Asynchronously Communicating Visibly Pushdown Systems

Domagoj Babić
UC Berkeley
babic@eecs.berkeley.edu

Zvonimir Rakamarić
Carnegie Mellon University
zvonimir.rakamaric@gmail.com

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 is strictly more general than the well-known communicating finite-state machines and allows: (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. And unfortunately, things occasionally do go wrong, often impacting millions of customers, or opening gaping security holes, which have been recently exploited to even "buy" goods on the Internet without paying for them [35].

On the positive side, this is a huge 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. In this paper, 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 [5, 18]) 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 [26] 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.

Enforcing Pachl's restrictions would be too restrictive in our model. First, it would make little sense to model individual processes as VPA, but not being able to remotely call their procedures. To model remote procedure calls, the queue languages need to be visibly pushdown, rather than just regular word languages. Second, 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 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. In practice, our relaxations enable formal analysis and verification of significantly more complex asynchronous message-passing systems than allowed by prior models.

In addition, there is an even larger class of relations that can be algorithmically translated into synchronized relations, i.e., resynchronized. We introduce a new type of automaton, called switching multitape automaton, that characterizes relations that can be resynchronized with prior preprocessing. We show that if a relation can be algorithmically translated into a relation accepted by a switch-

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

ing multitape automaton, then it can also be resynchronized. As our proofs rely solely on the properties of synchronized relations, we postpone the treatment of resynchronizable relations until after we state and prove the main results.

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. We summarize the contributions of this paper as follows:

- A new formal model for asynchronous message-passing programs.
- 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 global control state and global configuration reachability in our model.
- An introduction of switching multitape automata characterizing a family of relations that can be algorithmically resynchronized.

2. 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 [30] stating that the reachability of stack-based finite-data abstractions of concurrent programs with preemption is undecidable. Further research focused on computationally more tractable models, like context-bounded [28] 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. Like the task-based model, 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 the CFSM-based models. 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.

2.1 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 worker-pool-based multithreaded applications (e.g., servers).

Sen and Viswanathan [32] 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 [16] 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. [23] 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 deque messages from queues only when the local stack is empty. Each visibly pushdown automaton 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.

2.2 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. Pachl [25, 26] 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, 8, 9, 34]), 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. Second, we significantly relax the restrictions on queue relations, allowing more expressive communication patterns.

The first major relaxation allows 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. The second major relaxation is related to the queue relations. 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.

2.3 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 most expressive family of relations.

Rabin and Scott [29] introduced a generalization of the finite automata operating on words (single tape) to tuples of words (multiple tapes). There are two basic variants of such automata: nondeterministic and deterministic.[2] The relations accepted by the former are called *rational relations* (*Rat*), and by the later *deterministic rational relations* (*DRat*). While the equivalence problem of *DRat* is decidable [20], inclusion is unfortunately undecidable for both classes. Therefore, our proof technique, which generalizes Pachl's [25] technique, cannot be used to prove decidability of reachability in systems of CFMS or visibly pushdown transducers whose queue languages form (deterministic) rational relations.

Synchronized relations (*Sync*) [13] restrict the tapes of an n-tape automaton to move simultaneously at every step of any computation. Thus, 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 simple forms of counting in asynchronously communicating programs. For example, it is possible to describe a system that sends a number of requests asynchronously and expects to get the same number of responses. 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 classical 1-tape automata (closure under union, intersection, etc.) and their inclusion can be efficiently checked.

Frougny and Sakarovitch [15] 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. They showed that 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.

Finally, *recognizable relations* (*Rec*) [33] have the weakest expressive power of all mentioned families of relations. 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 (with a special delimiter symbol if the alphabets are not disjoint) and recognized by a 1-tape automaton. Pachl's work and, to our knowledge, all the work on regular model checking focuses on this family of relations, which are insufficiently expressive to describe complex inter-dependencies among queues.

3. A Motivating Example

We illustrate the expressive power of our formal model with a small, yet intricate, example. The example models a client and a server asynchronously communicating by sending messages over reliable queues. The client asynchronously sends a potentially unbounded number of data requests. The server receives and immediately acknowledges data requests, but postpones their processing until later. When the data becomes available, the server responds with data. When ready to process the received data, the client removes the data message from the queue and acknowledges the receipt of data. Both the client and the server count messages: the client checks that all the data requests have been served, and the server checks that all the requests have been responded to. Unlike our model, the previously proposed models of asynchronously communicating processes are unsuitable for modeling and verification of this example:

- CFSMs can count only a finite bounded number of messages. The example requires unbounded message counting.
- This example features complex inter-dependencies among queue and stack languages and previously proposed models are too inexpressive to deal with such inter-dependencies.
- Bounded state-space exploration (e.g., bounding queue sizes or the number of allowed context-switches) cannot explore the whole unbounded state-space of the example.
- Both the server and the client send and receive messages with non-empty stacks, which task-based models cannot model.

Fig. 2 illustrates the example. The communication between the client and the server is established using queues q_1, q_2, q_3, and q_4. The call (resp. return) messages push (resp. pop) symbols from a stack, and are denoted by an $\overline{overline}$ (resp. $\underline{underline}$), while messages triggering internal transitions have neither. The server is implemented using two automata communicating over q_5 and q_6. The client initiates asynchronous communication by sending $\overline{req}$ messages to the server. The server immediately acknowledges each $\overline{req}$, by sending $\overline{ack}$, but postpones processing the request until later. By pushing γ_3 or γ_4 on its stack, the server counts the number of pending requests. The server uses γ_3 to denote the bottom of its stack and to detect when all the received requests have been served. Upon receiving $\overline{ack}$, the client pushes γ_1 or γ_2 on its stack to count the number of the requests that the server has initiated processing. The client uses γ_1 to denote the bottom of its stack and to detect when all data requests (acknowledged by the server) have been responded to. When ready, the server responds with the requested data ($\underline{data}$). Upon receiving $\underline{data}$, the client pops γ_1 or γ_2 from its stack, indicating that one of the pending data requests has been responded to, and acknowledges receiving data, by sending $\underline{dack}$. The server pops its stack upon receiving $\underline{dack}$, indicating that one of the pending requests has been served.

The server is fully asynchronous and can postpone serving data. To model such asynchrony, the server would need to nondeterministically pop its stack on an empty (ε) transition, once the data is ready. However, visibly pushdown automata cannot access the stack on empty transitions, but rather all stack accesses have to be visible at the level of the input language. To work around this constraint, the server uses a small one-state repeater automaton on the side. The repeater receives internal messages *dcnt*, and bounces

$$\begin{array}{ccccccc} \checkmark & & \checkmark & & \oslash & & \oslash \\ Rec & \subset & Sync & \subset & DRat & \subset & Rat \\ [33] & & [13] & & [29] & & [29] \end{array}$$

Figure 1. Hierarchy of Relations Over Words and Trees. All the inclusions are known to be strict. The checkmark ($\checkmark$) denotes the families of relations for which inclusion is decidable. Any of those families, as well as relations reducible to those families, can be used to describe the queue and stack languages in our model, while maintaining the decidability of reachability.

[2] In the deterministic multitape automata, the current state determines which tape to read from, and given the state and the input symbol, there is only one possible next state.

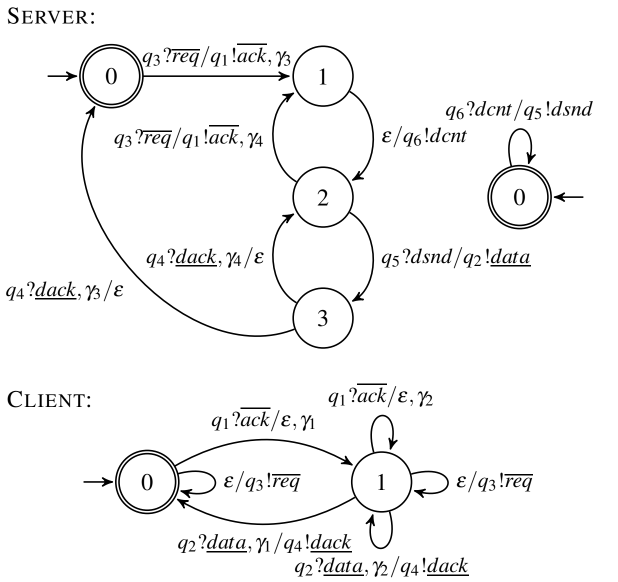

Figure 2. An Example of an Asynchronous Client-Server Service. Initial states are marked with arrows with no origin. Final states are drawn as double circles. Stacks and queues are not shown. A received message is preceded with the question mark and the queue it was received from; a sent message with the exclamation mark and the queue it was sent to. A received overline message (i.e., call message) pushes a symbol on a stack; a received underline message (i.e., return message) pops from a stack. For example, the transition $q_3?\overline{req}/q_1!\overline{ack},\gamma_3$ receives message $\overline{req}$ from queue q_3, sends message $\overline{ack}$ to queue q_1, and pushes γ_3 to the server's stack.

back *dsnd*. Since queues are assumed reliable, all yet unserved requests are pending on either q_5 or q_6. Effectively, those two queues keep track of the data being fetched (or computed) asynchronously. The repeater could as well represent a service fetching data from distributed storage, or a service computing the response. The server can (in state 2) either receive more requests or process one of the messages received from the repeater and send data back to the client. Since languages on both q_5 and q_6 are regular, and since neither the server nor the repeater perform any calls (or returns) on those messages, this workaround satisfies both the restrictions imposed by visibly pushdown automata and the conditions for decidability of reachability that we present later.

An example of a safety property one would want to establish for this protocol is absence of deadlocks (or livelocks). We can verify such properties as follows. Since there is a finite number of composite control states and messages (resp. symbols) to be removed (resp. popped) from queues (resp. stacks) in the next transition, we can enumerate all symbolic configurations leading to a deadlock (or livelock). For instance, if the client is in state 0 (waiting for $\overline{ack}$) and the server in state 3 (waiting for $\underline{dack}$), the service is livelocked if q_1 and q_4 are empty, irrespectively of the contents of other queues. That composite configuration turns out to be unreachable for the given example.

4. Notations and Terminology

Sets. Given a set S, we write $|S|$ to denote its size, i.e., the number of elements in S. We define *disjoint union* $A \uplus B$ as the standard set union $A \cup B$, but with an implicit side-constraint that the sets A and B are disjoint, i.e., $A \cap B = \emptyset$. 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., $A = \uplus_{i \in I} A_i$.

Tuples and Relations. *Tuples* are finite lists of objects. For instance, (obj_1, obj_2, obj_3) is a 3-tuple. We denote tuples by a vector sign, e.g., $\vec{t}$. If $\vec{t} = (obj_1, obj_2, \ldots, obj_n)$ 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 defined as $\{(a_1, \ldots, a_n) \mid a_i \in A_i\}$. 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, i.e., $\{\vec{t} \text{ such that } \vec{t}|_i \in S, 1 \le i \le n\}$. An *n-ary relation* is a subset of a cross-product of n sets.

Strings (Words). We shall often refer to finite sets of symbols as *alphabets* and denote them by Σ, possibly with indices. Let Σ be an alphabet; the set of all finite sequences of elements of Σ will be denoted by Σ^*; such sequences will be referred to as *strings* (or *words*) over Σ. More formally, Σ^* is the free monoid generated by Σ with *concatenation* $(\cdot)$ as the operation and the empty word ε as the identity. We shall also refer to concatenation as *product*. If $w = a_1 \cdot a_2 \cdots a_n$ is a word and $a_i \in \Sigma$, n is called the *length* of w and denoted $|w|$. Let $\vec{u}$ and $\vec{v}$ be tuples of words such that $|\vec{u}| = |\vec{v}|$, the *component-wise product* is defined as $\vec{u} \cdot \vec{v} = \vec{t}$, such that $\vec{t}|_i = \vec{u}|_i \cdot \vec{v}|_i$, for all $1 \le i \le |\vec{u}|$. The *power of a word* w is defined recursively: $w^0 = \varepsilon$, $w^{k+1} = w^k \cdot w$. 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 Σ. The *prefix order* $\le$ is defined as: $u \le v$ for $u, v \in \Sigma^*$ iff there exists $w \in \Sigma^*$ such that $v = u \cdot w$. We say that a set S is *prefix-closed* if $u \le v \wedge v \in S \Rightarrow u \in S$.

5. 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.

5.1 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 (Communicating Visibly Pushdown Transducer). *A communicating visibly pushdown transducer (CVPT) is a tuple of finite sets $T = (\Sigma_{rcv}, \Sigma_{snd}, Q, S, I, F, \Gamma, \Delta)$, where Σ_{rcv} is an input alphabet, Σ_{snd} is an output alphabet, Q is a set of unbounded FIFO queues, S is a set of states, $I \subseteq S$ is a set of initial states, $F \subseteq S$ is a set of final states, Γ is an alphabet of stack symbols, and Δ is a transition relation. The input alphabet Σ_{rcv} and the output alphabet Σ_{snd} are Q-indexed sets such that $\Sigma_{rcv} \cap \Sigma_{snd} = \emptyset$. Another way to partition the input alphabet is $\Sigma_{rcv} = \Sigma_c \uplus \Sigma_r \uplus \Sigma_i$. The Σ_c symbol is an alphabet of* call symbols*, while Σ_r is an alphabet of* return symbols*, such that for each return in Σ_r there exists a matching call in Σ_c, more formally: $\Sigma_r = \{\underline{c} \mid \overline{c} \in \Sigma_c\}$ and $|\Sigma_r| = |\Sigma_c|$. The Σ_i set denotes the* internal *alphabet. 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.*

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.

A *configuration C of a CVPT* is a tuple $(s, \sigma, \vec{\rho}) = (s, \sigma, \rho_1, \ldots, \rho_{|Q|}) \in S \times \Gamma^* \times \prod_{q \in Q} (\Sigma_{snd_q} \cup \Sigma_{rcv_q})^*$, 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 ρ_i represents the contents of queue $q_i \in Q$ and ρ the contents of queue q. We use the $C[oldstate \leftarrow newstate]$ parallel substitution notation to represent incremental modifications of configurations. For example, $C[s_1 \leftarrow s_2, \sigma \leftarrow a \cdot \sigma, \rho_3 \leftarrow \rho_3 \cdot b]$ 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; $C[m \cdot \rho \leftarrow \rho]$ 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 $m = \varepsilon$, then $q = \perp$. The transition relation $\Delta = \delta_c \dot\cup \delta_r \dot\cup \delta_i$, such that $\delta_c \subseteq S \times \Sigma_c \times (\Sigma_{snd} \cup \{\varepsilon\}) \times \Gamma \times S$, $\delta_r \subseteq S \times \Sigma_r \times \Gamma \times (\Sigma_{snd} \cup \{\varepsilon\}) \times S$, and $\delta_i \subseteq S \times (\Sigma_i \cup \{\varepsilon\}) \times (\Sigma_{snd} \cup \{\varepsilon\}) \times S$, is defined as follows ($\underset{x}{\longrightarrow}$ is the infix notation for δ_x):*

Call *If* $(s_1, q_1?\overline{m}_1, q_2!m_2, \gamma, s_2) \in \delta_c$, *then* $C \overset{q_1?\overline{m}_1/q_2!m_2,\gamma}{\underset{c}{\longrightarrow}} C[s_1 \leftarrow s_2, \sigma \leftarrow \gamma \cdot \sigma, \overline{m}_1 \cdot \rho_1 \leftarrow \rho_1, \rho_2 \leftarrow \rho_2 \cdot m_2]$;

Return *If* $(s_1, q_1?\underline{m}_1, \gamma, q_2!m_2, s_2) \in \delta_r$, *then* $C \overset{q_1?\underline{m}_1,\gamma/q_2!m_2}{\underset{r}{\longrightarrow}} C[s_1 \leftarrow s_2, \gamma \cdot \sigma \leftarrow \sigma, \underline{m}_1 \cdot \rho_1 \leftarrow \rho_1, \rho_2 \leftarrow \rho_2 \cdot m_2]$;

Internal *If* $(s_1, q_1?m_1, q_2!m_2, s_2) \in \delta_i$, *then* $C \overset{q_1?m_1/q_2!m_2}{\underset{i}{\longrightarrow}} C[s_1 \leftarrow s_2, m_1 \cdot \rho_1 \leftarrow \rho_1, \rho_2 \leftarrow \rho_2 \cdot m_2]$.

When we do not care about the exact type of a transition, we use $\longrightarrow$ to represent any of the $\underset{x}{\longrightarrow}$ 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 $C_{i-1} \longrightarrow C_i$. We extend the infix notation defined above to words: $C \overset{w/p}{\longrightarrow} C'$ 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., $C_1 \overset{w}{\longrightarrow} C_2$. The transitive closure of the $\longrightarrow$ relation is denoted $\overset{*}{\longrightarrow}$. Let $[\![T]\!]$ be the transduction induced by T: if there is a run $(s_0, \varepsilon, \vec{\varepsilon}) \overset{w/p}{\longrightarrow} (s, \sigma, \vec{\rho})$, where $s_0 \in I$ and $\vec{\varepsilon}$ 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 Σ_{snd} will be empty.

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

DEFINITION 3 (Visibly Pushdown Automaton and Language). *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 there exists a VPA A over Σ_{rcv} accepting the language, i.e., if $\exists A. \mathscr{L}(A) = L$. Let $\mathscr{V}$ be the set of all VPL languages.*

We now informally 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 call transition of T, and every return statement into a return transition. Statements that send or receive messages are mapped into send or receive transitions, respectively. 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.

5.2 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 $\vec{\rho}$ in T_i's configuration $(s, \sigma, \vec{\rho})$ 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 $C_i = (s_i, \sigma_i, \vec{\rho}_i)$ 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 $\vec{\rho}$* as a tuple of words $(\rho_{11}, \ldots, \rho_{1m_1}, \rho_{21}, \ldots, \rho_{2m_2}, \ldots, \rho_{n1}, \ldots, \rho_{nm_n})$, where $m_i = |\vec{\rho}_i|$ and $\rho_{ij} = \vec{\rho}_i|_j$. For a configuration $\vec{C}$, we write $\vec{C}.s$, $\vec{C}.\sigma$, and $\vec{C}.\rho$ for the composite control state $\vec{s}$, composite stack configuration $\vec{\sigma}$, and composite queue configuration $\vec{\rho}$.

We define the transition relation of a system in terms of transition relations of its individual components. Let $\vec{C}_0 = \prod_{1 \leq i \leq n} (s_i, \vec{\varepsilon}, \vec{\varepsilon})$, 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.

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).

[4] Note that we index Σ in three different ways: Σ_i is the alphabet of T_i, Σ_q is the alphabet of the messages on queue q, and Σ_{i_q} is T_i's alphabet projected on the set of messages allowed on queue q.

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

5.3 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 $\mathbf{L}_q \subseteq \prod_{1\le i\le n} S_i \times \prod_{q\in Q} \Sigma^*_{rcv_q}$ *as*

$$\mathbf{L}_q(\vec{s}) = \left\{ \vec{C}.\rho \mid \vec{C}_0 \xrightarrow{*} \vec{C} \wedge \vec{C}.s = \vec{s} \right\},$$

and the queue-stack relation $\mathbf{L}_{qs} \subseteq \prod_{1\le i\le n} S_i \times \prod_{q\in Q} \Sigma^*_{rcv_q} \times \prod_{1\le i\le n} \Gamma^*_i$*:*

$$\mathbf{L}_{qs}(\vec{s}) = \left\{ \vec{C}.\rho, \vec{C}.\sigma \mid \vec{C}_0 \xrightarrow{*} \vec{C} \wedge \vec{C}.s = \vec{s} \right\}.$$

where $Q = \bigcup_{1\le i\le n} Q_i$ 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. 2.2). Later, we prove that Problem 1 is decidable, despite our relaxations.

6. 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 [12, p. 75].

6.1 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.

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 $\forall 1 \le j \le n \;.\; u \cdot j \in D$. A* ranked alphabet *is a finite set $\mathscr{F}$ associated with a finite* ranking relation *$arity \subseteq \mathscr{F} \times \mathbb{N}$. Define $\mathscr{F}_n$ as a set $\{f \in \mathscr{F} \mid (f,n) \in arity\}$. The set $\mathbb{T}(\mathscr{F})$ of* terms *over the ranked alphabet $\mathscr{F}$ is the smallest set defined by:*

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

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

1. *if $t(u) \in \mathscr{F}_n$, $n \ge 1$ then $\{j \mid u \cdot j \in D\} = \{1, \ldots, n\}$;*
2. *if $t(u) \in \mathscr{F}_0$ 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., $max(\| t_1 \|, \ldots, \| t_k \|) + 1$.*

Fig. 4 shows an example of a tree and its tree domain. Following Alur and Madhusudan [2], we define an injective map $\eta : \Sigma^*_{rcv} \to \mathbb{T}(\mathscr{F})$, illustrated in Fig. 3, that translates VPL words to *stack-trees*

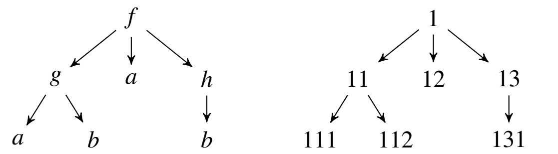

Figure 4. An Example of a Tree t and its Tree Domain. $D = \{1, 11, 111, 112, 12, 13, 131\}$, $\mathscr{F} = \{f, g, h, a, b\}$, $\| t \| = 3$, $t(1) = f$.

as follows:

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

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

We use this 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. 2.3, 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 $Rec^{\mathscr{V}}$. The $Rec^{\mathscr{V}}$ 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 $Rec^{\mathscr{V}}$, 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, $Rec^{\mathscr{V}}$ does not allow us to express even simple forms of counting and synchronization.

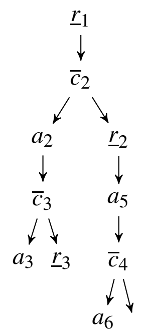

Figure 3. An Illustration of the $\eta : \Sigma^*_{rcv} \to \mathbb{T}(\mathscr{F})$ Mapping for a Word: $\underline{r}_1 \cdot \overline{c}_2 \cdot a_2 \cdot \overline{c}_3 \cdot a_3 \cdot \underline{r}_3 \cdot \underline{r}_2 \cdot a_5 \cdot \overline{c}_4 \cdot a_6$.

6.2 Synchronized Tree Relations

To define a more expressive class of recognizable relations, we use the concept of *overlap encoding* [12, p. 75], inductively defined for binary trees from $\mathbb{T}(\mathscr{F})$ as

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

where $t_i(1k)$ is equal to # if $k > arity(t_i(1))$. An example of the overlap encoding is shown in Fig. 5. Using the notion of the overlap encoding, we can define synchronized tree relations as follows:

DEFINITION 7 (Synchronized Tree Relations). *$Sync^{\mathscr{V}}$ is a family of relations $R \subseteq \mathbb{T}(\mathscr{F} \cup \{\#\})^n$ 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 $(\mathscr{F} \cup \{\#\})^n$.*

$Sync^{\mathscr{V}}$ inherits all the properties from regular tree languages: it is closed under Boolean operations and both the equality and

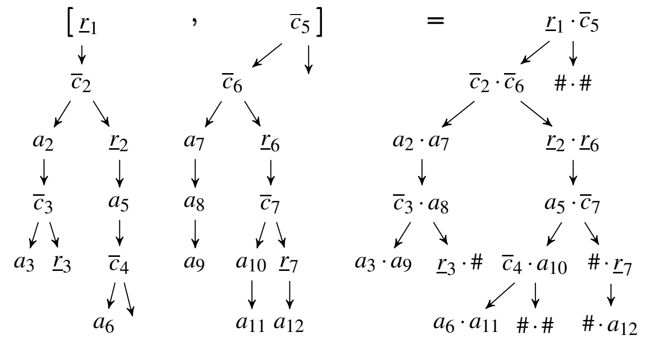

Figure 5. An Example of the Overlap Encoding. The left (resp. middle) tree represents the $\underline{r}_1 \cdot \overline{c}_2 \cdot a_2 \cdot \overline{c}_3 \cdot a_3 \cdot \underline{r}_3 \cdot \underline{r}_2 \cdot a_5 \cdot \overline{c}_4 \cdot a_6$ (resp. $\overline{c}_5 \cdot \overline{c}_6 \cdot a_7 \cdot a_8 \cdot a_9 \cdot \underline{r}_6 \cdot \overline{c}_7 \cdot a_{10} \cdot a_{11} \cdot \underline{r}_7 \cdot a_{12}$) VPL word.

containment are decidable. Furthermore, $Sync^{\mathscr{V}}$ is known to be a strict superclass of $Rec^{\mathscr{V}}$ [12, p. 79] and allows us to express limited forms of counting, e.g., $(a^n, b^n) \in Sync^{\mathscr{V}}$. We use the introduced family of relations, $Sync^{\mathscr{V}}$, to define sufficient conditions for the decidability of reachability for a system of CVPTs in the next section.

7. 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 and state the main theorem in Sec. 7.1.2, which we prove in the section that follows it. We end the section with a discussion on how programmers could help the model checking process by writing suitable invariants, and how the presented results on composite configuration reachability can be used to check composite control state reachability.

7.1 Sufficient Conditions for the Decidability of Reachability

As discussed in Sec. 2.2, 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.

7.1.1 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 [31]. 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 $\pi_X : \Sigma^* \rightarrow X^*$ 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: $\forall 1 \leq i \leq n \,.\, L \in \mathscr{V} \implies \pi_{\Sigma_{rcv_i}}(\llbracket T_j \rrbracket (L)) \in \mathscr{V}$.*

To understand the property better, suppose G is a graph representing a system M, such that vertices represent component CVPTs

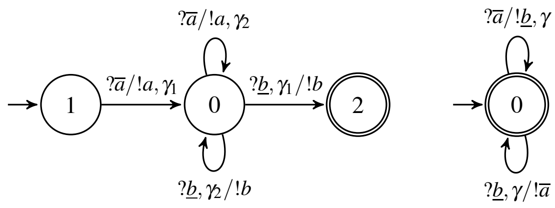

Figure 6. Examples of CVPTs not Satisfying the Composition Property. The left CVPT generates a context-free language $a^n b^n$, $a, b \in \Sigma_i$, while the right CVPT reverses calls and returns, generating $\underline{b}^n \overline{a}^n$, which is not a VPL. Each shown CVPT has one input and one output queue, which are omitted in the transition labels.

and edges represent communication between CVPTs; there is a directed edge between two nodes T_j and T_k if T_j sends messages to T_k. The above property assures that a non-VPL will never be generated on any path in G. Further on, we shall consider all CVPTs to have the composition property. Fig. 6 illustrates a few examples of CVPTs that do not have the composition property.

7.1.2 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. Similarly to word relations, such VPL relations can be recognizable, synchronized, or rational (see Fig. 1). We use the concept of synchronized tree relations, introduced in Sec. 6.2, 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.,*

$$\{[\eta(\sigma_1), \ldots, \eta(\sigma_n), \eta(\rho_1), \ldots, \eta(\rho_k)] \mid (\sigma_1, \ldots, \sigma_n, \rho_1, \ldots, \rho_k) \in \mathbf{L}_{qs}(\vec{s})\} \in Sync^{\mathscr{V}}.$$

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.*

7.2 Proof

The proof is structured similarly as Pachl's proof of decidability of reachability for CFSMs with a recognizable channel property[5] [25]. The structure of the proof is as follows. Before proving Theorem 1, we prove two helper lemmas. The first lemma proves that given a synchronized relation $\mathbf{L}$, it is decidable to check whether it is a sound over-approximation of the set of reachable composite configurations, i.e., whether it is consistent. The second lemma proves that a composite configuration $\vec{C}$ is unreachable if and only if there exists a consistent synchronized relation $\mathbf{L}$, such that $(\vec{C}.\sigma, \vec{C}.\rho) \notin \mathbf{L}(\vec{C}.s)$. Finally, the proof combines two semialgorithms, one of which is guaranteed to terminate. The first semialgorithm terminates if $\vec{C}$ is reachable, and the second if it is unreachable. The second semialgorithm enumerates synchronized relations and checks consistency of each relation, which is decidable

[5] Informally, a system has the recognizable channel property when a cross-product of queue languages in any composite state is a recognizable relation over words.

according to the first lemma. If $\vec{C}$ is unreachable, the semialgorithm is guaranteed to eventually find a relation that does not include $\vec{C}$, which exists according to the second lemma. We want to stress that although the proof is constructive, we focus on proving that reachability is decidable for our model, and ignore the issues of complexity. In other words, our proof is unlikely to serve as a starting point for an efficient algorithm. We discuss the issues of practicality and efficiency in Sec. 7.3, to which a reader not interested in the technicalities of the proof can safely jump to.

DEFINITION 8 (Consistency). *Let $M = (T_1, \ldots, T_n)$ be a system of CVPTs. We say that relation $\mathbf{L} \subseteq \prod_{1 \le i \le n} S_i \times \prod_{q \in Q} \Sigma^*_{rcv_q} \times \prod_{1 \le i \le n} \Gamma^*_i$ is* consistent *(with respect to M) if $\vec{C}_1 \xrightarrow{*} \vec{C}_2$ and $\left(\vec{C}_1.\rho, \vec{C}_1.\sigma\right) \in \mathbf{L}(\vec{C}_1.s)$ imply $\left(\vec{C}_2.\rho,\ \vec{C}_2.\sigma\right) \in \mathbf{L}(\vec{C}_2.s)$.*

Intuitively, a relation is consistent if it over-approximates the set of reachable composite configurations. For synchronized VPL relations, checking consistency effectively reduces to a few simple operations (quotient and concatenation) and language inclusion, all efficiently computable.

LEMMA 1. *Let M be a system of (restricted) CVPTs with the synchronized configuration property. Checking consistency (with respect to M) of $\mathbf{L}$ is decidable.*

PROOF Since each component has a finite number of control states, the number of composite control states is also finite. The number of possible transitions from those states is finite as well. Thus, by checking consistency with respect to every individual transition, we can check $\mathbf{L}$'s consistency.

Accordingly, the $\mathbf{L}$ relation is consistent iff for every two composite control states $\vec{s}_1$ and $\vec{s}_2$, such that $\vec{C}_1.s = \vec{s}_1$, $\vec{C}_2.s = \vec{s}_2$, and $\vec{C}_1 \longrightarrow \vec{C}_2$, it follows that $\left(\vec{C}_1.\rho, \vec{C}_1.\sigma\right) \in \mathbf{L}(\vec{s}_1)$ implies $\left(\vec{C}_2.\rho, \vec{C}_2.\sigma\right) \in \mathbf{L}(\vec{s}_2)$. There are three possible ways how a transition can change the queue configuration: send — appends a message, receive — removes a message, send-receive — does both, and two possible ways how a transition can change the stack configuration: call — pushes a symbol on a stack, and return — pops a symbol. All these six combinations can be synthesized by composing four basic operations: append to and remove from a queue, and push to and pop from a stack. Thus, we can check consistency by composing these basic operations and checking consistency of their compositions:

- *append* $(\mathbf{L}(\vec{s}), q_k, m) =$
  $\left\{(\vec{\sigma}, \vec{\rho}_1) \;\middle|\; (\vec{\sigma}, \vec{\rho}) \in \mathbf{L}(\vec{s}), \vec{\rho}_1|_k = \vec{\rho}|_k \cdot m, \vec{\rho}_1|_j = \vec{\rho}|_j \text{ for } j \neq k\right\}$
  where q_k denotes the queue to which a message m is appended,
- *remove* $(\mathbf{L}(\vec{s}), q_k, m) =$
  $\left\{(\sigma, \vec{\rho}_1) \;\middle|\; (\sigma, \vec{\rho}) \in \mathbf{L}(\vec{s}), m \cdot \vec{\rho}_1|_k = \vec{\rho}|_k, \vec{\rho}_1|_j = \vec{\rho}|_j \text{ for } j \neq k\right\}$,
- *push* $(\mathbf{L}(\vec{s}), \sigma_k, \gamma) =$
  $\left\{(\vec{\sigma}_1, \vec{\rho}) \;\middle|\; (\vec{\sigma}, \vec{\rho}) \in \mathbf{L}(\vec{s}), \vec{\sigma}_1|_k = \gamma \cdot \vec{\sigma}|_k, \vec{\sigma}_1|_j = \vec{\sigma}|_j \text{ for } j \neq k\right\}$
  where σ_k denotes T_k's stack and $\gamma \in \Gamma_k$, and
- *pop* $(\mathbf{L}(\vec{s}), \sigma_k) =$
  $\left\{(\vec{\sigma}_1, \vec{\rho}) \;\middle|\; (\vec{\sigma}, \vec{\rho}) \in \mathbf{L}(\vec{s}), \gamma \cdot \vec{\sigma}_1|_k = \vec{\sigma}|_k, \vec{\sigma}_1|_j = \vec{\sigma}|_j \text{ for } j \neq k\right\}$.

The effect of any transition on the queues and stacks of the system can be composed of the operations above. Computing the effect of these operations amounts to applying the quotient and concatenation operations, which are both efficiently computable for synchronized tree relations.

Thus, given $\mathbf{L}(\vec{s}_1)$, we can compute how the relation $\mathbf{L}$ changes after transitioning to $\vec{s}_2$. Let us name the computed relation $\mathbf{R}$. Hence, to check $\left(\vec{C}_2.\rho, \vec{C}_2.\sigma\right) \in \mathbf{L}(\vec{s}_2)$, it suffices to check $\mathbf{R} \subseteq \mathbf{L}(\vec{s}_2)$. Inclusion is decidable for synchronized tree relations and can be performed efficiently. □

The following lemma is the key component of the later proof of Theorem 1. The lemma says that if a composite configuration $\vec{C}$ is unreachable, there must exist a consistent relation $\mathbf{L}$ that does not include $\vec{C}$.

LEMMA 2. *Let M be a system of CVPTs. A configuration $\vec{C}$ is unreachable from an initial configuration iff there exists a consistent synchronized relation $\mathbf{L}$ and $\left(\vec{C}.\rho, \vec{C}.\sigma\right) \notin \mathbf{L}(\vec{C}.s)$.*

PROOF ($\Longleftarrow$) If $\vec{C}$ is reachable, then by the induction on the path by which is reachable, $\left(\vec{C}.\rho, \vec{C}.\sigma\right) \in \mathbf{L}(\vec{C}.s)$.

($\Longrightarrow$) Conversely, if $\vec{C}$ is unreachable, then the $\mathbf{L}_{qs}$ relation in Definition 5 satisfies the lemma condition, i.e., $(\vec{C}_0.\rho, \vec{C}_0.\sigma) \in \mathbf{L}_{qs}\left(\vec{C}_0.s\right)$ and $(\vec{C}.\rho, \vec{C}.\sigma) \notin \mathbf{L}_{qs}\left(\vec{C}.s\right)$, where $\vec{C}_0$ is an initial configuration. □

Finally, we have all the formal machinery to prove the main result of this paper:

PROOF OF THEOREM 1 We develop two semialgorithms, such that one of them always terminates. The first semialgoritm attempts to prove reachability of $\vec{C}$, while the second attempts to prove unreachability of $\vec{C}$.

Given a composite configuration $\vec{C}$, the first semialgorithm searches for a path from an initial composite configuration $\vec{C}_0 \xrightarrow{*} \vec{C}$ in a breadth-first manner. It terminates if $\vec{C}$ is reachable.

To prove unreachability, we use Gold's [17] language identification by enumeration method, which requires that the languages from a particular family can be effectively enumerated. Synchronized tree relations satisfy this requirement. One can keep enumerating (albeit not efficiently) $\mathbf{L}$ relations and check consistency (Lemma 1) of every guessed relation. If $\vec{C}$ is unreachable, there exists a consistent relation $\mathbf{L}$ and $\left(\vec{C}.\rho, \vec{C}.\sigma\right) \notin \mathbf{L}(\vec{C}.s)$ — according to Lemma 2 — and the second semialgorithm will eventually guess it, thereby proving unreachability of $\vec{C}$. □

Next, we discuss the efficiency of the consistency check, explain why it is important, and generalize our definition of reachability to symbolically defined composite configurations.

7.3 Discussion

While finding the relation that describes queue and stack configurations is likely to be computationally expensive (we ignored the complexity issues in the proof, and focused on the question of decidability), once the relation is known, one can efficiently check consistency and reachability. (CFSMs have a similar property, first noted by Pachl [25].) Thus, if programmers provided the relation, which can be seen as a system invariant, reachability can be efficiently checked, by computing quotient, concatenation, and containment of VPL relations. Asking programmers to provide such invariants for all composite states would be unproductive, but Pachl showed that if the invariants are provided for at least one edge in each loop in the composite state reachability graph, the remaining invariants can be automatically constructed. While we have not proven a generalization of Pachl's result to our model, we conjecture that the same principle applies. Therefore, if programmers provided only loop invariants (for the loops in the composite state transition graph), and if our conjecture is correct, reachability could be checked efficiently. Such a loop-invariant-based technique could be a viable path towards designing practical type-state (e.g., [14]) systems that would automatically check the communication contracts. Similar contracts, albeit far less expressive than ours, have been implemented in the Singularity [24] operating system.

An interesting research challenge is how to design a specification language for expressing such contracts (and the invariants that programmers would write). Such a language should not be too expressive; it is undecidable to check whether an arbitrary rational relation is synchronized [11]. Thus, the contract specification language should be able to express only the systems for which the queue and stack relations are algorithmically resynchronizable. The properties of such contracts could be checked automatically (Sec. 7.2). If programmers provided loop invariants, in addition to contracts, such checks could be done more efficiently.

Another point worth discussing is a generalization of the definition of reachability (Problem 1). Even if a target configuration were specified in terms of a tuple $(\vec{s}, \mathbf{L}'(\vec{s}))$, where $\vec{s}$ is a composite state and $\mathbf{L}'(\vec{s})$ is a synchronized relation, rather than in terms of a concrete stack and queue configuration, reachability is still decidable. To see that, we have to look at both components of the proof of Theorem 1. The first component is a semialgorithm that terminates if the target configuration is reachable. As it makes progress through the search space, that semialgorithm enumerates concrete configurations, and checks for each one whether it belongs to the target configuration (membership can be efficiently checked for synchronized relations). The second component is a semialgorithm that terminates if the target configuration is not reachable. As it keeps enumerating consistent $\mathbf{L}$ relations, it can check for each one whether the intersection with $\mathbf{L}'$ is empty (intersection and emptiness checks can be done efficiently for synchronized relations). Thus, it follows that the target configurations can be specified in terms of synchronized relations. It is easy to see that the composite control state reachability problem is equivalent to reachability of $(\vec{s}, (\Sigma^*)^n \times (\Gamma^*)^m)$, where n (resp. m) is the number of queues (resp. stacks).

8. Resynchronizable Relations

Out of the two properties sufficient for decidability of reachability, one requiring that CVPTs are composable (Sec. 7.1.1) and the other requiring that queue contents are representable by synchronized VPL relations (Sec. 7.1.2), the latter is less intuitive to understand. In particular, even during our research we found ourselves thinking hard about what kinds of communication patterns our formal model allows. The situation became even more complicated as we kept discovering relations that can obviously be synchronized, sometimes with a bit of additional computation, but did not directly fit into the definition of synchronized (tree) relations. In this section, we summarize our findings on what kinds of relations we found *resynchronizable*, i.e., reducible to synchronized relations. First, we discuss in greater depth the work of Frougny and Sakarovitch [15] on resynchronization of relations. Second, we introduce a new, to our knowledge, type of multitape finite-state automata that allow us to define an even larger set of resynchronizable relations. The purpose of introducing a new type of automata is purely to characterize a broader family of resynchronizable relations, and a detailed study of their properties is out of scope and focus of this paper.

8.1 Bounded Delay Multitape Automata

In their comprehensive study of synchronized relations, Frougny and Sakarovitch [15] introduce n-tape automata with an a-priori bounded delay, meaning that the allowed distance between the reading heads is always bounded. The relations accepted by such automata, called resynchronizable relations, have a bounded length difference property, i.e., for any tuple $(w_1, \ldots, w_n)$ of words accepted by an automaton with a bounded delay, the length difference of any two words, $|w_j| - |w_k|, j \neq k$, is bounded. Resynchronizable relations can be reduced to a finite union of the component-wise products of synchronized relations by finite sets. To illustrate that point, let us reconsider the example mentioned in Sec. 2: $R = (b^m aab^k, c^m d^k)$. It is easy to see that R can be expanded into a finite union of component-wise products of synchronized relations by finite sets: $(b^m, c^m) \cdot (a^2, \varepsilon^2)$ for $k = 0$, $(b^m, c^m) \cdot (a^2 b, d \cdot \varepsilon^2)$ for $k = 1$, $(b^m, c^m) \cdot (a^2 bb, d^2 \cdot \varepsilon^2)$ for $k = 2$, and $(b^m aab^{k-2}, c^m d^k) \cdot (bb, \varepsilon^2)$ for $k > 2$. The concept of resynchronizability easily generalizes to tree relations. Rational tree relations with a bounded height difference property (the difference in height between any pair of trees in the relation is bounded) can also be resynchronized [3].

Unfortunately, resynchronization (as proposed in [15]) does not help us with more complicated cases, like $(a^{m-k}, b^{k-l}, c^{m-l})$, which can appear even in relatively simple systems of CVPTs. Furthermore, reducing resynchronizable relations to a finite union of synchronized relations is not intuitive — a much more straightforward approach would be just to insert special symbols, where needed. For instance, the example discussed above could be synchronized by inserting two special # symbols: $(b^m aab^k, c^m\#\#d^k)$. Alternatively, those special symbols could be seen as a two-step pause for the right tape of a 2-tape automaton. Next, we introduce switching multitape automata that can switch reading heads on and off after reading special symbols.

8.2 Switching Multitape Automata

For humans, it is relatively easy to see that relations like $(b^m aab^k, c^m d^k)$ or $(a^{m-k}, b^{k-l}, c^{m-l})$ can be resynchronized — we can easily detect patterns in the language that should trigger a change in the behavior of individual tape heads of the automaton accepting the relation. For instance, the first letter a in the first example can be used as a cue for switching off the other (right) tape head, and the first letter b after the aa pattern for switching it on again. Similarly, in the second example, the third tape head is always on, the first one is initially on, while the second is initially off and turns on after the first head has reached the end of its tape. By replacing such pauses by special padding symbols #, we can construct synchronized relations: $(b^m aab^k, c^m\#\#d^k)$ and $(a^{m-k}\#^{k-l}, \#^{m-k}b^{k-l}, c^{m-l})$.

This section provides a characterization of such relations that are resynchronizable by insertion of special symbols. First, we introduce an automaton that can switch its tape heads on and off after reading special switching symbols. Such symbols cannot change the automaton's state, only which heads are enabled, and all enabled heads move synchronously in lock-step. Second, we show that with some restrictions the languages accepted by such automata can be resynchronized.

8.2.1 Automata-Based Characterization

Before giving a formal definition, we describe the intuition behind the introduced automaton. The automaton has a finite number of tape heads, each of which can be on or off at any time. The enabled heads all move together synchronously, as in the classical synchronized n-tape automata (see Sec. 2.3), one square (i.e., symbol) at a time. If any head reads a special switching symbol, the head moves over the special symbol (other heads reading non-switching symbols do not move), and then the automaton switches heads on and off according to the meaning of the special symbol. The switching is solely dependent on the special symbols, and not on the control state in which the automaton is in. More formally:

DEFINITION 9 (Switching Multitape Automata). *Let* $P = \{0, 1\}$ *represent the possible movements of each tape head;* 0 *(resp.* 1*) means that the head stays on the same (resp. moves to the next) symbol on the tape. Let* $W = \{0, 1\}$ *represent possible states of a* *tape head; 0 (resp. 1) means the head is off (resp. on). A* switching multitape automaton *is a tuple of finite sets* $(\Sigma, (\Omega, \prec), W^n \times S, I, F, \delta, switch)$*, where* Σ *is the input alphabet of symbols read from the tapes,* $(\Omega, \prec)$ *a totally ordered (according to* $\prec$*) set of special switching symbols disjoint from the input alphabet,* $W^n \times S$ *a set of states composed of a switch state of tape heads and control state,* $I \subseteq 1^n \times S$ *is a set of initial states,* $F \subseteq S$ *a set of final control states,* $\delta : W^n \times S \times \prod_{1 \le i \le n} (\Sigma \cup \Omega \cup \{\varepsilon\}) \times W^n \times S \times P^n$ *is a transition relation, and* $switch : \Omega \to W^n$ *is a total switching function that for each symbol from* Ω *determines which tape heads are switched on (and off).*

Let $f : W^n \times \prod_{1 \le i \le n} (\Sigma \cup \Omega \cup \{\varepsilon\}) \to \Omega \cup \{\perp\}$ *be a function that reads a symbol from all the tapes whose tape heads are turned on, reads* ε *from the remaining heads, and returns* $\perp$ *if none of the heads read any symbols from* Ω*, or the highest precedence symbol from* Ω *(according to* $\prec$*) read by any head otherwise.*

Let $g : W^n \times \prod_{1 \le i \le n} (\Sigma \cup \Omega \cup \{\varepsilon\}) \to P^n$ *be a function that computes how each head moves if any special symbols are read. The* g *function reads a tuple* $(a_1, \ldots, a_n)$ *from all the tapes — such that* a_i *represents the next symbol on the tape if the head is on, and* ε *if it is off — and returns a tuple* $\prod_{1 \le i \le n} p_i$ *such that* $p_i = 1$ *if* $a_i \in \Omega$*, and* $p_i = 0$ *otherwise.*

Let $\vec{w}_1, \vec{w}_2 \in W^n$ *be two switch states of* n *tape heads,* $\vec{p} \in P^n$ *a vector describing how each tape head moves, and* $\vec{a} \in \prod_{1 \le i \le n} (\Sigma \cup \Omega \cup \{\varepsilon\})$ *an input tuple. The transition relation is defined as follows:*

Move without switching $(\vec{w}_1, s_1, \vec{a}, \vec{w}_1, s_2, \vec{w}_1) \in \delta$ *if* $f(\vec{a}) = \perp$ *and* $(\vec{a}|_i = \varepsilon) \Leftrightarrow (\vec{w}_1|_i = 0)$*. Note that the state of the heads (on or off) does not change and that only the enabled heads move, all in lock-step by one square (i.e., symbol).*

Switch $(\vec{w}_1, s_1, \vec{a}, \vec{w}_2, s_1, \vec{p}) \in \delta$ *if* $f(\vec{a}) = \omega$*,* $switch(\omega) = \vec{w}_2$*,* $g(\vec{a}) = \vec{p}$*. Note that the control state of the automaton does not change, only the tapes that read a special symbol (*$\vec{p}$*) move, and the switch state of heads changes (to* $\vec{w}_2$*), according to the highest precedence switching symbol (*ω*) read.*

A run *of the automaton on a relation* $\vec{a}_0, \ldots, \vec{a}_k$ *is a finite sequence of states* $(\vec{w}_0, s_1), \ldots, (\vec{w}_k, s_k)$*, such that for each* $0 \le i < k$*, there exists a transition* $(\vec{w}_i, s_i, \vec{a}_i, \vec{w}_{i+1}, s_{i+1}, \vec{w}_{i+1})$ *, if* $f(\vec{a}_i) = \perp$*, and* $(\vec{w}_i, s_i, \vec{a}_i, \vec{w}_{i+1}, s_{i+1}, g(\vec{a}_i))$ *otherwise. A* complete run *is a run in which* $s_k \in F$*.*

As in the case of bounded delay multitape automata, the extension of switching multitape automata from words to trees is relatively straightforward. In case of trees, switching symbols are special unary symbols from $\mathscr{F}_1$, but have essentially the same effect as in the word case. Thus, we skip detailed discussion. In the following section, we show that relations accepted by switching multitape automata can be resynchronized.

8.2.2 Resynchronization

The point of introducing a new type of automata was to give a more accurate and encompassing characterization of relations that can be translated to synchronized relations. In this section, we prove that such resynchronization is indeed possible if there is an a-priori bounded number of switches on any complete run.

LEMMA 3. *If switching can happen only an a-priori bounded number of times on any complete run of a switching multitape automaton, then the relation it accepts can be translated (by padding with special # symbols) into a synchronized relation.*

PROOF Let A be a switching multitape automaton. If the relation accepted by A is finite, the conclusion follows. If the accepted relation is infinite, but the number of switches is bounded, it follows that the switches cannot happen within loops in the state transition graph of A. Let $G = (V, E)$ be a state transition graph of A, such that vertices $V = W^n \times S$ represent states and are labeled by switch and control state pairs, and edges $E \subseteq V \times V$ represent transitions labeled by symbols from Ω if it is a switching transition, and $(\Sigma \cup \{\varepsilon\})^n$ if it is a move without switching. In switching transitions $((\vec{w}_1, s_1), (\vec{w}_2, s_1)) \in E$, the control state does not change, and in non-switching transitions $((\vec{w}_1, s_1), (\vec{w}_1, s_2)) \in E$, the switch state does not change.

Since switches cannot happen within loops, all the strongly connected components in the graph can be temporarily replaced by a special super-node (like in hierarchical state machines). Let us call the graph having super-nodes instead of strongly connected components G'. Such a graph is acyclic and finite, and therefore can be expanded into a tree by duplicating parts of the graph and taking the initial state as the root.[6] Let us call the expanded tree G''.

Each branch of G'' has an a-priori bounded number of edges labeled by symbols from Ω. Now, we shall traverse the tree in preorder (root, left subtree, right subtree), performing the following operation: Let ω be the last switching symbol seen during the traversal. During the traversal, we remove all the transitions on input tuples $\vec{a}$ if the positions of the ε symbols in $\vec{a}$ do not match the positions of zeros in *switch*(ω). For instance, if the edge label is $(\varepsilon, a, b, \varepsilon)$ and *switch*$(\omega) = (0,0,1,1)$ (or $(0,1,1,1)$), we remove the edge, while we leave the edge if *switch*$(\omega) = (0,1,1,0)$. We perform the same operation for all the edges in the super-nodes as well.

After the traversal, let us prune away the nodes unreachable from the root. Each branch in the obtained tree has a bounded number of segments, separated by switching symbols, and all transitions in each segment have exactly the same heads active throughout the entire segment. Thus, each branch (together with the super-nodes) can be encoded as a concatenation of a finite number of synchronized relations. The number of branches is finite. It follows that relations accepted by switching multitape automata with a bounded number of switches on any complete run can be translated into a finite union of products of synchronized relations. □

Early on, in Sec. 2.3, we gave an example of a relation that is not resynchronizable: $R = ((a^*b)^m, c^m)$. Interestingly, relation R can be annotated so that a switching multitape automaton allowed to switch tape head states an unbounded number of times accepts the annotated relation: $((\omega_1 a^* \omega_2 b)^m, c^m)$, where ω_1 (resp. ω_2) switches off (resp. on) the right tape head. While it seems that relations allowing an unbounded number of switches are strictly more expressive than synchronized relations, studying their properties is out of scope of this paper.

The relations with an a-priori bounded number of switches clearly generalize the resynchronizable relations introduced by Frougny and Sakarovitch. On the other hand, it is easy to construct relations, like $(a^{m-k}, b^{k-l}, c^{m-l})$, that cannot be resynchronized as a finite union of the component-wise products of synchronized relations by finite sets, but can be resynchronized by introduction of switching symbols: $(a^{m-k}\omega_1, \omega_2 b^{k-l}, c^{m-l})$, where ω_1 switches the first head off and the second on, while ω_2 switches the second head off.

For some simple resynchronizable relations, like those of Frougny and Sakarovitch, it is easy to construct an algorithm that will take an arbitrary rational relation and synchronize it if it has an a-priori bounded length difference property, because the length dif-

[6] If there are multiple initial states, one can always create a single super-root node and the ε^n edges to the initial states.

ference of a rational relation is efficiently computable [15, p. 54]. Thus, it is possible to design an algorithm that will automatically insert all the switching symbols into Frougny and Sakarovitch's relations, although it is in general undecidable whether a rational relation is also synchronized [11]. However, the more general question of which relations can be algorithmically resynchronized (through insertion of switching symbols) is open.

Now, we describe how to symbolically represent one reachable composite configuration of our motivating example from Sec. 3 using a resynchronizable relation. Remaining reachable states can be represented similarly. A composite configuration of this system of CVPTs is a tuple of client, server, and repeater's configurations:

$$((s_1, \sigma_1, \rho_1, \rho_2), (s_2, \sigma_2, \rho_3, \rho_4, \rho_5), (s_3, \sigma_3, \rho_6)),$$

where s_1, s_2, s_3 represent control states, $\sigma_1, \sigma_2, \sigma_3$ words on stacks, and $\rho_1, \ldots, \rho_6$ words on queues. An example of a reachable (symbolic) composite configuration with complex inter-dependencies among queue and stack languages is:

$$\Big(\Big(1, \gamma_1 \gamma_2^{b-d-1}, \overline{ack}^{a-b}, \varepsilon\Big), \Big(2, \gamma_3 \gamma_4^{a-d-1}, \overline{req}^*, \varepsilon, dsnd^{c-d}\Big), \\ \Big(0, \varepsilon, dcnt^{a-c}\Big)\Big),$$

where a is the number of requests received by the server, b is the number of acknowledgments received by the client, c is the number of $dsnd$ messages sent back to the server, and d is the number of sent and acknowledged data messages. The queue-stack relation $\mathbf{L}_{qs}((1,2,0))$ is then

$$\Big(\overline{ack}^{a-b}, \varepsilon, \overline{req}^*, \varepsilon, dsnd^{c-d}, dcnt^{a-c}, \gamma_1 \gamma_2^{b-d-1}, \gamma_3 \gamma_4^{a-d-1}, \varepsilon\Big),$$

which is a resynchronizable relation

$$\Big(\overline{ack}^{a-b}\omega_2, \varepsilon, \overline{req}^*, \varepsilon, \omega_3 dsnd^{c-d}, dcnt^{a-c}\omega_4, \omega_1 \gamma_1 \gamma_2^{b-d-1}, \gamma_3 \gamma_4^{a-d-1}, \varepsilon\Big),$$

where symbol ω_1 switches the seventh head off, ω_2 switches the first head off and the seventh on, ω_3 switches the fifth head off, and ω_4 switches the sixth head off and the fifth on. This relation accurately models the dependencies among queue and stack languages, which is the key property required to precisely compute reachable states of the example.

9. 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. At the end of this section, we note some interesting limitations of our model.

9.1 Asynchronous Task-Based Paradigm

The task-based programming paradigm enables programmers to break up lengthy, unpredictable, 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 [27], routers [22], and embedded sensor networks [21].

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.

9.2 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 then 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 the Boolean 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 straightforward 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.

9.3 Limitations

While researching possible applications of our formal model, we also found an interesting limitation. Namely, it seems like our model is incapable of modeling distributed continuations. A continuation is an abstract representation of the control state (and stack) of a program. Continuations are a powerful concept that enables, for instance, a running task to be paused, its continuation stored, and then resumed later, possibly on a different machine. Our model has a limitation intrinsic to visibly pushdown transducers — a CVPT can push or pop symbols from its stack only when it receives a special call or return input symbol, respectively. Therefore, a CVPT cannot spontaneously empty its stack, which is a prerequisite for modeling continuations.

10. Future Work

We suspect that there exists another family of relations, in between *Sync* and *DRat* in Fig. 1, that still has decidable inclusion. If such a family exists, the queue and stack languages in our model could be even more expressive, allowing even more interesting communication inter-dependencies. We plan to study the switching multi-tape automata with an unbounded number of switches, which could possibly define the missing family. A related question is that of designing algorithms for inserting switching symbols into resynchronizable subclasses of rational relations. An important research question is to identify subclasses for which switching symbols can be inserted efficiently, like that of Frougny and Sakarovitch, and design algorithms for those subclasses.

Our result could probably be somewhat strengthened. For example, we conjecture that if $\mathbf{L}_q$ in Definition 5 is a consistent synchronized tree relation, so is $\mathbf{L}_{qs}$. We base this conjecture on the fact that the language of stack configurations of pushdown automata is regular [4], which can be proved by construction of a reachability set automaton. We managed to do such a construction for our model, proving that the product of stack languages is regular, but that result is too weak to prove the conjecture. Namely, we found no way to allow for synchronization between queue and stack languages, required to prove the conjecture.

Proving complexity bounds and designing model checking algorithms for our model is another important research step towards usage of our model in practice. We expect that a similar class of algorithms as those used in regular model checking, based on a combination of grammatical inference and more standard model checking techniques (e.g., [19]), could be used to model check our model as well.

Once model checking algorithms are developed, we expect that our model could serve as an underlying model for a type-state (e.g., [14]) system, in which programmers could describe contracts that could be checked automatically. Such contracts have been demonstrated in Singularity [24]. The model we introduce is significantly more expressive than the one in Singularity, and would therefore allow for much more complex communication protocols.

11. 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 interdependencies (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. Besides proving decidability of reachability, which is the main technical result of the paper, we also introduced switching multitape automata to characterize a set of relations that can be resynchronized, and therefore are allowed in our model. We believe this paper is opening a number of new, interesting research directions (both theoretical and practical), and could lead to novel languages and tools for design and analysis of asynchronous programs.

Acknowledgments

We would like to thank Brad Bingham, Jesse Bingham, Steven McCamant, Jan Pachl, Shaz Qadeer, and Serdar Tasiran for their feedback on the early drafts of this document, George Necula for sharing his insights on development of distributed message-passing systems in practice, and Jim Larus for pointing out the importance of synchronization in distributed message-passing protocols.

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 *Symp. on Theory of Computing (STOC)*, pages 202–211, 2004.

[3] Y. Andre and F. Bossut. Word-into-tree transducers with bounded difference. In *Intl. Joint Conf. on Theory and Practice of Software Development (TAPSOFT)*, pages 177–188, 1997.

[4] J.-M. Autebert, J. Berstel, and L. Boasson. Context-free languages and pushdown automata. In *Handbook of Formal Languages, Vol. 1*, pages 111–174. Springer-Verlag, 1997.

[5] 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.

[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] O. Carton, C. Choffrut, and S. Grigorieff. Decision problems among the main subfamilies of rational relations. *Informatique Théorique et Applications*, 40(2):255–275, 2006.

[12] 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/.

[13] S. Eilenberg, C. C. Elgot, and J. C. Shepherdson. Sets recognized by n-tape automata. *Journal of Algebra*, 13:447–464, 1969.

[14] J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Typestate verification: Abstraction techniques and complexity results. *Science of Computer Programming*, 58:57–82, 2005.

[15] C. Frougny and J. Sakarovitch. Synchronized rational relations of finite and infinite words. *Theoretical Comp. Sci.*, 108:45–82, 1993.

[16] P. Ganty and R. Majumdar. Algorithmic verification of asynchronous programs. *CoRR*, abs/1011.0551, 2010.

[17] E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[18] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Intl. Conf. on Computer Aided Verification (CAV)*, pages 72–83, 1997.

[19] P. Habermehl and T. Vojnar. Regular model checking using inference of regular languages. *Electronic Notes in Theoretical Computer Science*, 138:21–36, 2005.

[20] T. Harju and J. Karhumäki. The equivalence problem of multitape finite automata. *Theoretical Comp. Sci.*, 78:347–355, 1991.

[21] 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 Prog. Languages and Operating Systems (ASPLOS)*, pages 93–104, 2000.

[22] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Trans. Comp. Sys.*, 18(3):263–297, 2000.

[23] S. La Torre, P. Madhusudan, and G. Parlato. Context-bounded analysis of concurrent queue systems. In *Intl. Conf. on Tools and Alg. for the Construction and Analysis of Systems (TACAS)*, pages 299–314, 2008.

[24] J. Larus and G. Hunt. The Singularity system. *Communications of the ACM*, 53:72–79, 2010.

[25] J. K. Pachl. Reachability problems for communicating finite state machines. Technical Report CS-82-12, Department of Computer Science, University of Waterloo, 1982.

[26] 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.

[27] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *USENIX Annual Technical Conf.*, pages 199–212, 1999.

[28] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *Intl. Conf. on Tools and Alg. for the Construction and Analysis of Systems (TACAS)*, pages 93–107, 2005.

[29] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res. Dev.*, 3:114–125, 1959.

[30] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. on Prog. Lang. Sys.*, 22:416–430, 2000.

[31] J.-F. Raskin and F. Servais. Visibly pushdown transducers. In *Intl. Colloquium on Automata, Languages and Programming (ICALP), Part II*, pages 386–397, 2008.

[32] 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.

[33] W. Thomas. On logical definability of trace languages. In *Tech. Univ. of Munich Report TUM-I9002*, pages 172–182, 1990.

[34] 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.

[35] R. Wang, S. Chen, X. Wang, and S. Qadeer. How to shop for free online – security analysis of cashier-as-a-service based web stores. In *Symp. on Security and Privacy*, 2011.