Controllers for the Verification of Communicating Multi-Pushdown Systems[⋆]

C. Aiswarya[1], Paul Gastin[2], and K. Narayan Kumar[3]

[1] Uppsala University, Sweden
aiswarya.cyriac@it.uu.se
[2] LSV, ENS Cachan, CNRS & INRIA, France
gastin@lsv.ens-cachan.fr
[3] Chennai Mathematical Institute, India
kumar@cmi.ac.in

Abstract. Multi-pushdowns communicating via queues are formal models of multi-threaded programs communicating via channels. They are turing powerful and much of the work on their verification has focussed on under-approximation techniques. Any error detected in the under-approximation implies an error in the system. However the successful verification of the under-approximation is not as useful if the system exhibits unverified behaviours. Our aim is to design *controllers* that observe/restrict the system so that it stays within the verified under-approximation. We identify some important properties that a good controller should satisfy. We consider an extensive under-approximation class, construct a distributed controller with the desired properties and also establish the decidability of verification problems for this class.

1 Introduction

Most of the critical hardware and software consists of several parallel computing units/components. Each of these may execute recursive procedures and may also have several unbounded data-structures to enhance its computing power. Several of such components may be running on the same processor giving rise to a multi-threaded system with many unbounded data-structures. Furthermore, such complex infinite state systems may communicate over a network and be physically distributed. The high computational power in combination with unconstrained interactions make the analysis of these systems very hard.

The verification of such systems is undecidable in general. Even the basic problem of control state reachability (or emptiness checking) is undecidable as soon a program has two stacks or a self queue. However, these systems are so important, that several under-approximation techniques have been invented for their verification. If the under-approximation fails to satisfy a requirement, that immediately indicates an error in the system. However, if the system is verified correct under such restrictions, the correctness is compromised if the system

[⋆] Supported by LIA InForMel, and DIGITEO LoCoReP.

eventually exhibits behaviours outside the class. *Controlling* the system to only exhibit behaviours that have been verified to be correct is therefore crucial to positively use these under-approximation techniques. Alternately, we may use these controllers to raise a signal whenever the system behaviour departs from the verified class. For example, in the cruise control system of a car (or auto-pilot systems in trains/aircrafts), it will be useful to signal such a departure and switch from automatic to manual mode.

Our contributions We aim at obtaining a uniform controller for a class, which when run in parallel with the system, controls it so as to exhibit only those behaviours permitted by the class. Such a controller should possess nice properties like determinism, non-blocking, system independence etc. In Section 3, we identify and analyse such desirable features of a controller.

Our next contribution is to propose a very generous under-approximation class and to construct a controller satisfying all the desired properties. Our class bounds the number of phases – in a phase only one data-structure can be read in an unrestricted way though writes to all data-structures are allowed. But our notion of phases extends sensibly contexts of [14] and phases of [13]. In particular it permits *autonomous computations* within a phase instead of the well-queuing assumption. The latter corresponds to permitting reads from queues in the *main* program but not from any of the functions it calls. We permit recursive calls to be at any depth of recursion when reading from a queue. After such a read, however, returning from the function causes a phase change.

A concurrent system may be controlled in a global manner or in a distributed manner. If the concurrent processes are at a single location and communicate via shared variables, e.g., multi-threaded programs, a global controller is reasonable. We describe this sequential controller in Section 4. However, when these multi-threaded processes are physically distributed it is natural to demand a distributed controller. In Section 5, we illustrate the design of a controllable under-approximation class by extending our idea of phases to the distributed setting and constructing a distributed controller with all the desired properties.

Finally, we can prove using the split-width technique [5,6,8] that our generous under-approximation class can be model-checked against a wide variety of logics.

For lack of space, proofs of correctness and of decidability are omitted from this extended abstract and can be found in the full version [7].

Related Work: In the study of distributed automata a number of difficult synthesis theorems [9–11,19] have been proved. These theorems in conjunction with constructions for intersections yield controllers for these classes. Of particular interest is the theory of finite state machines communicating via queues, called message-passing automata (MPA). These have been well studied using labeled partial-orders (or graphs) called MSCs (Message-sequence charts) to represent behaviours. These systems are turing powerful and techniques restricting channel usage have been studied to obtain decidability. The most general class of this kind, called existentially k-bounded MSCs, consists of all behaviours (MSCs) that have at least one linearization in which the queue lengths are bounded by

k at every point. A deep result of [9] shows that for each k there is an MPA which accepts precisely the set of existentially k-bounded MSCs. Thus, if one uses such behaviours as an under-approximation class then this result implies the existence of a distributed controller. However, it is known that this controller cannot be made deterministic.

The bounding technique for verification has been extensively studied in the case of multi-pushdown systems (MPDS). For the restrictions studied in literature, bounded-context [18], bounded-phase [13], bounded-scope [16] and ordered stacks [2, 3], it is quite easy to construct deterministic controllers, though this question has not been addressed before. The context bounding technique is extended to pushdown systems communicating via queues under the restriction that queues may be read only when the stacks are empty (well-queuing) in [14], and under a dual restriction (on writes instead of reads) in [12]. Controllability is however not studied there. The k-Phase restriction we consider here is a natural joint generalization of these contexts (as well as the bounded-phase restriction for MPDS). In fact, for every bound k, there exist behaviours which are not captured by [13] and [14], but which are captured by our class with a bound of 3. (See Figure 1 for an example.)

2 Systems with stacks and queues

We provide a formal description of systems with data-structures and their behaviours. We restrict ourselves to systems with global states providing an (interleaved) sequential view. In Section 5 we extend this to the distributed case where there are a number of components each with their own collection of transitions. We consider a finite set $\mathbf{DS} = \mathbf{Stacks} \uplus \mathbf{Queues}$ of data-structures which are either stacks or queues and a finite set Σ of actions. Our systems have a finite set of control locations and use these (unbounded) stacks and queues. We obtain an interesting class of infinite state systems, providing an (interleaved) sequential view of multi-threaded recursive programs communicating via FIFO channels.

A stack-queue system (SQS) over data-structures $\mathbf{DS}$ and actions from Σ is a tuple $\mathcal{S} = (\mathsf{Locs}, \mathrm{Val}, \mathrm{Trans}, \mathsf{in}, \mathsf{Fin})$ where Locs is a finite set of locations, Val is a finite set of values that can be stored in the data-structures, $\mathsf{in} \in \mathsf{Locs}$ is the initial location, $\mathsf{Fin} \subseteq \mathsf{Locs}$ is the set of final locations, and Trans is the set of transitions which may write a value to, or read a value from, or do not involve a data-structure. For $\ell, \ell' \in \mathsf{Locs}$, $a \in \Sigma$, $d \in \mathbf{DS}$ and $v \in \mathrm{Val}$, we have

- internal transitions of the form $\ell \xrightarrow{\mathrm{a}} \ell'$,
- write transitions of the form $\ell \xrightarrow{a,d!v} \ell'$, and
- read transitions of the form $\ell \xrightarrow{a,d?v} \ell'$.

Intuitively, an SQS consists of a finite state system equipped with a collection of stacks and queues. In each step, it may use an internal transition to merely change its state, or use a write transition to append a value to the tail of a particular queue or stack or use a read transition to remove a value from the head (or tail) of a queue (of a stack respectively). The transition relation makes explicit the identity of the data-structure being accessed and the type of the operation. As observed in [1, 6, 13, 17] it is often convenient to describe the runs of such systems as a state-labeling of words decorated with a matching relation per data-structure instead of the traditional operational semantics using configurations and moves. This will prove all the more useful when we move to the distributed setting where traditionally semantics has always been given as state-labelings of appropriate partial orders [9, 11, 19].

A stack-queue word (SQW) over **DS** and Σ is a tuple $\mathcal{W} = (w, (\rhd^d)_{d \in \mathbf{DS}})$ where $w = a_1 a_2 \cdots a_n \in \Sigma^+$ is the sequence of actions, and for each $d \in \mathbf{DS}$, the matching relation $\rhd^d \subseteq \{1, \ldots, n\}^2$ relates write events to data-structure d to their corresponding read events. The following conditions should be satisfied:

- write events should precede read events: $e \rhd^d f$ implies $e < f$,
- data-structure accesses are disjoint: if $e_1 \rhd^d e_2$ and $e_3 \rhd^{d'} e_4$ are distinct edges ($d \neq d'$ or $(e_1, e_2) \neq (e_3, e_4)$) then they are disjoint ($|\{e_1, e_2, e_3, e_4\}| = 4$),
- $\forall d \in$ **Stacks**, $\rhd^d$ conforms to LIFO: if $e_1 \rhd^d f_1$ and $e_2 \rhd^d f_2$ are different edges then we do not have $e_1 < e_2 < f_1 < f_2$.
- $\forall d \in$ **Queues**, $\rhd^d$ conforms to FIFO: if $e_1 \rhd^d f_1$ and $e_2 \rhd^d f_2$ are different edges then we do not have $e_1 < e_2$ and $f_2 < f_1$.

We let $\rhd = \bigcup_{d \in \mathbf{DS}} \rhd^d$ be the set of all matching edges and $\mathcal{E} = \{1, \ldots, n\}$ be the set of events of $\mathcal{W}$. The set of all stack-queue words is denoted by $\mathbb{SQW}$.

We say that an event e is a *read event* (on data-strucutre d) if there is an f such that $f \rhd^d e$. We define *write events* similarly and an event is *internal* if it is neither a read nor a write. To define the run of an SQS over a stack-queue word $\mathcal{W}$, we introduce two notations. For $e \in \mathcal{E}$, we denote by e^- the immediate predecessor of e if it exists, and we let $e^- = \bot \notin \mathcal{E}$ otherwise. We let $\max(\mathcal{W})$ be the maximal event of $\mathcal{W}$.

A run of an SQS $\mathcal{S}$ on a stack-queue word $\mathcal{W}$ is a mapping $\rho \colon \mathcal{E} \to \mathsf{Locs}$ satisfying the following consistency conditions (with $\rho(\bot) = \mathsf{in}$):

- if e is an internal event then $\rho(e^-) \xrightarrow{\lambda(e)} \rho(e) \in \mathrm{Trans}$,
- if $e \rhd^d f$ for some data-structure $d \in \mathbf{DS}$ then for some $v \in \mathrm{Val}$ we have both $\rho(e^-) \xrightarrow{\lambda(e), d!v} \rho(e) \in \mathrm{Trans}$ and $\rho(f^-) \xrightarrow{\lambda(f), d?v} \rho(f) \in \mathrm{Trans}$.

The run is accepting if $\rho(\max(\mathcal{W})) \in \mathsf{Fin}$. The *language* $\mathcal{L}(\mathcal{S})$ accepted by an SQS $\mathcal{S}$ is the set of stack-queue words on which it has an accepting run.

Notice that SQSs are closed under intersection, by means of the cartesian product. Let $\mathcal{S}_i = (\mathsf{Locs}_i, \mathrm{Val}_i, \mathrm{Trans}_i, \mathsf{in}_i, \mathsf{Fin}_i)$ for $i \in \{1, 2\}$ be two SQSs. The cartesian product is $\mathcal{S}_1 \times \mathcal{S}_2 = (\mathsf{Locs}_1 \times \mathsf{Locs}_2, \mathrm{Val}_1 \times \mathrm{Val}_2, \mathrm{Trans}, (\mathsf{in}_1, \mathsf{in}_2), \mathsf{Fin}_1 \times \mathsf{Fin}_2)$ where the set of transitions is defined by

- $(\ell_1, \ell_2) \xrightarrow{a} (\ell'_1, \ell'_2) \in \mathrm{Trans}$ if $\ell_i \xrightarrow{a} \ell'_i \in \mathrm{Trans}_i$ for $i \in \{1, 2\}$,
- $(\ell_1, \ell_2) \xrightarrow{a, d!(v_1, v_2)} (\ell'_1, \ell'_2) \in \mathrm{Trans}$ if $\ell_i \xrightarrow{a, d!v_i} \ell'_i \in \mathrm{Trans}_i$ for $i \in \{1, 2\}$,
- $(\ell_1, \ell_2) \xrightarrow{a, d?(v_1, v_2)} (\ell'_1, \ell'_2) \in \mathrm{Trans}$ if $\ell_i \xrightarrow{a, d?v_i} \ell'_i \in \mathrm{Trans}_i$ for $i \in \{1, 2\}$.

In fact, $\mathcal{S}_1 \times \mathcal{S}_2$ has an (accepting) run on a stack-queue word $\mathcal{W}$ iff both $\mathcal{S}_1$ and $\mathcal{S}_2$ have an (accepting) run on $\mathcal{W}$. Therefore, $\mathcal{L}(\mathcal{S}_1 \times \mathcal{S}_2) = \mathcal{L}(\mathcal{S}_1) \cap \mathcal{L}(\mathcal{S}_2)$.

3 Controllers and Controlled Systems

SQSs are turing powerful as soon as **DS** contains two stacks or a queue, and hence their verification is undecidable. However, since it is an important problem, various under-approximation techniques have been invented in the recent years [2,3,13,14,16,18], starting with the *bounded-context restriction* [18] for systems with only stacks. Here, the number of times the system switches from using one stack to another is bounded by a fixed number k. Reachability and many other properties become decidable when restricted to such behaviours.

A typical under-approximation technique describes a whole family of classes $\mathbb{C}_k$ parametrized by an integer k which is proportional to the coverage: the higher the parameter, the more behaviours are covered. For example, the bound on number of context switches k serves as this parameter for the context bounding technique. Ideally, the under-approximations defined by the classes $(\mathbb{C}_k)_k$ should be *universal*, i.e., should cover all behaviours: every stack-queue word $\mathcal{W}$ should be in $\mathbb{C}_k$ for some k. This is true for the context bounding technique.

Traditionally under-approximations yield decidability for verification problems such as reachability [18] and model checking against linear time properties expressed in various logics upto MSO [17]. For such properties, if the model-checking problem yields a negative answer then this immediately means that the full system fails the verification as well.

However, assume that a system $\mathcal{S}$ has been verified against some linear-time or reachability property (or properties) wrt. some under-approximation class $\mathbb{C}$. This give us little information on whether the full system satisfies these properties. Hence we need a mechanism, which we call a *controller*, to restrict the system so that it does not exhibit behaviours outside $\mathbb{C}$. Observe that w.r.t. linear-time properties restricting the system to even a proper subset of $\mathbb{C}$ would still be acceptable though not desirable. However, for reachability properties a proper restriction might lead to a system that no longer satisfies the property. Therefore, a controller should allow all and only the behaviours of $\mathbb{C}$.

We now describe formally our notion of a controller for a class and examine some key properties that make it interesting.

A controller for a class $\mathbb{C} \subseteq \mathbb{SQW}$ is an SQS $\mathcal{C}$ such that $\mathcal{L}(\mathcal{C}) = \mathbb{C}$. We say that a class $\mathbb{C}$ is *controllable* if it admits a controller.

Suppose the restriction of the behaviours of a system $\mathcal{S}$ to a class $\mathbb{C}$ has been verified against some linear-time or reachability property φ. Further suppose that $\mathbb{C}$ admits a controller $\mathcal{C}$. Then, the *controlled system* $\mathcal{S}' = \mathcal{S} \times \mathcal{C}$ is such that $\mathcal{L}(\mathcal{S}') = \mathcal{L}(\mathcal{S}) \cap \mathbb{C}$, and therefore satisfies φ. Thus, a controller for a class is independent of the system $\mathcal{S}$ as well as the property. Once we identify a controllable class with decidable verification we may verify and control any system in a completely generic and transparent manner without any additional work.

Notice that we could have introduced more general controllability. For instance, a class $\mathbb{C}$ is non-uniformly controllable if for each system $\mathcal{S}$, there exists another system $\mathcal{S}'$ such that $\mathcal{L}(\mathcal{S}') = \mathcal{L}(\mathcal{S}) \cap \mathbb{C}$. While this would allow more classes to be controllable, it would not be very useful since it does not yield an automatic way to build $\mathcal{S}'$ from $\mathcal{S}$.

Using the cartesian product makes the controller integrable into the system. The controller, by definition, does not have its own auxiliary data-structures, but only shares the data-structures of the system. Moreover, it does not access a data-structure out of sync with the system. We could also give more intrusive power to a controller by allowing its transitions to depend on the current state of the system and on the current value read/written by the system on data-structures. But again, such a system would not be generic, and also, by its strong observation power, would compromise the privacy of the system.

We now consider other properties that a good controller must satisfy and use that to arrive at a formal definition of such a controller.

The under-aproximation classes are often defined based on the data-structure accesses, and do not depend on the action labels/internal actions. Hence an ideal controller should be definable independent of the action labels and must be oblivious to the internal moves. This can be done as follows.

We omit action labels from read/write transitions of $\mathcal{C}$: an abstract transition $\ell \xrightarrow{d!v} \ell'$ stands for transitions $\ell \xrightarrow{a,d!v} \ell'$ for all $a \in \Sigma$ and similarly for read transitions. Also, we do not describe internal transitions and assume instead that there are self-loops $\ell \xrightarrow{\mathrm{a}} \ell$ for all locations and actions.

This (abstract) controller should be deterministic and non-blocking, so that instantiating it with any alphabet will still be deterministic and non-blocking. Thus, the controller should have a unique run on any $\mathcal{W}$ and moreover this run does not depend on the internal events / action labels along the run, but depends only on the sequence of reads/writes on the different data-structures that appear along $\mathcal{W}$. The state of the controller at any point along this run unambiguously indicates whether the current prefix can be extended to a word that belongs to the class $\mathbb{C}$. With this we are ready to formalize our notion of a good controller.

A DS-controller is an SQS $\mathcal{C}$ which is oblivious to internal events and to action labels and which is deterministic and non-blocking. Formally, its (abstract) transitions should satisfy:

– for every $\ell \in \mathsf{Locs}$ and $d \in \mathbf{DS}$ there exists exactly one $\ell' \in \mathsf{Locs}$ and $v \in \mathrm{Val}$ such that $\ell \xrightarrow{d!v} \ell'$,
– for every $\ell \in \mathsf{Locs}$, $d \in \mathbf{DS}$ and $v \in \mathrm{Val}$ there exists exactly one $\ell' \in \mathsf{Locs}$ such that $\ell \xrightarrow{d?v} \ell'$.

All that we said so far suffices for a global (or seqeuntial) system. If the system to be verified and controlled is actually physically distributed, then a global sequential controller would not be integrable in the system. Instead we would need a distributed controller and this is much harder to achieve. We discuss this in Section 5.

Next we examine real examples of controllable under-approximations. While an under-approxiamation $\mathbb{C}_k$ is *nicely controllable* if it admits a controller with the above features, the class itself should satisfy some other properties for it to be useful. Firstly, $\mathbb{C}_k$ should have a wide coverage over the set of possible behaviours. A useful feature is that all behaviours fall in the class for an appropriately chosen parameter. Second, the definition of the class should be easy to describe. Finally, the verification problem for the class should be decidable. For instance, considering the collection of behaviours with clique/split/tree-width bounded by k satisfies the first and third properties but does not satisfy the second property. But more importantly, it is not clear that they have nice controllers of the form described above. We propose a meaningful class which has more coverage than bounded phase of [13], and is nicely controllable. We show the decidability of this class by demonstrating a bound on split-width.

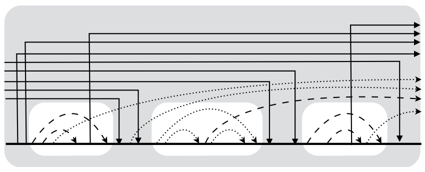
4 Class and Controller: Sequential case

We begin by identifying a class of behaviours, called k-Phase behaviours, which is verifiable and admits a **DS**-controller. Roughly speaking, a phase is a segment of the run where the reads are from a fixed data-structure. However, between successive reads, read-free recursive computations are permitted which may write to all data-structures, including their own call-stack. We formalize this below.

An autonomous computation involves a single recursive thread executing a recursive procedure without reading any other data structure. All read events are from a single stack while there is no restriction placed on the writes. We say that an edge $e \rhd f$ is *autonomous* if $e \rhd^s f$ for some $s \in \textbf{Stacks}$ and all in-between read events are from the same stack s: if $e' \rhd^d f'$ with $e \leq f' \leq f$ then $d = s$. We shall write $\rhd_a$ for the subset of $\rhd$ consisting of the autonomous edges and $\rhd_{na}$ for $\rhd \setminus \rhd_a$ and refer to them as the non-autonomous edges. If $e \rhd_a f$ then e and f are called *autonomous* write and read events respectively.

A d-phase is a sequence of consecutive events in which all non-autonomous reads are from the data-structure $d \in \textbf{DS}$. Writes to all data-structures are permitted. Moreover, a phase must not break an autonomous computation. Formally, a d-phase is identified by a pair of events $e \leq f$ (the first and the last events in the sequence) such that, if $e' \rhd_{na} f'$ with $e \leq f' \leq f$ then $e' \rhd^d f'$ and if $e' \rhd_a f'$ with $e \leq f' \leq f$ or $e \leq e' \leq f$ then $e \leq e' \leq f' \leq f$.

Example 1. Suppose $\textbf{DS} = \{q, s_1, s_2\}$. A q-phase is depicted on the right. Straight lines (resp. curved lines) represent $\rhd^d$ edges from queues (resp. stacks). Autonomous computations are highlighted in white.



Remark 2. Permitting autonomous (recursive) computations during a phase is a natural generalization of *well-queueing* assumption of [14] where reads from

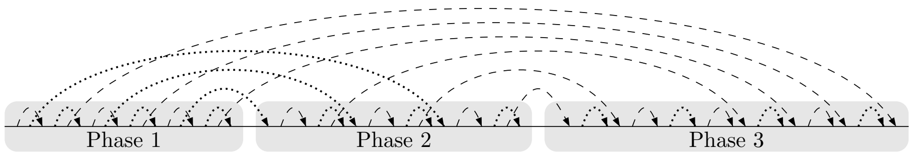


Fig. 1: A stack-queue word over two stacks and its maximal phase decomposition.

queues are permitted only when the stack associated with a process is empty. The latter corresponds to permitting reads from queues in the *main* program but not from any of the functions it calls. We permit recursive calls to be at any depth of recursion when reading from the queue. After such a read, however, returning from the function causes a phase change.

Our aim is to obtain a decidable and controllable class by bounding the number of phases. In the presence of queues, reading and writing on a queue during a phase can be used to simulate a turing machine using just 1-phase computations. Allowing autonomous computations on one stack while reading and writing on another also results in the same effect. This motivates the following definition which rules out such *self-loops.*

A phase identified by a pair (e, f) has a *self-loop* if it contains a non-autonomous edge: $e \leq e' \rhd_{na} f' \leq f$.

A phase decomposition is a partition of the set of events into phases with no self-loops. A k-phase decomposition is a phase decomposition with at most k phases. We denote by k-Phase the class of stack-queue words that admit a k-phase decomposition.

Remark 3. Observe that by freely allowing autonomous computations (as opposed to well-queuing), every stack-queue word is in k-Phase for some k.

Remark 4. When restricted to systems with only stacks, k-Phase subsumes the *k bounded phase restriction* for multi-pushdown systems [13]. It also subsumes the *k bounded context restriction* for systems with stacks and queues [14]. In fact, for every bound k, there exist stack-queue words which are not captured by [13] and [14], but which are in 3-Phase. (See Figure 1.)

A phase with no self-loops identified by (e, f) is *upper-maximal* if it cannot be extended upwards in a phase with no self-loops: if (e, g) is a phase with no self-loops then $g \leq f$. Given any k phase decomposition, we may extend the first phase to be upper-maximal and then extend the next (remaining) phase to be upper-maximal and so on till all the phases are upper maximal.

Lemma 5. *Every stack-queue word in k-*Phase *admits a* maximal *k-phase decomposition in which all phases are upper-maximal.*

Now we take up the task of constructing a **DS**-controller for the class k-Phase. A crucial step towards this end is to identify autonomous reads. We show below that this can be achieved with a multi-pushdown automaton $\mathcal{B}$ observing the data-structure access. When the system $\mathcal{S}$ writes/reads some value on a stack s the automaton $\mathcal{B}$ will simultaneously write/read a bit on the same stack. $\mathcal{B}$ is obtained as a cartesian product of automata $\mathcal{B}_s$ ($s \in \mathbf{Stacks}$) identifying the autonomous reads on stack s (described in Figure 2).

Here, $s!b$ (resp. $s?b$) means that the system $\mathcal{S}$ writes/reads on stack s and b is the tag bit that is simultaneously written/read by $\mathcal{B}_s$ on stack s. The other events do not change stack s. Moreover, $\bar{s}?$ is the observation of a read event of $\mathcal{S}$ which is *not* on stack s, and *else* means any event which is not explicitly specified.

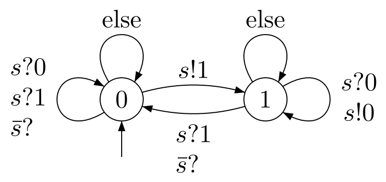


Fig. 2: The automaton $\mathcal{B}_s$.

We say that e is a *possibly autonomous write* to stack s at event g if $e \rhd^s f$ and $e \leq g < f$ and $e' \rhd^d f'$ with $e \leq f' \leq g$ implies $d = s$. Intuitively $\mathcal{B}_s$ will be in state 1 iff in the current prefix there is an unmatched write event e to stack s which is possibly autonomous. On a write to s the automaton moves from state 0 to 1 since this write is possibly autonomous, and pushes 1 on the stack to indicate that it is the first possibly autonomous write in the past. Then, as long as it does not read from a data-structure $d \neq s$, it stays in state 1, pushing 0 on the stack on a write to s and reading 0 from the stack on a read from s. If it reads 1 from the stack, then it has matched the first possibly autonomous write in the past, hence it goes back to state 0. On a read from $d \neq s$ it goes to state 0 since there cannot be any possibly autonomous write to s at this read event.

Lemma 6. *The automaton $\mathcal{B}_s$ is deterministic and non-blocking. Moreover, in the unique run of $\mathcal{B}_s$ on a word, the state b_s before a read from stack s determines whether this read is autonomous ($b_s = 1$) or not ($b_s = 0$).*

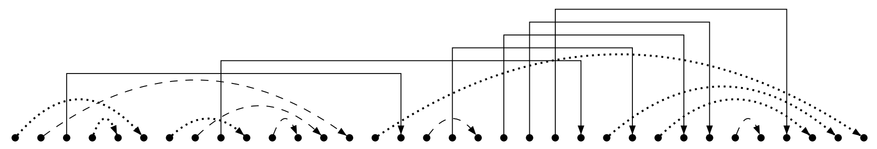
We now construct the deterministic $\mathbf{DS}$-controller $\mathcal{C}^k$ for k-Phase. This controller computes the maximal phase decomposition of a behaviour and uses the automaton $\mathcal{B}$ to identify autonomous reads. We denote by $\boldsymbol{b} = (b_s)_{s \in \mathbf{Stacks}}$ a state of $\mathcal{B}$. In addition, a state of $\mathcal{C}^k$ holds two other values:

- a counter $n \in \{1, \ldots, k, \infty\}$ which indicates the current phase number. The counter starts from value 1 and is non-decreasing along a run. The ∞ indicates that the number of phases has exceeded k. We follow the convention that $i + 1$ has the usual meaning if $i < k$, $k + 1 = \infty$ and $\infty + 1 = \infty$.
- a value $d \in \mathbf{DS} \cup \{?\}$ which indicates that the current phase has non-autonomous reads from $d \in \mathbf{DS}$ or that only autonomous reads have occurred so far ($d = ?$). Note that in the first phase all reads are autonomous (a non-autonomous read would create a self-loop). Hence, $d = ?$ iff $n = 1$.

The initial state of the controller is $(1, ?, \mathbf{0})$. On an internal event, the state remains unchanged. When the system writes to a data-structure the controller $\mathcal{C}^k$ writes its current phase number in addition to the bits written by $\mathcal{B}$.

$$(n, d, \boldsymbol{b}) \xrightarrow{d'!n} (n, d, \boldsymbol{b}) \quad \text{if } d' \in \mathbf{Queues} \tag{1}$$

$$(n, d, \boldsymbol{b}) \xrightarrow{d'!(n,c)} (n, d, \boldsymbol{b}') \quad \text{if } d' \in \mathbf{Stacks} \wedge \boldsymbol{b} \xrightarrow{d'!c} \boldsymbol{b}' \text{ in } \mathcal{B} \tag{2}$$



n	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	3	3	3	3	3	3	3	3	3	4	4	4	4	4	4	4	5	5	5
d	?	?	?	?	?	?	?	?	?	?	?	?	?	s_2	s_2	s_2	q	q	q	q	q	q	q	q	q	q	q	q	q	q	q	q	s_1	s_1	s_1
b_1	0	1	1	1	1	1	0	1	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
b_2	0	0	1	1	1	0	0	0	1	1	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0

Fig. 3: A run of the deterministic sequential controller $\mathcal{C}^k$.

Notice that in the first case, $\boldsymbol{b} \xrightarrow{d'!} \boldsymbol{b}$ is a transition in $\mathcal{B}$. A read event from a queue d' will stay in the same phase if d' is the current data-structure and the matching write comes from a previous phase (to avoid self-loops): if $d' \in \mathbf{Queues}$ then we have the following transitions in $\mathcal{C}^k$

$$(n, d, \boldsymbol{b}) \xrightarrow{d'?m} (n, d', \mathbf{0}) \quad \text{if } d' = d \wedge m < n \tag{3}$$
$$(n, d, \boldsymbol{b}) \xrightarrow{d'?m} (n+1, d', \mathbf{0}) \quad \text{otherwise} \tag{4}$$

Notice that in these cases, $\boldsymbol{b} \xrightarrow{d'?} \mathbf{0}$ is a transition in $\mathcal{B}$ since no stack can be on an autonomous computation at a read event from a queue. Further if $d = ?$, reading from a queue forces a phase change. This is needed, as otherwise there will be a self-loop on the first phase.

Finally, a read event from a stack s will stay in the same phase if it is an autonomous read ($b_s = 1$), or $s = d$ is the current data-structure and this read does not create a self-loop: if $s \in \mathbf{Stacks}$ then in $\mathcal{C}^k$ we have the transitions

$$(n, d, \boldsymbol{b}) \xrightarrow{s?(m,c)} (n, d, \boldsymbol{b}') \quad \text{if } (b_s = 1 \vee (s = d \wedge m < n)) \wedge \boldsymbol{b} \xrightarrow{s?c} \boldsymbol{b}' \text{ in } \mathcal{B} \tag{5}$$
$$(n, d, \boldsymbol{b}) \xrightarrow{s?(m,c)} (n+1, s, \mathbf{0}) \quad \text{otherwise} \tag{6}$$

Notice that in the last case, $\boldsymbol{b} \xrightarrow{s?c} \mathbf{0}$ is a transition in $\mathcal{B}$ and thus in all moves the third component stays consistent with moves of $\mathcal{B}$.

By construction the controller is deterministic and non-blocking. If the unique run of the controller on a $\mathcal{W}$ does not use a state of the form $(\infty, d, \boldsymbol{b})$ then $\mathcal{W}$ is in k-Phase. The set of positions labeled by states of the form $(i, d, \boldsymbol{b})$ identify the ith phase in a k phase decomposition. Conversely, let $\mathcal{W}$ be in k-Phase. Let $(\boldsymbol{b_e})$ be the state labeling position e in $\mathcal{W}$ in the unique run of $\mathcal{B}$ on $\mathcal{W}$. Let $(X_i)_{(i \leq l)}$ be the phases in the maximal decomposition of $\mathcal{W}$. It is easy to verify that the first position of X_i, $i \geq 2$ is a non-autonomous read and let d_i be the data-structure associated with this read. Then the labeling assigning $(1, ?, \boldsymbol{b_e})$ to any position $e \in X_1$ and $(i, d_i, \boldsymbol{b_e})$ to any event e in X_i, $2 \leq i \leq l$ is is an accepting run of the controller on $\mathcal{W}$.

Theorem 7. *The SQS $\mathcal{C}^k$ is a* $\mathbf{DS}$*-controller for the class k-*Phase *with* $(|\mathbf{DS}| \cdot (k+1) + 1)2^{|\mathbf{Stacks}|}$ *states.*

5 Class and Controller: Distributed case

In this section we describe a model intended to capture collections of SQS communicating via reliable FIFO channels (or queues). Such systems are called Stack-Queue Distributed System (SQDS). A behaviour of an SQDS is a tuple of stack-queue words with additional matching relations describing the inter-process communication via queues. Such behaviours extend Message Sequence Charts (MSCs) with matching relations for the internal stacks and queues. We call them stack-queue MSCs (SQMSC).

We then extend the notion of k-Phase to this distributed setting. We show that k-Phase enjoys a deterministic distributed controller with local acceptance conditions.

An architecture $\mathfrak{A}$ is a tuple $(\mathbf{Procs}, \mathbf{Stacks}, \mathbf{Queues}, \mathsf{Writer}, \mathsf{Reader})$ consisting of a set of processes $\mathbf{Procs}$, a set of stacks $\mathbf{Stacks}$, a set of queues $\mathbf{Queues}$ and functions Writer and Reader which assign to each stack/queue the process that will write (push/send) into it and the process that will read (pop/receive) from it respectively. We write $\mathbf{DS}$ for $\mathbf{Stacks} \uplus \mathbf{Queues}$.

A stack d must be local to its process, so $\mathsf{Writer}(d) = \mathsf{Reader}(d)$. On the other hand, a queue d may be local to a process p if $\mathsf{Writer}(d) = p = \mathsf{Reader}(d)$, otherwise it provides a FIFO channel from $\mathsf{Writer}(d)$ to $\mathsf{Reader}(d)$.

A Stack-Queue Distributed System (SQDS) over an architecture $\mathfrak{A}$ and an alphabet Σ is a tuple $\mathcal{S} = (\mathsf{Locs}, \mathrm{Val}, (\mathrm{Trans}_p)_{p\in\mathbf{Procs}}, \mathsf{in}, \mathsf{Fin})$ where each $\mathcal{S}_p = (\mathsf{Locs}, \mathrm{Val}, \mathrm{Trans}_p, \mathsf{in}, \emptyset)$ is an SQS over $\mathbf{DS}$ and Σ in which the transitions are compatible with the architecture: Trans_p may have a write (resp. read) transitions on data-structure d only if $\mathsf{Writer}(d) = p$ (resp. $\mathsf{Reader}(p) = d$). Moreover, $\mathsf{Fin} \subseteq \mathsf{Locs}^{\mathbf{Procs}}$ is the global acceptance condition. We say that the acceptance condition is local if $\mathsf{Fin} = \prod_{p\in\mathbf{Procs}} \mathsf{Fin}_p$ where $\mathsf{Fin}_p \subseteq \mathsf{Locs}$ for all $p \in \mathbf{Procs}$.

A stack-queue MSC (SQMSC) over architecture $\mathfrak{A}$ and alphabet Σ is a tuple $\mathcal{M} = ((w_p)_{p\in\mathbf{Procs}}, (\rhd^d)_{d\in\mathbf{DS}})$ where $w_p \in \Sigma^*$ is the sequence of events on process p and $\rhd^d$ is the relation matching write events on data-structure d with their corresponding read events. We let $\mathcal{E}_p = \{(p, i) \mid 1 \le i \le |w_p|\}$ be the set of events on process $p \in \mathbf{Procs}$. For an event $e = (p, i) \in \mathcal{E}_p$, we set $\mathsf{pid}(e) = p$ and $\lambda(e)$ be the ith letter of w_p. We write $\to$ for the successor relation on processes: $(p, i) \to (p, i + 1)$ if $1 \le i < |w_p|$ and we let $\rhd = \bigcup_{d\in\mathbf{DS}} \rhd^d$ be the set of all matching edges. We require the relation $< = (\to \cup \rhd)^+$ to be a strict partial order on the set of events. Finally, the matching relations should comply with the architecture: $\rhd^d \subseteq \mathcal{E}_{\mathsf{Writer}(d)} \times \mathcal{E}_{\mathsf{Reader}(d)}$. Moreover, data-structure accesses should be disjoint, stacks should conform to LIFO and queues should conform to FIFO (the formal definitions are taken verbatim from Section 2). An SQMSC is depicted in Figure 4.

As before, to define the run of an SQDS over a stack-queue MSC $\mathcal{M}$, we introduce two notations. For $p \in \mathbf{Procs}$ and $e \in \mathcal{E}_p$, we denote by e^- the unique event such that $e^- \to e$ if it exists, and we let $e^- = \perp_p \notin \mathcal{E}$ otherwise. We let $\max_p(\mathcal{M})$ be the maximal event of $\mathcal{E}_p$ if it exists and $\max_p(\mathcal{M}) = \perp_p$ otherwise.

A run of an SQDS $\mathcal{S}$ over a stack-queue MSC $\mathcal{M}$ is a mapping $\rho\colon \mathcal{E} \to \mathsf{Locs}$ satisfying the following consistency conditions (with $\rho(\bot_p) = \mathsf{in}$):

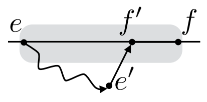
- if e is an internal event then $\rho(e^-) \xrightarrow{\lambda(e)} \rho(e) \in \mathrm{Trans}_{\mathsf{pid}(e)}$,
- if $e \rhd^d f$ for some data-structure $d \in \mathbf{DS}$ then for some $v \in \mathrm{Val}$ we have both $\rho(e^-) \xrightarrow{\lambda(e),d!v} \rho(e) \in \mathrm{Trans}_{\mathsf{pid}(e)}$ and $\rho(f^-) \xrightarrow{\lambda(f),d?v} \rho(f) \in \mathrm{Trans}_{\mathsf{pid}(f)}$.

The run is accepting if $(\rho(\max_p(\mathcal{M})))_{p \in \mathbf{Procs}} \in \mathsf{Fin}$. The *language* $\mathcal{L}(\mathcal{S})$ accepted by an SQDS $\mathcal{S}$ is the set of stack-queue MSCs on which it has an accepting run.

Notice that SQDSs are closed under intersection, by means of the cartesian product. The construction is similar to the one for SQSs in Section 2.

Bounded Acyclic Phase SQMSCs We generalize the under-approximation class k-Phase to the distributed setting. We allow at most k phases per process. As in the sequential case, autonomous computations are freely allowed. However, cycles on phases can be caused be the richer structure of the SQMSC than simple self loops.

In the distributed setting, the definitions of **autonomous computations** and of d**-phases** are identical to the sequential case, cf. Section 4. Again, we write $\rhd_a$ for autonomous edges and $\rhd_{na}$ for non-autonomous edges. A phase, which is a sequence of consecutive events executed by a single process, is identified by a pair of events (e, f) such that $e \to^* f$.



A phase (e, f) **has a cycle** if there is a non-autonomous edge $e' \rhd_{na} f'$ with $e \leq e'$ and $f' \to^* f$. Notice that e' needs not be in the phase. So a cycle starts from the phase at e then follows the partial order to some non-autonomous write e' whose read f' is in the phase. A phase is *acyclic* if it has no cycles. Notice that a non-autonomous edge within a phase induces a cycle (self-loop) whereas autonomous edges are freely allowed within phases. As a matter of fact, when there is exactly one process, a phase has a cycle iff it has a self-loop.

A phase decomposition of an SQMSC is a partition of its set of events into phases. A phase decomposition is *acyclic* if all phases are acyclic. It is a k-phase decomposition if there are at most k phases per process. We denote by k-Phase the set of SQMSCs that admits an acyclic k-phase decomposition.

An acyclic phase (e, f) is *upper-maximal* if extending it upwards would result in a cycle, i.e., for every other acyclic phase (e, f'), we have $f' \leq f$. See Figure 4 for an example. Lemma 5 easily lifts up to the distributed case as well.

Lemma 8. *Every SQMSC in* k-Phase *admits a* maximal *acyclic* k*-phase decomposition in which all phases are upper-maximal.*

Deterministic Distributed Controller We extend the notion of nice controllers to the distributed setting. That means controllers should be *distributed* and have *local* acceptance conditions. A local controller for one process should

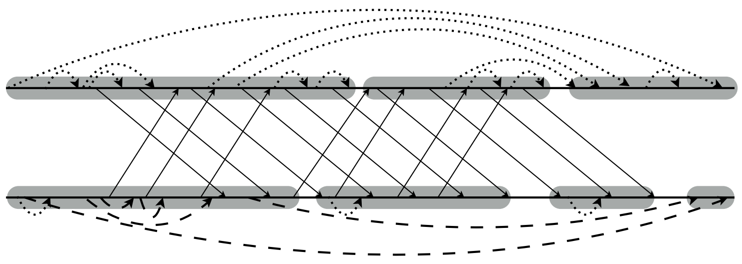


Fig. 4: A stack-queue MSC and its maximal phase decomposition.

be able to control the behaviour of that process regardless of the states of the other local controllers. The communication between the local controllers is also only by means of overloading the actual messages sent between the processes. The local controllers are not allowed to send messages out of sync, as it would create new behaviours in the controlled system. Thus a controlled system should be again obtained as a cartesian product of the system with a controller where both are SQDS, but in addition the controller has local acceptance conditions.

Theorem 9. *The class k-*Phase *admits a deterministic distributed controller $\mathcal{C}^k$ with $(|\mathbf{DS}| \cdot (k+2)^{|\mathbf{Procs}|} + 1)2^{|\mathbf{Stacks}|}$ states.*

The distributed controller is a generalisation of the sequential controller of Section 4. The main difference is that the local controller of process p remembers not only its current phase number, but a tuple $\boldsymbol{n} = (n_q)_{q \in \mathbf{Procs}}$ of phase numbers for each process. The intuition is that n_q is the largest phase of process q that is known to process p ($n_q = 0$ if no events of process q are in the past of the current event of process p).

For each stack s, we use the automaton $\mathcal{B}_s$ defined in Section 4 that identifies autonomous reads. For each process $p \in \mathbf{Procs}$, we let $\mathcal{B}_p$ be the product of the automata $\mathcal{B}_s$ where s is a stack of process p (i.e., $s \in \mathbf{Stacks}$ and $\mathsf{Writer}(s) = p$).

A state of the local controller $\mathcal{C}_p^k$ for process p is a tuple $(\boldsymbol{n}, d, \boldsymbol{b}_p)$ where $\boldsymbol{n} = (n_q)_{q \in \mathbf{Procs}}$ is the phase vector with $n_q \in \{0, 1, \ldots, k, \infty\}$, $d \in \mathbf{DS} \cup \{?\}$ with $\mathsf{Reader}(d) = p$ if $d \neq ?$, and $\boldsymbol{b}_p$ is a state of $\mathcal{B}_p$. The initial state of $\mathcal{C}_p^k$ is $\mathsf{in}_p = (\boldsymbol{n}, ?, \boldsymbol{0})$ with $n_p = 1$ and $n_q = 0$ for $q \neq p$. The local acceptance condition Fin_p is given by the set of states $(\boldsymbol{n}, d, \boldsymbol{b}_p)$ with $n_q \neq \infty$ for all $q \in \mathbf{Procs}$.

We describe now the local transitions of $\mathcal{C}_p^k$. They are similar to the transitions of the sequential controller given in Section 4. We start with write transitions, so let $d' \in \mathbf{DS}$ be such that $\mathsf{Writer}(d') = p$. On write events, the current phase vector is written on to the data-structure (in addition to the autonomous bit where needed).

$$(\boldsymbol{n}, d, \boldsymbol{b}_p) \xrightarrow{d'!\boldsymbol{n}} (\boldsymbol{n}, d, \boldsymbol{b}_p) \qquad \text{if } d' \in \mathbf{Queues} \tag{7}$$

$$(\boldsymbol{n}, d, \boldsymbol{b}_p) \xrightarrow{d'!(\boldsymbol{n},c)} (\boldsymbol{n}, d, \boldsymbol{b}'_p) \qquad \text{if } d' \in \mathbf{Stacks} \wedge \boldsymbol{b}_p \xrightarrow{d'!c} \boldsymbol{b}'_p \text{ in } \mathcal{B}_p \tag{8}$$

Let $d' \in \mathbf{Queues}$ be such that $\mathsf{Reader}(d') = p$. The transitions of $\mathcal{C}_p^k$ that read queue d' are given below. We should switch to the next phase 1) if $m_p = n_p$ since otherwise this non-autonomous read would close a cycle, 2) or if $d' \neq d \neq ?$ since in a phase all non-autonomous reads should be from the same data-structure.

$$(\boldsymbol{n}, d, \boldsymbol{b}_p) \xrightarrow{d'?\boldsymbol{m}} (\boldsymbol{n}', d', \boldsymbol{0}) \quad \text{if } m_p = n_p \vee (d' \neq d \neq ?) \quad \text{with } n'_p = n_p + 1 \wedge n'_q = \max(n_q, m_q) \text{ for } q \neq p \tag{9}$$

$$(\boldsymbol{n}, d, \boldsymbol{b}_p) \xrightarrow{d'?\boldsymbol{m}} (\boldsymbol{n}', d', \boldsymbol{0}) \quad \text{otherwise, with } \boldsymbol{n}' = \max(\boldsymbol{n}, \boldsymbol{m}) \tag{10}$$

Similarly, we give below read transitions from $d' \in \textbf{Stacks}$ with $\mathsf{Reader}(d') = p$. Here a switch of phase is required under the same conditions but only when the read is not autonomous.

$$(\boldsymbol{n}, d, \boldsymbol{b}_p) \xrightarrow{d'?(\boldsymbol{m},c)} (\boldsymbol{n}', d', \boldsymbol{0}) \quad \text{if } b_{d'} = 0 \wedge (m_p = n_p \vee (d' \neq d \neq ?)) \quad \text{with } n'_p = n_p + 1 \wedge n'_q = \max(n_q, m_q) \text{ for } q \neq p \tag{11}$$

$$(\boldsymbol{n}, d, \boldsymbol{b}_p) \xrightarrow{d'?(\boldsymbol{m},c)} (\boldsymbol{n}', d, \boldsymbol{b}'_p) \quad \text{otherwise, with } \boldsymbol{n}' = \max(\boldsymbol{n}, \boldsymbol{m}) \wedge \boldsymbol{b_p} \xrightarrow{d'?c} \boldsymbol{b}'_p \text{ in } \mathcal{B}_p \tag{12}$$

One of the differences of a local controller from a sequential controller is that the first phase may also perform non-autonomous reads. However, in such case, it must be from a queue.

On read transitions (10 and 12) which stay in the same phase, the phase vector is updated by taking the maximum between the current phase vector and the read-phase vector ($\boldsymbol{n}' = \max(\boldsymbol{n}, \boldsymbol{m})$). On a phase switch, a similar update is performed but the current phase number of process p is incremented.

6 Decidability

In this section we explain briefly why k-Phase is a verifiable under-approximation for SQDS. Consider the reachability problem which is equivalent to asking if given an SQDS $\mathcal{S}$ and $k \in \mathbb{N}$ whether $\mathcal{S}$ accepts at least one $\mathcal{M}$ from k-Phase. A non-trivial extension of the technique of [14] allows to reduce the reachability problem of SQDS restricted to k-Phase to the reachability problem of multi-pushdown systems for bounded phase.

A more general question is to model-check properties expressed in linear time logics ranging from temporal logics to MSO($\rightarrow$,$\rhd^d$). Given a formula φ we have to determine whether every $\mathcal{M} \in k$-Phase that is accepted by $\mathcal{S}$ satisfies φ. Observe that we may equivalently ask whether *every* behaviour of the controlled system $\mathcal{S}'$ satisfies φ. Using a slightly different approach we can obtain decidability not only for reachability but also for the linear-time model-checking problems.

In this approach we show that every behaviour in k-Phase has *split-width* [5, 6, 8] or tree-width [17] or clique-width [4] (measures of the complexity of graphs that happen to be equivalent for our class of graphs) bounded by some function $f(k)$. Here, we show an exponential bound on the split-width. Then, results from [6, 8, 17] imply that MSO model-checking for $\mathcal{S}'$ is decidable and results from [5,8] imply that model-checking linear-time temporal logic formulas can be solved in double exponential time. This is optimal, since reachability of k-phase multi-pushdown systems is double exponential time hard [15].

References

1. R. Alur and P. Madhusudan. Adding nesting structure to words. *Journal of the ACM*, 56:16:1–16:43, 2009.
2. M. F. Atig, B. Bollig, and P. Habermehl. Emptiness of multi-pushdown automata is 2ETIME-Complete. In *DLT*, LNCS 5257, pages 121–133. Springer, 2008.
3. L. Breveglieri, A. Cherubini, C. Citrini, and S. Crespi-Reghizzi. Multi-pushdown languages and grammars. *Int. J. Found. Comput. Sci.*, 7(3):253–292, 1996.
4. B. Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In G. Rozenberg, editor, *Handbook of Graph Grammars*, pages 313–400. World Scientific, 1997.
5. A. Cyriac. *Verification of Communicating Recursive Programs via Split-width*. PhD thesis, ENS Cachan, 2014. `http://www.lsv.ens-cachan.fr/~cyriac/download/Thesis_Aiswarya_Cyriac.pdf`.
6. A. Cyriac, P. Gastin, and K. Narayan Kumar. MSO decidability of multi-pushdown systems via split-width. In *CONCUR*, volume 7454 of *LNCS*, pages 547–561. Springer, 2012.
7. A. Cyriac, P. Gastin, and K. Narayan Kumar. Controllers for the Verification of Communicating Multi-pushdown Systems. Technical report, 2014. `http://hal.archives-ouvertes.fr/`.
8. A. Cyriac, P. Gastin, and K. Narayan Kumar. Verifying Communicating Multi-pushdown Systems. Technical report, Jan. 2014. `http://hal.archives-ouvertes.fr/hal-00943690`.
9. B. Genest, D. Kuske, and A. Muscholl. A Kleene theorem and model checking algorithms for existentially bounded communicating automata. *Information and Computation*, 204(6):920–956, 2006.
10. B. Genest, A. Muscholl, H. Seidl, and M. Zeitoun. Infinite-state high-level MSCs: Model-checking and realizability. *Journal of Computer and System Sciences*, 72(4):617–647, 2006.
11. J. G. Henriksen, M. Mukund, K. Narayan Kumar, M. A. Sohoni, and P. S. Thiagarajan. A theory of regular MSC languages. *Inf. Comput.*, 202(1):1–38, 2005.
12. A. Heußner, J. Leroux, A. Muscholl, and G. Sutre. Reachability analysis of communicating pushdown systems. In *FOSSACS*, volume 6014 of *LNCS*, pages 267–281. Springer, 2010.
13. S. La Torre, P. Madhusudan, and G. Parlato. A robust class of context-sensitive languages. In *LICS*, pages 161–170. IEEE Computer Society, 2007.
14. S. La Torre, P. Madhusudan, and G. Parlato. Context-bounded analysis of concurrent queue systems. In *TACAS*, volume 4963 of *LNCS*, pages 299–314. Springer, 2008.
15. S. La Torre, P. Madhusudan, and G. Parlato. An infinite automaton characterization of double exponential time. In *CSL*, volume 5213 of *LNCS*, pages 33–48. Springer, 2008.
16. S. La Torre and M. Napoli. Reachability of multistack pushdown systems with scope-bounded matching relations. In *CONCUR*, volume 6901 of *LNCS*, pages 203–218. Springer, 2011.
17. P. Madhusudan and G. Parlato. The tree width of auxiliary storage. In T. Ball and M. Sagiv, editors, *POPL*, pages 283–294. ACM, 2011.
18. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, volume 3440 of *LNCS*, pages 93–107. Springer, 2005.
19. W. Zielonka. Notes on finite asynchronous automata. *R.A.I.R.O. — Informatique Théorique et Applications*, 21:99–135, 1987.