Bounded Phase Analysis of Message-Passing Programs

Ahmed Bouajjani, Michael Emmi

Bounded Phase Analysis of Message-Passing Programs*

Ahmed Bouajjani and Michael Emmi†

LIAFA, Université Paris Diderot, France
{abou,mje}@liafa.jussieu.fr

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 "contexts-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.

1 Introduction

Software is becoming increasingly concurrent: reactivity (e.g., in 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 one or more processes execute tasks sequences which communicate by posting messages (alternatively, tasks) to message/task queues—Miller et al. [17] discuss why such models provide good programming abstractions. Single-process systems such as the JavaScript page-loading engine of modern web browsers [1], and the highly-scalable Node.js asynchronous web server [9], 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 highly-scalable and highly-reliable distributed systems are built.

Despite the increasing popularity of such programming models, little is known about precise algorithmic reasoning of such programs. This is perhaps not without

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

good reason: decision problems such as state-reachability for programs communicating with unbounded reliable queues are undecidable [8], even when there is only a single finite-state process (posting messages to itself). Furthermore, the known decidable under-approximations (e.g., bounding the size of queues) represent queues explicitly, are thus doomed to combinatorial explosion as the size and variability of queue content increases.

Some have proposed analyses which abstract message arrival order [20, 12, 11], or assume messages can be arbitrarily lost [2, 3]. Such analyses do not suffice when correctness arguments rely on reliable messaging—several systems specifically do ensure the ordered delivery of messages, including Erlang, Scala, and recent web-browser specifications [1]. Others have proposed analyses which compute finite symbolic representations of queue contents [5, 6]. 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" [14], 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 Section 3. All of the above techniques represent queues explicitly, though perhaps symbolically, 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 Section 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 [21] 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 true strength of phase-bounding by showing in Sections 4 and 5 that the bounded phase executions of a message-passing program can be concisely encoded as a non-deterministic 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 (Section 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. By reducing to sequential programs, we leverage highly-developed sequential program analysis tools for message-passing programs.

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. Though the distinction between queues containing messages and queues containing tasks is mostly aesthetic, our task-based treatment implies that queues are only read by idle processors; reading additional messages during a task's execution is prohibited. Though in principle many message-passing systems, e.g., in Erlang and Scala, allow reading additional messages at any program point, we have observed that common practice is to read messages only upon completing a task [22].

Though similar to Sen and Viswanathan [20]'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. For the case of single-processor programs, Sen and Viswanathan [20]'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 statements, s_p describes the entire body of p. The set of program statements s is denoted Stmts. Intuitively, a **post** ρ 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 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, and which easily reduce to the syntax of our grammar.

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

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

$$
\textsc{Dispatch}\quad \frac{}{\langle g, \varepsilon, f \cdot q\rangle \xrightarrow{\mathrm{S}} \langle g, f, q\rangle}
$$

$$
\textsc{Complete}\quad \frac{f = \langle \ell, \textbf{return}\ e\texttt{;}\ s\rangle}{\langle g, f, q\rangle \xrightarrow{\mathrm{S}} \langle g, \varepsilon, q\rangle}
$$

$$
\textsc{Self-Post}\quad \frac{s_1 = \textbf{post}\ \_\ p\ e\texttt{;}\ s_2 \qquad \ell_2 \in e(g, \ell_1) \qquad f = \langle \ell_2, s_p\rangle}{\langle g, \langle \ell_1, s_1\rangle\, w, q\rangle \xrightarrow{\mathrm{S}} \langle g, \langle \ell_1, s_2\rangle\, w, q \cdot f\rangle}
$$

Fig. 2. The single-processor transition rules $\rightarrow^{\mathrm{S}}$; see Appendix B for the standard sequential statements.

2.2 Single-Processor Semantics

A *(procedure) frame* $f = \langle \ell, s\rangle$ is a current valuation $\ell \in \mathsf{Vals}$ to the procedure-local variable $\texttt{l}$, along with a statement $s \in \mathsf{Stmts}$ to be executed. (Here s describes the entire body of a procedure p that remains to be executed, and is initially set to p's top-level statement s_p; we refer to initial procedure frames $t = \langle \ell, s_p\rangle$ as *tasks*, to distinguish the frames that populate processor queues.) The set of all frames is denoted Frames.

A *processor configuration* $\kappa = \langle g, w, q\rangle$ is a current valuation $g \in \mathsf{Vals}$ to the processor-global variable g, along with a procedure-frame stack $w \in \mathsf{Frames}^*$ and a pending-tasks queue $q \in \mathsf{Frames}^*$. A processor is idle when $w = \varepsilon$. The set of all processor configurations is denoted $\mathsf{Pconfigs}$. A processor configuration map $\xi : \mathsf{Pids} \to \mathsf{Pconfigs}$ maps each processor $\rho \in \mathsf{Pids}$ 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 $\rho' \in \mathsf{Pids} \setminus \{\rho\}$.

For expressions without program variables, we assume the existence of an evaluation function $[\![\cdot]\!]_{\mathrm{e}} : \mathsf{Exprs} \to \wp(\mathsf{Vals})$ such that $[\![\star]\!]_{\mathrm{e}} = \mathsf{Vals}$. For convenience we define $e(g, \ell) \stackrel{\text{def}}{=} [\![e[g/\mathsf{g}, \ell/\texttt{l}]]\!]_{\mathrm{e}}$ to evaluate the expression e in a global valuation g by substituting the current values for variables g and $\texttt{l}$. As these are the only program variables, the substituted expression $e[g/\mathsf{g}, \ldots]$ has no free variables.

Figure 2 defines the transition relation $\rightarrow^{\mathrm{S}}$ for the asynchronous behavior of each processor; the standard transitions for the sequential statements are listed in Appendix B. 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. In order to later restrict processor interleaving, we make explicit the *scheduler* which arbitrates the possible interleavings. Formally, a *scheduler* $M = \langle D, \mathsf{empty}, \mathsf{enabled}, \mathsf{step}\rangle$ consists of a data type D of scheduler objects $m \in D$, a scheduler constructor $\mathsf{empty} \in D$, a scheduler decision function $\mathsf{enabled} : (D \times (\mathsf{Pids} \to \mathsf{Pconfigs})) \to \wp(\mathsf{Pids})$, and a scheduler update function $\mathsf{step} : (D \times (\mathsf{Pids} \to \mathsf{Pconfigs}) \times (\mathsf{Pids} \to \mathsf{Pconfigs})) \to D$. The arguments to $\mathsf{enabled}$ and step allow a scheduler to decide which processors are enabled depending on the execution history. A scheduler is *deterministic* when $|\mathsf{enabled}(m, \xi)| \leq 1$ for all $m \in D$ and $\xi : \mathsf{Pids} \to \mathsf{Pconfigs}$, and is *non-blocking* when for all m and ξ, if there is some $\rho \in \mathsf{Pids}$ such that $\xi(\rho)$ is either non-idle or has pending tasks, then there exists $\rho' \in \mathsf{Pids}$ such that $\rho' \in \mathsf{enabled}(m, \xi)$ and $\xi(\rho')$ is either non-idle or has pending tasks. A *configuration* $c = \langle \rho, \xi, m\rangle$ is a currently executing processor $\rho \in \mathsf{Pids}$, along with a processor configuration map ξ, and a scheduler object m.

Figure 3 defines the multi-processor transition relation $\to_M$, parameterized by a scheduler M. The SWITCH rule non-deterministically 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 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 Section 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 $c_i \to_M c_{i+1}$ 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 $g_0 \in \mathsf{Vals}$, a local-variable valuation $\ell_0 \in \mathsf{Vals}$, and a procedure $p_0 \in \mathsf{Procs}$. A configuration $c = \langle \rho_0, \xi, \mathsf{empty}\rangle$ of a program P is *$\langle \rho_0, g_0, \ell_0, p_0\rangle$-initial* when $\xi(\rho_0) = \langle g_0, \varepsilon, \langle \ell_0, s_{p_0}\rangle\rangle$ and $\xi(\rho) = \langle g_0, \varepsilon, \varepsilon\rangle$ for all $\rho \neq \rho_0$. A configuration $\langle \rho, \xi, m\rangle$ is *g_f-final* when $\xi(\rho') = \langle g_f, w, q\rangle$ for some $\rho' \in \mathsf{Pids}$, and $w, q \in \mathsf{Frames}^*$. We say a global valuation g is *M-reachable* in P from ι when there exists an M-execution of P from some c_0 to some c such that c_0 is ι-initial and c is g-final[1].

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,

[1] In the presence of the **assume** statement, only values of completed executions are guaranteed to be valid.

$$\textsc{Switch}\quad \frac{\rho_2 \in \mathsf{enabled}(m, \xi)}{\langle \rho_1, \xi, m\rangle \underset{M}{\longrightarrow} \langle \rho_2, \xi, m\rangle}$$

$$\textsc{Step}\quad \frac{\xi_1(\rho) \xrightarrow{\mathrm{S}} \kappa \qquad \xi_2 = \xi_1\,(\rho \mapsto \kappa) \qquad \rho \in \mathsf{enabled}(m_1, \xi_1) \qquad m_2 = \mathsf{step}(m_1, \xi_1, \xi_2)}{\langle \rho, \xi_1, m_1\rangle \underset{M}{\longrightarrow} \langle \rho, \xi_2, m_2\rangle}$$

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

Fig. 3. The multi-processor transitions relation $\rightarrow_M$ parameterized by a scheduler $M = \langle D, \mathsf{empty}, \mathsf{enabled}, \mathsf{step}\rangle$.

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

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

// translation of g
G[phase]

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

// translation of post _ p e
if phase+1 < k then
    call p (e,phase+1)
```

Fig. 4. The k-phase sequential translation $(\!|P|\!)_k$ of a single-processor asynchronous message-passing program P.

even with only a single processor accessing finite-state data [8]. 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 Figure 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

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

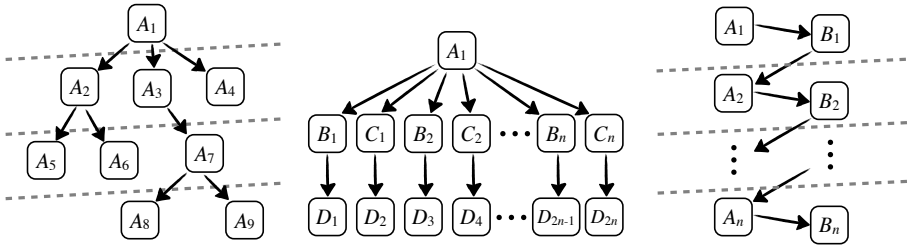


Fig. 5. Phase-bounded executions with processors A, B, C, and D; each task's label (e.g., A_i) indicates the processor it executes on (e.g., A). Arrows indicate the posting relation, indices indicate execution order on a given processor, and dotted lines indicate phase boundaries.

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*.*

The execution in Figure 5a is a 4-phase execution, since all tasks have phase less than 4. Despite there being an arbitrary number $3n + 1$ of posted tasks, the execution in Figure 5b is 1-phase, since there are no task-chains between same-processor tasks. Contrarily, the execution in Figure 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 $k \cdot |\mathsf{Pids}|$. 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 Figure 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 Figure 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, both schemes do capture all executions.

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

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. 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 Figure 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 i^{th} global valuation is set to the state g_i in which phase i is known to begin; tasks of phase i then reads from and writes to the i^{th} global valuation. It then only remains to ensure that the global valuations g_i used at the beginning of each phase $0 < i < k$ match the valuations reached at the end of phase $i - 1$.

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 i^{th} 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 i^{th} global valuation matches the initially-used valuation for phase $i+1$, for $0 \leq i < k-1$. When this condition holds, any global valuation observed along the execution is reachable within k phases in the original program. Figure 4 lists a code-to-code translation which implements this simulation.

Theorem 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 $(\!(P)\!)_k$—the k-phase sequential translation of P.*

When the underlying sequential program model has a decidable state-reachability problem, Theorem 2 gives 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.

Corollary 1. *The k-phase state-reachability problem is decidable for single-processor programs with finite data domains.*

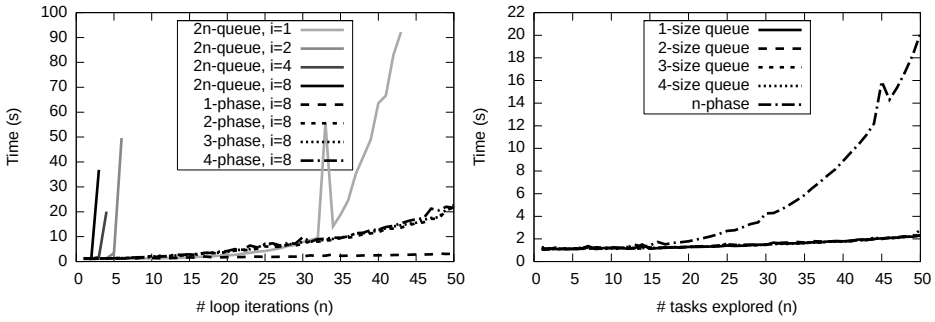
More generally, given any underlying sequential program model, our translation makes applicable any analysis tool for said model to message-passing programs, since the values of the additional variables are either from the finite domain $\{0, \ldots, k-1\}$, or in the domain of the original program variables.

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 (listed in full in Appendix C, 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 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 $P_1(i)$, 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 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 6a compares the time required to verify $P_1(i)$ (using the Boogie verification engine [4]) for various values of i, and various bounds n on loop unrolling. Note that although every execution of $P_1(i)$ has only 2 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, as the number of possible task-queues grows astronomically as the size of task-queues grow. This ultimately prohibits the bounded tasks-queue 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 is biased to support the bounded task-queue encoding. Following the example of Figure 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 6b compares the time required to verify P_2 for various bounds n on the number

[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.



(a) (b)

Fig. 6. 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. Time-out is set to 100s.

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

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 Figure 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 pre-determined: 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! In fact, due to the

[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.

capability of such unbounded interleaving, the problem of state-reachability under a phase-bound is undecidable for multi-processor programs, even for programs with finite data domains.

Theorem 3. *The k-phase bounded state-reachability problem is undecidable for multi-processor programs with finite data domains.*

Note that Theorem 3 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 two processors lends the power to simulate Post's correspondence problem [18].

Theorem 3 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 [10].

We define a *delaying scheduler* $M = \langle D, \mathsf{empty}, \mathsf{enabled}, \mathsf{step}, \mathsf{delay}\rangle$, as a scheduler $\langle D, \mathsf{empty}, \mathsf{enabled}, \mathsf{step}\rangle$, along with a function $\mathsf{delay} : (D \times \mathsf{Pids} \times (\mathsf{Pids} \to \mathsf{Pconfigs})) \to D$. Furthermore, we extend the transition relation of Figure 3 with a postponing rule of Figure 7 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. A delaying scheduler is *delay-accessible* when for every configuration c_1 and non-idle or task-pending processor ρ, there exists a sequence $c_1 \to_M \ldots \to_M c_j$ 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 Figure 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 Figure 5b is unique, since M is deterministic. However, as Figure 8 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.

Theorem 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 $h' \sim h$.*

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

Fig. 7. The delay operation.

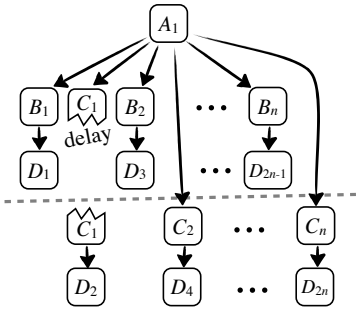


Fig. 8. A 2-phase delaying execution varying the 1-phase execution of Figure 5b.

```
// translation of var g: T
var G[k+d]: T
var shift[Pids][k], delay: d
var ancestors[Pids][k+d]: 𝔹

// translation of proc p (var l: T) s
proc p (var l: T, pid: Pids, phase: k)

// translation of g
G[ phase + shift[pid][phase] ]

// code to be sprinkled throughout
while ⋆ and delay < d do
    shift[pid][phase]++; delay++

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

// translation of post ρ p e
let p = phase + shift[pid][phase] in
let p' = p + (if ancestors[ρ][p] then 1 else 0) in
if p' < k then
    ancestors[ρ][p' + shift[ρ][p']] := true;
    call p (e, ρ, p')
    ancestors[ρ][p' + shift[ρ][p']] := false
```

Fig. 9. The k-phase d-delay sequential translation $(\!(P)\!)^{\text{bfs}}_{k,d}$ of a multi-processor message-passing asynchronous program P.

Note that Theorem 4 holds for *any* delay-accessible scheduler M—even deterministic schedulers. As it turns out there is one particular scheduler M_{bfs} for which we know a convenient sequential encoding, and this scheduler is described in Appendix D. For the moment, the important points to note are that M_{bfs} 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 9 lists a code-to-code translation which encodes bounded-phase and bounded-delay exploration of a given program according to the M_{bfs} scheduler as a sequential program.

Our translation closely follows the single-processor translation of Section 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, and
- 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 `shift` counter: an execution is only valid when for each processor

$\rho \in \mathsf{Pids}$ and each phase $0 < i < k$, $\texttt{shift}[\rho][i-1]$ matches the initial value of $\texttt{shift}[\rho][i]$.

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

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 tasks-queues explicitly.

6 Related Work

Our work follows the line of research on compositional reductions from concurrent to sequential programs. The initial so-called "sequentialization" [19] 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 [16]. La Torre et al. [15] later extended the approach to handle programs parameterized by an unbounded number of statically-determined threads, and shortly after, Emmi et al. [10] 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 [20]. Bouajjani et al. [7] pushed these results even further to a sequentialization which attempts to explore as many behaviors as possible within a given analysis budget. Each 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 implicitly abstracted away from the asynchronous program model [20]. Though Kidd et al. [13]'s sequentialization is sensitive to task priorities, their reduction assumes a finite number of statically-determined tasks.

In a closely-related work, La Torre et al. [14] propose a "context-bounded" analysis 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 work can be seen as an extension in a few ways. First, and most trivially, in their setting a process cannot post to its own message queue; this implies that at least $2k$ contexts must be used to simulate k phases of a single-processor program. Second, there are families of 1-phase executions which require an unbounded number of task-contexts to capture; the execution order $D_1 D_2 D_3 \ldots D_{2n}$ of Figure 5b is such an example. We conjecture that bounded phase and delay captures context-bounding—i.e., there exists a polynomial function $f : \mathbb{N} \to \mathbb{N}$ such that every k-context bounded execution of any program P is also a $f(k)$-phase and delay bounded execution. Finally, though phase-bounding leads to a convenient sequential encoding, we are unaware whether a similar encoding is possible for context-bounding.

Boigelot and Godefroid [5] and Bouajjani et al. [6] have proposed analyses of message-passing programs by computing explicit finite symbolic representations of message-queues. As our sequentialization does not represent queues explicitly, 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.

7 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 thank Constantin Enea, Cezara Dragoi, Pierre Ganty, and the anonymous reviewers for helpful feedback.

References

[1] HTML5: A vocabulary and associated APIs for HTML and XHTML. `http://dev.w3.org/html5/spec/Overview.html`.

[2] P. A. Abdulla and B. Jonsson. Verifying programs with unreliable channels. In *LICS '93: Proc. 8th Annual IEEE Symposium on Logic in Computer Science*, pages 160–170. IEEE Computer Society, 1993.

[3] P. A. Abdulla, A. Bouajjani, and B. Jonsson. On-the-fly analysis of systems with unbounded, lossy fifo channels. In *CAV '98: Proc. 10th International Conference on Computer Aided Verification*, volume 1427 of *LNCS*, pages 305–318. Springer, 1998.

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

[5] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. *Formal Methods in System Design*, 14(3):237–255, 1999.

[6] A. Bouajjani, P. Habermehl, and T. Vojnar. Verification of parametric concurrent systems with prioritised FIFO resource management. *Formal Methods in System Design*, 32(2):129–172, 2008.

[7] A. Bouajjani, M. Emmi, and G. Parlato. On sequentializing concurrent programs. In *SAS '11: Proc. 18th International Symposium on Static Analysis*, volume 6887 of *LNCS*, pages 129–145. Springer, 2011.

[8] D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.

[9] R. Dahl. Node.js: Evented I/O for V8 JavaScript. `http://nodejs.org/`.

[10] M. Emmi, S. Qadeer, and Z. Rakamarić. Delay-bounded scheduling. In *POPL '11: Proc. 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 411–422. ACM, 2011.

[11] P. Ganty and R. Majumdar. Algorithmic verification of asynchronous programs. *CoRR*, abs/1011.0551, 2010. `http://arxiv.org/abs/1011.0551`.

[12] R. Jhala and R. Majumdar. Interprocedural analysis of asynchronous programs. In *POPL '07: Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 339–350. ACM, 2007.

[13] N. Kidd, S. Jagannathan, and J. Vitek. One stack to run them all: Reducing concurrent analysis to sequential analysis under priority scheduling. In *SPIN '10: Proc. 17th International Workshop on Model Checking Software*, volume 6349 of *LNCS*, pages 245–261. Springer, 2010.

[14] S. La Torre, P. Madhusudan, and G. Parlato. Context-bounded analysis of concurrent queue systems. In *TACAS '08: Proc. 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *LNCS*, pages 299–314. Springer, 2008.

[15] S. La Torre, P. Madhusudan, and G. Parlato. Model-checking parameterized concurrent programs using linear interfaces. In *CAV '10: Proc. 22nd International Conference on Computer Aided Verification*, volume 6174 of *LNCS*, pages 629–644. Springer, 2010.

[16] A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1):73–97, 2009.

[17] M. S. Miller, E. D. Tribble, and J. S. Shapiro. Concurrency among strangers. In *TGC '05: Proc. International Symposium on Trustworthy Global Computing*, volume 3705 of *LNCS*, pages 195–229. Springer, 2005.

[18] E. L. Post. A variant of a recursively unsolvable problem. *Bull. Amer. Math. Soc*, 52(4):264–268, 1946.

[19] S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In *PLDI '04: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 14–24. ACM, 2004.

[20] K. Sen and M. Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. In *CAV '06: Proc. 18th International Conference on Computer Aided Verification*, volume 4144 of *LNCS*, pages 300–314. Springer, 2006.

[21] H. Svensson and T. Arts. A new leader election implementation. In *Erlang '05: Proc. 2005 ACM SIGPLAN Workshop on Erlang*, pages 35–39. ACM, 2005.

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

A Syntactic Extensions Used in Our Code Translations

The following syntactic extensions are reducible to the original program syntax of Section 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 $T = \sum_{i=1}^{j} T_i$. This allows local and global variables with distinct types.

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

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

and all occurrences of $\texttt{x}_i$ are replaced by $\texttt{x.f}_i$. 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** `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 $\star$, 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** $\texttt{x :=}\ p\ e$, where `x` is not subsequently used, can be written as **call** $\texttt{\_ :=}\ p\ e$, where $\texttt{\_:}\ T$ 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 $x\ \texttt{:=}\ x\ \texttt{+ 1}$.

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

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

B Sequential Program Semantics

For expressions without program variables, we assume the existence of an evaluation function $[\![\cdot]\!]_{\mathrm{e}} : \mathsf{Exprs} \to \wp(\mathsf{Vals})$ such that $[\![\star]\!]_{\mathrm{e}} = \mathsf{Vals}$. 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 `g` and `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

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

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 $s \in \mathsf{Stmts}$. 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 top-most 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 $e(C[\mathbf{skip}])$ by $e(C)$.

Figure 10 defines the transition relation $\rightarrow^{\mathrm{S}}$ 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 proceeds 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 $\rightarrow^{\mathrm{S}}$ is non-deterministic, since the evaluation of an expression e can result in an arbitrary set of possible values.

C Full Listing of Example Programs of Section 4

The first program $P_1(i)$, 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`

$$\textsc{Skip}\ \frac{}{C[\textbf{skip};\ s] \xrightarrow{\text{S}} C[s]} \qquad \textsc{Assume}\ \frac{\texttt{true} \in e(C)}{C[\textbf{assume}\ e] \xrightarrow{\text{S}} C[\textbf{skip}]} \qquad \textsc{Assign}\ \frac{v \in e(C)}{C[x\ \texttt{:=}\ e] \xrightarrow{\text{S}} C[\textbf{skip}]\ (x \leftarrow v)}$$

$$\textsc{If-Then}\ \frac{\texttt{true} \in e(C)}{C[\textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2] \xrightarrow{\text{S}} C[s_1]} \qquad \textsc{If-Else}\ \frac{\texttt{false} \in e(C)}{C[\textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2] \xrightarrow{\text{S}} C[s_2]}$$

$$\textsc{Loop-Do}\ \frac{\texttt{true} \in e(C)}{C[\textbf{while}\ e\ \textbf{do}\ s] \xrightarrow{\text{S}} C[s; \textbf{while}\ e\ \textbf{do}\ s]} \qquad \textsc{Loop-End}\ \frac{\texttt{false} \in e(C)}{C[\textbf{while}\ e\ \textbf{do}\ s] \xrightarrow{\text{S}} C[\textbf{skip}]}$$

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

Fig. 10. The single-processor transitions relation $\rightarrow^{\text{S}}$ for the standard sequential program statements.

to `true`, and 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.

```
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
```

Figure 6a compares the time required to verify $P_1(i)$ (using the Boogie verification engine [4]) for various values of i, and various bounds n on loop unrolling.Note that although every execution of $P_1(i)$ has only 2 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.

Our second program P_2 is biased to support the bounded task-queue encoding. 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.

```
var b: bool

proc main ()
    b := false;
    post p1 ()
    return

proc p1 ()
    assert !b;
    b := true;
    post p2 ();
    return

proc p2 ()
    b := false;
    post p1 ()
    return
```

Figure 6b compares the time required to verify P_2 for various bounds n on the number of tasks explored. 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.

D The Multi-Processor Breadth-First Scheduler

Here we define a deterministic, non-blocking, delay-accessible delaying scheduler M_{bfs} 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_{bfs} 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 $u_0 \in U$. Our scheduler 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 breadth-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 breadth-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 *Breadth-First Scheduler* $M_{\text{bfs}} = \langle D, \mathsf{empty}, \mathsf{enabled}, \mathsf{step}, \mathsf{delay} \rangle$ is defined over scheduler objects $m = \langle i, T, \surd, \Phi, \Delta \rangle \in D$ as described above; the initial object is $\mathsf{empty} = \langle 0, T_0, \surd_0, \Phi_0, \Delta_0 \rangle$, 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 $\rho \in \mathsf{Pids}$. The $\mathsf{enabled}(\langle i, T, \surd, \Phi, \Delta \rangle, \xi)$ operation uniquely returns the processor identifier ρ of the smallest task u—according to the breadth-first order of T—such that $\Phi(u) + \Delta(\rho) = i$. The $\mathsf{step}(\langle i_1, T_1, \surd_1, \Phi_1, \Delta \rangle, \xi_1, \xi_2)$ operation for a transition $\tau = \xi_1(\rho) \rightarrow^{\text{S}} \xi_2(\rho)$ returns $\langle i_2, T_2, \surd_2, \Phi_2, \Delta \rangle$ such that

- If τ is a COMPLETE-step of task u, then $\surd_2 = \surd_1(u \mapsto \texttt{true})$; 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 $\Phi_2 = \Phi_1(u' \mapsto \Phi_1(u) + \Delta(u))$; 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 $\mathsf{delay}(\langle i_1, T, \surd, \Phi, \Delta_1 \rangle, \rho, \xi)$ 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_{bfs} 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_{bfs} is a deterministic delaying scheduler which executes all tasks of a given phase before any task of a subsequent phase. Since M_{bfs} must pick an enabled task so long as there are pending tasks on some processor, M_{bfs} 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_{bfs} is delay accessible.

On non-delaying executions, M_{bfs} essentially performs a phase-by-phase breadth-first traversal of the task-posting tree T—a tree which includes tasks across all processors. Interestingly and essentially for our sequential encoding of M_{bfs} in Section 5, on a per-phase basis, with respect to any individual processor, the breadth-first traversal of the task-posting tree is identical to depth-first traversal. This follows from the fact that no task may have a same-processor ancestor in the same phase, and that processors do not share memory.

E Proofs to Selected Theorems

Theorem 3. *The k-phase bounded state-reachability problem is undecidable for multi-processor programs with finite data domains.*

Proof. We proceed by reduction from Post's correspondence problem [18]: 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 $i_1 \dots i_k \in \{1 \dots n\}^*$ such that $\alpha_{i_1}\alpha_{i_2} \dots \alpha_{i_k} = \beta_{i_1}\beta_{i_2} \dots \beta_{i_k}$. For this problem instance we build the following finite-data asynchronous message-passing program with four processors ρ_0, ρ_1, ρ_2, and ρ_3:

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

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

// 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
```

Initially, the `main` procedure is pending on processor ρ_0.

In each loop iteration, `main` chooses a branch corresponding to an index $i \in \{1 \dots n\}$ 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 $\alpha_{i_1} \dots \alpha_{i_k}$, 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. □