Verifying Communicating Multi-pushdown Systems

Aiswarya Cyriac, Paul Gastin, K. Narayan Kumar

Verifying Communicating Multi-pushdown Systems

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

[1] LSV, ENS Cachan, CNRS & INRIA, France
{cyriac,gastin}@lsv.ens-cachan.fr
[2] Chennai Mathematical Institute, India
kumar@cmi.ac.in

Abstract. Communicating multi-pushdown systems model networks of multi-threaded recursive programs communicating via reliable FIFO channels. Hence their verification problems are undecidable in general. The behaviours of these systems can be represented as directed graphs, which subsume both Message Sequence Charts and nested words. We extend the notion of split-width [8] to these graphs, defining a simple algebra to compose/decompose these behaviours using two natural operations: shuffle and merge. We obtain simple, uniform and optimal decision procedures for various verification problems parametrized by split-width, ranging from reachability to model-checking against MSO, PDL and Temporal Logics.

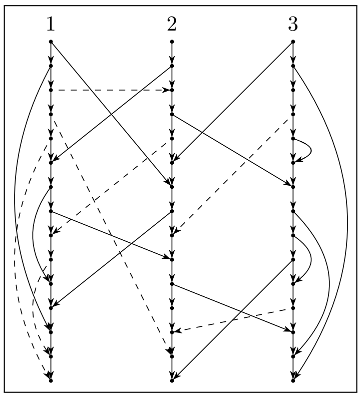
1 Introduction

Networks of multi-threaded recursive programs communicating via reliable unbounded channels are ubiquitous and their verification is as crucial as it is challenging. Recent researches have developed several approximation techniques for the verification of multi-threaded recursive programs (abstracted as multi-pushdown systems) and communicating machines. We continue this line of research. We propose a more general and unifying formalism for their modelling and consider various verification problems including reachability and model checking against logical specifications.

Towards getting a general formalism, we introduce a simple system model, Concurrent Processes with Store (CPS), which abstract the various data-structures (stacks for recursion and queues for channels) appearing in the system by a simple global store. A program in the system may write to the global store or read a value from the global store. Reads are destructive, as this is the case with stacks (pops) and queues (dequeue).

The behaviour of a CPS will be a tuple of sequences of events (one per program/process). In addition a binary matching relation links corresponding writes and reads. Such a graph is called a Concurrent Behaviour with Matching (CBM).

The matching relation can be further refined to retrieve the case when the store consists of several stacks and queues. The system model in this case is called CPSQ (Concurrent Processes with Stacks and Queues) and the behaviours are called MSCN (Message Sequence Charts with Nestings), which generalize nested words [1], multiply nested words [15], Message Sequence Charts [14] and stack-queue graphs [18].



For specifying their properties, we consider Monadic Second Order logic (MSO), Propositional Dynamic Logic with and without intersection (IPDL/PDL) and Temporal Logics (TL).

We show that the reachability problem for CPS is decidable, though with high complexity. It is infact equivalent to reachability in Petri-nets/VAS. Model checking and satisfiability of PDL without intersection is also shown to be decidable. All other problems for CPS/CBM turns out to be undecidable. Moreover, *all* the verification problems are undecidable for CPSQ/MSCN.

However, the verification of these systems is an important concern. Hence, we consider upper-approximate verification. We extend the technique of split-width to cope with CPS/CPSQ. Split-width was introduced in [8] for getting decidability for MSO model checking of multi-pushdown systems.

Split-width is based on a simple and intuitive algebra which generates CBMs. We first generalize CBMs to split-CBMs which are CBMs with holes. The holes may be seen as place holders where an actual behaviour may be inserted later. Our algebra is over split-CBMs. The atomic elements are (i) the single matching edge relating a write and the corresponding read and (ii) a single internal event. The *shuffle* operator lets us combine two split-CBMs by inserting parts of one into the holes of the other (and vice-versa). The *merge* operator allows us to close some of the holes in a split-CBM. Every CBM can be generated from the atomic ones using these two operations. split-CBM is the number of holes it has. We say that a CBM has split-width k if it can be generated by this algebra in such a way that each split-CBM obtained in any intermediate step has at most k holes. We call a term in this algebra a split-term.

The algebra allows us to recursively divide a CBM into independent parts which can be reasoned about separately. Thus, it provides a divide-and-conquer approach (or compositional reasoning) for the analysis of a behaviour. Note that, two events linked by a matching read-write are never separated in this algebra. The algebra also gives a natural embedding of CBMs into trees via split-terms, similar to how parse trees give a tree-representation for a word in a context-free language.

The valid tree-embeddings of CBMs with split-width at most k form a regular tree-language. Thus we can translate every problem on CBM/CPS to an equivalent one on tree-domain. With split-width as a parameter, we get *uniform* decision procedures with *optimal complexities* for *all* verification problems on CBM/CPS as well as MSCN/CPSQ. The complexities range from non-elementary for MSO to 2EXPTIME for IPDL to EXPTIME for PDL/TL to PTIME for reachability.

The simple notion of split-width, while giving us uniform and optimal decision procedures for a wide range of verification problems, is also very powerful. In [8,7] a bound on split-width is established for many behaviour restrictions for which decidability of reachability was known before, e.g., for multi-pushdown systems having a bound on context [20], scope [16], phase [15], ordered [6,2] and for communicating pushdown systems over acyclic topologies [12]. The systematic way of bounding the split-width also helped us to generalise these restrictions and to discover several new decidable classes.

In [7] it is further shown that a class of CBMs has bounded split-width if and only if it has bounded clique-width, and the corresponding bounds lie in a linear factor. Thus, by Courcelle's results, if a class of CBMs have a decidable MSO theory, then it also has bounded split-width. Thus split-width is not only sufficient but also necessary for MSO decidability of CBMs.

Related work [12] identifies topologies which give decidability of reachability for communicating pushdown systems. A unified proof of decidability of reachability of the known decidable restrictions (having a bound on context [20], scope [16], phase [15], ordered [6,2] for multi-pushdown systems and communicating pushdown systems over acyclic topologies) is given in [18] via demonstrating their bounded tree-width. An abstract representation of these tree-decompositions is studied in [13]. Reachability in acyclic networks of pushdown systems connected via lossy FIFO channels is considered in [3].

2 Concurrent Behaviours with Matching (CBM)

Let **Procs** $= \{1, \ldots, \mathsf{p}\}$ be a finite set of processes and Σ be a finite set of actions. We are interested in systems of concurrent processes with data structures such as queues, stacks, bags etc. With such data structures, a write action adds a value to the store, while a read action removes a value from it. We describe a behaviour of such a system as a tuple of sequences of actions (one for each process) enriched with a relation matching write events to their corresponding read events. Thus an action can be part of at most one matching edge. Furthermore, since writes must precede reads, the resulting graph is a partial order.

Definition 1. *A concurrent behaviour with matching (abbraviated as CBM) over an alphabet Σ and a set* **Procs** *of processes is a tuple $\mathcal{M} = (\mathcal{E}, \lambda, \mathsf{pid}, \rightarrow, \rhd)$ where*

- *$\mathcal{E}$ is a non-empty finite set of* events*,*
- *$\lambda \colon \mathcal{E} \to \Sigma$ labels each event with an action,*
- *$\mathsf{pid} \colon \mathcal{E} \to$* **Procs** *assigns a process to each event,*
- *$\rightarrow \subseteq \biguplus_{p \in \mathbf{Procs}} \mathsf{pid}^{-1}(p) \times \mathsf{pid}^{-1}(p)$ describes the linear orders on processes: for each process $p \in$* **Procs***, the restriction of $\rightarrow$ to the events of process p is the direct-successor relation of some total order on $\mathsf{pid}^{-1}(p)$,*

– $\rhd \subseteq \mathcal{E} \times \mathcal{E}$ *is the relation matching write events with their corresponding read events: it is irreflexive and disjoint, i.e., if* $e_1 \rhd e_2$ *then* $e_1 \neq e_2$ *and two distinct edges* $e_1 \rhd e_2$ *and* $e_3 \rhd e_4$ *must be disjoint* ($|\{e_1, e_2, e_3, e_4\}| = 4$*),*
– *The underlying graph* $(\mathcal{E}, (\to \cup \rhd)^*)$ *is a partial order.*

The set of all CBMs over **Procs** and Σ is denoted $\mathbb{CBM}(\mathbf{Procs}, \Sigma)$. Notice that, the definition of CBMs does not involve any specific data-structures such as queues or stacks. However, the $\rhd$ edges may be thought of as those arising from a global *store* shared by all processes.

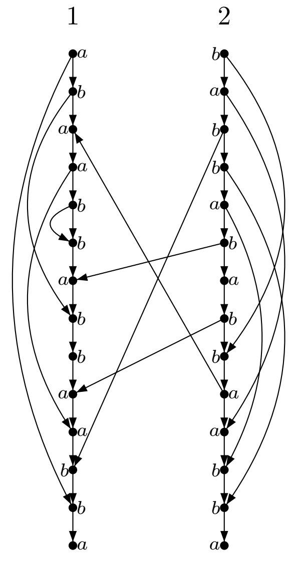


Fig. 1: A CBM $\mathcal{M}$

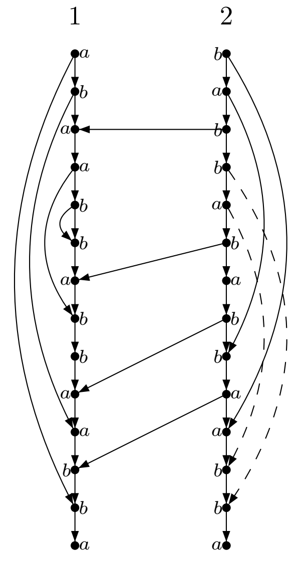


Fig. 2: An MSCN

Example 1. An example of a CBM over $\Sigma = \{a, b\}$, and $\mathbf{Procs} = \{1, 2\}$ is shown in Fig. 1. The vertical arrows denote the $\to$ edges and the other arrows denote the $\rhd$ edges. Observe that the underlying graph has no cycles. Notice also that the $\rhd$ edges do not follow any LIFO or FIFO policy.

In the following, an event which is the source (resp. the target) of a $\rhd$ edge is called a *write event* (resp. *read event*), and all other are called *internal events.*

Stacks and Queues To model behaviours of concurrent recursive multithreaded programs communicating via FIFO channels, we will consider restrictions of CBMs called Message Sequence Charts with Nestings (MSCNs) that subsume stack-queue graphs [18]. They are obtained by assuming that the global store shared by all processes is actually composed of disjoint data structures that are either stacks or queues. So we consider a finite set $\mathbf{DS} = \mathbf{Stacks} \uplus \mathbf{Queues}$ of data-structure labels. The read and write accesses are specified by mappings $\mathsf{Writer}\colon \mathbf{DS} \to \mathbf{Procs}$ and $\mathsf{Reader}\colon \mathbf{DS} \to \mathbf{Procs}$. We restrict to self-stacks, i.e., for all $d \in \mathbf{Stacks}$ we have $\mathsf{Writer}(d) = \mathsf{Reader}(d)$. We define an *architecture* to be a tuple $\mathfrak{A} = (\mathbf{Procs}, \mathbf{Stacks}, \mathbf{Queues}, \mathsf{Writer}, \mathsf{Reader})$.

An MSCN is a CBM in which events accessing the store will be additionally labeled with a data-structure and the matching relation is restricted so that it complies with the with LIFO or FIFO policies.

Definition 2. *An MSCN over architecture $\mathfrak{A}$ and alphabet Σ is a tuple $\mathcal{M} = (\mathcal{E}, \lambda, \mathsf{pid}, \delta, \to, \rhd)$ where $\mathcal{M} = (\mathcal{E}, \lambda, \mathsf{pid}, \to, \rhd)$ is a CBM and $\delta\colon \mathcal{E} \to \mathbf{DS}$ is a partial map satisfying*

- $\mathsf{dom}(\delta) = \{e \in \mathcal{E} \mid e \rhd f \text{ or } f \rhd e \text{ for some } f \in \mathcal{E}\}$,
- *If $e \rhd f$, then $\delta(e) = \delta(f)$, $\mathsf{Writer}(\delta(e)) = \mathsf{pid}(e)$ and $\mathsf{Reader}(\delta(f)) = \mathsf{pid}(f)$.*
- *For each $d \in \mathbf{Stacks}$, $\rhd^d = \rhd \cap (\delta^{-1}(d))^2$ 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 $e_1 \to^+ e_2 \to^+ f_1 \to^+ f_2$.*
- *For each $d \in \mathbf{Queues}$, $\rhd^d = \rhd \cap (\delta^{-1}(d))^2$ conforms to FIFO: if $e_1 \rhd^d f_1$ and $e_2 \rhd^d f_2$ are different message edges then we do not have $e_1 \to^+ e_2$ and $f_2 \to^+ f_1$.*

We denote by $\mathbb{MSCN}(\mathfrak{A}, \Sigma)$ the set of MSCNs over architecture $\mathfrak{A}$ and action labels Σ.

Example 2. An MSCN is shown in Fig. 2. The data-structures consist of one stack on process 1, two stacks on process 2 (one using solid arrows and the other dashed arrows) and one queue from process 2 to process 1. For readability, we do not include the **DS** labels to the events, they can be recovered from the various matching edges.

When data structures are queues between pairs of processes, MSCNs extend message sequence charts which are scenarios standardised by ITU [14]. When on the other hand there is a single process and our data structures are stacks, our setting extends nested words [1] and multiply nested words [15].

3 Concurrent Processes with Store (CPS)

We now describe a system model whose behaviours are CBMs. It consists of a set of finite state processes which share a single *store* data-structure. The store is just an unordered multi-set of unbounded size supporting two actions — add a value and remove a value.

Definition 3. *A system of concurrent processes with store (CPS) over a set of processes* **Procs** *and actions Σ is a tuple $\mathcal{S} = (\mathsf{Locs}, (\mathrm{Trans}_p)_{p \in \mathbf{Procs}}, \ell_{in}, \mathsf{Locs}_{fin})$ where*

- Locs *is the finite set of local control locations,*
- $\ell_{in} \in \mathsf{Locs}$ *is local initial state of each process,*

– $\mathsf{Locs}_{fin} \subseteq \mathsf{Locs}^{\mathbf{Procs}}$ *is the set of global final states,*
– $\mathrm{Trans}_p = \mathrm{Trans}_{p,i} \uplus \mathrm{Trans}_{p!} \uplus \mathrm{Trans}_{p?}$ *is the set of local transitions of process* p *where*

$$\begin{aligned} \mathrm{Trans}_{p,i} &\subseteq \mathsf{Locs} \times \Sigma \times \mathsf{Locs} \\ \mathrm{Trans}_{p!} &\subseteq \mathsf{Locs} \times \Sigma \times \mathsf{Locs} \times \mathsf{Locs} \\ \mathrm{Trans}_{p?} &\subseteq \mathsf{Locs} \times \mathsf{Locs} \times \Sigma \times \mathsf{Locs} \end{aligned}$$

$\mathrm{Trans}_{p,i}$ *denotes the internal transitions of Process* p*, i.e., those that do not involve the store,* $\mathrm{Trans}_{p!}$ *denotes the set of write transitions of Process* p*, i.e., transitions which write to the store and* $\mathrm{Trans}_{p?}$ *are the transitions of* p *which read from the store.*

The intended operational semantics is the following. A write transition of the form $(\ell_1, a, \ell_2, \ell)$ results in process p performing an action labeled a while moving from location ℓ_1 to ℓ_2 and inserting value ℓ into the store. A read transition of the form $(\ell_1, \ell, a, \ell_2)$ results in process p performing an action labeled a whilst consuming a value ℓ from the store and moving from ℓ_1 to ℓ_2.

As is well known in concurrency theory, it is also possible to provide the semantics of such systems directly on a partially ordered structure that represents the causal relationship between the events in the execution. For a CPS, the appropriate structure is that of CBMs, where the relation $\rhd$ relates the insertion of a value into the store and its corresponding consumption. It is easy to provide an equivalent operational semantics (see [7]).

We now define the notion of a *run* of a CPS over a CBM. Such a run labels the events of a CBM with the control locations of the CPS. This is given by the control location labeling $\rho\colon \mathcal{E} \to \mathsf{Locs}$. Further, in order to keep track of the values in the store, a run also labels write events by the value that was added to the store. This is specified by a *partial data mapping* $\nu\colon \mathcal{E} \to \mathsf{Locs}$ such that $e \in \mathsf{dom}(\nu)$ iff e is a write event.

A *run* of a CPS $\mathcal{S}= (\mathsf{Locs}, (\mathrm{Trans}_p)_{p\in\mathbf{Procs}}, \ell_{\mathrm{in}}, \mathsf{Locs}_{\mathrm{fin}})$ over a CBM $\mathcal{M} = (\mathcal{E}, \lambda, \mathsf{pid}, \to, \rhd)$ is a pair (ρ, ν) such that the following additional consistency conditions hold for all $e \in \mathcal{E}$. We denote by e^-, the unique event such that $e^- \to e$ if it exists, and otherwise $e^- = \perp_{\mathsf{pid}(e)} \notin \mathcal{E}$. We set $\rho(\perp_p) = \ell_{\mathrm{in}}$ as a convention.

1. if $e \rhd f$ then $(\rho(e^-), \lambda(e), \rho(e), \nu(e)) \in \mathrm{Trans}_{\mathsf{pid}(e)!}$,
2. if $f \rhd e$ then $(\rho(e^-), \nu(f), \lambda(e), \rho(e)) \in \mathrm{Trans}_{\mathsf{pid}(e)?}$,
3. if e is internal then $(\rho(e^-), \lambda(e), \rho(e)) \in \mathrm{Trans}_{\mathsf{pid}(e),i}$.

Let f_p denote the maximal event on process p if it exists. Otherwise, we set $f_p = \perp_p \notin \mathcal{E}$ (recall that $\rho(\perp_p) = \ell_{\mathrm{in}}$). A run is *accepting* if $(\rho(f_1), \ldots, \rho(f_{\mathsf{p}})) \in \mathsf{Locs}_{\mathrm{fin}}$.

The *language* accepted by a CPS $\mathcal{S}$ is the set of CBMs on which it has an accepting run. We denote it by $\mathscr{L}(\mathcal{S})$.

Stacks and Queues Let $\mathfrak{A} = (\mathbf{Procs}, \mathbf{Stacks}, \mathbf{Queues}, \mathsf{Writer}, \mathsf{Reader})$ be an architecture. When operating over $\mathfrak{A}$, the transitions of a CPS are refined into $\mathrm{Trans}_{p!} = \biguplus_{d \in \mathsf{Writer}^{-1}(p)} \mathrm{Trans}_{p!d}$ and $\mathrm{Trans}_{p?} = \biguplus_{d \in \mathsf{Reader}^{-1}(p)} \mathrm{Trans}_{p?d}$.

For the operational semantics, the store is split into stacks and queues whose contents are words in Locs^*. A write transition $(\ell_1, a, \ell_2, \ell) \in \mathrm{Trans}_{p!d}$ adds the value ℓ on the right of the word holding the contents of data-structure d. A read transition $(\ell_1, \ell, a, \ell_2) \in \mathrm{Trans}_{p?d}$ removes the value ℓ on the left (resp. right) of the word for queue (resp. stack) d.

The modification is even easier for runs defined over an MSCN. We only have to change the conditions above for the write and read transitions to:

1. if $e \rhd^d f$ then $(\rho(e^-), \lambda(e), \rho(e), \nu(e)) \in \mathrm{Trans}_{\mathsf{pid}(e)!d}$,
2. if $f \rhd^d e$ then $(\rho(e^-), \nu(f), \lambda(e), \rho(e)) \in \mathrm{Trans}_{\mathsf{pid}(e)?d}$.

A CPS $\mathcal{S}$ over an architecture $\mathfrak{A}$ is called a system of concurrent processes with stacks and queues (CPSQ). It defines the language $\mathscr{L}_{\mathbb{MSCN}}(\mathcal{S})$ of MSCNs in $\mathbb{MSCN}(\mathfrak{A}, \Sigma)$ admitting an accepting run.

4 Specification formalisms

We now describe three classical logical formalisms that will be used for specifying properties of CBMs. We only give their syntax and intuitive semantics here.

Monadic Second Order Logic over $\mathbb{CBM}(\mathbf{Procs}, \Sigma)$ is denoted $\mathsf{MSO}(\mathbf{Procs}, \Sigma)$. Its syntax is given below, where $p \in \mathbf{Procs}$ and $a \in \Sigma$.

$$\begin{aligned} \varphi ::= {} & a(x) \mid p(x) \mid x = y \mid x \in X \mid x \to y \mid x \rhd y \\ & \mid \varphi \vee \varphi \mid \neg\varphi \mid \exists x\, \varphi \mid \exists X\, \varphi \end{aligned}$$

Every sentence in MSO defines a language of CBMs consisting of all those that satisfy that sentence. As is well known, monadic second-order quantifiers can be used to express transitive closures — that is, for each MSO formula $\varphi(x, y)$ defining a relation R over its domain, one can construct another formula $\varphi^*(x, y)$ which defines the transitive closure of R. Thus, the predicates $\leq$ and $\leq_p$, denoting transitive closures of $(\rhd \cup \to)$ and $\to$ respectively, are $\mathsf{MSO}(\mathbf{Procs}, \Sigma)$ definable.

The formula $\forall x \forall y. a(x) \implies (x \leq y) \vee (y \leq x)$ asserts that events labeled a cannot occur *concurrently* to some other event. The formula

$$\forall x \forall y. (x \leq_p y \wedge x \rhd y) \implies \neg\exists z. (x \leq_p z \wedge z \leq_p y \wedge b(z))$$

prohibits, in any process, occurrence of b events from the point a value is inserted into the store to the point it is taken back.

Remark 1. The language of a CPS $\mathcal{S}$ over $\mathbf{Procs}$ and Σ can be described in $\mathsf{MSO}(\mathbf{Procs}, \Sigma)$. We use second order variables $\{X_\ell\}_{\ell \in \mathsf{Locs}}$ to identify the ρ mapping and $\{Y_\ell\}_{\ell \in \mathsf{Locs}}$ to identify the ν mapping and the rest of the formula asserts that the second order variable assignment identifies a valid run of the CPS.

This generous expressive power of MSO comes at a cost — in general, satisfiability and model-checking are undecidable and even for decidable fragments the decision procedures have non-elementary complexity. Therefore we examine less expressive logics with reasonable decision procedures.

Propositional Dynamic Logic (PDL) A well-studied logical formalism for describing properties of programs is that of PDL, a generalization of modal logic. As in a modal logic, formulas in PDL assert properties of nodes in a graph, in our case events in a CBM. Unlike a modal logic where modal operators only refer to neighbours of the current node, PDL uses *path modalities* to assert properties on nodes reachable via paths conforming to some regular expressions. Traditionally, PDL is used to express *branching-time* properties of *transition systems* (or *kripke structures*). However, in the study of concurrent systems where each behaviour has a graph-like structure, PDL may be used to express properties of behaviours (i.e., linear-time properties of the system under consideration) as illustrated in [11,5,4]. PDL and its extensions with *converse* and *intersection* are studied in this sense, as linear-time logics, here. The syntax of state formulas (σ) and path formulas (π) of $\mathsf{ICPDL}(\mathbf{Procs}, \Sigma)$ are given by

$$\sigma ::= \top \mid p \mid a \mid \sigma \vee \sigma \mid \neg\sigma \mid \langle\pi\rangle\sigma$$
$$\pi ::= \underline{\sigma} \mid \rightarrow \mid \rhd \mid \rightarrow^{-1} \mid \rhd^{-1} \mid \pi + \pi \mid \pi \cap \pi \mid \pi \cdot \pi \mid \pi^*$$

where $p \in \mathbf{Procs}$ and $a \in \Sigma$. If backward edges $\rightarrow^{-1}$ and $\rhd^{-1}$ are not allowed the fragment is called PDL with intersection (IPDL). If intersection $\pi \cap \pi$ is not allowed, the fragment is PDL with converse (CPDL). The basic PDL is when neither backwards edges nor intersection are allowed.

The formula p asserts that current event belongs to process p while a asserts that it is labeled by a. The formula $\langle\pi\rangle\sigma$ at e asserts the existence of an e' satisfying σ and a path $e = e_1, e_2, \ldots, e_k = e'$ that conforms to π. The only paths that conform to $\underline{\sigma}$ are the trivial paths from e to e for any e that satisfies σ. Similarly $\rightarrow$ and $\rhd$ identify pairs related by the corresponding edge relation in the CBM. Finally $\cdot$, $+$ and $*$ correspond to composition, union and iteration of paths as in regular expressions.

The formula $\langle\rightarrow^*\rangle\alpha$ asserts that α holds at some future event on the same process while the formula $\langle(\underline{\beta} \cdot \rightarrow^{-1})^*\rangle\alpha$ asserts that, β has been true at all the events in the current process since the last event (on this process) that satisfied α.

Intersection of path expressions is a powerful feature. The formula $\langle\pi_1 \cap \pi_2\rangle\alpha$ at an event e asserts the existence of an e' satisfying α, a path from e to e' conforming to π_1 and a path from e to e' conforming to π_2. For instance, the formula $\langle\rhd \cap (\rightarrow^* \cdot \underline{b} \cdot \rightarrow^*)\rangle\mathtt{true}$ is true at an event e only if there is an e' with $e \rhd e'$ on the same process and b holds somewhere between e and e' (on this process).

Observe that PDL formulas have implicit free variables. To define languages of CBMs with PDL we introduce sentences ϕ of $\mathsf{ICPDL}(\mathbf{Procs}, \Sigma)$ with the following syntax: $\phi = \top \mid \mathsf{E}\,\sigma \mid \phi \vee \phi \mid \neg\phi$ where σ is an $\mathsf{ICPDL}(\mathbf{Procs}, \Sigma)$ state formula. The sentence $\mathsf{E}\,\sigma$ is true on CBM $\mathcal{M}$ if $\mathcal{M}, e \models \sigma$ for some event e of $\mathcal{M}$. With this interpretation the formula $\neg\,\mathsf{E}\langle \rhd \cap (\to^* \cdot \underline{b} \cdot \to^*)\rangle\mathtt{true}$ is equivalent to the second MSO property written above.

Temporal Logics Another reason to study PDL over CBMs is that it naturally subsumes an entire family of temporal logics. The classical linear time temporal logic (LTL) is interpreted over discrete linear orders and comes with two basic temporal operators: the *next state* ($\mathsf{X}\varphi$) which asserts the truth of φ at the next position and the *until* ($\varphi_1 \mathsf{U} \varphi_2$) which asserts the existence of some future position where φ_2 holds such that φ_1 holds everywhere in between. In the setting of CBMs, following [11], it is profitable to extend this to a whole family of temporal operators by parametrizing the steps used by *next* and *until* with path expressions. Despite of their differing expressive powers, they can all be translated into PDL uniformly.

The syntax of local temporal logics $\mathsf{TL}(\mathbf{Procs}, \Sigma)$ is as follows, where $a \in \Sigma$, $p \in \mathbf{Procs}$ and π is a path expression:

$$\varphi = a \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathsf{X}_\pi\varphi \mid \varphi \,\mathsf{U}_\pi\, \varphi$$

For example, $\varphi \,\mathsf{U}_\pi\, \psi$ asks for the existence of a sequence of events related by π-steps and such that ψ holds at the last event of the sequence and φ holds at intervening events in the sequence. The translation in PDL gives $\langle(\underline{\Phi} \cdot \pi)^*\rangle\Psi$, where Φ and Ψ are the PDL translations of φ and ψ respectively. When π is $\to$ it corresponds to the classical until along a process, and when π is $\to + \rhd$ it corresponds to an *existential until* in the partial order of the CBM. We may also use backward steps such as $\to^{-1}$ or $\to^{-1} + \rhd^{-1}$ and thus $\mathsf{TL}(\mathbf{Procs}, \Sigma)$ has both future and past modalities.

Stacks and Queues Let $\mathfrak{A} = (\mathbf{Procs}, \mathbf{Stacks}, \mathbf{Queues}, \mathsf{Writer}, \mathsf{Reader})$ be an architecture. The logics above may be used to specify properties of MSCNs over $(\mathfrak{A}, \Sigma)$. We provide the ability to talk about the data-structures by adding atomic formulas. For MSO logic, we add $d(x)$, meaning that event x is labeled $d \in \mathbf{DS}$. For PDL or TL, we simply add atomic propositions $d \in \mathbf{DS}$ to the syntax of state formulas. Then, every sentence of $\mathsf{MSO}(\mathfrak{A}, \Sigma)$, $\mathsf{ICPDL}(\mathfrak{A}, \Sigma)$ and $\mathsf{TL}(\mathfrak{A}, \Sigma)$ defines a language of $\mathbb{MSCN}(\mathfrak{A}, \Sigma)$.

Note that MSO logic or IPDL are powerful enough to express the LIFO or FIFO restrictions. More precisely, let $x \rhd^d x' = x \rhd x' \wedge d(x) \wedge d(x')$. The LIFO condition for data-structure d can be expressed with the MSO formula

$$\neg\exists x, x', y, y',\ x \rhd^d x' \wedge y \rhd^d y' \wedge x < y < x' < y' \,.$$

The same property is expressed by the IPDL formula

$$\neg\,\mathsf{E}\langle(\rhd^d \cdot \to \cdot (\rhd^d \cdot \to + \underline{\neg\langle\rhd^d\rangle\top} \cdot \to)^*) \cap (\to^+ \cdot \rhd^d)\rangle\mathtt{true}$$

where the path expression $\rhd^d$ stands for $\underline{d} \cdot \rhd \cdot \underline{d}$.

5 Split concurrent behaviour with matching

Our analysis of CBMs via split-width will be based on a divide and conquer approach. More precisely, we will *divide* a CBM in smaller behaviours by splitting (cutting) some of the process (linear) edges so that the graph becomes disconnected. Such subgraphs are called *split*-CBMs. Once a CBM is split, a process may have several connected components and we relate them with *elastic* edges to remember the original ordering between them.

Example 3. The split-CBM $\not{\mathcal{M}}$ of Fig. 3 is obtained by splitting 5 process edges from the CBM $\mathcal{M}$ of Fig. 1, replacing them by elastic edges (wavy edges in the picture).

If we omit the elastic edges, this split-CBM is composed of 3 disconnected subgraphs which can be easily seen in Fig. 4. Hence, $\not{\mathcal{M}}$ can be divided into the two split-CBMs of Fig. 4. Notice that in these split-CBMs, we still include elastic edges allowing to recover the original ordering between events. Also, the split-CBM on the right of Fig. 4 can be further divided into two split-CBMs, one where only process 1 is active, the other where only process 2 is active.

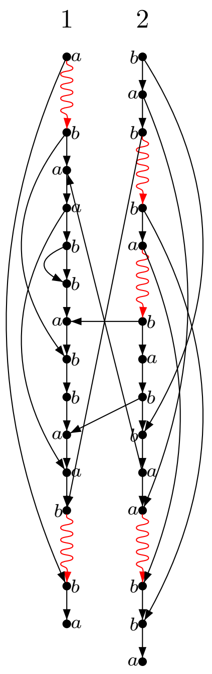


Fig. 3: Split-CBM $\not{\mathcal{M}}$ obtained by splitting $\mathcal{M}$ (Fig. 1).

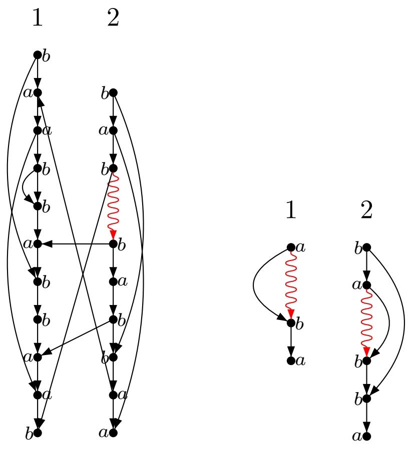


Fig. 4: $\not{\mathcal{M}}$ divided into two smaller split-CBMs.

Definition 4. *A* split-CBM *is a CBM in which $\to$ edges are partitionned into* rigid *edges (denoted $\xrightarrow{r}$) and* elastic *edges (denoted $\xrightarrow{e}$). It is a tuple $\mathcal{M} = (\mathcal{E}, \lambda, \mathsf{pid}, \xrightarrow{r}, \xrightarrow{e}, \rhd)$ such that $\mathcal{M} = (\mathcal{E}, \lambda, \mathsf{pid}, \to, \rhd)$ is a CBM with $\to = \xrightarrow{r} \uplus \xrightarrow{e}$, which we call the* underlying CBM.

The *elasticity* of a split-CBM $\mathcal{M}$ is its number of elastic edges: $\mathsf{elasticity}(\mathcal{M}) = |\xrightarrow{e}|$. A *component* of $\mathcal{M}$ is a maximal connected component of the graph restricted to rigid edges ($\xrightarrow{r}$). For instance, the split-CBM $\mathcal{M}$ of Fig. 3 has elasticity five and it has seven components. The set of *active processes* in $\mathcal{M}$ is $\mathsf{Procs}(\mathcal{M}) = \{p \in \mathbf{Procs} \mid \mathsf{pid}^{-1}(p) \neq \emptyset\}$. The number of components of a split-CBM $\mathcal{M}$ is $\mathsf{elasticity}(\mathcal{M}) + \mathsf{Procs}(\mathcal{M})$.

Stacks and Queues Indeed, this definition carries over to MSCNs: a split-MSCN over some architecture $\mathfrak{A}$ is simply a tuple $\mathcal{M} = (\mathcal{E}, \lambda, \mathsf{pid}, \delta, \xrightarrow{r}, \xrightarrow{e}, \rhd)$ such that $\mathcal{M} = (\mathcal{E}, \lambda, \mathsf{pid}, \delta, \xrightarrow{r} \uplus \xrightarrow{e}, \rhd)$ is an MSCN over $\mathfrak{A}$.

6 Merge and Shuffle

We have motivated split-CBMs with a top-down divide and conquer approach: starting from a CBM, we split some process edges and then divide the graph in smaller split-CBMs that are no more connected. Dually, we may reconstruct bottom-up CBMs. The dual operation of *divide* is *shuffle*, it takes two split-CBMs and shuffles their components to get a single split-CBM. The dual for *split* is *merge*, it turns an elastic edge split-CBM into a rigid edge.

The merge operation may transform *any* of the elastic edges into a rigid one, though only one at a time. Hence $\mathsf{merge}(\mathcal{M})$ is a set of split-CBMs.

Definition 5. *Formally, let $\mathcal{M} = (\mathcal{E}, \lambda, \mathsf{pid}, \xrightarrow{r}, \xrightarrow{e}, \rhd)$ be a split-CBM. Then, $\mathsf{merge}(\mathcal{M})$ consists of the split-CBMs $\mathcal{M}' = (\mathcal{E}, \lambda, \mathsf{pid}, \xrightarrow{r}', \xrightarrow{e}', \rhd)$ such that $\xrightarrow{r}' \uplus \xrightarrow{e}' = \xrightarrow{r} \uplus \xrightarrow{e}$, $\xrightarrow{e}' \subseteq \xrightarrow{e}$ and $|\xrightarrow{e}| - |\xrightarrow{e}'| = 1$.*

Notice that the number of components and the elasticity decrease by 1 as the result of a merge and that $|\mathsf{merge}(\mathcal{M})| = \mathsf{elasticity}(\mathcal{M})$. For instance, the split-CBM $\mathcal{M}$ of Fig. 3 has 5 elastic edges, hence $\mathsf{merge}(\mathcal{M})$ consists of five split-CBMs, each of elasticity 4.

The binary shuffle operation, denoted $\sqcup\!\sqcup$, when applied to two split-CBMs, gives a set of split-CBMs, obtained by shuffling on each process their components. For instance, the split-CBM of Fig. 3 belongs to the shuffle of the two split-CBMs of Fig. 4.

Definition 6. *Let $\mathcal{M}_i = (\mathcal{E}_i, \lambda_i, \mathsf{pid}_i, \xrightarrow{r}_i, \xrightarrow{e}_i, \rhd_i)$ for $i \in \{1, 2\}$ be two split-CBMs with $\mathcal{E}_1 \cap \mathcal{E}_2 = \emptyset$. Then, $\mathcal{M}_1 \sqcup\!\sqcup \mathcal{M}_2$ is the set of split-CBMs $\mathcal{M} = (\mathcal{E}, \lambda, \mathsf{pid}, \xrightarrow{r}, \xrightarrow{e}, \rhd)$ such that*

- *apart from the elastic edges, $\mathcal{M}$ is the disjoint union of $\mathcal{M}_1$ and $\mathcal{M}_2$, i.e., $\mathcal{E} = \mathcal{E}_1 \uplus \mathcal{E}_2$, $\lambda = \lambda_1 \uplus \lambda_2$, $\mathsf{pid} = \mathsf{pid}_1 \uplus \mathsf{pid}_2$, $\rhd = \rhd_1 \uplus \rhd_2$ and $\xrightarrow{r} = \xrightarrow{r}_1 \uplus \xrightarrow{r}_2$,*

– *for each i, the order of the components of $\mathcal{M}_i$, as prescribed by the elastic edges $\xrightarrow{e}_i$, is preserved in $\mathcal{M}$: $\xrightarrow{e}_1 \cup \xrightarrow{e}_2 \subseteq (\xrightarrow{r} \cup \xrightarrow{e})^*$.*

Note that, two consecutive components in $\mathcal{M}_i$ may either be still consecutive in $\mathcal{M}$ or may be separated by components from $\mathcal{M}_{3-i}$. An elastic edge between two components of a split-CBM is a place holder where some other components may be inserted during a shuffle. If we decide that no more components should be inserted, the elastic edge can be replaced with a rigid edge using the merge operation.

Example 4. Considering again the two split-CBMs from Fig. 4, we can see that their shuffle contains 18 split-CBMs, one of which is presented in Fig. 3, two others are presented in Fig. 5.

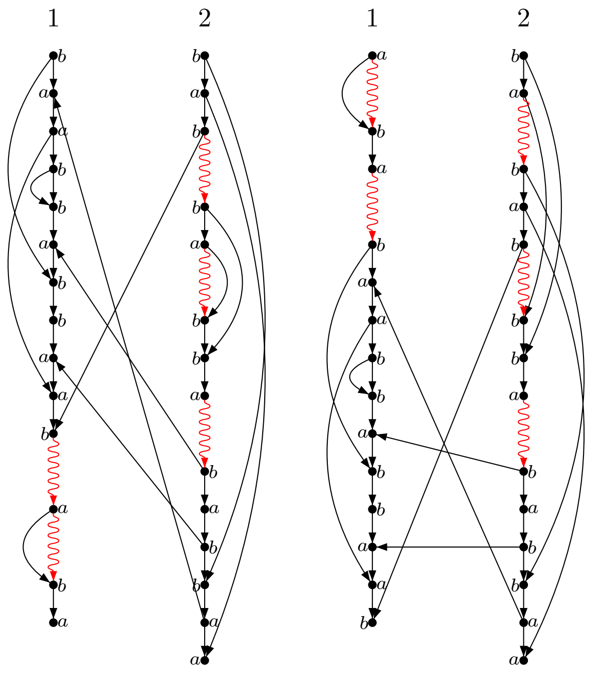


Fig. 5: Two shuffled split-CBMs.

Note that, the number of components of any split-CBM $\mathcal{M} \in \mathcal{M}_1 \sqcup \mathcal{M}_2$ is the sum of the number of components in $\mathcal{M}_1$ and the number of components in $\mathcal{M}_2$. We deduce that

$$\mathsf{elasticity}(\mathcal{M}) = \mathsf{elasticity}(\mathcal{M}_1) + \mathsf{elasticity}(\mathcal{M}_2) + \\ |\mathsf{Procs}(\mathcal{M}_1) \cap \mathsf{Procs}(\mathcal{M}_2)| \,.$$

We can lift the definition of merge and shuffle to sets of split-CBMs in the natural way. Let L, L_1 and L_2 be sets of split-CBMs. We define

$$\mathsf{merge}(L) = \bigcup_{\mathcal{M} \in L} \mathsf{merge}(\mathcal{M}) \tag{1}$$

$$L_1 \sqcup\!\sqcup L_2 = \bigcup_{\mathcal{M}_1 \in L_1, \mathcal{M}_2 \in L_2} \mathcal{M}_1 \sqcup\!\sqcup \mathcal{M}_2 \tag{2}$$

Stacks and Queues Again, merge and shuffle carry over to MSCNs by replacing split-CBMs by split-MSCNs in Def. 5 and Def. 6. Any elastic edge of a split-MSCN may be turned into a rigid edge with the merge operation. On the other hand, notice that, the shuffle operation of split-MSCNs is more restricted than for split-CBMs since the LIFO and FIFO policies of the data-structures must be respected.

7 An algebra over split-CBMs

We define an algebra which allows to reconstruct bottom-up (split)-CBMs from atomic ones (single events) using three very simple constructs: adding a matching edge between two events, shuffling two split-CBMs into a single one, or merging two consecutive components of a split-CBM into a single one.

We describe a combination of these constructs with *split-terms* over $(\mathbf{Procs}, \Sigma)$ whose syntax is given by:

$$s ::= (a,p) \mid (a,p) \rhd (a',p') \mid \mathsf{merge}(s) \mid s \sqcup\!\sqcup s$$

where $a, a' \in \Sigma$ and $p, p' \in \mathbf{Procs}$. Note that, in this algebra, the binary operator $\rhd$ may only be applied to atomic terms.

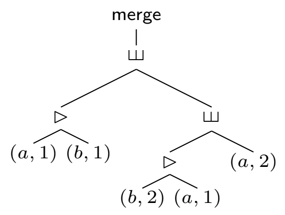
Each split-term s represents a set $[\![s]\!]$ of split-CBMs. This semantics is defined as follows:

- $[\![(a,p)]\!]$ is the CBM consisting of a single event labeled a on process p.
- $[\![(a,p) \rhd (a',p')]\!]$ is the CBM consisting of two events, e labeled a on process p, and e' labeled a' on process p'. These events are connected by a matching edge: $e \rhd e'$. Moreover, if $p = p'$, these two events are also linked by an elastic edge: $e \xrightarrow{\mathsf{e}} e'$.
- $[\![\mathsf{merge}(s)]\!] = \mathsf{merge}([\![s]\!])$ (cf. (1) in Sec. 6).
- $[\![s_1 \sqcup\!\sqcup s_2]\!] = [\![s_1]\!] \sqcup\!\sqcup [\![s_2]\!]$ (cf. (2) in Sec. 6).

Example 5. For instance, consider the split-term

$$s = \mathsf{merge}(((a,1) \rhd (b,1)) \sqcup\!\sqcup (((b,2) \rhd (a,1)) \sqcup\!\sqcup (a,2))).$$

This term can be depicted as the following tree



and its semantics consists of 18 split-CBMs, two of which are given in Fig. 6.

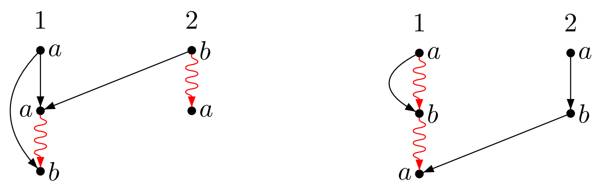


Fig. 6: Two split-CBMs

We can easily check that all the split-CBMs in the semantics $[\![s]\!]$ of a split-term s have the same set of non-empty processes, denoted $\mathsf{Procs}(s)$, the same number of components, hence also the same elasticity, donoted $\mathsf{elasticity}(s)$. We have

- $\mathsf{elasticity}((a,p)) = 0$,
- $\mathsf{elasticity}((a,p) \triangleright (a',p')) = \begin{cases} 0 \text{ if } p \neq p' \\ 1 \text{ if } p = p', \end{cases}$
- $\mathsf{elasticity}(\mathsf{merge}(s)) = \mathsf{elasticity}(s) - 1$, and
- $\mathsf{elasticity}(s_1 \sqcup s_2) = \mathsf{elasticity}(s_1) + \mathsf{elasticity}(s_2) + |\mathsf{Procs}(s_1) \cap \mathsf{Procs}(s_2)|$.

The *width* of a split-term s, denoted $\mathsf{swd}(s)$, is the maximum elasticity of all its sub-terms. For instance, the elasticity of the split-term in Example 5 is two and its width is three.

The *split-width* of a split-CBM $\mathcal{M}$, denoted $\mathsf{swd}(\mathcal{M})$, is the minimum width of all split-terms s such that $\mathcal{M} \in [\![s]\!]$. For instance, both split-CBMs of Fig. 6 have split-width two since they are respectively in the semantics of

$$s_1 = \mathsf{merge}(((a,1) \triangleright (b,1)) \sqcup ((b,2) \triangleright (a,1))) \sqcup (a,2)$$
$$s_2 = ((a,1) \triangleright (b,1)) \sqcup \mathsf{merge}(((b,2) \triangleright (a,1)) \sqcup (a,2)).$$

We denote by k-$\mathbb{CBM}$ the set of split-CBMs whose split-width is bounded by k.

Remark 2. The split-width algebra over $(\mathbf{Procs}, \Sigma)$ can generate any CBM $\mathcal{M} = (\mathcal{E}, \lambda, \mathsf{pid}, \rightarrow, \triangleright)$ over $(\mathbf{Procs}, \Sigma)$. In fact a sequence of shuffles of basic split-terms will generate a split-CBM $\mathcal{M}_1 = (\mathcal{E}, \lambda, \mathsf{pid}, \xrightarrow{\mathrm{r}}_1 = \emptyset, \xrightarrow{\mathrm{e}}_1 = \rightarrow, \triangleright)$. This will then be followed by a sequence of merges to get the split-CBM $\mathcal{M}_2 = (\mathcal{E}, \lambda, \mathsf{pid}, \xrightarrow{\mathrm{r}}_2 = \rightarrow, \xrightarrow{\mathrm{e}}_2 = \emptyset, \triangleright) = \mathcal{M}$.

Remark 3 (Split-width of Nested-Words). Nested words are MSCNs over an architecture with only one process and one stack. The split-width of any nested-word is at most two. This can be checked easily since a nested word is either the concatenation of two nested words, or of the form $\begin{tikzcd} a \arrow[r] \arrow[rr, bend left] & w_1 \arrow[r] & b \end{tikzcd}$ where w_1 is a nested word, or a basic nested word of the form a or $\begin{tikzcd} a \arrow[r] \arrow[r, bend left] & b \end{tikzcd}$.

Stacks and Queues Split-terms are easily extended to an architecture $\mathfrak{A}$ by adding the data-structure label to terms denoting a matching edge: $(a, p, d) \rhd (a', p', d)$ means that data-structure d is accessed by the write event a on process p and the matching read event a' on process p'. The semantics of merge and shuffle is as above, restricting to $\mathbb{MSCN}(\mathfrak{A}, \Sigma)$.

8 Main Results

We are interested in various verification problems (Table 1) for CPS/CBM and CPSQ/MSCN. The Emptiness Problem (1, resp. 1') asks whether a given CPS (resp. CPSQ) is empty. The satisfiability problems for MSO (2) and the different variants of PDL (3,4, 5) ask whether a given sentence is satisfied by any CBM. The same questions can be asked in the case of MSCNs as well (2',3',4',5'). The model checking problems on the other hand, check for the existence of a satisfying model in the language of a CPS (6,7,8,9) or respectively CPSQ (6',7',8',9'). As we will see soon, most of these problems are undecidable, and the decidable ones are of very high complexity. Hence we consider the parametrised (by split-width) version of the above problems (10-18, 10'-18'). The various verification problems are stated in Table 1. A main contribution of our paper is the decidability and complexity results of these problems:

Theorem 1. *The decidability and complexity of the problems 1-18 and 1'-18' (Table 1) are as stated in Table 2.*

In this section, we proceed to prove the results for problems (1-9) and (1'-9'). The results for the remaining problems are proved in the following sections.

First, we consider control state reachability or equivalently the (language) emptiness problem. For CPSQ the emptiness problem (1') is undecidable due to the presence of multiple stacks (and/or queues). The reachability problem for CPS (1) is equivalent to the reachability problem for *vector addition systems* (VAS) or Petri Nets, a problem whose decidability is known but highly non-trivial [19,17].

Let $\mathbf{L}$ be a set of labels. A *vector addition system* V over $\mathbf{L}$ is a finite set of functions from the set $\{-1, 0, 1\}^{\mathbf{L}}$. Elements of V are called *transitions.* A *marking* of V is an element of $\mathbb{N}^{\mathbf{L}}$.

We write $v+w$ for the function defined as $(v+w)(i) = v(i)+w(i)$. A transition w is *enabled* at v if $(v + w)(i) \geq 0$ for all $i \in \mathbf{L}$. We write $v \xrightarrow{w} v'$ for markings v, v' and a transition $w \in V$ if w is enabled at v and $v' = v + w$. Let $v \rightarrow v'$ if there is some $w \in V$ such that $v \xrightarrow{w} v'$ and let $\rightarrow^*$ be the reflexive-transitive closure of $\rightarrow$.

Problem		Input	Question
ID	Name		
1	Emptiness	$\mathcal{S}$: CPS	Is $\mathscr{L}(\mathcal{S}) = \emptyset$?
2	MSO-Sat	ϕ: MSO sentence	Is there a CBM $\mathcal{M}$ s.t. $\mathcal{M} \models \phi$?
3,4,5	(ϵ,C,IC)PDL-Sat	ϕ: (ϵ,C,IC)PDL sentence	
6	MSO-MC	ϕ: MSO sentence $\mathcal{S}$: CPS	Is there a CBM $\mathcal{M} \in \mathscr{L}(\mathcal{S})$ s.t. $\mathcal{M} \models \phi$?
7,8,9	(ϵ,C,IC)PDL-MC	ϕ: (ϵ,C,IC)PDL sentence $\mathcal{S}$: CPS	
10	par-Emptiness	$\mathcal{S}$: CPS, $k \in \mathbb{N}$	Is $\mathscr{L}(\mathcal{S}) \cap k\text{-}\mathbb{CBM} = \emptyset$?
11	par-MSO-Sat	ϕ: MSO sentence $k \in \mathbb{N}$	Is there a CBM $\mathcal{M} \in k\text{-}\mathbb{CBM}$ s.t. $\mathcal{M} \models \phi$?
12,13,14	par-(ϵ,C,IC)PDL-Sat	ϕ: (ϵ,C,IC)PDL sentence $k \in \mathbb{N}$	
15	par-MSO-MC	ϕ: MSO sentence $\mathcal{S}$: CPS $k \in \mathbb{N}$	Is there a CBM $\mathcal{M} \in \mathscr{L}(\mathcal{S}) \cap k\text{-}\mathbb{CBM}$ s.t. $\mathcal{M} \models \phi$?
16,17,18	par-(ϵ,C,IC)PDL-MC	ϕ: (ϵ,C,IC)PDL sentence $\mathcal{S}$: CPS $k \in \mathbb{N}$	

Table 1: Problems for CPS and CBM. Primed versions refer to the corresponding problems for CPSQ/MSCN.

ID	Complexity
1,3,7	Decidable (VAS Reachability)
2,5,6,9, 1'-9'	Undecidable
4,8	Open
10,10'	ExpTime in k PTime in $\lvert\mathcal{S}\rvert$
11,11', 15,15'	Non-elementary
12,12', 13,13'	ExpTime in k and $\lvert\phi\rvert$
14,14'	2ExpTime in $\lvert\phi\rvert$ ExpTime in k
16,16', 17,17'	ExpTime in k, $\lvert\phi\rvert$ PTime in $\lvert\mathcal{S}\rvert$
18,18'	2ExpTime in $\lvert\phi\rvert$ ExpTime in k PTime in $\lvert\mathcal{S}\rvert$

Table 2: Decidability and complexities

The *reachability problem* for VAS is the following: Given a VAS V and an *initial* marking u determine whether $u \rightarrow^* \mathbf{0}$ where $\mathbf{0}$ is the constant function returning 0. The reachability problem is known to be decidable [19,17] but its precise complexity is not known.

Remark 4. The classical definition of VAS permits transitions to come from the set $\mathbb{Z}^{\mathbf{L}}$ and further the reachability problem is often formulated as given u and v in $\mathbb{N}^{\mathbf{L}}$ whether $u \rightarrow^* v$. As is well known, as far as decidability goes, our version is equivalent to the general one.

As a first step we show that the emptiness problem for CPS can be reduced to the reachability problem for VAS.

Lemma 1. *Given a CPS $\mathcal{S} = (\mathsf{Locs}, (\mathrm{Trans}_p)_{p \in \mathbf{Procs}}, \ell_{in}, \mathsf{Locs}_{fin})$ over the set of processes* **Procs** *and alphabet Σ, we can construct a VAS $V_{\mathcal{S}}$ and an initial marking $u \in V_{\mathcal{S}}$ such that the language of $\mathcal{S}$ is non-empty if and only if $u \rightarrow^* \mathbf{0}$.*

Proof. Clearly, as far as the emptiness problem goes, one may assume that Σ is a singleton, by relabeling all transitions to use this single letter if necessary. So, we assume that $\Sigma = \{a\}$ is singleton. $V_{\mathcal{S}}$ is defined over the set of labels

$$\mathbf{L} = \mathbf{Procs} \times \mathsf{Locs} \ \uplus \ \mathsf{Locs}\,.$$

For each transition $\tau \in \biguplus \mathrm{Trans}_p$, we have an element v_τ in $V_{\mathcal{S}}$ given by

1. Case $\tau = (\ell_1, a, \ell_2) \in \mathrm{Trans}_{p,i}$: $v_\tau(p, \ell_1) = -1$, $v_\tau(p, \ell_2) = 1$ and $v_\tau(x) = 0$ for all other $x \in \mathbf{L}$.
2. Case $\tau = (\ell_1, a, \ell_3, \ell) \in \mathrm{Trans}_{p!}$: $v_\tau(p, \ell_1) = -1$, $v_\tau(p, \ell_2) = 1$, $v_\tau(\ell) = 1$ and $v_\tau(x) = 0$ for all other $x \in \mathbf{L}$.
3. Case $\tau = (\ell_1, \ell, a, \ell_2)$: $v_\tau(p, \ell_1) = -1$, $v_\tau(p, \ell_2) = 1$, $v_\tau(\ell) = -1$ and $v_\tau(x) = 0$ for all other $x \in \mathbf{L}$.

In addition for each $f = (f_p)_{p \in \mathbf{Procs}} \in \mathsf{Locs}_{\mathrm{fin}}$, we have a transition τ_f given by

4) $\tau_f(p, f_p) = -1$ for $p \in \mathbf{Procs}$ and $\tau_f(x) = 0$ for all other $x \in \mathbf{L}$.

We let u be the marking with $u(p, \ell_{in}) = 1$ for all $p \in \mathbf{Procs}$ and $u(x) = 0$ for all other $x \in \mathbf{L}$. Then, $u \rightarrow^* \mathbf{0}$ iff $\mathcal{S}$ accepts at least one CBM. $\square$

Lemma 2. *Let V be a VAS over the set of labels* **L** *and let u be a marking. Then there is a CPS $\mathcal{S} = (\mathsf{Locs}, (\mathrm{Trans}_p)_{p \in \mathbf{Procs}}, \ell_{in}, \mathsf{Locs}_{fin})$ with* $\mathbf{Procs} = \{p\}$ *a singleton and $\Sigma = \{a\}$ a singleton such that $u \rightarrow^* \mathbf{0}$ if and only if the language of $\mathcal{S}$ is non-empty.*

Proof. W.l.o.g we assume that $u \neq \mathbf{0}$. The idea is to use one location for each $\ell \in \mathbf{L}$ and in the simulation keep c copies of ℓ in the store whenever the marking of the VAS assigns the value c to ℓ. Thus, to begin the simulation from the marking u we add we add $u(\ell)$ copies of ℓ to the store. Subsequently, each transition v is simulated using $|\mathbf{L}|$ transitions, one transition to modify the store content of each $\ell \in \mathbf{L}$ as specified in v.

Let $\leq_{\mathbf{L}}$ be some linear order on $\mathbf{L}$, $\lessdot_{\mathbf{L}}$ be the covering relation of this linear order and let b and e be the minimum and maximum elements under $\leq_{\mathbf{L}}$. The state space is

$$\mathsf{Locs} = \{(\ell, i) \mid \ell \in \mathbf{L}, i \leq u(\ell)\} \uplus \{(v, \ell) \mid v \in V, \ell \in \mathbf{L}\} \uplus \mathbf{L}$$

with $\ell_{\text{in}} = (b, u(b))$ and $\mathsf{Locs}_{\text{fin}} = \{(e, 0)\}$.

States of the from (ℓ, i), $i \geq 1$, are used to initialize the store to u (using the first 2 types of transitions below). The state $(e, 0)$ has a special role, which is to initiate the simulation of the next transition (using transitions of type 3 given below). States of the form (v, ℓ) are used is to modify the number of copies of ℓ in the store as required by v (using transitions of types 4-9). Observe at the end of the simulation of a transition, the simulation reverts back to $(e, 0)$ so that a new simulation may begin.

The set of transitions is as follows

$$\begin{aligned}
((\ell, i), a, (\ell, i-1), \ell) &\in \text{Trans}_{p!} \text{ if } i > 0.\\
((\ell, 0), a, (\ell', u(\ell'))) &\in \text{Trans}_{p,i} \text{ if } \ell \lessdot_{\mathbf{L}} \ell'.\\
((e, 0), a, (v, b) &\in \text{Trans}_{p,i} \ \forall v \in V.\\
((v, \ell), a, (v, \ell'), \ell) &\in \text{Trans}_{p!} \text{ if } \ell \lessdot_{\mathbf{L}} \ell' \text{ and } v(\ell) = 1.\\
((v, \ell), \ell, a, (v, \ell')) &\in \text{Trans}_{p?} \text{ if } \ell \lessdot_{\mathbf{L}} \ell' \text{ and } v(\ell) = -1.\\
((v, \ell), a, (v, \ell')) &\in \text{Trans}_{p,i} \text{ if } \ell \lessdot_{\mathbf{L}} \ell' \text{ and } v(\ell) = 0.\\
((v, e), a, (e, 0), e) &\in \text{Trans}_{p!} \text{ if } v(e) = 1.\\
((v, e), e, a, (e, 0)) &\in \text{Trans}_{p?} \text{ if } v(e) = -1.\\
((v, e), a, (e, 0)) &\in \text{Trans}_{p,i} \text{ if } v(e) = 0. \qquad \Box
\end{aligned}$$

Next we consider decision problems for the logics described in Sec. 4. Given a sentence in such a logic we would like to check whether it is absurd. This amounts to asking whether there is at least one CBM which can model the given sentence, the so-called *satisfiability* problem for the logic. The satisfiability problems (2',5') for MSO and IPDL are undecidable over MSCNs, even when $\mathfrak{p} = 1$. Indeed, the set of runs of a multi-pushdown system or an automaton with queues is expressible in these logics and as a result the satisfiability problem for these logics is undecidable.

The satisfiability problems (2,5) for MSO and IPDL over CBMs are also undecidable. Recall, from Sec. 4, that MSO and IPDL can both express LIFO (as well as FIFO) restrictions. This can be used to show that the set of runs of a multi-pushdown system or an automaton with queues is also expressible in these logics over CBMs.

Another important verification problem is model checking of CPS (resp. CPSQ) against specifications given in some logic, that is, given a system and a formula, decide whether some CBM (resp. MSCN) accepted by the CPS (resp. CPSQ) satisfies the formula. As usual, the satisfiability problem reduces to the model checking problem since it is easy to describe a CPS (resp. CPSQ) accepting all CBMs (resp. MSCNs) over $(\mathbf{Procs}, \Sigma)$. Thus, due to the presence of multiple-stacks/queues in CPSQs, model-checking problems (6'–9') are all undecidable. Since MSO and IPDL can express LIFO or FIFO restrictions, the model checking problems (6,9) are also undecidable.

For PDL and TL, both satisfiability (3) and model-checking for CPS (7) are decidable with high complexity. The solution presented below uses the classical tableaux construction. These problems remain open for PDL with converse.

To prove that the satisfiability problem for PDL sentences over CBMs is decidable, we consider an extended labeling of a CBM with maximal consistent subsets of the Fischer-Ladner closure of the formula [9]. The proof is then done in two steps. In the first step, we describe consistency conditions on the labelings. These conditions essentially relate the labeling of a node with that of its neighbouring nodes. In the second step we argue that these conditions can be checked by a CPS. For the satisfiability of the PDL sentence, the CPS in addition needs to witness some nodes in which some state formulas hold. Thus, the satisfiability problem for PDL sentences can be reduced to the emptiness checking of CPS, which is decidable (with complexity equivalent to reachability in VAS).

First, consider a PDL formula $\langle\pi\rangle\sigma$. This formula is equivalent to $\langle\pi\cdot\underline{\sigma}\rangle\mathtt{true}$. Hence, for the sake of notations, we may assume that if $\langle\pi\rangle\sigma$ appears as a subformula then σ is $\mathtt{true}$. Furthermore, we simply denote it by $\langle\pi\rangle$ (which means $\langle\pi\rangle\mathtt{true}$).

Next, we describe the Fischer-Ladner closure of a PDL formula. The path expressions present some subtleties. We need to indicate the current "progress" of a path expression. Hence we first consider the automaton for a path expression seen as a regular expression.

Let π be a path expression. Let $\text{Tests}(\pi) = \{\underline{\beta_1}, \ldots, \underline{\beta_k}\}$ be the set of tests appearing in π. Now, π is a regular expression over the alphabet $\{\to, \rhd\}\cup\text{Tests}(\pi)$. Let $\mathcal{A}_\pi$ be a finite state automaton for π (obtained from any standard regular expressions to finite state automata translation).

Now we are ready to describe the closure of a PDL state formula σ, denoted $\mathtt{CL}(\sigma)$.

- $\sigma \in \mathtt{CL}(\sigma)$.
- If $\neg\sigma_1 \in \mathtt{CL}(\sigma)$ then $\sigma_1 \in \mathtt{CL}(\sigma)$.
- If $\sigma_1 \in \mathtt{CL}(\sigma)$ and σ_1 is not a negation then $\neg\sigma_1 \in \mathtt{CL}(\sigma)$.
- If $\sigma_1 \vee \sigma_2 \in \mathtt{CL}(\sigma)$ then $\sigma_1, \sigma_2 \in \mathtt{CL}(\sigma)$.
- If $\langle\pi\rangle \in \mathtt{CL}(\sigma)$ then
  - $\langle\pi@q\rangle \in \mathtt{CL}(\sigma)$ for all states q of $\mathcal{A}_\pi$,
  - $\beta \in \mathtt{CL}(\sigma)$ for all $\underline{\beta} \in \text{Tests}(\pi)$.

Now we are ready to describe the first step of the proof. We consider the labeling of the events of a CBM by maximal consistent subsets of $\mathtt{CL}(\sigma)$. The labeling of an event is intended to describe the precise set of formulas in $\mathtt{CL}(\sigma)$ which hold true at that event. If $\langle\pi@q\rangle$ is present at an event, the intended meaning is that a run of the automaton $\mathcal{A}_\pi$ from state q to a final state can be simulated as a walk on the CBM from the current event. We will now describe the consistency conditions on the labeling which ensure that the labeling indeed does what we want.

A set $X \subseteq 2^{\mathtt{CL}(\sigma)}$ can label an event labeled a on process p only if it agrees to the following *static* consistency:

– if $b \in \Sigma \cap \mathtt{CL}(\sigma)$ then $b \in X \Leftrightarrow b = a$,
– if $q \in \mathbf{Procs} \cap \mathtt{CL}(\sigma)$ then $q \in X \Leftrightarrow q = p$,
– if $\neg\sigma_1 \in \mathtt{CL}(\sigma)$ then $\sigma_1 \in X \Leftrightarrow \neg\sigma_1 \notin X$,
– if $\sigma_1 \vee \sigma_2 \in \mathtt{CL}(\sigma)$ then $\sigma_1 \vee \sigma_2 \in X \Leftrightarrow \sigma_1 \in X$ or $\sigma_2 \in X$,
– if $\langle\pi\rangle \in \mathtt{CL}(\sigma)$ then
  - $\langle\pi\rangle \in X \Leftrightarrow \langle\pi@q\rangle \in X$ for some initial state q of $\mathcal{A}_\pi$,
  - $\langle\pi@q'\rangle \in X$ and $\beta \in X$ and $q \xrightarrow{\beta} q'$ in $\mathcal{A}_\pi$ implies $\langle\pi@q\rangle \in X$,
  - q is a final state of $\mathcal{A}_\pi$ implies $\langle\pi@q\rangle \in X$.

Moreover the labeling of an event should be consistent with that of its $\rightarrow$ and $\rhd$ successors. Let e be an event of the CBM and let e' and e'' be events of the CBM such that $e \rightarrow e'$ and $e \rhd e''$ if they exist. Let X, X' and X'' be their respective labelings. Then the following (local) *dynamic* consistency must also hold. If $\langle\pi\rangle \in \mathtt{CL}(\sigma)$ and q is a state of $\mathcal{A}_\pi$ then $\langle\pi@q\rangle \in X$ only if there is a path $q = q_0 \xrightarrow{\beta_1} q_1 \xrightarrow{\beta_2} q_2 \ldots \xrightarrow{\beta_k} q_k$ in $\mathcal{A}_\pi$ with $k \geq 0$, $\beta_1, \beta_2, \ldots, \beta_k \in X$ and

– either q_k is a final state of $\mathcal{A}_\pi$,
– or $q_k \xrightarrow{\rightarrow} q'$ in $\mathcal{A}_\pi$ and $\langle\pi@q'\rangle \in X'$,
– or $q_k \xrightarrow{\rhd} q'$ in $\mathcal{A}_\pi$ and $\langle\pi@q'\rangle \in X''$.

Furthermore, for all $\langle\pi@q'\rangle \in \mathtt{CL}(\sigma)$, if $\langle\pi@q'\rangle \in X'$ and $q \xrightarrow{\rightarrow} q'$ in $\mathcal{A}_\pi$ then $\langle\pi@q\rangle \in X$. Also, for all $\langle\pi@q''\rangle \in \mathtt{CL}(\sigma)$, if $\langle\pi@q''\rangle \in X''$ and $q \xrightarrow{\rhd} q''$ in $\mathcal{A}_\pi$ then $\langle\pi@q\rangle \in X$.

Let $\chi : \mathcal{E} \rightarrow 2^{\mathtt{CL}(\sigma)}$ be a labeling of the events. We say that χ is *consistent* if it respects the static and (local) dynamic consistency conditions described above. The consistency condition ensures the correctness of the labeling:

Lemma 3. *χ is consistent if and only if $\chi(e) = \{\alpha \in \mathtt{CL}(\sigma) \mid \mathcal{M}, e \models \alpha\}$.*

The first step of the proof is complete. In the second step we argue that the above consitency conditions can be checked by a CPS. The CPS remembers a pair (X_1, X_2) in its locations. X_1 is the labeling of the current event and X_2 is the expected labeling of the successor event. Then,

– $((X_1, X_2), a, (X_2, X_3), X_4) \in \text{Trans}_{p!}$ if $(X, X', X'') = (X_2, X_3, X_4)$ satisfies the consistency conditions,
– $((X_1, X_2), a, (X_2, X_3)) \in \text{Trans}_{p,i}$ if $(X, X', X'') = (X_2, X_3, \emptyset)$ satisfies the consistency conditions,
– $((X_1, X_2), X_2, a, (X_2, X_3)) \in \text{Trans}_{p?}$ if $(X, X', X'') = (X_2, X_3, \emptyset)$ satisfies the consistency conditions.

From the initial state, there is a transition to a pair of static consistent subsets of $\mathtt{CL}(\sigma)$. That is, the pair (X_1, X_2) may be replaced with ℓ_{in} in the above paragraph to get the initial transitions.

A pair (X_2, X_3) is a final state on a process if $X_3 = \emptyset$ and $(X, X', X'') = (X_2, \emptyset, \emptyset)$ satisfies the consistency conditions.

In order to evaluate a PDL sentence ϕ, we add a flag bit ε_σ for each subformula $\mathsf{E}\,\sigma$ of ϕ. The flag bit ε_σ is set to 0 initially and it is set to 1 as soon as a transition (as described above) with $\sigma \in X_2$ is taken. The global final acceptance then permits all combinations of locations which are final states on respective processes and in addition, the formula ϕ evaluates to $\mathtt{true}$ when its subformulas $\mathsf{E}\,\sigma$ are replaced with their associated flag bits ε_σ.

The construction is complete. The sentence ϕ is satisfiable if and only if the language accepted by the constructed CPS is non-empty.

Remark 5. The construction will not go through as such if we consider CPDL instead of PDL. However, if the path-expressions inside any iteration $(*)$ are one-way then the above construction can be adapted to check its satisfiability.

Thus, we see that most verification problems are undecidable. Moreover, the decidable cases have high complexity. However, if we restrict these decision problems to CBMs (or MSCNs) with split-width bounded by some *parameter* k, we obtain tractable decision procedures for almost all of them in a uniform manner (the only exception is in problems where the MSO formula is part of the input, where non-elementary lower bounds are known even for ordinary words, leave alone CBMs). The following sections describe this technique in detail.

9 Split-CBMs to Trees

Here, we show how to encode CBMs of bounded split-width with finite binary trees. This is a crucial step in all the decision procedures described in Sec. 10 that exploit the bound on split-width.

As it stands, each split-term s defines a set of split-CBMs $[\![s]\!]$. In order to reason about each split-CBM we decorate split-terms with additional labels so that each such labeled term denotes a unique split-CBM. The reason $[\![s]\!]$ is a set is that the operations merge and $\sqcup\!\sqcup$ are *ambiguous*. For instance, the merge operation replaces one elastic edge by a rigid edge, but does not specify which. By decorating each merge operation with the identity of this edge we resolve this ambiguity. The shuffle operation permits the interleaving of the components coming from its two operands in multiple ways and we disambiguate it by decorating each shuffle operation with the precise ordering of these components.

The key observation is that we only need a finite set of labels to disambiguate every split-term of width at most k. Our labels consist of a word per process, containing one letter per component indicating the origin of that component. At merge nodes we use letters m to denote that it is the result of a merge and i to indicate that it is inherited as it is from the operand. At a shuffle node we use letters ℓ to indicate it comes from the left operand and r to indicate it comes from the right. For instance, Fig. 7 and 8 give the 3-DSTs corresponding to the split-CBMs in Fig. 6 arising from the split-term s from Ex. 5.

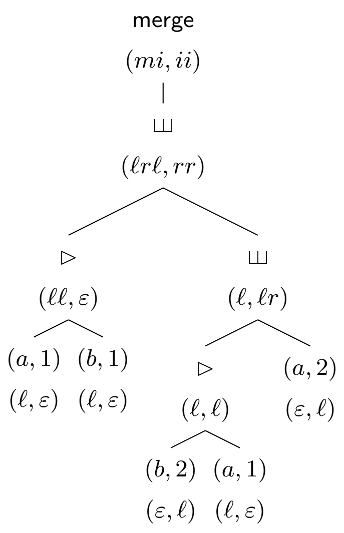


Fig. 7: A 3-DST for the split-term in Ex. 5.

Consider the set of labels

$$L_k = (\{i, m\}^{\leq k})^{\mathbf{Procs}} \cup (\{\ell, r\}^{\leq k})^{\mathbf{Procs}}.$$

A *k-disambiguated split-tree* or k-DST is a split-term s of width at most k, treated as a binary tree, labeled by L_{k+1}, and satisfying some validity conditions. Hence, each node n of t corresponds to a subterm s_n of s, and we denote by $(W_p(n))_{p\in\mathbf{Procs}}$ its labeling. We define simultaneously, the validity condition for the labeling at each node n and the unique split-CBM $\mathcal{M}_n$ belonging to $[\![s_n]\!]$ identified by this labeling.

1. If n is a leaf with associated split-term $s_n = (a, p) \in \Sigma \times \mathbf{Procs}$, then $\mathcal{M}_n$ is the unique split-CBM in $[\![s_n]\!]$ and we set $W_p(n) = \ell$ and $W_{p'}(n) = \varepsilon$ if $p' \neq p$.
2. If n is a $\rhd$ node then its children must be leaves n' and n'' and $s_n = s_{n'} \rhd s_{n''}$. Again, $\mathcal{M}_n$ is the unique split-CBM in $[\![s_n]\!]$ and we set $W_p(n) = W_p(n') \cdot W_p(n'')$ for all $p \in \mathbf{Procs}$.
3. If n is a merge node, then it has a single child n' and $s_n = \mathsf{merge}(s_{n'})$. In this case, there must be exactly one process p such that $W_p(n) \in i^*mi^*$ and $|W_p(n)| = |W_p(n')| - 1$ and for all other $p' \neq p$, we have $W_{p'}(n) \in i^*$ and $|W_{p'}(n)| = |W_{p'}(n')|$. Further, $\mathcal{M}_n$ is the split-CBM obtained from $\mathcal{M}_{n'}$ by merging on process p the component indicated by m in $W_p(n)$ with the next component. Clearly $\mathcal{M}_n \in [\![s_n]\!]$.

4. If n is a shuffle node, it has two children n' and n'' and $s_n = s_{n'} \sqcup s_{n''}$. Then, for each process $p \in \mathbf{Procs}$, we have $W_p(n) \in \{\ell, r\}^{\leq k}$ and $\#_\ell(W_p(n)) = |W_p(n')|$ and $\#_r(W_p(n)) = |W_p(n'')|$. Moreover, $\mathcal{M}_n$ is the unique split-CBM obtained by shuffling the components of $\mathcal{M}_{n'}$ and $\mathcal{M}_{n''}$ as indicated by $(W_p(n))_{p\in\mathbf{Procs}}$ and once again $\mathcal{M}_n \in [\![s_n]\!]$.

Clearly, the validity conditions above for k-DSTs can be checked with a deterministic bottom-up tree automaton.

Lemma 4. *The set of k-DSTs is a regular tree language recognized by a tree automaton $\mathcal{A}_{k\text{-valid}}$ whose size is $2^{\mathcal{O}(k|\mathbf{Procs}|)} + \mathcal{O}(|\Sigma| \cdot |\mathbf{Procs}|)$.*

We write $\mathcal{M}_t$ for the split-CBM described by the root of t. When a split-term s has width k, it is not difficult to see that, any split-CBM $\mathcal{M} \in [\![s]\!]$ can be obtained as $\mathcal{M}_t$ for some k-DST t with associated split-term s.

We can recover $\mathcal{M}_t$ from t and hence *reason* about $\mathcal{M}_t$ using t. Clearly the events in $\mathcal{M}_t$ are in bijective correspondence with the leaves of t and we identify the two. If n and n' are leaves of t then $n \rhd n'$ in $\mathcal{M}_t$ iff there is a $\rhd$ node in t whose left child is n and right child is n'.

When is $n \xrightarrow{\mathrm{r}} n'$ in $\mathcal{M}_t$? The $\xrightarrow{\mathrm{r}}$ edge connecting them is to be found in some merge common ancestor of n and n'. We walk up the tree starting at leaf n tracking the identity of the component whose last event is n (this component may grow in size as previous components are merged with it), till a merge node x merging this component with the next is encountered. We also walk up the tree starting at the leaf n' tracking the identity of its component till a merge node x' merging this component with the previous one is encountered. These routes from n and n' are marked in red and blue in Fig. 8.

Clearly, $n \xrightarrow{\mathrm{r}} n'$ iff $x = x'$. It is easy to build a bottom-up tree automaton to carry out this tracking and to check if $x = x'$. This gives us the first part of the following Proposition. The second part follows from the observation that having found x one may walk down from there to the leaf n'.

Proposition 1. – *There is a deterministic bottom-up tree automaton with at most $3k\mathfrak{p} + 2$ states which accepts the set of k-DSTs t having exactly two marked leaves n and n' such that $n \xrightarrow{\mathrm{r}} n'$ in the split-CBM $\mathcal{M}_t$.*
– *There is a deterministic tree-walking automaton with at most $2k\mathfrak{p}$ states which has an accepting run on a k-DST t from leaf n to leaf n' iff $n \xrightarrow{\mathrm{r}} n'$ in the split-CBM $\mathcal{M}_t$.*

Remark 6. It is also possible to restrict Lemma 4 and Proposition 1 to k-DSTs that identify CBMs (as opposed to split-CBMs). An analogue of Proposition 1 can also be established for the relation $\xrightarrow{\mathrm{r}}^{-1}$ as well as $\xrightarrow{\mathrm{e}}$ and $\xrightarrow{\mathrm{e}}^{-1}$.

10 Split-width and Decidability

10.1 Satisfiability

An easy consequence of Proposition 1 is that the satisfiability problem for MSO becomes decidable when restricted to CBMs of split-width at most k.

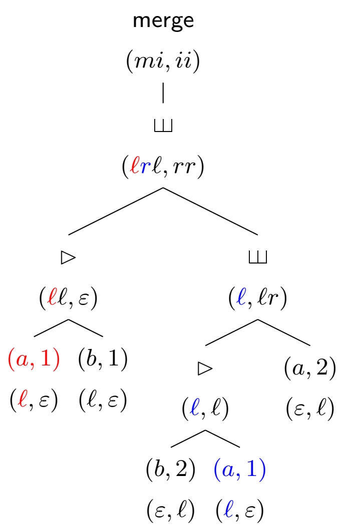


Fig. 8: Recovering a rigid edge in a 3-DST.

Theorem 2. *From any* $\mathsf{MSO}(\mathbf{Procs},\Sigma)$ *formula* ψ *one can effectively construct a tree automaton* $\mathcal{A}^k_\psi$ *such that*

$$L(\mathcal{A}^k_\psi) = \{t \mid t \text{ is a } k\text{-DST},\ \mathcal{M}_t \models \psi\}\,.$$

By Lemma 4 and Remark 6 we may assume that the input is a k-DST representing a CBM. The argument is quite standard: construct the automaton inductively, using closure under union, intersection, complement and projection to handle the boolean operators and quantifiers. This leaves the atomic formulas. The formula $a(x)$ is translated to a tree automaton that verifies that x is a leaf and that it is labeled a and similarly for $p(x)$. The formula $x \in X$ is translated to a tree automaton that verifies that x is a leaf and belongs to the set of leaves labeled by X. $x \triangleright y$ just requires us to verify that the leaves labeled x and y have a parent labeled $\triangleright$. Finally, $x \to y$ is handled using Proposition 1, completing the proof. As always, the combination of projection and complementation means that the size of the constructed automaton grows as a non-elementary function.

Decidability for MSO immediately implies decidability for all the variants of PDL (and the Temporal Logics). However, we obtain more efficient decision procedures by working directly with these logics.

Theorem 3. *From any CPDL sentence* ψ *one can effectively construct a tree automaton* $\mathcal{A}^k_\psi$ *whose size is* $2^{\mathcal{O}(\mathfrak{p}^2.k^2.|\sigma|^2)}$ *such that* $L(\mathcal{A}^k_\psi) = \{t \mid t \text{ is a } k\text{-DST},\ \mathcal{M}_t \models \psi\}$.

The idea here is to use alternating 2-way tree automata (A2A). For a PDL sentence $\mathsf{E}\,\sigma$ the A2A walks down to a leaf and starts a single copy of the A2A that will verify the formula σ. For each σ we construct an automaton A_σ such that A_σ has an accepting run from a leaf n if and only if the event n in the associated CBM satisfies σ. The automata for the atomic formulas $\texttt{true}$, p and a are self-evident. For $\vee$ and $\neg$ we use the constructions for union and complementation for A2A. The case where $\sigma = \langle\pi\rangle\sigma'$ needs a little bit of work. Suppose π does not use any state formulas then we construct a finite automaton $\mathcal{B}_\pi$ equivalent to the regular expression π (over the alphabet $D = \{\rightarrow, \rightarrow^{-1}, \rhd, \rhd^{-1}\}$). We non-deterministically guess an accepting run of $\mathcal{B}_\pi$, simulating each move labeling this run using the tree-walking automata given by Proposition 1 and Remark 6. Notice that each such simulation of a move from D begins and ends at a leaf. Finally, when reaching a final state of $\mathcal{B}_\pi$, we start a copy of the automaton A'_σ. Checking state formulas in π adds no complication due to the power of alternation. To verify the formula $\underline{\alpha}$ we simply propagate a copy of the automaton A_α at the current node (leaf). All this can be formalized to get an A2A $\mathcal{A}_\sigma$ of size $\mathcal{O}(k \cdot \mathtt{p} \cdot |\sigma|)$. We then use Vardi's result [21] to convert this into an ordinary tree automaton of size $2^{\mathcal{O}(k^2 \cdot \mathtt{p}^2 \cdot |\sigma|^2)}$.

Since temporal logic formulas can be translated into CPDL formulas with linear increase in size, it follows that

Corollary 1. *The satisfiability problem for $\mathsf{TL}(\mathbf{Procs}, \Sigma)$ over k-split-width CBMs is decidable in* EXPTIME.

The intersection operator in IPDL adds an additional level of complexity, since the path expression $\pi_1 \cap \pi_2$ requires that the tree walking automata propagated to handle π_1 and π_2 have to end up at the same leaf. However, the technique of [10] to decide IPDL over trees can be adapted to our setting as well. As in [10] this results in an additional exponential increase in size.

Theorem 4. *From any ICPDL sentence ψ one can effectively construct a tree automaton $\mathcal{A}_\psi^k$ whose size is doubly exponential such that $L(\mathcal{A}_\psi^k) = \{t \mid t \text{ is a } k\text{-DST}, \mathcal{M}_t \models \psi\}$.*

We explain the construction of alternating 2-way tree automata (A2A) for ICPDL. We have already explained the construction for CPDL in Sec. 10. The modification required to deal with intersection is borrowed from [10] and explained below.

Recall, from Section 10, that from any state formula σ in CPDL, we construct by structural induction an A2A $\mathcal{A}_\sigma$ such that $\mathcal{A}_\sigma$ has an accepting run starting from some leaf n of a k-DST t if and only if $\mathcal{M}_t, n \models \sigma$. In this construction, to deal with any formula of the form $\langle\pi\rangle\alpha$, we construct an alternating A2A automaton with a specific structure: it consists of a tree-walking automaton which navigates a walk from a leaf n to another leaf n' (possibly going through many leaves on the way) in the given k-DST t (propagating state-formula checking automata on the way) and transfers control to the automaton for α at n'. Let us call this automaton $\mathcal{A}_\pi$. Then, $\mathcal{A}_\pi$ has an accepting run starting at a leaf n in which its tree-walking component reaches the leaf n' iff $\mathcal{M}_t, n, n' \models \pi$.

Extending this to the case where $\pi = \pi_1 \cap \pi_2$, turns out to be a direct application of the technique of [10], where an A2A construction for model-checking ICPDL over trees is described. Consider any accepting walk w of the tree-walking component of $\mathcal{A}_{\pi_1}$ from a leaf n to a leaf n'. This may be broken up as $w_0 e_1 w_1 \cdots e_m$ where $e_1 e_2 \cdots e_m$ is the (unique) shortest path from n to n' in the tree and further there is no occurrence of e_i in $w_i e_{i+1} \cdots e_m$. If w' is an accepting walk of $\mathcal{A}_{\pi_2}$ then we may write $w' = w'_0 e_1 w'_1 \cdots e_m$ similarly. The idea is to synchronize the tree-walking part of the automata $\mathcal{A}_{\pi_1}$ and $\mathcal{A}_{\pi_2}$ on the edges along the path $e_1 e_2 \cdots e_m$ while allowing them to evolve asynchronously along w_i (w'_i).

As shown in [10], it is easy with an A2A to check, given a pair of states (q, q') of $\mathcal{A}_{\pi_1}$ (resp. of $\mathcal{A}_{\pi_2}$) whether there is a looping path in the tree starting from state q and ending in state q'. The states of the walking automaton part of $\mathcal{A}_{\pi_1 \cap \pi_2}$ are pairs of states, one each form the walking automaton part of $\mathcal{A}_{\pi_1}$ and $\mathcal{A}_{\pi_2}$. At each step along the path $e_1 e_2 \cdots e_m$, this walking automaton from a state (q_1, q_2) first guesses the existence of looping paths q_1 to q'_1 and q_2 to q'_2 in $\mathcal{A}_{\pi_1}$ and $\mathcal{A}_{\pi_2}$ respectively and propogates automata to verify the same. It then synchronises a pair of transitions, q'_1 to q''_1 from $\mathcal{A}_{\pi_1}$ and q'_2 to q''_2 from $\mathcal{A}_{\pi_2}$ to take one more step along the walk $e_1 \cdots e_m$ and move to (q''_1, q''_2). The details are easy to formalise and thus the construction extends to the case where intersection is permitted as part of path formulas.

For the complexity, we have already argued in Sec. 10 that the number of states of $\mathcal{A}_\sigma$ is $\mathcal{O}(k \cdot \mathfrak{p} \cdot |\sigma|)$ when σ is a formula from CPDL. This is because a basic move along a linear edge $\rightarrow$ is simulated with $\mathcal{O}(k \cdot \mathfrak{p})$ states thanks to Prop. 1, and all constructs from CPDL only increase the size of the automata linearly. Now, intersection induces a quadratic increase in the number of states. Indeed, it requires pairs of states, either from $\mathcal{A}_{\pi_1}$ *or* from $\mathcal{A}_{\pi_2}$ to check the existence of looping paths, or from $\mathcal{A}_{\pi_1}$ *and* $\mathcal{A}_{\pi_2}$ to synchronously follow the shortest path between two leaves. Due to this, we can show that the size of the A2A $\mathcal{A}_\sigma$ constructed from a formula σ in ICPDL is $2^{\mathrm{poly}(|\sigma|)}$, and we deduce that the size of the corresponding tree automaton is doubly exponential in $|\sigma|$. We refer to [10, Section 3.3] for a more detailed study of the complexity, which can be further refined if we consider the intersection width of the ICPDL formula.

10.2 Emptiness and other Problems on CPS

We now show that the emptiness, universality and the containment problems for CPS w.r.t. CBMs with split-width bounded by k have reasonable complexity.

Lemma 5. *Given a CPS $\mathcal{S}$ over $(\mathbf{Procs}, \Sigma)$ and any integer k, one can effectively construct a tree automaton $\mathcal{A}^k_\mathcal{S}$ with $|\mathsf{Locs}|^{\mathcal{O}(\mathfrak{p}.k)}$ many states such that*

$$L(\mathcal{A}^k_\mathcal{S}) = \{t \mid t \textit{ is a } k\textit{-DST},\ \mathcal{M}_t \in \mathscr{L}(\mathcal{S})\}\,.$$

Recall, from Sec. 3, that a run is just a labeling of the events by locations via two functions (ρ, ν) satisfying 3 requirements. Our aim is to construct a bottom-up tree automaton that simulates such runs on any k-DST.

Let t be a k-DST. The events of $\mathcal{M}_t$ are the leaves of t. The bottom-up tree automaton guesses a possible labeling of the events (the leaves) and verifies that it defines a run as it walks up the DST. Actually, at each leaf e, the automaton guesses the location labels assigned to e as well as to e^-, the reason for which will be clear soon. Due to this double labeling, if e is an internal event then conformance to requirement 3 in the definition of a run can be checked immediately. Similarly, if $e \rhd f$ then, by nondeterministically guessing the identity of the value that is placed and removed from the store by transitions at these two events, the automaton can verify the conformance to requirement 1 at e and to requirement 2 at f simultaneously. This is done as it visits the parent of these two nodes (labeled $\rhd$). We are almost done, except that all this assumes the correctness of the guess about the labeling of e^- at each e. It remains to verify the correctness of these guesses.

The correctness of the guess at leaf e is verified at the unique merge node m_e in the tree that adds the $\xrightarrow{\mathsf{r}}$ (or equivalently $\rightarrow$) edge connecting e^- and e. Thus, the guessed location labels of e^- and e need to be carried in the state of the automaton till this node is reached. The key observation is that at every node in the path from e to m_e, e is the left-most event in its component and similarly, e^- is the right-most event in its component along the path from e^- to m_e. In other words, as the automaton walks up the tree, it only needs to keep the guesses for the first and last events in each component (in each process). The number of such events is bounded by $\mathfrak{p}(k+1)$, explaining the complexity stated in Lem. 5. (Actually, for the first event e in any component it suffices to keep the labels associated with e^- and for the last event e it suffices to keep the labels for e.)

It is easy to maintain this information. At a merge node, apart from checking the correctness as explained above, the unnecessary labels (e/e^- if they are not the first or last events of the merged component) are dropped and other labels are inherited. At a shuffle node, the labels for each component is simply inherited. Finally, when the automaton reaches the root, there is only one component per process. The entire run accepts if in each process the location labeling e^- of the first event is ℓ_{in} and the tuple of locations labeling the last events of each process is a final state in $\mathsf{Locs}_{\text{fin}}$. In all this we have assumed that the automaton reads a k-DST, but that can be arranged using Lemma 4, completing the proof. As an immediate application we have the following theorem.

Theorem 5. *The emptiness problem for CPS over* $(\mathbf{Procs}, \Sigma)$ *w.r.t. k-split-width CBMs is decidable in* EXPTIME. *The universality problem and inclusion problems w.r.t. k-split-width CBMs are decidable in* 2-EXPTIME.

Emptiness problem for $\mathcal{S}$ reduces to the emptiness problem for $\mathcal{A}^k_{\mathcal{S}}$. Notice that a CBM $\mathcal{M}$ of split-width at most k is accepted by $\mathcal{S}$ iff *all* k-DSTs representing $\mathcal{M}$ are accepted by $\mathcal{A}^k_{\mathcal{S}}$. Hence, the universality problem reduces to checking whether $\mathcal{A}^k_{\mathcal{S}}$ accepts *all* k-DSTs representing CBMs (as opposed to split-CBMs), which is just the equivalence problem of tree automata. Finally, the containment problem reduces to the containment problem for associated tree automata.

10.3 Model-checking

The k-split-width model-checking problem for a logic $\mathcal{L}$ determines, given a CPS and a formula φ in $\mathcal{L}$, whether some CBM of split-width at most k accepted by the CPS satisfies φ.

Theorem 6. *The k-split-width model-checking problem for MSO can be solved with non-elementary complexity. The k-split-width model-checking problem for CPDL and Temporal Logic are in* EXPTIME. *The k-split-width model-checking problem for ICPDL is in* 2-EXPTIME.

In all the cases, we use results from Sections 10.1 and 10.2, we first construct from the formula φ a tree automaton $\mathcal{A}^k_{\neg\varphi}$ that recognizes all k-DSTs representing CBMs that do not satisfy φ. The model-checking problem then reduces to the emptiness of the intersection of the tree automata $\mathcal{A}^k_{\neg\varphi}$ and $\mathcal{A}^k_{\mathcal{S}}$.

11 Discussions

Optimal complexities of the decision procedures Optimality of the reachability in CPS and of PDL satisfiability follows from the equivalence to (reduction from) VAS. The PTIME hardness on the size of the system $|\mathcal{S}|$ for the problems 10, 16-18, 10', 16'-18' follow from the PTIME hardness of the emptiness checking of nested-word automata, since nested-words have split-width bounded by 2 (cf. Remark 3). The hardness of problems 14, 18, 12'-14', 16'-18' with respect to the size of the (ϵ,C,IC)PDL formula and more specifically with respect to the bound on split-width follow from the corresponding problems in the case of bounded phase multi-pushdown systems [5] and the bound on their split-width [8].

Open problems In addition to problems 4 and 8 from Table 1 a number of interesting problems remain open (see [7]). Given an architecture $\mathfrak{A}$, a set of actions Σ and an integer k, is it possible to design a CPSQ $\mathcal{S}^{\text{univ}}_k$ over $\mathfrak{A}$ and Σ such that it accepts exactly the MSCNs whose split-width is bounded by k? If so, one can restrict any CPSQ to its (verified) k-bounded split-width behaviours by intersecting it with $\mathcal{S}^{\text{univ}}_k$. This problem is open, however see [7] for subclasses which can be easily implemented as CPSQs. These subclasses generalize many known decidable system restrictions, giving elementary decision procedures for various model-checking problems. Another open problem is whether we can construct the CPS (or CPSQ) corresponding to a regular language of k-DSTs. An affirmative answer would immediately yield a complementation procedure for CPS w.r.t. behaviours of split-width at most k.

References

1. R. Alur and P. Madhusudan. Adding nesting structure to words. *Journal of the ACM*, 56(3):1–43, 2009.

2. Mohamed Faouzi Atig, Benedikt Bollig, and Peter Habermehl. Emptiness of multi-pushdown automata is 2ETIME-Complete. In Masami Ito and Masafumi Toyama, editors, *Developments in Language Theory*, volume 5257, pages 121–133. Springer, 2008.
3. Mohamed Faouzi Atig, Ahmed Bouajjani, and Tayssir Touili. On the reachability analysis of acyclic networks of pushdown systems. In *CONCUR*, pages 356–371, 2008.
4. B. Bollig, D. Kuske, and I. Meinecke. Propositional dynamic logic for message-passing systems. *Logical Methods in Computer Science*, 6(3:16), 2010.
5. Benedikt Bollig, Aiswarya Cyriac, Paul Gastin, and Marc Zeitoun. Temporal logics for concurrent recursive programs: Satisfiability and model checking. In *MFCS*, volume 6907 of *Lecture Notes in Computer Science*, pages 132–144. Springer, 2011.
6. Luca Breveglieri, Alessandra Cherubini, Claudio Citrini, and Stefano Crespi-Reghizzi. Multi-pushdown languages and grammars. *Int. J. Found. Comput. Sci.*, 7(3):253–292, 1996.
7. Aiswarya Cyriac. *Verification of Communicating Recursive Programs via Split-width.* PhD thesis, ENS Cachan, 2014.
8. Aiswarya Cyriac, Paul Gastin, and K. Narayan Kumar. MSO decidability of multi-pushdown systems via split-width. In *CONCUR*, volume 7454 of *Lecture Notes in Computer Science*, pages 547–561. Springer, 2012.
9. M.J. Fischer and R.E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979.
10. Stefan Göller, Markus Lohrey, and Carsten Lutz. Pdl with intersection and converse: satisfiability and infinite-state model checking. *J. Symb. Log.*, 74(1):279–314, 2009.
11. Jesper G. Henriksen and P. S. Thiagarajan. Dynamic linear time temporal logic. *Ann. Pure Appl. Logic*, 96(1-3):187–207, 1999.
12. Alexander Heußner, Jérôme Leroux, Anca Muscholl, and Grégoire Sutre. Reachability analysis of communicating pushdown systems. In C.-H. Luke Ong, editor, *FOSSACS*, volume 6014, pages 267–281. Springer, 2010.
13. O. Inverso, S. La Torre, E. Tomasco, and G. Parlato. Looking at computations from a different angle. Technical report, Univ. of Southampton, 2013.
14. ITU-TS. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC).* ITU-TS, Geneva, February 2011.
15. Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. A robust class of context-sensitive languages. In *LICS*, pages 161–170. IEEE Computer Society, 2007.
16. Salvatore La Torre and Margherita Napoli. Reachability of multistack pushdown systems with scope-bounded matching relations. In Joost-Pieter Katoen and Barbara König, editors, *CONCUR*, volume 6901, pages 203–218. Springer, 2011.
17. Jérôme Leroux. Vector addition system reachability problem: a short self-contained proof. In *POPL*, pages 307–316, 2011.
18. P. Madhusudan and Gennaro Parlato. The tree width of auxiliary storage. In Thomas Ball and Mooly Sagiv, editors, *POPL*, pages 283–294. ACM, 2011.
19. Ernst W. Mayr. An algorithm for the general petri net reachability problem. In *STOC*, pages 238–246, 1981.
20. Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In Nicolas Halbwachs and Lenore D. Zuck, editors, *TACAS*, volume 3440, pages 93–107. Springer, 2005.
21. M. Y. Vardi. The taming of converse: Reasoning about two-way computations. In *Proc. of the Conference on Logic of Programs*, pages 413–423. Springer, 1985.