# Asynchronously Communicating Visibly Pushdown Systems

Domagoj Babić[1] and Zvonimir Rakamarić[2]

[1] Facebook, Inc., USA, babic.domagoj@gmail.com
[2] University of Utah, USA, zvonimir@cs.utah.edu

**Abstract.** We introduce an automata-based formal model suitable for specifying, modeling, analyzing, and verifying asynchronous task-based and message-passing programs. Our model consists of visibly pushdown automata communicating over unbounded reliable point-to-point first-in-first-out queues. Such a combination unifies two branches of research, one focused on task-based models, and the other on models of message-passing programs. Our model generalizes previously proposed models that have decidable reachability in several ways. Unlike task-based models of asynchronous programs, our model allows sending and receiving of messages even when stacks are not empty, without imposing restrictions on the number of context-switches or communication topology. Our model also generalizes the well-known communicating finite-state machines with recognizable channel property allowing (1) individual components to be visibly pushdown automata, which are more suitable for modeling (possibly recursive) programs, (2) the set of words (i.e., languages) of messages on queues to form a visibly pushdown language, which permits modeling of remote procedure calls and simple forms of counting, and (3) the relations formed by tuples of such languages to be synchronized, which permits modeling of complex interactions among processes. In spite of these generalizations, we prove that the composite configuration and control-state reachability are still decidable for our model.

## 1 Introduction

The asynchronous message-passing programming paradigm is becoming a de facto standard for parallel and distributed computing (e.g., cloud applications, web services, scientific computing). Programming such asynchronous systems is, however, difficult. In addition to having to reason about concurrency, programmers typically do not have full control over all the services they use. Therefore, failures are rarely reproducible, rendering debugging all but impossible. In response, programmers succumb to logging interesting events and gathering various statistics, hoping that if something goes wrong the logs will reveal the source of failure.

On the positive side, this is an opportunity for the scientific community to provide appropriate computationally tractable formal models, as well as programming paradigms, languages, and analysis tools based on such models. We

propose such a formal model for asynchronous message-passing programs. The model generalizes several existing well-known models, but we prove that checking the system's safety properties is still decidable. More precisely, we propose an abstract automata-based model, in which individual processes are modeled by visibly pushdown automata (VPA) [2] that communicate via unbounded point-to-point reliable first-in-first-out (FIFO) queues. VPA are single-stack pushdown automata where all stack push and pop operations must be visible (i.e., explicit) in the input language. Such automata are commonly used to represent abstractions (e.g., computed using predicate abstraction [15, 4]) of possibly recursive programs.

Unfortunately, reachability is undecidable even for finite-state machines communicating over unbounded queues (a.k.a. CFSMs) [10]. Researchers proposed a number of restrictions to regain decidability: bounding the size of queues to some fixed size, restricting the communication topology, and restricting the expressiveness of the languages representing the messages on queues. Pachl [21] proved that if a CFSM has a recognizable channel property — all the queue languages are regular and all those languages form a recognizable relation,[1] then reachability is decidable.

Pachl's restrictions are too restrictive in practice. Recognizable relations are a very inexpressive class of relations that can model inter-dependencies among queues only if languages describing the contents of each queue are finite. For instance, if we have an invariant that there should be the same number of messages, say $a$ and $b$, on two queues in some composite control state, the relation representing the configuration of queues would be $(a^n, b^n)$, which is not a recognizable relation. Even simple systems, like a client sending some number of requests and expecting the same number of responses, require queue relations that allow inter-dependencies (i.e., synchronization) among individual queue languages.

We relax Pachl's restrictions by allowing (but not requiring!) queue (and stack) configurations to form synchronized visibly pushdown relations, which significantly broadens the applicability of our model. Although in our model the two extensions, from regular to visibly pushdown languages and from recognizable to synchronized relations, go hand-in-hand, it is worth noting that they are orthogonal and each is valuable on its own. For instance, our relaxation from recognizable to synchronized relations is applicable to other models as well — a straightforward consequence of our results is that reachability of CFSMs with synchronized channel property is decidable.

The main technical contribution of this paper is a proof that model checking safety properties — global control state and global configuration reachability — is decidable for the model we propose. The result is non-trivial as the introduced model allows unbounded stacks and queues, arbitrary communication topologies, as well as complex inter-dependencies of queue and stack languages.

[1] Informally, a relation is recognizable if the concatenation of all the languages that are elements of the relation tuple is a regular language (see Sec. 3.3).

These extensions allow our model to capture the following distributed programming patterns:

**Remote procedure calls.** Processes can (recursively) call procedures to be executed on behalf of remote processes.

**Message counting.** Processes can use their local stack to count the number of messages and assure that the number of responses matches the number of requests.

**Asynchronous communication.** Processes send and receive messages asynchronously. While we present the model with FIFO channels, bag-like (i.e., multiset-like) queues can be simulated by receiving each message (assuming there are finitely many) on a separate queue. The receiver can nondeterministically choose which queue to process next. These features allow us to model both asynchronous communication often used for hiding latency in web services as well as non-deterministic interleaving of messages from different senders in publish-subscribe settings.

**Synchronization.** Often in practice, processes send multiple messages at once to different receivers. CFSMs with recognizable channel property are unable to model that behavior, as it creates dependencies among queues. Our generalization to synchronized relations allows us to handle non-trivial queue dependencies.

We summarize the contributions of this paper as follows:

- A new formal model for asynchronous message-passing programs. It is a relaxation of known communication restrictions along two dimensions: from regular to visibly pushdown languages, and from recognizable to synchronized relations.
- A proof of decidability of control state and configuration reachability in our model.

Our technical report [3] provides a more comprehensive coverage of this material.

## 2 Applications

In this section, we discuss the applications of the introduced model in the context of two major asynchronous programming paradigms: the task-based and the message-passing paradigm.

**Asynchronous Task-Based Paradigm.** The task-based programming paradigm enables programmers to break up time-consuming operations into a collection of shorter tasks. This adds reactivity to the system, and typically improves responsiveness and performance of long-running programs. Tasks can be either asynchronously posted for execution by other tasks, or triggered by events. These two approaches (and their combination) have been successfully employed in many domains: they form the basis of JavaScript and Silverlight (client-side) web applications, and have been shown useful for building fast servers [22], routers [18], and embedded sensor networks [17].

The formal system we propose can model this class of applications as follows. Each task (and there is a finite number of them) is executed on a single VPA. During execution, each task can change the state of its VPA and send messages

to other VPAs, which is sufficient for modeling the global shared state changes that the task-based models can model. The task buffer is modeled as a FIFO queue: posting a task amounts to sending an invocation message to the task buffer queue.

**Message-Passing Paradigm.** The message-passing paradigm, in which processes communicate exclusively by sending messages to each other, has been implemented in a number of different ways: as an integral part of a programming language (e.g., Erlang, Scala), as a message-passing API implemented as a library (e.g., MPI, SOAP, Java Message Service, Microsoft Message Queuing), or as a software as a service model (e.g., Amazon Simple Queue Service). Message-passing applications can be viewed as a network of processes communicating over FIFO queues. It is straightforward to model such networks as a system of CVPTs: each (recursive) process can be abstracted into a Boolean program that sends and receives messages, and in turn the language of traces such program generates is accepted by a visibly pushdown transducer. For example, Erlang's message send and receive operations (i.e., `!` and `receive`) closely match send and receive operations in our model. It is also natural to map basic MPI asynchronous blocking send and receive operations (i.e., `MPI_Send` and `MPI_Recv`) to our model. Web services, another class of message-passing applications, are Internet-based applications that communicate and exchange data with other available web services in order to implement required functionality. The services typically communicate via asynchronous message-passing (e.g., SOAP, Ajax), and therefore again fit into our model.

## 3 Background and Related Work

The research on abstract models of asynchronous computation has progressed along two, mostly disjoint, paths. The first path stemmed from the classical negative result of Ramalingam [25] stating that the reachability for stack-based finite-data abstractions of concurrent programs with preemption is undecidable. Further research focused on computationally more tractable models, like context-bounded [23] and task-based non-preemptive models. The latter are more relevant to this paper. The second path originated in the study of finite-state machines communicating over reliable unbounded queues [10], known as CFSMs. Reachability is undecidable for general CFSMs. Further research focused on restrictions of CFSMs, especially of queue relations, and development of model checking algorithms exploiting such restrictions. In this paper, we unify the two paths, proposing a model suitable for the same class of applications as the task-based non-preemptive models, while significantly generalizing CFSMs with recognizable channel property, one of the more popular decidable restrictions. We proceed by surveying the related work along both paths, and then providing the necessary background on synchronized relations, which allow us to express complex inter-dependencies among queues and stacks.

### 3.1 Task-Based Models

The task-based model consists of a pushdown automaton and a task buffer for storing asynchronous task invocations. After the currently executing task

returns, a scheduler takes another task from the task buffer and executes it on the automaton. Tasks execute atomically and can change the global state (of the automaton), but new tasks can start executing only when the stack is empty, i.e., the model is non-preemptive. Tasks cannot send messages to each other and communicate only by changing the global state. The model is suitable for modeling event-based applications (e.g., JavaScript programs) and simple worker-pool-based multithreaded applications (e.g., servers).

Sen and Viswanathan [27] proposed a task-based model where the task buffer is modeled as a multiset (i.e., a bag) and showed that the state exploration problem is EXPSPACE-hard. Ganty and Majumdar [14] did a comprehensive study of multiset task-based models, proving EXPSPACE-completeness of safety verification, and proposed a number of extensions. For instance, they show that the configuration reachability problem for the task-based model with task cancellation is undecidable. Our model does not allow message cancellation, but once the execution starts on some automaton, it is possible to send an abort message, which can change the course of execution.

La Torre et al. [19] studied a different set of trade-offs. Similarly to ours, their model allows unbounded reliable queues instead of multisets, but either bounds the number of allowed context-switches or restricts the communication topology to assure computational tractability. Similarly to the multiset-based model, their model can dequeue messages from queues only when the local stack is empty. Each VPA in our model can both send and receive messages, independently of the state of the local stack, as long as the languages of all the queues and stacks in the system can be described by a synchronized relation. Furthermore, we neither impose restrictions on the communication topology, nor require the number of context-switches to be bounded.

### 3.2 Communicating Finite-State Machines

Another line of research on formal models of asynchronous computation focused on CFSMs [10]. A CFSM is a system of finite-state machines operating in parallel and sending messages to each other via unbounded FIFO queues. CFSMs can model preemption, but finite-state machines are an overly coarse abstraction of (possibly recursive) programs. As discussed earlier, reachability is undecidable for CFSMs, in general. Basu et al. [5] show that a sub-class of asynchronous CFSMs can be encoded into synchronous systems, where reachability can be efficiently computed. They present an approach for deciding whether an asynchronous system belongs to that sub-class. Pachl [20, 21] found that if the language of messages on each queue is regular and the tuple of such languages for all queues is a recognizable relation, then the reachability problem is decidable for CFSMs.

His work was followed by extensive research on, so called, regular model checking (e.g., [6, 9, 29, 8]), where queue contents are described using recognizable relations over words. Model checking is then done by computing (sometimes approximations of) a transitive closure of the system's transition relation, and checking whether the image of the transitive closure is contained in the relation describing the queue contents. As the focus of this paper is on proposing a

new formal model for modeling asynchronously communicating programs, and proving that the model has a decidable reachability problem, rather than on algorithms, we omit an extensive account of the regular model checking work. Instead, we direct an interested reader to a survey [1]. We suspect that techniques similar to the ones developed for regular model checking, especially to those for regular tree model checking (e.g., [7]), could be applied to model check the formal model we propose.

We generalize Pachl's results along two dimensions. First, the components of our formal model are visibly pushdown transducers, which can closely model the control flow of recursive programs. Therefore, they are a better candidate for modeling asynchronously communicating programs (e.g., event-based programs, web services, cloud applications, scientific computing applications) than the less powerful finite-state machines. Accordingly, we allow queue relations in our model to be visibly pushdown [2], rather than just regular. This relaxation enables us to support remote procedure calls and limited forms of unbounded message counting. Second, we significantly relax the restrictions on queue relations, allowing more expressive communication patterns. More precisely, we show that our model has a decidable reachability problem even when we move one step up in the hierarchy of families of relations from the family used by Pachl. Such more expressive relations allow us to model complex inter-dependencies among queue and stack configurations.

### 3.3 Relations Over Words and Trees

In this section, we give an overview of the main results on relations over regular sets (of words and trees) relevant to this paper. The properties of those relations are the key to understanding the presented contributions. A property that we are particularly interested in is the decidability of language inclusion ($\subseteq$), which we use in the proof of the decidability of reachability in our formal model. We start with the least expressive family of relations (see Fig. 1).

| ✓ | ✓ | ⊘ | ⊘ |
|---|---|---|---|
| $Rec$ | $\subset Sync \subset$ | $DRat \subset$ | $Rat$ |
| [28] | [12] | [24] | [24] |

**Fig. 1.** Hierarchy of Relations Over Words and Trees. The checkmark (✓) denotes the relations for which inclusion is decidable. Such relations when used to describe the queue and stack languages in our model maintain the decidability of reachability.

*Recognizable relations* (*Rec*) [28] have the weakest expressive power of all families we discuss. Each tape of an $n$-tape automaton operates independently of others and has its own memory. The relations accepted by such automata can be represented as finite unions of cross-products of regular component languages. Effectively, the component languages can be all concatenated together and recognized by a 1-tape automaton. Pachl's work, as well as most work on regular model checking, focuses on this family of relations, which are insufficiently expressive to describe complex inter-dependencies among queues.

*Synchronized relations* (*Sync*) [12] are strictly more expressive than recognizable relations. Synchronized $n$-tape automata over $\Sigma^* \times \cdots \times \Sigma^*$ can be seen as the classical 1-tape automata over the alphabet that is a cross-product of alphabets of all tapes, i.e., $(\Sigma \times \cdots \times \Sigma)^*$. Such automata move all their tape heads

synchronously in lock-step, as if it is a single head reading a tuple of symbols. Synchronized relations are sufficiently expressive to describe languages such as $(a^m, b^m)$, which is useful for describing asynchronously communicating programs in which processes can send multiple messages to different queues at the same time. By adding a special padding symbol (#), synchronized relations can also be used to describe languages such as $(a^m, b^k)$, where $k > m$. Synchronized relations have essentially the same properties as the 1-tape automata (closure under union, intersection, etc.) and their inclusion can be efficiently checked.

Frougny and Sakarovitch [13] defined *resynchronizable relations*, which describe languages of $n$-tape automata whose tapes are not synchronized, but the distance between tape heads is a-priori bounded. Such relations can be characterized as a finite union of the component-wise products of synchronized relations by finite sets, which in turn means they can be reduced to synchronized relations. For instance, $(b^m aab^k, c^m d^k)$ is an example of a resynchronizable relation: after reading $(b^m, c^m)$, the first tape reads two more symbols (increasing the distance between tape heads to two), and then both tapes can again move together in sync. Relations like $((a^*b)^m, c^m)$ are not resynchronizable, as the distance between tape heads can become arbitrarily large. Our proof technique is applicable to all families of relations reducible to synchronized relations.

Rabin and Scott [24] introduced a generalization of the finite automata operating on words (single tape) to tuples of words (multiple tapes). Such automata realize regular transductions. The basic variants of such automata are non-deterministic and deterministic, accepting *rational relations* (*Rat*) and *deterministic rational relations* (*DRat*), respectively. While the equivalence problem of *DRat* is decidable [16], inclusion is unfortunately undecidable for both classes. Therefore, our proof technique cannot be used to prove decidability of reachability in systems of CFSM or visibly pushdown transducers whose queue languages form (deterministic) rational relations.

## 4 The Formal Model

In this section, we describe our formal model. We begin by describing the basic component — a visibly pushdown transducer, continue with a definition of a system of such transducers communicating over reliable unbounded queues, and finish with a discussion of relations describing queue and stack configurations.

**Notations and Terminology.** We define *disjoint union* $A \dot\cup B$ as the standard set union $A \cup B$, but with an implicit side-constraint that the sets $A$ and $B$ are disjoint. Let $I$ be a set. An *$I$-indexed* set $A$ is defined as a disjoint union of sets indexed by elements of $I$, i.e., $A = \dot\cup_{i \in I} A_i$. We denote tuples by a vector sign, e.g., $\vec{t}$. If $\vec{t} = (obj_1, \ldots, obj_n)$ is a tuple, $n$ is called the *size* of the tuple and denoted $|\vec{t}|$. The *cross-product* of sets $A_1, \ldots, A_n$, denoted $\prod_{1 \le i \le n} A_i$, is a set of $n$-tuples $\{(a_1, \ldots, a_n) \mid a_i \in A_i\}$. The $i$-th element of a tuple $\vec{t}$ is denoted $\vec{t}|_i$. We write $S^n$ for a set of $n$-tuples in which all elements are from $S$. Let $\vec{u}$ and $\vec{v}$ be tuples of words such that $|\vec{u}| = |\vec{v}|$; the *component-wise product* of $\vec{u}$ and $\vec{v}$, denoted $\vec{u} \cdot \vec{v}$, is a tuple of words $\vec{t}$ such that for

$1 \leq i \leq |\vec{u}|$, $\vec{t}|_i = \vec{u}|_i \cdot \vec{v}|_i$. The *power of a word* $w$ is defined recursively: $w^0 = \epsilon$, $w^{k+1} = w \cdot w^k$. If $A, B$ are languages, then their concatenation $A \cdot B$ is the language $\{u \cdot v \mid u \in A, v \in B\}$. If $w$ is a word and $A$ is a language, then $w \cdot A = \{w \cdot u \mid u \in A\}$, $A \cdot w = \{u \cdot w \mid u \in A\}$. *(Left) language quotient* $a^{-1}A$ is the language $\{u \mid a \cdot u \in A\}$. Let $u, v \in \Sigma^*$ be words over $\Sigma$. The *prefix order* $\leq$ is defined as: $u \leq v$ iff there exists $w \in \Sigma^*$ such that $v = u \cdot w$. We say that a set $S$ is *prefix-closed* if $u \leq v \wedge v \in S \Rightarrow u \in S$.

### 4.1 Visibly Pushdown Transducers

The individual processes receive and process words of input messages, and generate words of output messages. Thus, they can be modeled as transducers — state machines translating one language into another. We introduce such a machine with a finite-state control, a single stack, and a finite set of unbounded FIFO queues. On each step it can read a symbol from one queue and write to another; if the symbol read is a call (resp. return), it can simultaneously push to (resp. pop from) its stack.

**Definition 1.** *A* communicating visibly pushdown transducer *(CVPT) is a tuple* $T = (\Sigma_{rcv}, \Sigma_{snd}, Q, S, I, F, \Gamma, \Delta)$ *of finite sets, where* $\Sigma_{rcv}$ *is an input alphabet,* $\Sigma_{snd}$ *an output alphabet,* $Q$ *a set of unbounded FIFO queues,* $S$ *a set of states,* $I \subseteq S$ *a set of initial states,* $F \subseteq S$ *a set of final states,* $\Gamma$ *an alphabet of stack symbols, and* $\Delta$ *a transition relation. The input alphabet* $\Sigma_{rcv}$ *and the output alphabet* $\Sigma_{snd}$ *are disjoint sets indexed by* $Q$*, which means that the set of messages that can be sent to any* $q_i \in Q$ *is disjoint from messages that can be sent to all other queues* $Q \setminus q_i$*. Another way to partition the input alphabet is* $\Sigma_{rcv} = \Sigma_c \cup \Sigma_r \cup \Sigma_i$*, where* $\Sigma_c$ *is an alphabet of* call symbols*,* $\Sigma_r$ *an alphabet of* return symbols*, and* $\Sigma_i$ *the* internal *alphabet. For each return in* $\Sigma_r$ *there exists a matching call in* $\Sigma_c$*, more formally:* $\Sigma_r = \{\underline{c} \mid \overline{c} \in \Sigma_c\}$ *and* $|\Sigma_r| = |\Sigma_c|$.[2] *The set of queues* $Q$ *contains a special symbol* $\perp \in Q$ *used in transitions that do not receive input from (or send output to) a queue.*

A *configuration* $C$ *of a CVPT* is a tuple $(s, \sigma, \vec{\varrho}) = (s, \sigma, \varrho_1, \ldots, \varrho_{|Q|}) \in S \times \Gamma^* \times \prod_{q \in Q} (\Sigma_{snd_q} \cup \Sigma_{rcv_q})^*$, representing a control state, a word on the stack, and contents (represented as words) of each of CVPT's queues. For stacks, the leftmost symbol of the word is the top of the stack. For queues, the leftmost symbol of the word represents the next message to be processed (i.e., the oldest yet unprocessed message), while the rightmost symbol represents the most recently received message. To simplify the notation, we assume that $\varrho_i$ represents the contents of queue $q_i \in Q$ and $\varrho$ the contents of queue $q$. We use the $C[\mathit{oldstate} \leftarrow \mathit{newstate}]$ parallel substitution notation to represent incremental modifications of configurations. For example, $C[s_1 \leftarrow s_2, \sigma \leftarrow a \cdot \sigma, \varrho_3 \leftarrow \varrho_3 \cdot b]$ denotes a configuration $C$ modified so that the control state is changed from $s_1$ to $s_2$, message $a$ is pushed on the stack, and message $b$ is appended to queue $q_3$;

[2] We denote call symbols with overline (e.g., $\overline{c}$), and return symbols with underline (e.g., $\underline{c}$).

$C[m \cdot \varrho \leftarrow \varrho]$ denotes a configuration $C$ modified so that message $m$ is removed from queue $q$. We define the transition relation of a CVPT as follows.

**Definition 2 (CVPT Transition Relation).** *Let $C$ be a configuration of a CVPT $T$. If $m \in \Sigma_{snd_q}$, for some $q \in Q$, let $q!m$ (resp. $q?m$) be an alias name for message $m$ sent to (resp. received from) queue $q$.*[3] *If $m = \epsilon$, then $q = \perp$. The transition relation $\Delta = \delta_c \cup \delta_r \cup \delta_i$, such that $\delta_c \subseteq S \times \Sigma_c \times (\Sigma_{snd} \cup \{\epsilon\}) \times \Gamma \times S$, $\delta_r \subseteq S \times \Sigma_r \times \Gamma \times (\Sigma_{snd} \cup \{\epsilon\}) \times S$, and $\delta_i \subseteq S \times (\Sigma_i \cup \{\epsilon\}) \times (\Sigma_{snd} \cup \{\epsilon\}) \times S$, is defined as follows ($\underset{x}{\longrightarrow}$ is the infix notation for $\delta_x$):*

**Call.** *If $(s_1, q_1?\overline{m}_1, q_2!m_2, \gamma, s_2) \in \delta_c$, then $C \xrightarrow[c]{q_1?\overline{m}_1/q_2!m_2,\gamma} C[s_1 \leftarrow s_2, \sigma \leftarrow \gamma \cdot \sigma, \overline{m}_1 \cdot \varrho_1 \leftarrow \varrho_1, \varrho_2 \leftarrow \varrho_2 \cdot m_2]$;*

**Return.** *If $(s_1, q_1?\underline{m}_1, \gamma, q_2!m_2, s_2) \in \delta_r$, then $C \xrightarrow[r]{q_1?\underline{m}_1,\gamma/q_2!m_2} C[s_1 \leftarrow s_2, \gamma \cdot \sigma \leftarrow \sigma, \underline{m}_1 \cdot \varrho_1 \leftarrow \varrho_1, \varrho_2 \leftarrow \varrho_2 \cdot m_2]$;*

**Internal.** *If $(s_1, q_1?m_1, q_2!m_2, s_2) \in \delta_i$, then $C \xrightarrow[i]{q_1?m_1/q_2!m_2} C[s_1 \leftarrow s_2, m_1 \cdot \varrho_1 \leftarrow \varrho_1, \varrho_2 \leftarrow \varrho_2 \cdot m_2]$.*

When we do not care about the exact type of a transition, we use $\longrightarrow$ to represent any of the $\underset{x}{\rightarrow}$ transitions defined above. A *run* of a CVPT on a word $w = a_0 \cdots a_n \in \Sigma^*_{rcv}$ from a configuration $C$ is a finite sequence of configurations $C_0, C_1, \ldots, C_n$, such that $C_0 = C$ and for each $0 < i \leq n$ there exist a transition $C_{i-1} \longrightarrow C_i$. An *accepting run* is a run in which $C_n = (s_n, \sigma_n, \vec{\varrho}_n)$ and $s_n \in F$. A word $w \in \Sigma^*_{rcv}$ is accepted by a CVPT $A$ if there is an accepting run of $A$ on $w$. The language of $A$, denoted $\mathcal{L}(A)$, is the set of words accepted by $A$. We extend the infix notation, defined above for transitions, to words: $C \xrightarrow{w/p} C'$ if there exists a run on $w$ from $C$ to $C'$ yielding output word $p$. When we are interested only in the input word, say $w$, we omit the output word, e.g., $C_1 \xrightarrow{w} C_2$. The transitive closure of the $\longrightarrow$ relation is denoted $\xrightarrow{*}$. Let $[\![T]\!]$ be the transduction induced by $T$: if there is a run $(s_0, \epsilon, \vec{\epsilon}) \xrightarrow{w/p} (s, \sigma, \vec{\varrho})$, where $s_0 \in I$ and $\vec{\epsilon}$ is a tuple of empty strings, then $p \in [\![T]\!](w)$. We generalize the transduction $[\![T]\!]$ to languages as usual, i.e., $[\![T]\!](L) = \{[\![T]\!](w) \mid w \in L\}$.

We use Definition 1 for two purposes. First, we use it to define individual components of a system of asynchronously communicating processes. The set of final states could be empty for such components, if we are interested in the computation those components perform, rather than the language they accept. Second, we use Definition 1 to define visibly pushdown languages (VPLs), which in turn we use to define conditions under which reachability is still decidable for our model. When defining VPLs, the set of final states will be non-empty, but the output alphabet $\Sigma_{snd}$ will be empty.

**Definition 3.** *A CVPT with $\Sigma_{snd} = \emptyset$ is a* visibly pushdown automaton *(VPA). A language of finite words $L \subseteq \Sigma^*_{rcv}$ is a* visibly pushdown language *(VPL) if*

[3] Note that $m = q!m = q?m$ for any $q$ and $m$. The alias names are just a notational convenience.

*there exists a VPA $A$ over $\Sigma_{rcv}$ accepting the language, i.e., if $\exists A.\mathcal{L}(A) = L$. Let $\mathcal{V}$ be the set of all VPL languages.*

We now sketch how to model a (possibly recursive) Boolean program $P$ (i.e., a program with a finite number of variables over a finite domain) as a CVPT $T$. We choose a suitable alphabet of call, return, and internal symbols for representing statements of $P$. Then, every call statement of $P$ is mapped into a $\underset{c}{\rightarrow}$ transition of $T$, and every return statement into a $\underset{r}{\rightarrow}$ transition. All other statements are mapped into simple internal transitions. Furthermore, the model could be extended with pre-initialized queues, with only one receiver and no senders, initialized to the language describing the program to be executed on the receiving automaton. However, such extensions significantly complicate the exposition, without contributing to the expressiveness of our model.

### 4.2 Systems of CVPTs

We compose CVPTs into more complex systems as follows.

**Definition 4 (Asynchronous System of CVPTs).** *An* asynchronous system of CVPTs $M = (T_1, \ldots, T_n)$*, where* $T_i = (\Sigma_{rcv_i}, \Sigma_{snd_i}, \ Q_i, S_i, I_i, F_i, \Gamma_i, \Delta_i)$*, is a tuple of CVPTs, such that each FIFO queue has exactly one receiver and one sender. Any pair of CVPTs, say $T_j$ and $T_k$, can share one or more queues $q \in \bigcup_{1 \leq i \leq n} Q_i$ such that sender's $\Sigma_{snd_{j_q}}$ is equivalent*[4] *to receiver's $\Sigma_{rcv_{k_q}}$.*

Since each queue in the system has a single receiver, we introduce a convention to avoid redundancy in specifying contents of the same queue: when we refer to a CVPT $T_i$ as a part of a system, we consider that $\vec{\varrho}$ in $T_i$'s configuration $(s, \sigma, \vec{\varrho})$ represents only the contents of $T_i$'s input queues, i.e., the queues are considered to belong to the receiver.

A *composite configuration* is a tuple $\vec{C} = (C_1, \ldots, C_n)$. Let $C_i = (s_i, \sigma_i, \vec{\varrho}_i)$ represent the configuration of the $i$-th CVPT in the system. We define the *composite control state* $\vec{s}$ of a system as a tuple of states $(s_1, \ldots, s_n)$, *composite stack configuration* $\vec{\sigma}$ as a tuple of words $(\sigma_1, \ldots, \sigma_n)$, and *composite queue configuration* $\vec{\varrho}$ as a tuple of words $(\varrho_{11}, \ldots, \varrho_{1m_1}, \varrho_{21}, \ \ldots, \varrho_{2m_2}, \ldots, \varrho_{n1}, \ldots, \varrho_{nm_n})$, where $m_i = |\vec{\varrho}_i|$ and $\varrho_{ij} = \vec{\varrho}_i|_j$. For a configuration $\vec{C}$, we write $\vec{C}.s$, $\vec{C}.\sigma$, and $\vec{C}.\varrho$ for the composite control state $\vec{s}$, composite stack configuration $\vec{\sigma}$, and composite queue configuration $\vec{\varrho}$.

We define the transition relation of a system in terms of transition relations of its individual components. Let $\vec{C}_0 = \prod_{1 \leq i \leq n} (s_i, \vec{\epsilon}, \vec{\epsilon})$, where $s_i \in I_i$, be an initial composite configuration. A *run of a system* is a finite sequence $\vec{C}_0 \longrightarrow \vec{C}_1 \longrightarrow \cdots \longrightarrow \vec{C}_k$, where $\longrightarrow$ is defined as in Definition 2, with a minor difference that the output queues belong to another component, and not the one making the transition.

[4] Note we index $\Sigma$ in three different ways: $\Sigma_i$ is the alphabet of $T_i$, $\Sigma_q$ is the alphabet of the messages on queue $q$, and $\Sigma_{i_q}$ is $T_i$'s alphabet projected on the set of messages allowed on $q$.

Now, we have all the formal machinery needed to define the configuration reachability problem for a system of CVPTs. Further discussion will focus on the composite configuration reachability, but we show later that our results can be somewhat generalized (e.g., to the composite control state reachability problem).

*Problem 1.* For a given composite configuration $\vec{C}$ of an asynchronous system $M$ of CVPTs, does there exist a run of $M$ ending in $\vec{C}$?

### 4.3 Relations Describing Configurations

For a given composite state $\vec{s}$ and a particular queue (resp. stack), we refer to the set of all words over messages (resp. stack symbols) describing the possible queue (resp. stack) contents in $\vec{s}$ as a queue (resp. stack) language. To define relations among those languages, we introduce stack and queue relations:

**Definition 5 (Stack and Queue Relations).** *Let $M = (T_1, \ldots, T_n)$ be a system of CVPTs. Let $\vec{C}_0$ be an initial composite configuration. We define the* queue relation $\mathbf{L}_q \subseteq \prod_{1 \leq i \leq n} S_i \times \prod_{q \in Q} \Sigma^*_{rcv_q}$ *as* $\mathbf{L}_q(\vec{s}) = \left\{ \vec{C}.\varrho \mid \vec{C}_0 \xrightarrow{*} \vec{C} \wedge \vec{C}.s = \vec{s} \right\}$, *and the* queue-stack relation $\mathbf{L}_{qs} \subseteq \prod_{1 \leq i \leq n} S_i \times \prod_{q \in Q} \Sigma^*_{rcv_q} \times \prod_{1 \leq i \leq n} \Gamma_i^*$ *as* $\mathbf{L}_{qs}(\vec{s}) = \left\{ \vec{C}.\varrho, \vec{C}.\sigma \mid \vec{C}_0 \xrightarrow{*} \vec{C} \wedge \vec{C}.s = \vec{s} \right\}$, *where $Q = \bigcup_{1 \leq i \leq n} Q_i$ is the set of all queues in $M$.*

In the next section, we introduce a family of synchronized tree relations, which we use to relax Pachl's restrictions (Sec. 3.2). Later, we prove that Problem 1 is decidable, despite our relaxations.

## 5 Tree Relations

In this section, we first develop a connection between VPLs and regular tree languages, building on top of prior work by Alur and Madhusudan [2]. We then define synchronized tree relations, using the appropriate encoding operator [11, p. 75].

### 5.1 Isomorphism Between VPLs and Stack-Tree Languages

VPLs can be characterized in terms of, so called, stack-tree languages. We use this characterization to define the relations we are interest in. We start by defining trees and then develop the connection to VPLs.

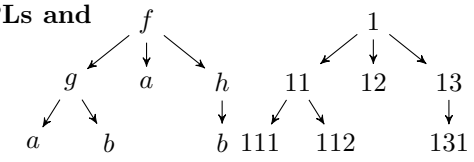


**Fig. 2.** An Example of a Tree $t$ and its Tree Domain. $D = \{1, 11, 111, 112, 12, 13, 131\}$, $\mathcal{F} = \{f, g, h, a, b\}$, $\| t \| = 3$, $t(1) = f$.

**Definition 6 (Trees).** *Let $\mathbb{N}$ be the set of natural numbers. A* tree domain *is a finite non-empty prefix-closed set $D \subset \mathbb{N}^*$ satisfying the following property: if $u \cdot n \in D$ then $\forall 1 \leq j \leq n \;.\; u \cdot j \in D$. A* ranked alphabet *is a finite set $\mathcal{F}$ associated with a finite* ranking relation *arity $\subseteq \mathcal{F} \times \mathbb{N}$. Define $\mathcal{F}_n$ as a set*

$\{f \in \mathcal{F} | \, (f,n) \in \mathit{arity}\}$. *The set* $\mathbb{T}(\mathcal{F})$ *of* terms *over the ranked alphabet* $\mathcal{F}$ *is the smallest set defined by:*

1. $\mathcal{F}_0 \subseteq \mathbb{T}(\mathcal{F})$;
2. *if* $n \geq 1,\ f \in \mathcal{F}_n,\ t_1, \ldots, t_n \in \mathbb{T}(\mathcal{F})$ *then* $f(t_1, \ldots, t_n) \in \mathbb{T}(\mathcal{F})$.

*Each term can be represented as a finite ordered* tree $t : D \to \mathcal{F}$, *which is a mapping from a tree domain into the ranked alphabet such that* $\forall u \in D$:

1. *if* $t(u) \in \mathcal{F}_n,\ n \geq 1$ *then* $\{j \mid u \cdot j \in D\} = \{1, \ldots, n\}$;
2. *if* $t(u) \in \mathcal{F}_0$ *then* $\{j \mid u \cdot j \in D\} = \emptyset$.

*The* height $\| \ t \ \|$ *of a tree* $t = f(t_1, \ldots, t_k)$ *is the number of symbols along the longest branch in the tree, i.e.,* $\mathit{max}\,(\| \ t_1 \ \|, \ldots, \| \ t_k \ \|) + 1$.

Fig. 2 shows an example of a tree and its tree domain.

Given a word $v \in \Sigma^*_{rcv}$, we say that $v$ has *matched returns* (resp. *calls*) if it is a production of the grammar $W ::= a \mid W \cdot W \mid V \mid U,\ \ V ::= a \mid V \cdot V \mid \overline{c} \cdot V \cdot \underline{c}$ such that $U ::= \overline{c}$ (resp. $U ::= \underline{c}$), where $a \in \Sigma_i$, $\overline{c} \in \Sigma_c$, $\underline{c} \in \Sigma_r$. A word is *well-matched* if it has both matched returns and calls. Following Alur and Madhusudan [2], we define an injective map $\eta : \Sigma^*_{rcv} \to \mathbb{T}(\mathcal{F})$, illustrated in Fig. 3, that translates VPL words to *stack-trees* as follows:

$$
\begin{array}{l}
\eta(\epsilon) = \#; \\
\eta(\overline{c}w) = \overline{c}(\eta(w), \#), \text{ if there is no return } \underline{c} \text{ matching } \overline{c} \text{ in } w; \\
\eta(\overline{c}w\underline{c}w') = \overline{c}(\eta(w), \eta(\underline{c}w')), \text{ assuming } w \text{ is well-matched}; \\
\eta(aw) = a(\eta(w)), \text{ if } a \in \Sigma_i \cup \Sigma_r.
\end{array}
$$

The ranked alphabet $\mathcal{F} = \mathcal{F}_0 \cup \mathcal{F}_1 \cup \mathcal{F}_2$ used in the translation is defined as follows: $\mathcal{F}_0 = \{\#\}$, where $\#$ is a special symbol, $\mathcal{F}_1 = \Sigma_i \cup \Sigma_r$, and $\mathcal{F}_2 = \Sigma_c$. Regular sets of stack-trees form stack-tree languages, which are isomorphic to VPLs [2].

We use the $\eta(\Sigma^*_{rcv})$ isomorphism to define, indirectly, VPL relations. Such relations can, broadly, be classified into those recognizable by various types of finite automata and those that are not recognizable. For instance, $(a^n, b^{2^n})$ is an example of a relation not recognizable by any finite-state machine. The *Rec* class of recognizable relations, introduced in Sec. 3.3, can be extended to regular (stack-) tree languages, in which case it correspond to relations that are finite unions of cross-products of regular (stack-) tree languages, denoted $\mathit{Rec}^{\mathcal{V}}$. The $\mathit{Rec}^{\mathcal{V}}$ class is recognizable by a tree automaton, but is insufficiently expressive. In particular, the languages that are elements of the cross-product are independent and cannot express relations like $(a^n, b^n)$. This means that if we restricted the cross-product of queue languages to belong to $\mathit{Rec}^{\mathcal{V}}$, we could not express protocols that send $n$ messages (say $a$) asynchronously and than expect the same number of acknowledgments (say $b$). In other words, $\mathit{Rec}^{\mathcal{V}}$ does not allow us to express even simple forms of counting and synchronization.

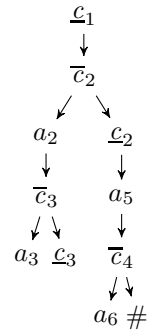


**Fig. 3.** Mapping $\eta$ for a Word: $\underline{c}_1 \cdot \overline{c}_2 \cdot a_2 \cdot \overline{c}_3 \cdot a_3 \cdot \underline{c}_3 \cdot \underline{c}_2 \cdot a_5 \cdot \overline{c}_4 \cdot a_6$.

### 5.2 Synchronized Tree Relations

We define a more expressive class of recognizable relations using *overlap encoding* [11, p. 75], inductively defined for a sequence of binary trees from $\mathbb{T}(\mathcal{F})$ as

$$[t_1, .., t_n] = \begin{cases} t_1(1) \cdots t_n(1) & \text{if } \mathit{arity}\,(t_i(1)) = 0 \\ t_1(1) \cdots t_n(1)\,([t_1(11), .., t_n(11)]) & \text{if } \mathit{arity}\,(t_i(1)) \leq 1 \\ t_1(1) \cdots t_n(1)\,([t_1(11), .., t_n(11)]\,, & \text{otherwise} \\ \quad [t_1(12), .., t_n(12)]) & \end{cases}$$

where $t_i(1k)$ is equal to $\#$ if $k > \mathit{arity}\,(t_i(1))$. An example of the overlap encoding is shown in Fig. 4. Using the notion of the overlap encoding, we can define synchronized tree relations as follows:

**Definition 7 (Synchronized Tree Relations).** *$Sync^{\mathcal{V}}$ is a family of relations $R \subseteq \mathbb{T}(\mathcal{F} \cup \{\#\})^n$ such that $\{[t_1, \ldots, t_n] \mid (t_1, \ldots, t_n) \in R\}$ is recognized by a finite tree automaton over the alphabet $(\mathcal{F} \cup \{\#\})^n$.*

$Sync^{\mathcal{V}}$ inherits all the properties from regular tree languages: it is closed under Boolean operations and both the equality and containment are decidable. Furthermore, $Sync^{\mathcal{V}}$ is known to be a strict superclass of $Rec^{\mathcal{V}}$ [11, p. 79] and allows us to express limited forms of counting, e.g., $(a^n, b^n) \in Sync^{\mathcal{V}}$. We use the introduced family of relations, $Sync^{\mathcal{V}}$, to define sufficient conditions for the decidability of reachability for a system of CVPTs in the next section.

## 6 Decidability of Reachability

In this section, we state and prove the main result of this paper. We begin by introducing sufficient conditions for decidability of reachability of a system of asynchronously communicating CVPTs, state the main theorem in Sec. 6.1, and prove it in the end.

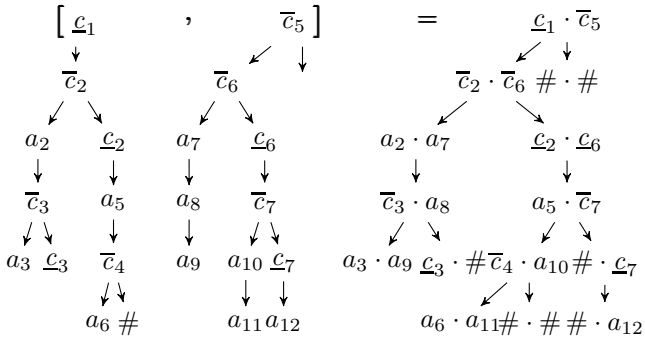


**Fig. 4.** An Example of the Overlap Encoding. The left (resp. middle) tree represents the $\underline{c}_1 \cdot \overline{c}_2 \cdot a_2 \cdot \overline{c}_3 \cdot a_3 \cdot \underline{c}_3 \cdot \underline{c}_2 \cdot a_5 \cdot \overline{c}_4 \cdot a_6$ (resp. $\overline{c}_5 \cdot \overline{c}_6 \cdot a_7 \cdot a_8 \cdot a_9 \cdot \underline{c}_6 \cdot \overline{c}_7 \cdot a_{10} \cdot a_{11} \cdot \underline{c}_7 \cdot a_{12}$) VPL word.

### 6.1 Sufficient Conditions for the Decidability of Reachability

As discussed in Sec. 3.2, reachability is undecidable even for CFSMs. However, if relations representing queue configurations are restricted to regular and recognizable, reachability is decidable. In this section, we relax those restrictions, while maintaining decidability. First, we allow the languages representing contents of each queue to be visibly pushdown, rather than just regular. We require CVPTs not to generate context-free outputs, to assure that CVPTs in a system are composable. Second, we allow relations to be synchronized, rather than just recognizable. These relaxations are orthogonal and each is valuable on its own, but the combination is, of course, more powerful.

**CVPT Composition.**CVPTs are, in general, not closed under composition. As defined in Definition 1, CVPTs accept exactly VPLs. However, even for VPL inputs, CVPTs can generate context-free outputs [26]. As context-free relations do not have the properties we require (e.g., containment is undecidable), we introduce the following requirement:

*Property 1 (Composition Property).* Let $\pi_X : \Sigma^* \to X^*$ be a projection operator that erases all symbols from a word that are not in set $X$. For instance, if $X = \{a, b\}$ then $\pi_X(a \cdot d \cdot d \cdot b \cdot d) = a \cdot b$. Let $M = (T_1, \ldots, T_n)$ be a system of CVPTs. A CVPT $T_j$ is said to be *composable* if a projection of its output language (i.e., a transduction of some VPL $L$) onto the input alphabet of any $T_i$ is a VPL. More formally: $\forall 1 \leq i \leq n \;.\; L \in \mathcal{V} \implies \pi_{\Sigma_{rcv_i}}(\llbracket T_j \rrbracket(L)) \in \mathcal{V}$.

To understand the property better, suppose $G$ is a graph representing a system $M$, such that vertices represent component CVPTs and edges represent communication between CVPTs; there is a directed edge between two nodes $T_i$ and $T_j$ if $T_i$ sends messages to $T_j$. The above property assures that a non-VPL will never be generated on any path in $G$. Further on, we shall assume that all CVPTs have the composition property.

**Synchronization.**According to Property 1, the language representing the contents of each queue is a VPL. Thus, the contents of queues can be described by a cross-product of VPLs in every composite control state. Such VPL relations can be recognizable, synchronized, or rational (see Fig. 1). We use the concept of synchronized tree relations, introduced in Sec. 5.2, to define synchronized VPL relations. As we prove later, if queue and stack relations in every reachable composite control state are synchronized VPL relations, then the reachability is decidable for our model.

*Property 2 (Synchronized Configuration Property).* We say that an asynchronous system of CVPTs has the *synchronized configuration property* iff in every composite control state $\vec{s}$ reachable from an initial configuration $\vec{C}_0$, the encoding $[\eta(\mathbf{L}_{qs}(\vec{s}))]$ is a synchronized tree relation, i.e.,
$\{[\eta(\sigma_1), \ldots, \eta(\sigma_n), \eta(\varrho_1), \ldots, \eta(\varrho_k)] \mid (\sigma_1, \ldots, \sigma_n, \varrho_1, \ldots, \varrho_k) \in \mathbf{L}_{qs}(\vec{s})\} \in Sync^{\mathcal{V}}$.

We now state the main result of this paper:

**Theorem 1.** *Reachability is decidable for a system of composable CVPTs with the synchronized configuration property.*

For the proof, please see our technical report [3].

## 7 Conclusions

In this paper, we proposed a new formal model for asynchronously communicating message-passing programs. The model is composed of visibly pushdown transducers communicating over unbounded reliable point-to-point FIFO queues. The proposed model is intended for specifying, modeling, analysis, and verifying of asynchronous message-passing programs and makes it possible to model (possibly recursive) programs and complex communication patterns. Our results generalize the prior work on communicating finite state machines along two directions — by allowing visibly pushdown languages on queues, and by allowing complex inter-dependencies (i.e., synchronization) among stack and queue languages. Our work also unifies two branches of research — one focused on task-based and the other on queue-based message-passing models. The results are non-trivial, because there are two sources of infiniteness: stacks and queues.

## Acknowledgment

We would like to thank Brad Bingham, Jesse Bingham, Matko Botinčan, Steven McCamant, Jan Pachl, Shaz Qadeer, and Serdar Tasiran for their feedback on the early drafts of this document. The first author did this work while he was at UC Berkeley, where his research was sponsored by LLNL.

## References

1. P. A. Abdulla, B. Jonsson, M. Nilsson, and M. Saksena. A survey of regular model checking. In *Intl. Conf. on Concurrency Theory (CONCUR)*, pages 35–48, 2004.
2. R. Alur and P. Madhusudan. Visibly pushdown languages. In *Annual ACM Symp. on Theory of Computing (STOC)*, pages 202–211, 2004.
3. D. Babić and Z. Rakamarić. Asynchronously communicating visibly pushdown systems. Technical Report UCB/EECS-2011-108, University of California, Berkeley, Oct 2011.
4. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 203–213, 2001.
5. S. Basu, T. Bultan, and M. Ouederni. Synchronizability for verification of asynchronously communicating systems. In *Intl. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 56–71, 2012.
6. B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. The power of QDDs. In *Intl. Symp. on Static Analysis (SAS)*, pages 172–186, 1997.
7. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular tree model checking. *Electronic Notes in Theoretical Computer Science*, 149:37–48, 2006.
8. A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *Intl. Conf. on Computer Aided Verification (CAV)*, pages 372–386, 2004.
9. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *Intl. Conf. on Computer Aided Verification (CAV)*, pages 403–418, 2000.

10. D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of ACM*, 30:323–342, 1983.
11. H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 2007. http://tata.gforge.inria.fr/.
12. S. Eilenberg, C. C. Elgot, and J. C. Shepherdson. Sets recognized by $n$-tape automata. *Journal of Algebra*, 13:447–464, 1969.
13. C. Frougny and J. Sakarovitch. Synchronized rational relations of finite and infinite words. *Theoretical Computer Science*, 108:45–82, 1993.
14. P. Ganty and R. Majumdar. Algorithmic verification of asynchronous programs. *Computing Research Repository (CoRR)*, abs/1011.0551, 2010.
15. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Intl. Conf. on Computer Aided Verification (CAV)*, pages 72–83, 1997.
16. T. Harju and J. Karhumäki. The equivalence problem of multitape finite automata. *Theoretical Computer Science*, 78:347–355, 1991.
17. J. L. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 93–104, 2000.
18. E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.
19. S. La Torre, P. Madhusudan, and G. Parlato. Context-bounded analysis of concurrent queue systems. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 299–314, 2008.
20. J. K. Pachl. Reachability problems for communicating finite state machines. Technical Report CS-82-12, Department of Computer Science, University of Waterloo, 1982.
21. J. K. Pachl. Protocol description and analysis based on a state transition model with channel expressions. In *Intl. Conf. on Protocol Specification, Testing and Verification (PSTV)*, pages 207–219, 1987.
22. V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *USENIX Annual Technical Conference*, pages 199–212, 1999.
23. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 93–107, 2005.
24. M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3:114–125, 1959.
25. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems*, 22:416–430, 2000.
26. J.-F. Raskin and F. Servais. Visibly pushdown transducers. In *Intl. Colloquium on Automata, Languages and Programming (ICALP), Part II*, pages 386–397, 2008.
27. K. Sen and M. Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. In *Intl. Conf. on Computer Aided Verification (CAV)*, pages 300–314, 2006.
28. W. Thomas. On logical definability of trace languages. In *ASMICS workshop, Technical University of Munich, Report TUM-I9002*, pages 172–182, 1990.
29. A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Learning to verify safety properties. In *Intl. Conf. on Formal Engineering Methods (ICFEM)*, pages 274–289, 2004.