

Context-Bounded Analysis of Concurrent Queue Systems*

Salvatore La Torre¹, P. Madhusudan², and Gennaro Parlato^{1,2}

¹ Università degli Studi di Salerno, Italy

² University of Illinois at Urbana-Champaign, USA

Abstract. We show that the bounded context-switching reachability problem for concurrent finite systems communicating using unbounded FIFO queues is decidable, where in each context a process reads from only one queue (but is allowed to write onto all other queues). Our result also holds when individual processes are finite-state recursive programs provided a process dequeues messages only when its local stack is empty. We then proceed to classify architectures that admit a decidable (unbounded context switching) reachability problem, using the decidability of bounded context switching. We show that the precise class of decidable architectures for recursive programs are the forest architectures, while the decidable architectures for non-recursive programs are those that do not have an undirected cycle.

1 Introduction

Networks of concurrent processes communicating via message queues form a very natural and useful model for several classes of systems with inherent parallelism. Two natural classes of systems can be modeled using such a framework: asynchronous programs on a multi-core computer and distributed programs communicating on a network.

In parallel programming languages for multi-core or even single-processor systems (e.g., Java, web service design), *asynchronous programming* or *event-driven programming* is a common idiom that programming languages provide [19,7,13]. In order to obtain higher performance and low latency, programs are equipped with the ability to issue tasks using *asynchronous* calls that immediately return, but are processed later, either in parallel with the calling module or perhaps much later, depending on when processors and other resources such as I/O become free. Asynchronous calls are also found in event-driven programs where a program can register *callback* functions that are associated to particular events (such as a new connection arriving on a socket), and are called when the event occurs. Programs typically call several other functions asynchronously so that they do not get blocked waiting for them to return. The tasks issued by a system are typically handled using queues, and we can build faithful models of these systems as networks of processes communicating via queues.

* The first and third authors were partially supported by the MIUR grants ex-60% 2006 and 2007 Università degli Studi di Salerno.

Distributed systems communicating via FIFO message channels also form a natural example of networks of processes communicating via queues. Motivated by the verification problem of distributed communication protocols, the model-checking problem for these systems has been studied extensively, where each process in the system is modeled using a finite-state transition system [1,15,10].

In this paper, we study the reachability problem for finite-state processes (and finite-state recursive processes) communicating via FIFO queues. We follow the paradigm of *abstraction*, where we assume that each program has been abstracted (using, for example, predicates) or modeled as a finite-state process, and algorithmically subject to model-checking.

The main barrier to model-check queue systems is that the FIFO message queues give an infinite state-space that is intractable: even reachability of two processes communicating via queues with each other is *undecidable*. There have been several ways to tackle the undecidability in this framework. One line of attack has been to weaken the power of queues by assuming that messages can get arbitrarily lost in queues; this leads to a decidable reachability problem [1] that is appealing in the distributed protocol domain as messages indeed can get lost, but is less natural in the event-driven programming domain as enlisted tasks seldom get lost.

Another technique is to ignore the FIFO order of messages and model the queue as a bag of messages, where any element of the bag can be delivered at any time [18,11]. When the number of kinds of messages is bounded, this amounts to keeping track of how many messages of each kind are present in the bag, which can be tracked using counters. Using the fact that counter systems without a zero-check admit a decidable reachability problem, model-checking of these systems can be proved decidable. In the realm of event-driven programming the assumption of modeling pending tasks as a bag is appealing because of the nondeterministic nature of scheduling.

In this paper, we do not destroy the contents of queues nor destroy the FIFO order, but model queues accurately. To curb undecidability, we show that the *bounded context-switching reachability problem* is decidable. More precisely, a *context* of a queueing network is defined as an (arbitrarily long) evolution of one process that is allowed to dequeue messages from (only) one queue and enqueue messages on all its outgoing message queues. The bounded context-switching reachability problem asks whether a global control state is reachable through a run that switches contexts at most k times, for a fixed value k .

Bounded context switching for recursive concurrent (non-queueing) programs was introduced in [16] in order to find a meaningful way to explore initial parts of executions using algorithmic techniques. The intuition is that many errors manifest themselves even with a short number of context-switches, and hence a model-checker that explores thoroughly the states reached within a few context-switches can prove effective in finding bugs. Bounded context-switching for non-queueing programs have exhibited good coverage of state-spaces [14] and are an appealing restriction for otherwise intractable verification problems of concurrent programs.

We show the decidability of bounded context switching reachability for queueing finite-state programs, which in addition can have a finite *shared memory*. We show that the problem is also decidable for *recursive programs*, wherein each process has a local call-stack that it can manipulate on its moves, provided each process is *well-queueing*. A set of programs is well-queueing if each process dequeues messages only when its local stack is empty. This model allows us to capture general event-driven programs that have recursive synchronous calls, and the well-queueing assumption is natural as the most prevalent programs dequeue a task and execute it to completion before dequeuing the next task (see [18,5] for similar restrictions). We show both the above decidability results by a reduction to the bounded phase reachability of multistack machines, which was recently proven by us to be decidable [12].

We also study the *unbounded context-switching* reachability problem for queue systems by classifying the architectures that admit decidable reachability problems. An architecture is a set of process sites and queues connecting certain pairs of process sites. For the class of recursive programs, we show that the only architectures with a decidable reachability problem are the directed forest architectures. This decidability result is shown using the *bounded context switching decidability* result. We find this surprising: the decidability of bounded context switching (including the notion of a context) stems from a technical result on bounded phase multistack automata, which were defined with no queues in mind, and yet proves to be sufficient to capture all decidable queueing architectures.

Turning to non-recursive architectures, we again provide an exact characterization: the precise class of decidable architectures are those whose underlying *undirected* graph is a forest. The decidability for this result uses a simple idea that queues in architectures can be reversed, and the proof of decidability of tree architectures is considerably simpler as we can build a global finite-state machine simulating the network with bounded-length message queues.

The paper is organized as follows. The next section defines networks of processes communicating via queues, and the reachability and bounded context switching reachability problems. Section 3 establishes our results on bounded context switching reachability of queue systems using multi-stack pushdown automata. Section 4 encompasses our results on the exact class of decidable architectures for recursive and non-recursive programs, and Section 5 ends with some conclusions.

Related Work. The idea of context-bounded analysis for concurrent systems was introduced in [16] where it was shown that it yields a decidable reachability problem for shared memory recursive Boolean programs, and is a generalization of the KISS framework proposed by Shaz Qadeer [17]. The last two years has seen an increasing interest in context-bounded analysis for otherwise intractable systems, including context-bounded analysis for asynchronous dynamic pushdown networks [2], for systems with bounded visible heaps [3], and for a more general notion of context-switching for recursive Boolean programs [12]. A recent paper [14] shows experimentally that a few number of context-switches achieves large coverage of state-space in multithreaded programs.

Message-passing queue systems has been a well-studied problem over the last two decades, and several restrictions based on automata-theoretic analysis have been proposed to verify such systems. These include systems with lossy channels [1], and several restricted models of queue systems, such as systems with a single queue [10], systems where message queues contain only one kind of message [15], half-duplex (and quasi-duplex) systems where only one queue can be active and contain messages at any point [4], and reversal bounded multicounter machines connected via a single queue [9].

Finally, asynchronous programs have been shown to have a decidable reachability problem when at any point only a single recursive process runs, enqueueing tasks onto a bag of tasks, where enqueueing of tasks can only be performed when the local stack is empty [18]. In a recent paper, an under and over approximation scheme based on bounding the counters representing messages has been proposed, and implemented, to solve dataflow analysis problems for asynchronous programs [11].

2 Queue Systems

In this section, we define networks of shared-memory processes communicating via unbounded FIFO queues. The number of processes will be bounded, and the state of each process will be modeled using a global finite-state control that models the control locations of the processes, the local variables, and the global shared memory they access. The global finite-state control can be either a non-recursive program, or a recursive program modeled by each process having its own call-stack that it can push and pop from.

We model any number of queues using which the processes can communicate; each queue has a unique sending process and a unique receiving process. There will be a finite message alphabet, but each queue has an unbounded capacity to store messages. Processes hence communicate either using the shared-memory (which carries only a bounded amount of information) or through queues (which carry unbounded information).

An *architecture* is a structure $(P, Q, \text{Sender}, \text{Receiver})$ where P is a finite set of processes, Q is a finite set of queues, and $\text{Sender}: Q \rightarrow P$ and $\text{Receiver}: Q \rightarrow P$ are two functions that assign a unique *sender* process and *receiver* process for each queue in Q , respectively. We assume that the sender of a queue cannot be its receiver as well: i.e. for every $q \in Q$, $\text{Sender}(q) \neq \text{Receiver}(q)$.

We refer to processes in P using notations such as p, p', p_i, \hat{p} , etc., and queues using q, q' , etc.

Recursive Programs Communicating Via Queues

Let Π be a finite message alphabet. Consider an architecture $\mathcal{A}=(P, Q, \text{Sender}, \text{Receiver})$. An *action of a process* $p \in P$ (over Π) is of one of the following forms:

- $p: \text{send}(q, m)$ where $m \in \Pi$, $q \in Q$, and $\text{Sender}(q) = p$.

- $p: \text{recv}(q, m)$ where $m \in \Pi$, $q \in Q$, and $\text{Receiver}(q) = p$.
- $p: \text{int}$ or $p: \text{call}$ or $p: \text{ret}$.

Intuitively, a “ $p: \text{int}$ ” action is an internal action of process p that does not manipulate queues, “ $p: \text{send}(q, m)$ ” is an action where process p enqueues the message m on queue q (the receiver is predetermined as the receiver process for the queue), and “ $p: \text{recv}(q, m)$ ” corresponds to the action where p receives and dequeues the message m from queue q .

The *stack actions* are those of the form $p: \text{call}$ or $p: \text{ret}$. The stack action $p: \text{call}$ corresponds to a *local call of a procedure* in process p , where the process pushes onto its local stack some data (the valuation of its local variables) and moves to a new state. The stack action $p: \text{ret}$ corresponds to a return from a procedure where the local stack is popped and the process moves to a new state that depends on the current state and the data popped from the stack.

Let Act_p denote the set of actions of p , and let $\text{Act} = \bigcup_{p \in P} \text{Act}_p$ denote the set of all actions. Let Calls denote the set of call actions $\{p: \text{call} \mid p \in P\}$ and Rets denote the set of return actions $\{p: \text{ret} \mid p \in P\}$.

Definition 1. A recursive queueing concurrent program (RQCP) over an architecture $(P, Q, \text{Sender}, \text{Receiver})$ is a structure $(S, s_0, \Pi, \Gamma, \{T_p\}_{p \in P})$, where S is a finite set of states, $s_0 \in S$ is an initial state, Π is a finite message alphabet, and Γ is a finite stack alphabet. If Act_p is the set of actions of process p on the message alphabet Π , then T_p is a set of transitions:

$$T_p \subseteq (S \times (\text{Act}_p \setminus \{p: \text{call}, p: \text{ret}\}) \times S) \cup (S \times \{p: \text{call}\} \times S \times \Gamma) \cup (S \times \{p: \text{ret}\} \times \Gamma \times S).$$

The size of an RQCP as above is the size of the tuple representation.

A *configuration* of an RQCP $R = (S, s_0, \Pi, \Gamma, \{T_p\}_{p \in P})$ is a tuple $(s, \{\sigma_p\}_{p \in P}, \{\mu_q\}_{q \in Q})$ where $s \in S$, for each $p \in P$, $\sigma_p \in \Gamma^*$ is the content of the local stack of p , and for each queue $q \in Q$, $\mu_q \in \Pi^*$ is the content of q .¹

Transitions between configurations are defined as follows:

$$(s, \{\sigma_p\}_{p \in P}, \{\mu_q\}_{q \in Q}) \xrightarrow{\text{act}} (s', \{\sigma'_p\}_{p \in P}, \{\mu'_q\}_{q \in Q}) \text{ if}$$

[Internal] $\text{act} = \hat{p}: \text{int}$ and there is a transition $(s, \hat{p}: \text{int}, s') \in T_{\hat{p}}$ such that

- for every $q \in Q$, $\mu'_q = \mu_q$, and
- for every $p \in P$, $\sigma'_p = \sigma_p$.

[Send] $\text{act} = \hat{p}: \text{send}(\hat{q}, m)$ and there is a transition

$$(s, \hat{p}: \text{send}(\hat{q}, m), s') \in T_{\hat{p}} \text{ such that}$$

- $\mu'_{\hat{q}} = m \cdot \mu_{\hat{q}}$, and for every $q \neq \hat{q}$, $\mu'_q = \mu_q$
- for every $p \in P$, $\sigma'_p = \sigma_p$.

[Receive] $\text{act} = \hat{p}: \text{recv}(\hat{q}, m)$ and there is a transition

$$(s, \hat{p}: \text{recv}(\hat{q}, m), s') \in T_{\hat{p}} \text{ such that}$$

- $\mu_{\hat{q}} = \mu'_{\hat{q}} \cdot m$ and for every $q \neq \hat{q}$, $\mu'_q = \mu_q$
- for every $p \in P$, $\sigma'_p = \sigma_p$.

¹ The top of the stack of p is at the beginning of σ_p , and the last message enqueued onto q is at the beginning of μ_q , by convention.

- [**Call**] $act = \widehat{p}: call$ and there is a transition $(s, \widehat{p}: call, s', \gamma) \in T_{\widehat{p}}$ such that
- $\sigma'_{\widehat{p}} = \gamma\sigma_{\widehat{p}}$, and for every $p \neq \widehat{p}$, $\sigma'_p = \sigma_p$,
 - for every $q \in Q$, $\mu'_q = \mu_q$.
- [**Return**] $act = \widehat{p}: ret$ and there is a transition $(s, \widehat{p}: ret, \gamma, s') \in T_{\widehat{p}}$ such that
- $\sigma_{\widehat{p}} = \gamma\sigma'_{\widehat{p}}$ and for every $p \neq \widehat{p}$, $\sigma'_p = \sigma_p$,
 - for every $q \in Q$, $\mu'_q = \mu_q$.

A *run* of an RQCP is a sequence of transitions $c_0 \xrightarrow{act_1} c_1 \xrightarrow{act_2} c_2 \dots \xrightarrow{act_n} c_n$ with $c_0 = (s, \{\sigma_p\}_{p \in P}, \{\mu_q\}_{q \in Q})$, where $s = s_0$ is the initial state, $\sigma_p = \epsilon$ for each $p \in P$ (initial stacks are empty), and $\mu_q = \epsilon$ for each $q \in Q$ (initial queues are empty). A state \widehat{s} is said to be *reachable* if there is a run $c_0 \xrightarrow{act_1} c_1 \xrightarrow{act_2} c_2 \dots \xrightarrow{act_n} c_n$ such that $c_n = (\widehat{s}, \{\sigma_p\}_{p \in P}, \{\mu_q\}_{q \in Q})$.

The *reachability problem* for recursive programs communicating via queues is to determine, given an RQCP $(S, s_0, \Pi, \Gamma, \{T_p\}_{p \in P})$ and a set of target states $T \subseteq S$, whether any $\widehat{s} \in T$ is reachable.

Non-recursive programs communicating via queues

A non-recursive queueing concurrent program (QCP) over the processes P , message alphabet Π , and queues Q is an RQCP $(S, s_0, \Pi, \Gamma, \{T_p\}_{p \in P})$ in which there are no transitions on calls and returns (i.e. there is no transition on an action of the form $p: call$ or $p: ret$ in T_p). Consequently, we remove the stack alphabet Γ from its description, and its configurations do not involve the local stacks of each process. A QCP hence is of the form $(S, s_0, \Pi, \{T_p\}_{p \in P})$ where $T_p \subseteq (S \times (Act_p \setminus \{p: call, p: ret\}) \times S)$, a configuration of a QCP is of the form $(s, \{\mu_q\}_{q \in Q})$, and the semantics of transitions on configurations are the appropriately simplified versions of the rules for internal, send, and receive actions described above. The reachability problem is analogously defined.

Bounded context switching

It is well-known that the reachability problem for even non-recursive queueing concurrent programs is undecidable (see Lemma 8 later). The undecidability result holds even for a very simple architecture with only two processes p and p' , and two queues, one from p to p' and the other from p' to p .

Since reachability is undecidable for queue systems, we study bounded context-switching of queue systems. Intuitively, a context of a queueing system is a sequence of moves where only one process evolves, dequeuing at most one queue q (but possibly enqueueing messages on any number of queues that it can write to). The bounded-context switching reachability problem for an RQCP (or QCP) is the problem of finding whether a target set of states is reachable by some run that switches contexts at most k times, for an a priori fixed bound k .

Formally, for any $p \in P$, $q \in Q$ such that $Receiver(q) = p$, let

$$Act_{p,q} = \{p: int, p: call, p: ret\} \cup \{p: send(q', m) \mid q' \in Q, Sender(q') = p, m \in \Pi\} \\ \cup \{p: recv(q, m) \mid m \in \Pi\}$$

denote the set of actions of p with dequeue actions acting only on queue q . A run $c_0 \xrightarrow{act_1} c_1 \xrightarrow{act_2} \dots \xrightarrow{act_n} c_n$ has at most k context switches if the cardinality of the set $\{i \mid act_i \in Act_{p,q}, act_{i+1} \notin Act_{p,q}, p \in P, q \in Q\}$ is bounded by k .

The bounded context-switching reachability problem is to determine, given an RQCP (or QCP), a target set of states T , and a bound $k \in \mathbb{N}$, whether any state in T is reachable on a run that has at most k context switches.

Well-queueing processes

An RQCP is said to be *well-queueing* if every process p dequeues a message from a queue only when its local stack is empty. Formally, an RQCP is well-queueing if there is no run of the form $c_0 \xrightarrow{act_1} c_1 \xrightarrow{act_2} \dots c_{n-1} \xrightarrow{act_n} c_n$ where $c_{n-1} = (s, \{\sigma_p\}_{p \in P}, \{\mu_q\}_{q \in Q})$, $act_{n-1} = p:recv(q, m)$ and $\sigma_p \neq \epsilon$.

3 The Bounded Context-Switching Reachability Problem

In this section, we recall multi-stack pushdown systems and show that bounded context-switching reachability for both non-recursive and well-queueing recursive concurrent programs can be decided via a reduction to bounded phase reachability for multi-stack pushdown systems [12].

Multi-stack Pushdown Systems

A multi-stack pushdown system is the natural extension of standard pushdown system with multiple stacks. Formally, a *multi-stack pushdown system* (MSPS) is $M = (S, s_0, St, \Gamma, \Delta)$ where S is a finite set of states, $s_0 \in S$ is the initial state, St is a finite set of stacks, Γ is the stack alphabet and $\Delta = \Delta_{int} \cup \Delta_{push} \cup \Delta_{pop}$ is the transition relation with $\Delta_{int} \subseteq S \times S$, $\Delta_{push} \subseteq S \times St \times \Gamma \times S$, and $\Delta_{pop} \subseteq S \times St \times \Gamma \times S$.

A *configuration* c of an MSPS M is a tuple $\langle s, \{\sigma_{st}\}_{st \in St} \rangle$, where $s \in S$ is the current control state of M , and for every $st \in St$, $\sigma_{st} \in \Gamma^*$ denotes the content of stack st (we assume that the leftmost symbol of st is the top of the stack). The *initial configuration* of M is $\langle s_0, \{\sigma_{st}\}_{st \in St} \rangle$, where for every $st \in St$, $\sigma_{st} = \epsilon$ denoting that each stack is empty. The semantics of M is given by defining the transition relation induced by Δ on the set of configurations of M . We write $\langle s, \{\sigma_{st}\}_{st \in St} \rangle \xrightarrow{\delta} \langle s', \{\sigma'_{st}\}_{st \in St} \rangle$ iff one of the following cases holds: (unless it is differently specified, we assume that $\sigma_{st} = \sigma'_{st}$ for every $st \in St$)

[Internal move] δ is $(s, s') \in \Delta_{int}$.

[Push onto stack \widehat{st}] δ is $(s, \widehat{st}, a, s') \in \Delta_{push}$ and $\sigma'_{\widehat{st}} = a.\sigma_{\widehat{st}}$.

[Pop from stack \widehat{st}] δ is $(s, \widehat{st}, a, s') \in \Delta_{pop}$ and $\sigma_{\widehat{st}} = a.\sigma'_{\widehat{st}}$.

A *run* of an MSPS M is a sequence of transitions $c_0 \xrightarrow{\delta_1} c_1 \xrightarrow{\delta_2} c_2 \dots \xrightarrow{\delta_n} c_n$. A state $\widehat{s} \in S$ is reachable if there exists a run $c_0 \xrightarrow{\delta_1} c_1 \xrightarrow{\delta_2} c_2 \dots \xrightarrow{\delta_n} c_n$ such that c_0 is the initial configuration, and c_n is a configuration of the form $\langle \widehat{s}, \{\sigma_{st}\}_{st \in St} \rangle$.

A *phase* of a run is a portion of the run in which the pop moves are all from the same stack. For a positive integer k , a *k-phase* run is a run that is composed of at most k phases. Formally, an M run $c_0 \xrightarrow{\delta_1} c_1 \xrightarrow{\delta_2} c_2 \dots \xrightarrow{\delta_n} c_n$ is *k-phase* if we can split the sequence $\delta_1 \dots \delta_n$ into $\alpha_1 \dots \alpha_k$ such that: for each $i = 1, \dots, k$,

there is a stack $st \in St$ such that each rule $\delta \in \Delta_{pop}$ within α_i is of the form (s, st, a, s') . Therefore, in a k -phase run the stack from which we pop symbols is changed at most $k - 1$ times (*phase-switches*). A state \hat{s} is k -phase reachable if it is reachable on a k -phase run. The *bounded phase reachability problem* is the problem of determining whether, given an MSPS M , a set of states T , and a positive integer k , there is a state of T that is reachable on a k -phase run.

Theorem 1. [12] *The bounded phase reachability problem for MSPSs is decidable. Moreover, the problem can be solved in time exponential in the number of states and double exponential in the number of phases.*

Decidability of Bounded Context-Switching Reachability

We start showing that context-switching reachability for QCPs is decidable.

Theorem 2. *The bounded context-switching reachability problem for non-recursive queuing concurrent programs is decidable. Moreover, the problem can be solved in time double exponential in the number of context-switches, and exponential in the size of the program.*

Proof. We reduce the reachability problem up to k context switches for QCPs to the reachability problem up to $2k + 1$ phases for MSPSs. Fix a QCP A over an architecture $(P, Q, Sender, Receiver)$ and let S be the set of states of A . We construct an MSPS M which simulates A by keeping track of the state of A in its control state, and stores the contents of each queue q in a stack st_q , and has an additional work stack st .

Let us denote a context with (\hat{p}, \hat{q}) , where \hat{p} is the active process dequeuing from q in the context. Fix a run of A and let (\hat{p}, \hat{q}) be the context at a particular point in the run. M is defined such that the following invariant is preserved: the content of any queue $q \neq \hat{q}$ is stored in st_q with the rear at the top, stack $st_{\hat{q}}$ is empty and the content of queue \hat{q} is stored in st with the front at the top.

An internal move of A is simulated by an internal move of M ; sending a message m to a queue $q \neq \hat{q}$ corresponds to push m onto stack st_q ; receiving a message m from \hat{q} corresponds to pop m from the work stack st . Consequently, in one context, there are no phase switches in the simulating machine M . On switching context from (\hat{p}, \hat{q}) to (\hat{p}', \hat{q}') , M moves the content of st onto stack $st_{\hat{q}'}$ and then the content of stack $st_{\hat{q}'}$ onto st . Observe that the first of these two tasks does not cause a change of phase in the run on M since st is the stack which is popped while simulating the context (\hat{p}, \hat{q}) . The second task requires popping from a new stack and thus causes a change of phase.

We can design the described MSPS M such that it has states polynomial in $|S|$. Therefore, reachability in A within k context-switches reduces to reachability within $2k + 1$ phases in M ($k + 1$ phases are required for the $k + 1$ contexts and k additional phases for context switching). The stated complexity bound thus follows from Theorem 1. \square

The construction sketched in the above proof can be adapted to show the decidability of bounded context-switching reachability for well-queueing RQCPs:

Theorem 3. *The bounded context-switching reachability problem for well-queueing recursive concurrent programs is decidable. Moreover, the problem can be solved in time double exponential in the number of context-switches, and exponential in the size of the program.*

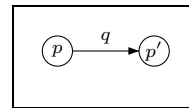
Proof. Let R be a well-queueing RQCP. We will simulate R using an MSPS as in the proof of Theorem 2. The MSPS will have one stack st_q for every queue q , and an extra work stack st , as before, but in addition it will have one stack st_p for every process p .

When the current context is (\hat{p}, \hat{q}) , we will maintain the invariant that the local stack of all processes p ($p \neq \hat{p}$) in the RQCP is stored in the reverse order in stack st_p , and the queue contents of each queue q ($q \neq \hat{q}$) are stored in the stack st_q as before; the stacks $st_{\hat{p}}$ and $st_{\hat{q}}$ will be empty and the stack st will have the content of queue \hat{q} and on top of it the content of the local stack of \hat{p} . Internal moves and enqueueing operations are performed as before, and calls and returns are performed by pushing and popping the work-stack. When process \hat{p} dequeues from queue \hat{q} , its local stack must be empty (by the well-queueing assumption), and hence the next message to be dequeued from q will be at the top of the stack st , and can hence be popped. When the context switches from (\hat{p}, \hat{q}) to (\hat{p}', \hat{q}') , we transfer the top portion of stack st onto stack $st_{\hat{p}'}$ and the bottom portion onto the stack $st_{\hat{q}'}$, and then transfer the contents of stack $st_{\hat{q}'}$ to st followed by the contents of stack $st_{\hat{p}'}$ to st . This requires two extra phases and maintains the invariant. The complexity follows from Theorem 1. \square

The Well-Queueing Assumption and the Notion of Context

Reachability for recursive queueing concurrent programs that are not well-queueing is complex and even the simplest of architectures has an undecidable bounded context-switching reachability problem:

Theorem 4. *The bounded context-switching reachability problem for RQCPs (which need not be well-queueing) is undecidable for the architecture containing two processes p and p' with a single queue connecting p to p' . The undecidability result holds even if we restrict to runs with at most a single context switch.*



Also relaxing the requirement that in each context a process can dequeue at most from one queue immediately leads to undecidability.

Theorem 5. *The bounded context-switching reachability problem for QCPs (hence for well-queueing RQCPs) where a process can dequeue from more than one queue in each context is undecidable. The undecidability result holds even if we restrict to runs with just one context switch and allow processes to dequeue from at most two queues in each context.*

4 Unbounded Context-Switching: Decidable Architectures

In this section, we study the class of architectures for which *unbounded* context-switching reachability (or simply reachability) is decidable. Our goal is to give exact characterizations of decidable architectures for the framework where individual processes are non-recursive, as well as the framework where individual processes are recursive. We restrict ourselves to studying the reachability problem for programs that have *no shared memory* and are *well-queueing*. As we show later in this section (Section 4.3), programs with shared memory and recursive programs that are not well-queueing are undecidable even for the simplest of architectures. We prove that for recursive well-queueing concurrent programs with no shared memory, the class of decidable architectures is precisely the class of directed forest architectures. For the non-recursive queueing concurrent programs with no shared memory, we show that the class of decidable architectures is precisely the polyforest architectures (a polyforest is a set of disjoint polytrees; a polytree is an architecture whose underlying undirected graph is a tree).

Processes with no shared memory: A recursive queueing concurrent program $(S, s_0, \Pi, \Gamma, \{T_p\}_{p \in P})$ is said to have *no shared memory* if its state space is the product of local state-spaces and each move of a process depends only on its local state, and updates only its local state. In other words, $S = \prod_{p \in P} S_p$, where S_p is a finite set of *local* states of process p , and there is a *local* transition relation LT_p (for each $p \in P$) where

$$LT_p \subseteq (S_p \times (Act_p \setminus \{p: call, p: ret\}) \times S_p) \cup (S_p \times \{p: call\} \times S_p \times \Gamma) \cup (S_p \times \{p: ret\} \times \Gamma \times S_p)$$

such that for all $p \in P$ and $s, s' \in S$:

- for every $a \in (Act_p \setminus \{p: call, p: ret\})$,
 $(s, a, s') \in T_p$ iff $(s[p], a, s'[p]) \in LT_p$ and $s'[p'] = s[p']$ for every $p \neq p'$;
- for every $\gamma \in \Gamma$, $(s, p: call, s', \gamma) \in T_p$ iff
 $(s[p], p: call, s'[p], \gamma) \in LT_p$ and $s'[p'] = s[p']$ for every $p \neq p'$;
- for every $\gamma \in \Gamma$, $(s, p: ret, \gamma, s') \in T_p$ iff
 $(s[p], p: call, \gamma, s'[p]) \in LT_p$ and $s'[p'] = s[p']$ for every $p \neq p'$.

In fact, for RQCPs with no shared memory, we can assume that the RQCP is presented in terms of its local transition relations, and model it as a tuple $(\{S_p\}_{p \in P}, s_0, \Pi, \Gamma, \{LT_p\}_{p \in P})$ where $s_0 \in \prod_{p \in P} S_p$. The *size* of an RQCP with no shared memory will be in terms of this representation: i.e. the size of this tuple. Note that this size is possibly exponentially smaller than the size of the RQCP with the global transition relations. The complexity results in this section will refer to the size of RQCPs with no shared memory measured with the local transition relations.

The graph of an architecture: We will characterize decidable architectures based on properties of the underlying graphs. The *graph of an architecture* $\mathcal{A}=(P, Q, \text{Sender}, \text{Receiver})$ is $G=(V, E)$ where $V=P$ and E is the set of labeled edges $E=\{(p, q, p') \mid \text{Sender}(q) = p, \text{Receiver}(q) = p', q \in Q, p, p' \in P\}$.

4.1 Decidable Architectures for Recursive Programs

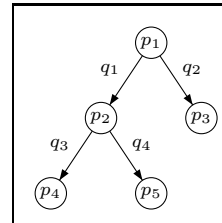
We now show that the only architectures that admit a decidable reachability problem for well-queueing recursive concurrent programs (with no shared memory) are the class of directed forest architectures.

An architecture is said to be a *directed tree architecture* if its graph is a rooted tree, i.e. there is a root process p_0 , every other process p is reachable from p_0 using directed edges, and there is no undirected cycle in the graph. An architecture is said to be a *directed forest architecture* if its graph is the disjoint union of rooted trees. The main theorem of this section is:

Theorem 6. *An architecture admits a decidable reachability problem for well-queueing RQCPs with no shared memory iff it is a directed forest architecture. Moreover, the reachability problem is decidable in time doubly exponential in the number of processes and singly exponential in the size of the RQCP.*

The above theorem is proved using Lemma 1 and Lemma 3 below.

The decidability result is obtained using the decidability of bounded context-switching reachability established in the previous section. Intuitively, given a directed tree architecture, any execution of the processes is equivalent to a run where the root process first runs, enqueueing messages to its children, and then its children run (one after another) dequeuing messages from the incoming queue and writing to their children, and so on. For example, for the directed tree architecture shown on the right, any reachable state of the system can be reached by a run that has 4 context switches: in the first context p_1 runs enqueueing messages on q_1 and q_2 , then p_2 runs dequeuing messages from q_1 and enqueueing messages in q_3 and q_4 , and then in three contexts, p_3 , p_4 and p_5 run, one after the other, dequeuing messages from their incoming queues.



Lemma 1. *The reachability problem for well-queueing RQCPs with no shared memory is decidable for all directed forest architectures, and is decidable in time doubly exponential in the number of processes, and singly exponential in the size of the RQCP.*

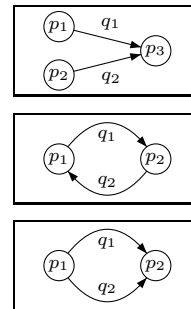
Proof. On directed tree architectures, unbounded reachability reduces to bounded context-switching reachability, where each process in the tree runs at most once, processing messages from its only incoming queue and writing to its outgoing queues. Directed forest architectures can be analyzed by executing its component directed trees one after another. Note that the fact that in a tree there is at most one incoming edge to a node is crucial; and so is the assumption that

there is no shared memory. Hence, given an RQCP M with no shared memory over a directed forest architecture, we can reduce it to the problem of reachability within n contexts (where n is the number of processes) of a new RQCP M' (with shared memory); furthermore, the number of states of M' is linearly proportional to the local states of M . The lemma now follows from Theorem 3. \square

Let us now show that all other architectures are undecidable for well-queueing RQCPs with no shared memory. First, we establish that three architectures are undecidable:

Lemma 2. *The following architectures are undecidable for all well-queueing recursive concurrent programs with no shared memory:*

- the architecture consisting of three processes p_1 , p_2 and p_3 , with a queue from p_1 to p_3 , and another from p_2 to p_3 ;
- the two-process cyclic architecture consisting two processes p_1 and p_2 , with two queues, one from p_1 to p_2 , and the other from p_2 to p_1 ;
- the architecture with processes p_1 and p_2 , with two queues from p_1 to p_2 .



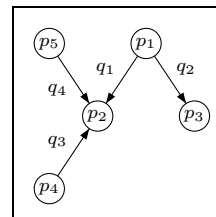
The above lemma can be extended to show that any architecture embedding any of the above architectures is undecidable:

Lemma 3. *Any architecture that has (a) a process with two incoming queues, or (b) a set of processes forming a cycle, or (c) has two distinct paths from one process to another, is undecidable for well-queueing RQCPs with no shared memory. Consequently, any architecture that is not a directed forest is undecidable for recursive well-queueing concurrent programs with no shared memory.*

Lemma 1 and Lemma 3 establish Theorem 6.

4.2 Decidable Architectures for Non-recursive programs

We now turn to the classification of architectures that admit a decidable reachability problem for non-recursive queueing programs with no shared memory. A directed graph is a polytree if it does not have any undirected cycles, i.e. the undirected graph corresponding to it is a tree. A polyforest is a disjoint union of polytrees. An architecture is a polytree (or a polyforest) if its graph is a polytree (or polyforest). We show that the class of decidable architectures for non-recursive programs is precisely the polyforest architectures. For example, the architecture depicted on the right is a polytree architecture, but not a directed tree architecture; hence it admits decidable reachability for non-recursive programs but not for well-queueing recursive programs.



Theorem 7. *The class of architectures that admit a decidable reachability problem for QCPs with no shared memory are precisely the polyforest architectures.*

We prove the above theorem using Lemma 7 and Lemma 9 below.

First, we can reduce the reachability problem to the reachability problem on empty queues (where a state is deemed reachable only if it is reachable with all queues emptied). Any QCP can be transformed so that any individual process, for any of its outgoing queues, stops sending messages (throwing away future messages sent on this queue) and instead sends a special symbol on the queues signaling that this is the last message that will be received by the recipient. All processes must receive these last messages before they reach the target states.

Lemma 4. *The reachability problem for QCPs (or even RQCPs) on any architecture is polynomial time reducible to the reachability problem on empty queues for QCPs (or RQCPs, respectively) on the same architecture.*

Now, we show a crucial lemma: for non-recursive programs the direction of a queue in an architecture does not matter for decidability. Intuitively, consider two architectures that are exactly the same except for a queue q which connects p_1 to p_2 in A_1 , and connects p_2 to p_1 in A_2 instead. Then, reachability on empty queues for a QCP on A_1 can be transformed to reachability on empty queues for a corresponding program over A_2 by letting p_1 receive in A_2 the messages from p_2 which it would have instead sent to p_2 in A_1 : the program at p_1 simply dequeues from q whenever the original program at p_1 enqueued onto q ; similarly process p_2 enqueues onto q whenever the original program at p_2 dequeued q .²

Lemma 5. *Let A_1 and A_2 be two architectures whose underlying undirected graphs are isomorphic. Then, reachability on empty queues for QCPs with no shared memory on A_1 can be effectively (and in polynomial time) reduced to reachability on empty queues for QCPs with no shared memory on A_2 .*

We now show that reachability on all directed-forest architectures is decidable, which, combined with the above lemmas will show that all polyforest architectures are decidable. While this already follows from our result for recursive programs on directed forest architectures, we can give a much simpler proof for non-recursive architectures. Essentially, a QCP with no shared memory on a directed tree architecture can be simulated by a *finite-state* process that keeps track of the global state of each process and synchronizes processes on sending/receiving messages. Since a process lower in a tree can never enable or disable a transition in a process higher in the tree, and since there is no shared memory, we can argue that this finite-state process will discover all reachable states. The argument easily extends to directed forest architectures, and it is easy to see that it results in a PSPACE decision procedure. The problem is PSPACE-hard as even reachability of synchronizing finite-state machines is PSPACE-hard [6].

² The reader may wonder why this transformation cannot work recursive programs; it does indeed work. However, it may make a well-queueing program non well-queueing!

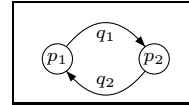
Lemma 6. *The reachability problem for QCPs with no shared memory over directed forest architectures is PSPACE-complete.*

Combining the above with Lemmas 4 and 5, we establish the upper bounds:

Lemma 7. *The reachability problem for QCPs with no shared memory over polyforest architectures is decidable and is PSPACE-complete.*

Let us now show that all but the polyforest architectures are undecidable.

Lemma 8. *The reachability problem for QCPs with no shared memory over the architecture consisting of two processes, p_1 and p_2 , with one queue from p_1 to p_2 , and the other from p_2 to p_1 , is undecidable.*



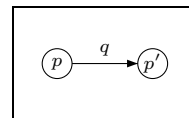
The above proof can be extended to show that any architecture with a directed cycle is undecidable. Combining this with Lemmas 4 and 5 we get that all architectures whose graphs have an undirected cycle are undecidable.

Lemma 9. *The reachability problem for QCPs with no shared memory over any architecture that is not a polyforest architecture is undecidable.*

4.3 The Well-Queueing Assumption and Absence of Shared Memory

This section has dealt with *well-queueing* processes communicating with each other through unbounded queues and *without any shared memory*. Reachability in shared-memory concurrent queue systems is more complex and even the simplest of architectures is undecidable:

Theorem 8. *The reachability problem for QCPs (and hence well-queueing RQCPs) is undecidable for the architecture containing two processes p and p' with a single queue connecting p to p' (depicted on the right). For well-queueing RQCPs, the undecidability result holds even if there are two processes and no queues.*



Similarly, recursive queueing concurrent programs that are not well-queueing are complex too and even the simplest of architectures is undecidable:

Theorem 9. *The reachability problem for RQCPs with no shared memory (which need not be well-queueing) is undecidable for the architecture containing two processes p and p' with a single queue connecting p to p' .*

Consequently, the classification of decidable architectures is interesting only under the assumptions of no shared memory and well-queueing.

5 Conclusions

We have shown that bounded context-switching reachability is decidable for queueing non-recursive programs and well-queueing recursive programs. Using this result, we have precisely characterized the architectures that admit a decidable reachability problem for both recursive and non-recursive programs. Our contribution is theoretical, but addresses an important problem involving a model that can capture both asynchronous programs as well as distributed communicating processes.

The most important future direction we see is in designing *approximate* analysis for queue systems based on the theory we have presented that will work well on domain-specific applications. Two recent papers give us hope: in [18], the authors addressed the reachability problem for asynchronous programs communicating via unbounded *bags* of messages using counter systems, and a year later, a convergent under- and over-approximation of counter contents led to a practical implementation of dataflow analysis for asynchronous programs [11]. A similar scheme for queue systems would be interesting and useful.

References

1. Abdulla, P., Jonsson, B.: Verifying programs with unreliable channels. In: LICS, pp. 160–170. IEEE Computer Society, Los Alamitos (1993)
2. Bouajjani, A., Esparza, J., Schwoon, S., Strejcek, J.: Reachability analysis of multithreaded software with asynchronous communication. In: Ramanujam, R., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821, pp. 348–359. Springer, Heidelberg (2005)
3. Bouajjani, A., Fratani, S., Qadeer, S.: Context-bounded analysis of multithreaded programs with dynamic linked structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 207–220. Springer, Heidelberg (2007)
4. Cécé, G., Finkel, A.: Programs with quasi-stable channels are effectively recognizable (extended abstract). In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 304–315. Springer, Heidelberg (1997)
5. Chadha, R., Viswanathan, M.: Decidability results for well-structured transition systems with auxiliary storage. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR. LNCS, vol. 4703, Springer, Heidelberg (2007)
6. Cheng, A., Esparza, J., Palsberg, J.: Complexity Results for 1-Safe Nets. Theor. Comput. Sci. 147(1-2), 117–136 (1995)
7. Gay, D., Levis, P., von Behren, J.R., Welsh, M., Brewer, E.A., Culler, D.E.: The Nesc language: A holistic approach to networked embedded systems. In: PLDI, pp. 1–11. ACM Press, New York (2003)
8. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading (1979)
9. Ibarra, O.H.: Verification in queue-connected multicounter machines. Int. J. Found. Comput. Sci. 13(1), 115–127 (2002)
10. Ibarra, O.H., Dang, Z., San Pietro, P.: Verification in loosely synchronous queue-connected discrete timed automata. Theor. Comput. Sci. 290(3), 1713–1735 (2003)
11. Jhala, R., Majumdar, R.: Interprocedural analysis of asynchronous programs. In: POPL, pp. 339–350. ACM Press, New York (2007)

12. La Torre, S., Madhusudan, P., Parlato, G.: A robust class of context-sensitive languages. In: LICS, pp. 161–170. IEEE Computer Society Press, Los Alamitos (2007)
13. Libasync, <http://pdos.csail.mit.edu/6.824-2004/async/>.
14. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI, pp. 446–455. ACM, New York (2007)
15. Peng, W., Purushothaman, S.: Analysis of a class of communicating finite state machines. *Acta Inf.* 29(6/7), 499–522 (1992)
16. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
17. Qadeer, S., Wu, D.: Kiss: keep it simple and sequential. In: PLDI, pp. 14–24. ACM, New York (2004)
18. Sen, K., Viswanathan, M.: Model checking multithreaded programs with asynchronous atomic methods. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 300–314. Springer, Heidelberg (2006)
19. Zeldovich, N., Yip, A., Dabek, F., Morris, R., Mazières, D., Kaashoek, M.F.: Multiprocessor support for event-driven programs. In: USENIX (2003)