is not limited by message-queue capacity nor the number of processors. We show that the resulting phase-bounded analysis problems can be solved by concise reduction to sequential program analysis. Preliminary evidence suggests our approach is at worst competitive with known task-order respecting bounded analysis techniques, and can easily scale where those techniques quickly explode.

Acknowledgments We graciously thank Constantin Enea, Cezara Dragoi, Pierre Ganty, Salvatore La Torre, and the anonymous TACAS and STTT reviewers for helpful feedback.

Appendix A: Syntactic extensions used in our translations

The following syntactic extensions are reducible to the original program syntax of Sect. 2.1. Here we freely assume the existence of various type- and expression-constructors. This does not present a problem since our program semantics does not restrict the language of types nor expressions.

Multiple types. Multiple type labels $T_1, \ldots, T_j$ can be encoded by systematically replacing each T_i with the sum-type $T = \Sigma_{i=1}^{j} T_i$. This allows local and global variables with distinct types.

Multiple variables. Additional variables $\mathtt{x}_1$: T_1, …, $\mathtt{x}_j$: T_j can be encoded with a single record-typed variable $\mathtt{x}$: T, where T is the record type

$$\{\mathtt{f}_1 : T_1, \ldots, \mathtt{f}_j : T_j\}$$

and all occurrences of $\mathtt{x}_i$ are replaced by $\mathtt{x}.\mathtt{f}_i$. When combined with the extension allowing multiple types, this allows each procedure to declare any number and type of local variable parameters, distinct from the number and type of global variables.

Local variable declarations. Additional (non-parameter) local variable declarations **var** $\mathtt{l}$': T to a procedure p can be

encoded by adding `l'` to the list of parameters, and systematically adding an initialization expression (e.g., the choice expression *, or **false**) to the corresponding position in the list of arguments at each call site of p to ensure that `l'` begins correctly (un)initialized.

Unused values. Call assignments **call** x := p e, where `x` is not subsequently used, can be written as **call** _:= p e, where _: T is an additional unread local variable, or simpler yet as **call** p e.

Unused branches. **if** e **then** s **else skip** is abbreviated by **if** e **then** s.

Increment. Increment operations x++ are encoded as $x := x + 1$.

Let bindings. Let bindings of the form **let** x: $T = e$ **in** can be encoded by declaring `x` as a local variable **var** x: T immediately followed by an assignment x := e. This construct is used to explicate that the value of x remains constant once initialized. The binding **let** x: T **in** is encoded by the binding **let** x: T = * **in** where * is the choice expression.

Arrays. Finite arrays with j elements of type T can be encoded as records of type $\{\mathtt{f}_1 : T, \ldots, \mathtt{f}_j : T\}$, where $\mathtt{f}_1 \ldots \mathtt{f}_j$ are fresh names. Occurrences of terms a[i] are replaced by $\mathtt{a.f}_i$, and array-expressions $[e_1, \ldots, e_j]$ are replaced by record-expressions $\{\mathtt{f}_1 = e_1, \ldots, \mathtt{f}_j = e_j\}$.

Appendix B: Source code of programs analyzed in Sect. 6

```
// Cyclicity Predicate
function is_cycle(
   parent: [pid] pid, reported: [pid] bool,
   c: [int] pid, size: int )
   returns (bool)
{
   size > 1

   // only consider nodes who have set their parent
        links
   && ( forall i: int :: i >= 0 && i < size ==> reported
        [ c[i] ])

   // the parent link of each node points to the
        previous
   && ( forall i: int :: i > 0 && i < size ==> (parent[
        c[i] ] == c[i-1]) )

   // the parent link of the first node points to the
        last
   && parent[ c[0] ] == c[size-1]
}
procedure assert_cycle ( parent: [pid] pid,
                  reported: [pid] bool )
{
   var n: int;
   var c: [int] pid;
   assume is_cycle(parent,reported,c,n);
   assert false;
   return;
}
```

```
// Spanning Tree algorithm, bug with unordered buffers
type pid;
const root: pid;
var parent: [pid] pid;
var reported: [pid] bool;

procedure Main ()
{
   var self: pid where self == root;
   assume (forall p: pid :: !reported[p]);
   call {:async} search (self, self);
   return;
}

procedure search (self: pid, sender: pid)
{
   var neighbor: pid;
   var sent: [pid] bool where (forall p: pid :: !sent[p
        ]);

   if (! reported[self]) {

      // BUG: this should be done synchronously
      // so that when the message comes back we
      // don't continue to propagate.
      if (self != sender) {
         call {:async} parent (self, sender);
      }

      while (*) {
         havoc neighbor;
         assume neighbor != self;
         assume neighbor != sender;
         assume neighbor != root;
         assume !sent[neighbor];
         sent[neighbor] := true;
         call {:async} search (neighbor, self);
      }
   }

   return;
}

procedure parent (self: pid, p: pid)
{
   parent[self] := p;
   reported[self] := true;

   if (*) { call assert_cycle(parent, reported); }

   return;
}
```

```
// Spanning Tree algorithm, bug with FIFO buffers
type pid;
const root: pid;
var parent: [pid] pid;
var reported: [pid] bool;

procedure Main ()
{
   var self: pid where self == root;
   assume (forall p: pid :: !reported[p]);
   call {:async} search (self, self);
   return;
}

procedure search (self: pid, sender: pid)
{
   var neighbor: pid;
   var sent: [pid] bool where (forall p: pid :: !sent[p
        ]);
   if (! reported[self]) {

      while (*) {
         havoc neighbor;
         assume neighbor != self;
         assume neighbor != sender;
         assume neighbor != root;
```

```
            assume !sent[neighbor];
            sent[neighbor] := true;

            // BUG: making both calls asynchronously
            // from another process allows multiple
            // pending parent calls from different
            // nodes, even with FiFo queues.
            // Note: in the other order (i.e. parent
            // then search) this bug goes away with
            // FiFo queues.
            call {:async} search( neighbor, self );
            call {:async} parent( neighbor, self );
         }
      }

      return;
   }

   procedure parent (self: pid, p: pid)
   {
      parent[self] := p;
      reported[self] := true;

      if (*) { call assert_cycle(parent, reported); }

      return;
   }


// Bellman-Ford algorithm
type pid;
type val;
const root: pid;
var dist: [pid] int;
var parent: [pid] pid;
var reported: [pid] bool;

const weight: [pid, pid] int;
axiom (forall p: pid :: weight[p,p] == 0);
axiom (forall p, q: pid :: weight[p,q] >= 0);

// const unique P1, P2, P3, P4: pid;
const NULL: val;
const INF: int;
axiom (INF == 10000);

procedure Main ()
{
   var self: pid where self != root;
   var w: val;
   assume (w != NULL);
   assume (forall p: pid :: dist[p] == INF);
   assume (forall p: pid :: !reported[p]);

   call {:async} bellmanFord (self, 0, root);
   return;
}

procedure bellmanFord (self: pid, w: int, sender: pid)
{
   var neighbor: pid;

   // BUG: need to check "<" not "<="
   if ( w + weight[self,sender] <= dist[self] ) {
      dist[self] := w + weight[self,sender];
      parent[self] := sender;
      reported[self] := true;

      while (*) {
         havoc neighbor;
         assume neighbor != self;
         assume neighbor != sender;
         assume neighbor != root;
         call {:async} bellmanFord (neighbor, dist[
              self], self);
      }
   }
   if (*) { call assert_cycle(parent, reported); }
   return;
}
```

```
// Breadth-First Spanning Tree algorithm
type pid;
const root: pid;
var dist: [pid] int;
var parent: [pid] pid;
var reported: [pid] bool;
const INF: int;
axiom (INF == 10000);

procedure Main ()
{
   var self: pid where self != root;
   assume (forall p: pid :: dist[p] == INF);
   assume (forall p: pid :: !reported[p]);
   call {:async} bfs(self, -1, root);
   return;
}

// choose a new parent if she is closer to the source
     than the previous
procedure bfs (self: pid, m: int, sender: pid)
{
   var neighbor: pid;

   if ( m <= dist[self] ) { // m + 1
      dist[self] := m; // m + 1;
      parent[self] := sender;
      reported[self] := true;

      while (*) {
         havoc neighbor;
         assume neighbor != self;
         assume neighbor != sender;
         assume neighbor != root;
         call {:async} bfs (neighbor, dist[self], self)
              ;
      }

      // Possible bugs? cannot forever get decreasing
           distances...
   }

   if (*) { call assert_cycle(parent, reported); }

   return;
}
```

References

1. Abdulla, P.A., Jonsson, B.: Verifying programs with unreliable channels. In: LICS '93: Proceedings of the 8th Annual IEEE Symposium on Logic in Computer Science, pp. 160–170. IEEE Computer Society (1993)
2. Abdulla, P.A., Bouajjani, A., Jonsson, B.: On-the-fly analysis of systems with unbounded, lossy fifo channels. In: CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification, vol. 1427 of LNCS, pp. 305–318. Springer, Berlin (1998)
3. Atig, M.F., Bollig, B., Habermehl, P.: Emptiness of multi-pushdown automata is 2etime-complete. In: DLT '08: Proceedings of the 12th International Conference on Developments in Language Theory, vol. 5257 of LNCS, pp. 121–133. Springer, Heidelberg (2008)
4. Atig, M.F., Bouajjani, A., Emmi, M., Lal, A.: Detecting fair non-termination in multithreaded programs. In: CAV '12: Proceedings of the 24th International Conference on Computer Aided Verification, vol. 7358 of LNCS. Springer, Heidelberg (2012)
5. Barnett, M., Leino, K.R.M. Moskal, M., Schulte W.: Boogie: an intermediate verification language. http://research.microsoft.com/en-us/projects/boogie/. Accessed 1 Jan 2012
6. Boigelot, B., Godefroid, P.: Symbolic verification of communication protocols with infinite state spaces using QDDs. Form. Methods Syst. Design **14**(3), 237–255 (1999)

7. Bouajjani, A., Emmi, M.: Bounded phase analysis of message-passing programs. In: TACAS '12: Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, vol. 7214 of LNCS, pp. 451–465. Springer, Heidelberg (2012)
8. Bouajjani, A., Habermehl, P.: Symbolic reachability analysis of fifo-channel systems with nonregular sets of configurations. Theor. Comput. Sci. **221**(1–2), 211–250 (1999)
9. Bouajjani, A., Habermehl, P., Vojnar, T.: Verification of parametric concurrent systems with prioritised FIFO resource management. Form. Methods Syst. Design **32**(2), 129–172 (2008)
10. Bouajjani, A. Emmi, M., Parlato G.: On sequentializing concurrent programs. In: SAS '11: Proceedings of the 18th International Symposium on Static Analysis, vol. 6887 of LNCS, pp. 129–145. Springer, Heidelberg (2011)
11. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM **30**(2), 323–342 (1983)
12. Chaudhuri, S.: Subcubic algorithms for recursive state machines. In: POPL '08: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 159–169. ACM, New York (2008)
13. Dahl, R.: Node.js: Evented I/O for V8 JavaScript. http://nodejs.org/. Accessed 1 Jan 2012
14. Emmi, M., Lal, A.: Finding non-terminating executions in distributed asynchronous programs. In: SAS '12: Proceedings of the 19th International Static Analysis Symposium, LNCS, pp. 439–455. Springer, Berlin (2012)
15. Emmi, M. Qadeer, S., Rakamaric, Z.: Delay-bounded scheduling. In: POPL '11: Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 411–422. ACM, New York (2011)
16. Emmi, M., Lal, A., Qadeer, S.: Asynchronous programs with prioritized task-buffers. In: FSE '12: Proceedings of the 20th International Symposium on the Foundations of Software Engineering. ACM, New York (2012)
17. Esparza, J., Kucera, A., Schwoon, S.: Model checking ltl with regular valuations for pushdown systems. Inf. Comput. **186**(2), 355–376 (2003)
18. Ganty, P., Majumdar, R.: Algorithmic verification of asynchronous programs. ACM Trans. Program. Lang. Syst. **34**(1), 6 (2012)
19. Ganty, P., Majumdar, R., Rybalchenko, A.: Verifying liveness for asynchronous programs. In: POPL 09: Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 102–113. ACM, New York (2009)
20. Haller, P., Odersky, M.: Scala actors: unifying thread-based and event-based programming. Theor. Comput. Sci. **410**(2–3), 202–220 (2009)
21. Heußner, A., Leroux, J., Muscholl, A., Sutre, G.: Reachability analysis of communicating pushdown systems. In: FOSSACS '10: Proceedings of the 13th International Conference on Foundations of Software Science and Computational Structures, vol. 6014 of LNCS, pp. 267–281. Springer, Heidelberg (2010)
22. HTML5: A vocabulary and associated APIs for HTML and XHTML. http://dev.w3.org/html5/spec/Overview.html. Accessed 1 Jan 2012
23. Jhala, R., Majumdar, R.: Interprocedural analysis of asynchronous programs. In: POPL '07: Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 339–350. ACM, New York (2007)
24. Kidd, N., Jagannathan, S., Vitek, J.: One stack to run them all: reducing concurrent analysis to sequential analysis under priority scheduling. In: SPIN '10: Proceedings of the 17th International Workshop on Model Checking Software, vol. 6349 of LNCS, pp. 245–261. Springer, Heidelberg (2010)
25. La Torre S., Parlato, G.: Scope-bounded multistack pushdown systems: fixed-point, sequentialization, and tree-width. In: FSTTCS '12: Proceedings of the IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, vol. 18 of LIPIcs, pp. 173–184. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2012)
26. La Torre, S., Madhusudan, P., Parlato, G.: A robust class of context-sensitive languages. In: LICS '07: Proceedings of the 22nd IEEE Symposium on Logic in Computer Science, pp. 161–170. IEEE Computer Society (2007)
27. La Torre, S. Madhusudan, P., Parlato G.: Context-bounded analysis of concurrent queue systems. In: TACAS '08: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, vol. 4963 of LNCS, pp. 299–314. Springer, Heidelberg (2008)
28. La Torre, S. Madhusudan, P., Parlato, G.: Reducing context-bounded concurrent reachability to sequential reachability. In: CAV '09: Proceedings of the 21st International Conference on Computer Aided Verification, vol. 5643 of LNCS, pp. 477–492. Springer, Heidelberg (2009)
29. La Torre, S., Madhusudan, P., Parlato, G.: Model-checking parameterized concurrent programs using linear interfaces. In: CAV '10: Proceedings of the 22nd International Conference on Computer Aided Verification, vol. 6174 of LNCS, pp. 629–644. Springer, Heidelberg (2010)
30. La Torre, S., Madhusudan, P., Parlato, G.: Sequentializing parameterized programs. In: FIT '12: Proceedings of the Fourth Workshop on Foundations of Interface Technologies, vol. 87 of EPTCS, pp. 34–47 (2012)
31. Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. Form. Methods Syst. Design **35**(1), 73–97 (2009)
32. Lal, A. Qadeer, S., Lahiri, S.K.: A solver for reachability modulo theories. In: CAV '12: Proceedings of the 24th International Conference on Computer Aided Verification, vol. 7358 of LNCS, pp. 427–443. Springer, Heidelberg (2012)
33. Lynch, N.A.: Distributed algorithms. ISBN 1-55860-348-4. Morgan Kaufmann, San Francisco (1996)
34. Miller, M.S., Tribble, E.D., Shapiro, J.S.: Concurrency among strangers. In: TGC '05: Proceedings of the International Symposium on Trustworthy Global Computing, vol. 3705 of LNCS, pp. 195–229. Springer, Heidelberg (2005)
35. Post, E.L.: A variant of a recursively unsolvable problem. Bull. Am. Math. Soc **52**(4), 264–268 (1946)
36. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: TACAS '05: Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, vol. 3440 of LNCS, pp. 93–107. Springer, Heidelberg (2005)
37. Qadeer, S., Wu, D.: KISS: keep it simple and sequential. In: PLDI '04: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 14–24. ACM, New York (2004)
38. Reps, T.W., Horwitz, S., Sagiv, S.: Precise interprocedural dataflow analysis via graph reachability. In: POPL '95: Proceedings of the 22th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 49–61. ACM, New York (1995)
39. Sen, K., Viswanathan, M.: Model checking multithreaded programs with asynchronous atomic methods. In: CAV '06:

Proceedings of the 18th International Conference on Computer Aided Verification, vol. 4144 of LNCS, pp. 300–314. Springer, Heidelberg (2006)
40. Sharir, M., Pnueli, A.: Two approaches to interprocedural data-flow analysis. In: Muchnick, S.S., Jones, N.D. (eds.) Program Flow Analysis: Theory and Applications, chapter 7, pp. 189–234. Prentice-Hall, Englewood Cliffs (1981)
41. Svensson, H., Arts, T.: A new leader election implementation. In: Erlang '05: Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang, pp. 35–39. ACM, New York (2005)
42. Trottier-Hebert, F.: Learn you some Erlang for great good! http://learnyousomeerlang.com/. Accessed 1 Jan 2012