# Verifying Communicating Multi-pushdown Systems via Split-width*

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.** Communicating multi-pushdown systems model networks of multi-threaded recursive programs communicating via reliable FIFO channels. We extend the notion of split-width [8] to this setting, improving and simplifying the earlier definition. Split-width, while having the same power of clique-/tree-width, gives a divide-and-conquer technique to prove the bound of a class, thanks to the two basic operations, shuffle and merge, of the split-width algebra. We illustrate this technique on examples. We also obtain simple, uniform and optimal decision procedures for various verification problems parametrised by split-width.

## 1 Introduction

This paper is about the formal verification of multi-threaded recursive programs communicating via reliable FIFO channels. This is an important but highly challenging problem. 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 generic under-approximation class, and give uniform decision procedures for a variety of verification problems including reachability and model-checking against logical specifications.

We model the system as a collection of finite state machines, equipped with unbounded stack and queue data-structures. Thus, we get a faithful modelling of programs using such data-structures. They can also be used to model implicit features in a distributed setting, e.g., stack models recursion and queues model communication channels. Such systems are called stack-queue distributed system (SQDS) in this paper. The behaviour of an SQDS, called a stack-queue MSC (SQMSC), is a tuple of sequences of events (one per program/process). In addition a binary matching relation links corresponding writes (push/send) and reads (pop/receive). These were called stack-queue graphs in [20], run graphs

* This work is partially supported by LIA InForMel.

in [15] and they jointly generalise nested words [1], multiply nested words [17] and Message Sequence Charts (MSC) [16]. An example is given is Fig. 1.

These systems are Turing powerful, and hence their verification is undecidable. Several under-approximations [11,13,10,21,17,19,2,5] have been studied in the literature. SQMSCs form a class of graphs, and hence bounds on clique-/tree-width can also be used as an under-approximation, as they give decidability for model-checking against powerful MSO logic. Since SQMSCs have bounded degree, it follows from Courcelle's result that a bound on tree-width is necessary for obtaining decidability against MSO model-checking. In fact, a bound on tree-width is established in [20] for many of the known decidable classes. Thus [20] gives the first unified proof of decidability of reachability of these classes. In [15], a bound on the tree-width of the run graphs of restricted communicating pushdown systems is shown via hyper-edge replacement grammars.

We propose another measure called split-width, which is specific to SQMSCs (as opposed to generic graphs) and hence simpler. It is based on a divide-and-conquer decomposition mechanism[4] (or dually an algebra[5]) for SQMSCs, and provide a natural tree-embedding of SQMSCs. This way, every verification problem can be stated equivalently over trees, and hence can be solved efficiently.

Furthermore, split-width is as powerful as tree-width (see [7]), and the respective bounds lie in a linear factor of each other. Thus bounding split-width can also be seen as a way to bound tree-width, which is often a difficult task. A bound on split-width has been established for many known decidable classes in [8,7]. The systematic way of bounding the split-width helped in generalising these classes and in discovering new decidable classes.

As said before, split-width is a measure based on decomposing an SQMSC into atomic pieces. The atomic pieces are single events and edges linking writes and reads. The idea is to decompose an SQMSC by the repeated application of two operations: split and divide. *Split* chops the edge between neighbouring events and *divide* separates such split-SQMSCs into independent parts. We may need several splits before it can be divided, and the maximum number of such splits on a decomposition is its width. The split-width is the width of an optimal decomposition – one that minimises the maximum number of splits.

The above decomposition procedure can be abstracted as a term in an algebra. This gives a natural embedding of SQMSCs into trees, similar to how parse trees give a tree-representation for a word in a context-free language. The valid tree-embeddings of SQMSC with split-width at most $k$ form a regular tree-language. Thus we can translate every problem (see below) on SQDS/SQMSC to an equivalent one on tree-domains.

We consider several verification problems starting from reachability. We use Monadic Second Order logic (MSO), Propositional Dynamic Logic with and

[4] $k$-decompositions are a divide and conquer technique for bounding tree-width. The role played by edges in split-width is played by vertices there. Further, split-width decompositions are duplication free allowing us to reason about SQMSCs easily.

[5] Split-width algebra is in some sense a restriction of special tree-width algebra [6] resulting in decomposition trees where the matching edges occur at the lowest level.

without intersection (IPDL/PDL) and Temporal Logics (TL) as specification languages. Satisfiability checking, and model-checking of SQDS against specifications given in these formalisms are also addressed.

With split-width as a parameter, we get *uniform* decision procedures with *optimal complexities* for *all* verification problems on SQDS/SQMSC. The complexities range from non-elementary for MSO to 2EXPTIME for IPDL to EXPTIME for PDL/TL and reachability. However, the complexity is only *polynomial* in the number of states of the SQDS. Thus, if the bound on split-width is fixed a priori, then reachability is in PTIME.

Split-width was originally introduced in [8] as a technique to prove decidability of MSO model checking of multi-pushdown systems. In this paper, we generalise this notion to more complex behaviours involving multiple processes with several local stacks and queues and multiple channels between the processes for communication. In another dimension, we address more verification problems, for example model checking an SQDS against PDL. Finally, we describe a uniform approach to address these variety of problems simplifying the original proofs and constructions in [8].

## 2 Preliminaries: Systems and Behaviours

In this section we describe our formal model called *Stack-Queue Distributed Systems* (SQDS). Such a system consists of a finite collection of *processes* each of which having finitely many control locations. Further the collection has access to a set of stacks and queues. Each data-structure (stack/queue) is written to by a unique process (called its *writer*) and read from by a unique process (called its *reader*). For stacks we additionally require that the reader and writer are the same process. Such a system is a formal model of distributed multi-threaded programs communicating via reliable FIFO channels. The call-stack of one thread is modelled with a stack, while the reliable FIFO channels connecting the processes are modelled with queues. In addition, processes may use stacks and queues as local data structures, e.g. in task schedulers, in resource request managers etc. They can also arise when a process represents the global behaviour of a collection of distributed processes along with the channels interconnecting them.

**An architecture** $\mathfrak{A}$ is a tuple ($\mathbf{Procs}, \mathbf{Stacks}, \mathbf{Queues}, \mathsf{Writer}, \mathsf{Reader}$) consisting of a fintie set **Procs** of processes, a finite set of **Stacks**, a finite set of **Queues** and functions $\mathsf{Writer}$ and $\mathsf{Reader}$ 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 **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 $\Sigma$ is a tuple $\mathcal{S} = (\mathsf{Locs}, \mathrm{Val}, (\mathrm{Trans}_p)_{p \in \mathbf{Procs}}, \ell_{\mathrm{in}}, \mathsf{Fin})$ where $\mathsf{Locs}$ is a finite set of locations, Val is a finite set of values that can be stored in the data-structures, $\ell_{\text{in}} \in \mathsf{Locs}$ is the initial location, $\mathsf{Fin} \subseteq \mathsf{Locs}^{\mathbf{Procs}}$ is the set of global final locations, and $\text{Trans}_p$ is the set of transitions of process $p$. $\text{Trans}_p$ may have write (resp. read) transitions on data-structure $d$ only if $\mathsf{Writer}(d) = p$ (resp. $\mathsf{Reader}(p) = d$). For $\ell, \ell' \in \mathsf{Locs}$, $a \in \Sigma$, $d \in \mathbf{DS}$ and $v \in \text{Val}$, $\text{Trans}_p$ has

– internal transitions of the form $\ell \xrightarrow{a} \ell'$,
– write transitions of the form $\ell \xrightarrow{a,d!v} \ell'$ with $\mathsf{Writer}(d) = p$, and
– read transitions of the form $\ell \xrightarrow{a,d?v} \ell'$ with $\mathsf{Reader}(d) = p$.

For an SQDS $\mathcal{S}$, by $\mathsf{sizeof}(\mathcal{S})$ we denote its number of states ($|\mathsf{Locs}|$).

Intuitively, an SQDS consists of a collection of finite state automata equipped with a collection of stacks and queues. In each step, a process uses an internal transition to simply change its state, or uses a write transition to append a value to the tail of a particular queue or stack, or uses a read transition to remove a value from the head (or tail) of a queue (resp. of a stack). The transition relation makes explicit the identity of the data-structure being accessed and the type of the operation. As observed in [1,17,8,20] it is often convenient to describe the semantics of such systems as a labeling of words decorated with a matching relation per data-structure instead of using configurations and moves.

This is also consistent with the usual semantics of distributed systems given as labelings of appropriate partial orders [23,13,11]. We now describe formally the structures that represent behaviours of such SQDS.

**A stack-queue MSC (SQMSC)** over architecture $\mathfrak{A}$ and alphabet $\Sigma$ is a tuple $\mathcal{M} = ((\mathcal{E}_p)_{p\in\mathbf{Procs}}, \rightarrow, \lambda, (\rhd^d)_{d\in\mathbf{DS}})$ where $\mathcal{E}_p$ is the finite set of events on process $p$, $\rightarrow$ relates only events within a process, i.e., $\rightarrow \,=\, \uplus \rightarrow_p$ where $\rightarrow_p \,=\, \rightarrow \cap\, (\mathcal{E}_p \times \mathcal{E}_p)$, $\rightarrow_p$ is the covering relation of a linear order on $\mathcal{E}_p$, $\rhd^d$ is the relation matching write events on data-structure $d$ with their corresponding read events and $\lambda(e) \in \Sigma$ is the letter labeling event $e$. We set $\mathsf{pid}(e) = p$ if $e \in \mathcal{E}_p$, and $\mathcal{E} = \uplus_p \mathcal{E}_p$. We also let $\rhd \,=\, \bigcup_{d\in\mathbf{DS}} \rhd^d$ be the set of all matching edges. We require the relation $< \,=\, (\rightarrow \cup \rhd)^+$ to be a strict partial order on the set of events. Finally, the matching relations should comply with the architecture: $\rhd^d \subseteq \mathcal{E}_{\mathsf{Writer}(d)} \times \mathcal{E}_{\mathsf{Reader}(d)}$.

– 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 denote by $\mathbb{SQMSC}(\mathfrak{A}, \Sigma)$ the set of SQMSCs over $\mathfrak{A}$ and $\Sigma$. An SQMSC over an architecture with one process and one stack is a nested word [1]. An SQMSC over an architecture with no stacks and at most one queue between every pair of processes is a Message Sequence Chart [16]. An SQMSC is depicted in Figure 1.

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 SQDS over an SQMSC $\mathcal{M}$, we introduce two notations. For $p \in \mathbf{Procs}$ and $e \in \mathcal{E}_p$, we denote by $e^-$ the unique event such

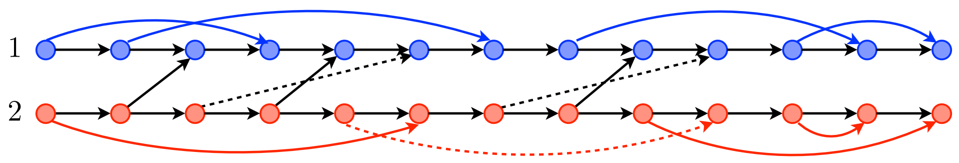


Fig. 1: An SQMSC over 2 processes, 3 queues and 2 stacks. Two queues form channels from Proc. 2 to Proc. 1, while the 3rd queue is a self-queue on Proc. 1.

that $e^- \rightarrow e$ if it exists, and we let $e^- = \bot_p \notin \mathcal{E}$ otherwise. We let $\max_p(\mathcal{M})$ be the maximal event of $\mathcal{E}_p$ if it exists and $\max_p(\mathcal{M}) = \bot_p$ otherwise.

**A run of an SQDS** $\mathcal{S}$ over an SQMSC $\mathcal{M}$ is a mapping $\rho \colon \mathcal{E} \rightarrow \mathsf{Locs}$ satisfying the following consistency conditions (with $\rho(\bot_p) = \ell_{\text{in}}$):

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

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

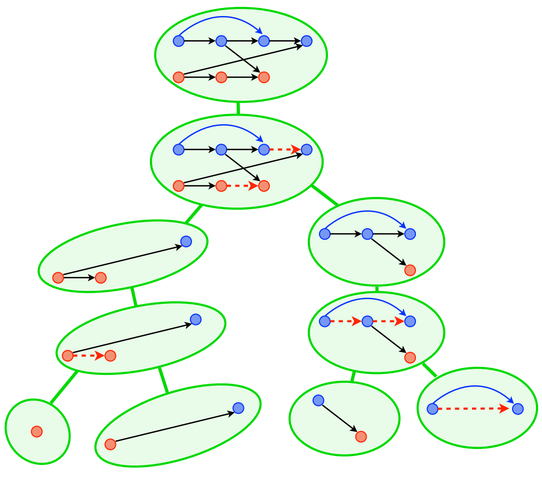
The reachability problem asks, given an SQDS $\mathcal{S}$, whether some global final locations from $\mathsf{Fin}$ is reachable in $\mathcal{S}$? This is equivalent to the language emptiness problem for SQDS and is undecidable. We get decidability (cf. Theorem 9) for the bounded split-width reachability problem: given a parameter $k$, does $\mathscr{L}(\mathcal{S})$ contain at least one SQMSC with split-width at most $k$? The complexity of our decision procedure is only polynomial in $\mathsf{sizeof}(\mathcal{S})$ though exponential in $k$ and $|\mathfrak{A}|$. We also obtain optimal decision procedures (cf. Theorem 16) for the bounded split-width model-checking problems wrt. MSO (non-elementary) and PDL (polynomial in $\mathsf{sizeof}(\mathcal{S})$, and exponential in $k$, $|\mathfrak{A}|$ and the formula).

## 3 The Split-Width Algebra for SQMSCs

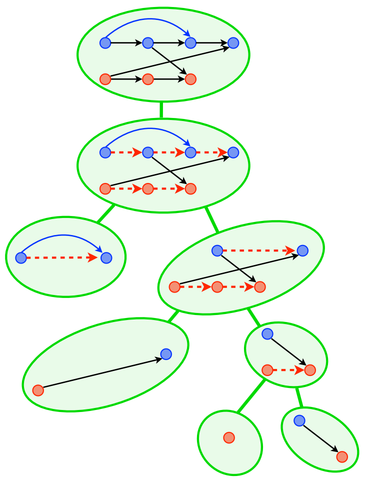
The idea is to decompose each SQMSC into *atomic pieces*. We begin by removing some of the $\rightarrow$ edges to create holes which we call *elastic edges*. This operation is called *split*. We call an SQMSC with elastic edges a *split-SQMSC*.

After removing some $\rightarrow$ edges it is possible that the entire split-SQMSC consists of two disjoint parts with only elastic edges connecting them. At this point we may break-up this split-SQMSC into these two parts and then continue decomposing them separately. This operation is called *divide*. Our aim is to use split and divide repeatedly until we are left with the atomic parts, which are either internal events or an edge of the form $e \rhd^d f$. Figure 2a describes this decomposition on an SQMSC where the elastic edges are dashed ($\dashrightarrow$).

For any such complete break up of an SQMSC, its width is the maximum number of elastic edges of all the split-SQMSCs produced. The break-up described in Figure 2a has width 2. There may be several ways of starting with



(a) – of width 2.



(b) – of width 5.

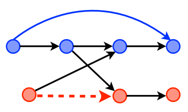
Fig. 2: Two decompositions of an SQMSC (consecutive splits are contracted)

an SQMSC and breaking it down into its atomic components. A different and somewhat more trivial decomposition with width 5 is described in Figure 2b.

The *split-width* of an SQMSC is the minimum width among all possible ways of breaking it up into its atomic parts using split and divide. A class $C$ of SQMSCs has split-width $k$ if each of its members has split-width at most $k$.

We begin by formalizing the idea of SQMSCs with holes. As explained above, it is done by identifying a subset of the $\rightarrow$ edges as *elastic edges* standing for holes which may be filled later (or dually for a part that has been removed). Then we describe the split-width algebra.

**A split-SQMSC** is an SQMSC in which $\rightarrow$ edges are partitioned into *rigid* edges (denoted $\xrightarrow{\text{r}}$) and *elastic* edges (denoted $\xrightarrow{\text{e}}$). It is a pair $\mathcal{M\!\!M} = (\mathcal{M}, \xrightarrow{\text{e}})$ where $\mathcal{M} = ((\mathcal{E}_p)_{p\in\mathbf{Procs}}, \rightarrow, \lambda, (\rhd^d)_{d\in\mathbf{DS}})$ is an SQMSC and $\xrightarrow{\text{e}} \subseteq \rightarrow$. We let $\xrightarrow{\text{r}} = \rightarrow \setminus \xrightarrow{\text{e}}$ be the rigid edges of $\mathcal{M\!\!M}$.



The *elasticity* of a split-SQMSC is the number of elastic edges it has. A emphcomponent is a maximal connected component of the graph when restricted to only rigid edges ($\xrightarrow{\text{r}}$). For instance, the split-SQMSC on right has elasticity one, and three components.

**The split-width Algebra** over $\mathfrak{A}$ and $\Sigma$ is given by the following syntax:

$$s ::= (a, p) \mid (a, \mathsf{Writer}(d)) \rhd^d (a', \mathsf{Reader}(d)) \mid \mathsf{merge}(s) \mid s \sqcup s$$

where $a, a' \in \Sigma$, $p \in \mathbf{Procs}$ and $d \in \mathbf{DS}$. We use *split-terms* to refer to the terms of this algebra. Note that, the binary operators $\rhd^d$ may only be applied to atomic terms. The operator $\mathsf{merge}$ denotes the dual of the split operation while $\sqcup$ represents the dual of the divide operation. The terms of this algebra represent sets of split-SQMSCs (rather than SQMSCs) as there are many ways of shuffling together two split-SQMSCs (and also many choices for converting some elastic edge into a rigid one).

- $[\![(a,p)]\!]$ is the SQMSC consisting of a single event labeled $a$ on process $p$.
- $[\![(a,p) \rhd^d (a',p')]\!]$ is the split-SQMSC 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: $e \rhd^d e'$. Moreover, if $p = p'$, these two events are also linked by an elastic edge: $e \xrightarrow{e} e'$.

The $\mathsf{merge}$ operator when applied to a split-SQMSC $\not{\mathcal{M}}$ returns the set of split-SQMSCs obtained by replacing one elastic edge by a rigid edge and is extended naturally to sets of split-SQMSCs. The $\sqcup$ operator applied to two split-SQMSCs $\not{\mathcal{M}}_1$ and $\not{\mathcal{M}}_2$ returns the set of split-SQMSCs that can be divided into $\not{\mathcal{M}}_1$ and $\not{\mathcal{M}}_2$ and is once again extended naturally to an operation on sets.

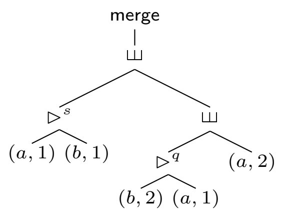
Formally, $[\![\mathsf{merge}((\mathcal{M}, \xrightarrow{e}))]\!]$ contains any split-SQMSC $(\mathcal{M}, \xrightarrow{e}{}')$ such that $\xrightarrow{e}{}' \subseteq \xrightarrow{e}$ and $|\xrightarrow{e}{}'| = |\xrightarrow{e}| - 1$. The number of components and the elasticity decrease by 1 as the result of a merge. Further, $|\mathsf{merge}(\not{\mathcal{M}})| = \mathsf{elasticity}(\not{\mathcal{M}})$.

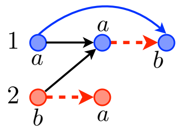
Let $\not{\mathcal{M}}_i = (\mathcal{M}_i, \xrightarrow{e}_i)$ with $\mathcal{M}_i = ((\mathcal{E}^i_p)_{p\in\mathbf{Procs}}, \rightarrow_i, \lambda_i, (\rhd^d_i)_{d\in\mathbf{DS}})$ for $i \in \{1,2\}$ be two split-SQMSCs with disjoint sets of events. Then, $\not{\mathcal{M}}_1 \sqcup \not{\mathcal{M}}_2$ is the set of split-SQMSCs $\not{\mathcal{M}} = (\mathcal{M}, \xrightarrow{e})$ with $\mathcal{M} = ((\mathcal{E}_p)_{p\in\mathbf{Procs}}, \rightarrow, \lambda, (\rhd)_{d\in\mathbf{DS}})$ such that

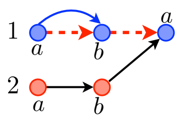
- apart from the elastic edges, $\not{\mathcal{M}}$ is the disjoint union of $\not{\mathcal{M}}_1$ and $\not{\mathcal{M}}_2$, i.e., $\mathcal{E}_p = \mathcal{E}^1_p \uplus \mathcal{E}^2_p$, $\lambda = \lambda_1 \uplus \lambda_2$, $\mathsf{pid} = \mathsf{pid}_1 \uplus \mathsf{pid}_2$, $\rhd^d = \rhd^d_1 \uplus \rhd^d_2$ and $\xrightarrow{r} = \xrightarrow{r}_1 \uplus \xrightarrow{r}_2$,
- for each $i$, the order of the components of $\not{\mathcal{M}}_i$, as prescribed by the elastic edges $\xrightarrow{e}_i$, is preserved in $\not{\mathcal{M}}$: $\xrightarrow{e}_1 \cup \xrightarrow{e}_2 \subseteq \rightarrow^*$.

Note that, the number of components of any split-SQMSC $\not{\mathcal{M}} \in (\not{\mathcal{M}}_1 \sqcup \not{\mathcal{M}}_2)$ is the sum of the number of components in $\not{\mathcal{M}}_1$ and $\not{\mathcal{M}}_2$.

*Example 1.* The tree on right depicts the split-term $s = \mathsf{merge}(((a,1) \rhd^s (b,1)) \sqcup (((b,2) \rhd^q (a,1)) \sqcup (a,2)))$. $[\![s]\!]$ has 18 split-SQMSCs , of which two are shown below:



$\not{\mathcal{M}}_1 =$ 

$\not{\mathcal{M}}_2 =$ 

We can easily check that all the split-SQMSC in the semantics $[\![s]\!]$ of any 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 $\mathsf{elasticity}(s)$.

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 $s$ of Ex. 1 is two and its width is three. The *split-width* of a split-SQMSC $\not{\mathcal{M}}$, denoted $\mathsf{swd}(\not{\mathcal{M}})$, is the minimum width of all split-terms $s$ such that $\not{\mathcal{M}} \in [\![s]\!]$. For instance, $\not{\mathcal{M}}_1$ and $\not{\mathcal{M}}_2$ of Ex. 1 have split-width two since they are respectively in the semantics of

$$s_1 = (\mathsf{merge}(((a,1) \rhd^s (b,1)) \sqcup ((b,2) \rhd^q (a,1)))) \sqcup (a,2)$$
$$s_2 = ((a,1) \rhd^s (b,1)) \sqcup \mathsf{merge}(((b,2) \rhd^q (a,1)) \sqcup (a,2)).$$

*Remark 2.* The split-width algebra over $\mathfrak{A}$ can generate any SQMSC $\mathcal{M} = ((\mathcal{E}_p)_{p\in\mathbf{Procs}}, \rightarrow, \lambda, (\rhd)_{d\in\mathbf{DS}})$ over $\mathfrak{A}$. In fact a sequence of shuffles of basic split-terms will generate a split-SQMSC $\mathcal{A\!\!\!M}_1 = (\mathcal{M}, \xrightarrow{e}_1 = \rightarrow)$. This will then be followed by a sequence of merges to get the split-SQMSC $\mathcal{A\!\!\!M}_2 = (\mathcal{M}, \xrightarrow{e}_2 = \emptyset)$.

Some classes of bounded split-width are given below. (See [8,7] for more involved classes).

**Nested words** have split-width bounded by 2. This follows 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 $\begin{tikzcd}a \arrow[r] \arrow[r, bend left] & b\end{tikzcd}$.

**Message sequence graphs** are graphs in which the edges are labelled by MSCs from a finite set $\Gamma$. A path in an MSG generates an MSC $\mathcal{M} \in \Gamma^*$ by concatentation of the edge labels along the path. It is easy to see that all MSCs in $\Gamma^*$ have split-width bounded by $|\mathbf{Procs}| + m$ where $m$ the maximum size of an MSC from $\Gamma$. Indeed, let $\mathcal{M} = \mathcal{M}_1 \cdot \mathcal{M}_2 \cdot \ldots \cdot \mathcal{M}_n = \mathcal{M}' \cdot \mathcal{M}_n \in \Gamma^+$. We decompose this MSC recursively as follows: If $\mathcal{M}'$ is nonempty, then remove the $\rightarrow$ edges added at the last concatenation by splits, and then divide into $\mathcal{M}'$ and $\mathcal{M}_n$; recursively decompose each of them. If $\mathcal{M} \in \Gamma$, then it can be decomposed naively, with split-width bounded by its size.

**Proposition 3.** *Let $G$ be an MSG over $\Gamma$ and let $m = \max\{|M| : M \in \Gamma\}$. Then the split-width of any MSC generated by $G$ is bounded by $|\mathbf{Procs}| + m$.*

In fact, MSCs generated from a finite set $\Gamma$ are existentially bounded. This larger class also has bounded split-width as demonstrated below.

**Existentially $k$-bounded MSCs** are those in which the events can be linearly ordered (extending the partial order $\leq$) such that the number of unmatched writes at any point is bounded by $k$. Let this linear order be denoted $\preceq$. We decompose an existentially $k$ bounded MSC as follows. We split the $\rightarrow$ edge originating from the first $k+1$ events in the $\preceq$ order. Let us call these events *detached* events. Among the detached events, we divide 1) all read events along with their partners (since the partner write event precedes the read, it is also detached), and 2) all internal events, as atomic pieces from the rest of the split-MSC. At this point, all detached events are writes. Since there cannot be more than $k$ unmatched writes at any point, the number of detached events is strictly less than $k+1$. Hence, we proceed by splitting more $\rightarrow$ edges in the $\preceq$ order until there are $k+1$ detached events. Then we follow with the divide operation as before, and repeat until the whole MSC is decomposed. Thus,

**Proposition 4.** *Existentially $k$-bounded MSCs have split-width at most $k+1$.*

## 4 Split-SQMSC to Trees

One key interest in defining the split-width algebra is that the class of $k$ split-width SQMSCs can be seen as a regular tree language and that model-checking problems of this class can be reduced to problems on tree automata. As a first step we show how to encode SQMSC of bounded split-width as binary trees.

As it stands, each split-term $s$ defines a set of split-SQMSCs $[\![s]\!]$. In order to reason about each split-SQMSC we decorate split-terms with additional labels so that each such labeled term denotes a unique split-SQMSC. The reason $[\![s]\!]$ is a set is that the operations $\mathsf{merge}$ 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 $\ell$ to indicate it comes from the left operand and $r$ to indicate it comes from the right. For instance, the figure on the right dismabiguates the split-term $s$ of Ex. 1 to represent $\mathcal{M}_1$.

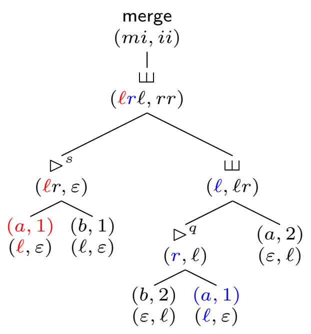


Fig. 3: A 3-DST

Consider the set of labels $L_k = (\{i, m\}^{\leq k+1})^{\mathbf{Procs}} \cup (\{\ell, r\}^{\leq k+1})^{\mathbf{Procs}}$. 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 $L_k$, and satisfying some validity conditions. Hence, each node $n$ of $t$ corresponds to a subterm $s_n$ of $s$, and we denote its labeling by $(W_p(n))_{p\in\mathbf{Procs}}$. We define simultaneously, the validity condition for the labeling at each node $n$, and the split-SQMSC $\mathcal{M}_n$ identified by this labeling.

1. If $n$ is a leaf with associated split-term $s_n = (a, p)$, then $\mathcal{M}_n$ is the unique split-SQMSC in $[\![s_n]\!]$. We set $W_p(n) = \ell$ as a convention and $W_{p'}(n) = \varepsilon$ for all $p' \neq p$.
2. If $n$ is a $\rhd^d$ node, then its children must be leaves $n'$ and $n''$ and $s_n = s_{n'} \rhd^d s_{n''}$. Again, $\mathcal{M}_n$ is the unique split-SQMSC in $[\![s_n]\!]$. Let $p'$ and $p''$ be the processes of the children. We let $W_p(n) = \varepsilon$ for all $p \notin \{p', p''\}$. If $p' = p''$ then we let $W_{p'}(n) = \ell r$, and otherwise we let $W_{p'}(n) = \ell$ and $W_{p''}(n) = r$.
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-SQMSC 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 $s_n = s_{n'} \sqcup\!\sqcup s_{n''}$. Then, for each process $p \in \mathbf{Procs}$, we have $W_p(n) \in \{\ell, r\}^{\leq k+1}$ and $\#_\ell(W_p(n)) = |W_p(n')|$ and $\#_r(W_p(n)) = |W_p(n'')|$. Moreover, $\mathcal{M}_n$ is the unique split-SQMSC (if it exists) 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. However, the above validity conditions are not sufficient, as the LIFO/FIFO policies on the data-structure may be violated at a shuffle, and hence its semantics could be empty. But we can modify this bottom-up automaton to check whether the shuffles respect the LIFO/FIFO policies. To this end, the automaton keeps a subset of $\mathbf{DS} \times \{1, \ldots, k\} \times \{1, \ldots, k\}$ in its state. If a tuple $(d, i, j)$ is in the state labelling a node $n$, then it means that there is some $\rhd^d$ edge from the $i$th component of $\mathsf{Writer}(d)$ to the $j$th component of $\mathsf{Reader}(d)$ in $[\![s_n]\!]$. This information can be consistently updated at every transition, and used to forbid invalid shuffles. Thus,

**Proposition 5.** *The set of $k$-DSTs is a regular tree language recognized by a tree automaton with $2^{\mathcal{O}(k|\mathfrak{A}|)}$ many states.*

We write $\mathcal{M}_t$ for the split-SQMSC described by the root of some DST $t$. When a split-term $s$ has width $k$, it is not difficult to see that, any split-SQMSC $\mathcal{M} \in [\![s]\!]$ 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 $n \rhd^d n'$ in $\mathcal{M}_t$ iff there is a $\rhd^d$ 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. 3.

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 6.**
- *There is a deterministic bottom-up tree automaton with at most $3k|\mathbf{Procs}| + 2$ 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-SQMSC $\mathcal{M}_t$.*
- *There is a deterministic tree-walking automaton with at most $2k|\mathbf{Procs}|$ 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-SQMSC $\mathcal{M}_t$.*

*Remark 7.* It is also possible to restrict Prop. 5 and Prop. 6 to $k$-DSTs that identify SQMSCs (as opposed to split-SQMSCs). An analogue of Prop. 6 can also be established for the relation $\xrightarrow{\mathrm{r}}^{-1}$ as well as $\xrightarrow{\mathrm{e}}$ and $\xrightarrow{\mathrm{e}}^{-1}$.

**Reachability and other problems on SQDS** We now use the above results to show decidability of reachability of SQDS parametrized by a bound on split-width. Our decision procedure is only polynomial in the number of states of the SQDS, while it is exponential in the number of processes, number of data-structures and split-width.

**Proposition 8.** *Given an SQDS $\mathcal{S}$ over $(\mathfrak{A}, \Sigma)$ and any integer $k > 0$, one can effectively construct a tree automaton $\mathcal{A}^k_{\mathcal{S}}$ with $|\mathsf{Locs}|^{\mathcal{O}(k|\mathfrak{A}|)}$ many states such that $L(\mathcal{A}^k_{\mathcal{S}}) = \{t \mid t \text{ is a } k\text{-DST},\ \mathcal{M}_t \in \mathscr{L}(\mathcal{S})\}$.*

Recall, from Sec. 2, that a run of an SQDS is just a labeling $\rho$ of the events by locations. 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 $e^-$. Due to this double labeling, if $e$ is an internal event then consistency can be checked immediately. Similarly, if $e \rhd^d f$ then, by nondeterministically guessing the value that is placed and removed from the data-structure by transitions at these two events, the automaton checks for consistency. This is done as it visits the parent of these two nodes (labeled $\rhd^d$). It remains to verify the correctness of the guess about the labeling of $e^-$ at each $e$.

The correctness of the guess at leaf $e$ is verified at the unique merge node $m_e$ in the tree that adds the $\xrightarrow{r}$ (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 $|\mathbf{Procs}|(k+1)$, explaining the complexity stated in Prop. 8.

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 are 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 $\ell_{\text{in}}$ and the tuple of locations labeling the last events of each process is a final state in $\mathsf{Fin}$. In all this we have assumed that the automaton reads a $k$-DST, but that can be arranged using Prop. 5, completing the proof. As an immediate application we have the following theorem.

**Theorem 9.** *The bounded split-width reachability problem for SQDS over $(\mathfrak{A}, \Sigma)$ is* EXPTIME*-complete. The complexity is, however, only polynomial in the size of the SQDS.*

Reachability problem for $\mathcal{S}$ reduces to the emptiness problem for $\mathcal{A}^k_{\mathcal{S}}$. Notice that an SQMSC $\mathcal{M}$ 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 SQMSCs, which is just the equivalence problem of tree automata. Finally, the containment problem reduces to the containment problem for associated tree automata.

**Corollary 10.** *The universality problem for SQDS and inclusion problem for SQDSs wrt. k-split-width SQMSCs are decidable in* 2-Exptime.

## 5 Further results

We now describe logical formalisms for specifying properties of SQMSCs and we give optimal decision procedures for the satisfiability problem of these logics as well as for model-checking SQDSs against specifications in these logics, when the problems are parametrised by split-width.

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

$$\varphi ::= a(x) \mid p(x) \mid x \leq y \mid x \in X \mid x \rightarrow y \mid x \rhd^d y \mid \varphi \vee \varphi \mid \neg\varphi \mid \exists x\, \varphi \mid \exists X\, \varphi$$

Every sentence in MSO defines a language of SQMSCs consisting of all those that satisfy that sentence. Note that, for an SQDS $\mathcal{S}$ over $\mathfrak{A}$ and $\Sigma$, $\mathscr{L}(\mathcal{S})$ can be described in $\mathsf{MSO}(\mathfrak{A}, \Sigma)$. While the satisfiability and model-checking are undecidable for MSO, it becomes decidable when parametrised by split-width.

**Theorem 11.** *From any* $\mathsf{MSO}(\mathfrak{A}, \Sigma)$ *sentence* $\psi$ *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 Prop. 5 and Remark 7 we may assume that the input is a $k$-DST representing an SQMSC. 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 $X$. $x \rhd^d y$ just requires us to verify that the leaves labeled $x$ and $y$ have a parent labeled $\rhd^d$. Finally, $x \rightarrow y$ is handled using Prop. 6.

As always, the combination of projection and complementation means that the size of the constructed automaton grows as a non-elementary function wrt. the size of the formula. Now, we examine other logics having decision procedures with reasonable complexity.

**Propositional Dynamic Logic (PDL)** is a well-studied logical formalism for describing properties of programs. As in a modal logic, formulas in PDL assert properties of nodes in a graph, in our case events in an SQMSC. 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 [14,3,4]. PDL and its extensions with *converse* and *intersection* are studied in this sense here. The syntax of state formulas ($\sigma$) and path formulas ($\pi$) of $\mathsf{ICPDL}(\mathfrak{A}, \Sigma)$ are given by

$$\sigma ::= \top \mid p \mid a \mid \sigma \vee \sigma \mid \neg\sigma \mid \langle\pi\rangle\sigma$$

$$\pi ::= \underline{\sigma} \mid \rightarrow \mid \rhd^d \mid \rightarrow^{-1} \mid (\rhd^d)^{-1} \mid \pi + \pi \mid \pi \cap \pi \mid \pi \cdot \pi \mid \pi^*$$

where $p \in \mathbf{Procs}$, $d \in \mathbf{DS}$ and $a \in \Sigma$. If backward edges $\rightarrow^{-1}$ and $(\rhd^d)^{-1}$ 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). In simple PDL neither backwards edges nor intersection are allowed.

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

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

The formula $\langle\pi_1 \cap \pi_2\rangle\alpha$ at an event $e$ asserts the existence of an $e'$ satisfying $\alpha$, a path from $e$ to $e'$ conforming to $\pi_1$ and a path from $e$ to $e'$ conforming to $\pi_2$. For instance, the formula $\langle\rhd^d \cap (\rightarrow^* \cdot \underline{b} \cdot \rightarrow^*)\rangle\top$ holds at event $e$ only if there is an $e'$ with $e \rhd^d e'$ and $b$ holds somewhere between $e$ and $e'$, all 3 events being on the same process.

Observe that PDL formulas have implicit free variables. To define languages of SQMSC with PDL we introduce sentences with the following syntax: $\phi = \top \mid \mathsf{E}\,\sigma \mid \phi \vee \phi \mid \neg\phi$ where $\sigma$ is an $\mathsf{ICPDL}(\mathfrak{A}, \Sigma)$ state formula. The sentence $\mathsf{E}\,\sigma$ is true on SQMSC $\mathcal{M}$ if $\mathcal{M}, e \models \sigma$ for some event $e$ of $\mathcal{M}$.

Decidability for MSO implies decidability for all the variants of PDL. However, we get more efficient decision procedures by working directly on it.

**Theorem 12.** *From any CPDL sentence $\phi$ one can effectively construct a tree automaton $\mathcal{A}^k_\phi$ whose size is $2^{\mathcal{O}(|\mathfrak{A}|^2 . k^2 . |\phi|^2)}$ such that*

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

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 $\sigma$. For each $\sigma$ we construct an automaton $\mathcal{A}_\sigma$ such that $\mathcal{A}_\sigma$ has an accepting run from a leaf $n$ if and only if the event $n$ in the associated SQMSC satisfies $\sigma$. The automata for the atomic formulas $\top$, $p$ 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 $\pi$ does not use any state formulas then we construct a finite automaton $\mathcal{B}_\pi$ equivalent to the regular expression $\pi$ (over the alphabet $D = \{\rightarrow, \rightarrow^{-1}$

$, \rhd^d, (\rhd^d)^{-1} \mid d \in \mathbf{DS}\})$. We non-deterministically guess an accepting run of $\mathcal{B}_\pi$, simulating each move labeling this run using the tree-walking automata given by Prop. 6 and Remark 7. 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 $\mathcal{A}_{\sigma'}$. Checking state formulas in $\pi$ adds no complication due to the power of alternation. To verify the formula $\underline{\alpha}$ we simply propagate a copy of the automaton $\mathcal{A}_\alpha$ at the current node (leaf). All this can be formalized to get an A2A $\mathcal{A}_\sigma$ of size $\mathcal{O}(k \cdot |\mathbf{Procs}| \cdot |\sigma|)$. We then use Vardi's result [22] to convert this into an ordinary tree automaton of size $2^{\mathcal{O}(k^2 \cdot |\mathbf{Procs}|^2 \cdot |\sigma|^2)}$.

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 $\pi_1$ and $\pi_2$ have to end up at the same leaf. However, the technique of [12] to decide IPDL over trees can be adapted to our setting as well. As in [12] this results in an additional exponential increase in size. The details of this construction is provided in a preliminary version of this paper [9].

**Theorem 13.** *From any ICPDL sentence $\phi$ one can effectively construct a tree automaton $\mathcal{A}^k_\phi$ of doubly exponential size such that*
$$L(\mathcal{A}^k_\phi) = \{t \mid t \text{ is a } k\text{-DST},\ \mathcal{M}_t \models \phi\}.$$

**Theorem 14.** *The satisfiability problem for $\mathsf{CPDL}(\mathfrak{A}, \Sigma)$ over $k$-split-width SQMSCs is* Exptime*-complete. The satisfiability problem for $\mathsf{ICPDL}(\mathfrak{A}, \Sigma)$ over $k$-split-width SQMSCs is* 2-Exptime*-complete.*

**Temporal Logics** Another reason to study PDL over SQMSCs 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 $\varphi$ at the next position and the *until* ($\varphi_1 \mathsf{U} \varphi_2$) which asserts the existence of some future position where $\varphi_2$ holds such that $\varphi_1$ holds everywhere in between. In the setting of SQMSCs, following [14], 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.

The syntax of local temporal logics $\mathsf{TL}(\mathfrak{A}, \Sigma)$ is as follows, where $a \in \Sigma$, $p \in \mathbf{Procs}$ and $\pi$ 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 $\pi$-steps and such that $\psi$ holds at the last event of the sequence and $\varphi$ holds at intervening events in the sequence. The translation in PDL gives $\langle(\underline{\Phi} \cdot \pi)^*\rangle\Psi$, where $\Phi$ and $\Psi$ are the PDL translations of $\varphi$ and $\psi$ respectively. When $\pi$ is $\rightarrow$ it corresponds to the classical until along a process, and when $\pi$ is $\rightarrow + \rhd$ it corresponds to an *existential until* in the partial order of the SQMSC. We may also use backward steps such as $\rightarrow^{-1}$ or $\rightarrow^{-1} + \rhd^{-1}$ and thus $\mathsf{TL}(\mathfrak{A}, \Sigma)$ has both future and past modalities. Hence, from Theorem 12 we obtain

**Corollary 15.** *The satisfiability problem for $\mathsf{TL}(\mathfrak{A}, \Sigma)$ over $k$-split-width SQMSCs is* Exptime*-complete.*

**Model-checking** The $k$-split-width model-checking problem for a logic $\mathcal{L}$ determines, given an SQDS and a formula $\varphi$ in $\mathcal{L}$, whether some SQMSC of split-width at most $k$ accepted by the SQDS satisfies $\varphi$.

**Theorem 16.** *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 Logics are* EXPTIME*-complete. The $k$-split-width model-checking problem for ICPDL is* 2-EXPTIME*-complete. The complexities of these three problems are only polynomial in the size of the SQDS.*

We use Prop. 8 to construct $\mathcal{A}^k_{\mathcal{S}}$ from $\mathcal{S}$, and Theorems 11, 12 and 13 to construct from the formula $\varphi$ a tree automaton $\mathcal{A}^k_\varphi$ that recognizes all $k$-DSTs representing SQMSCs that satisfy $\varphi$. The model-checking problem then reduces to the emptiness of the intersection of the tree automata $\mathcal{A}^k_\varphi$ and $\mathcal{A}^k_{\mathcal{S}}$.

## 6 Discussions

**Optimal complexities of the decision procedures** The PTIME hardness on the size of the SQDS follows from the PTIME hardness of the emptiness checking of nested-word automata, since nested-words have split-width bounded by 2. The hardness wrt. size of the ($\epsilon$,C,IC)PDL formula follows from the corresponding case of nested words [3] (or equivalently trees [12]). The hardness wrt. the bound on split-width is a consequence of the following facts. 1. Reachability problem can be reduced to both satisfiability problem and model-checking problem of temporal logics. 2. Satisfiability and model-checking problems of temporal logics reduce to the corresponding problems of PDL and MSO. 3. Split-width of $m$-phase bounded multi-pushdown systems is bounded by $2^m$ [8]. 4. Reachability of bounded phase multi-pushdown systems is 2ETIME-hard [18].

**Further optimisations** A split-term is almost a path if one operand of any shuffle node is an atomic piece (or a subterm whose size is bounded by an a priori fixed constant). Such split-terms are said to be *word-like*. If a class admits bounded split-width via word-like decompositions, then we can obtain a better space complexity for our procedures by using finite state word-automata instead of tree-automata. Thus, the complexity for reachability in this case would be only NLOGSPACE in the number of states of the SQDS though PSPACE in $\mathfrak{A}$ and the bound on split-width. It would be PSPACE in the size of the PDL and temporal logic formula for model-checking, matching the lower bounds for the case of words. The class of SQMSCs having split-width bounded via word-like split-terms subsume interesting classes, like existentially $k$-bounded MSCs since the decomposition given on page 8 yields word like split-terms.

**Bag data-structure** We could have added bags as a possible data-structure and extended our definition of systems and behaviours accordingly. For a discussion of such systems and associated results, the reader is referred to a preliminary version of this paper [9].

**Acknowledgement** We thank Benedikt Bollig for many helpful comments.

## References

1. R. Alur and P. Madhusudan. Adding nesting structure to words. *Journal of the ACM*, 56(3):1–43, 2009.
2. M. Faouzi Atig, B. Bollig, and P. Habermehl. Emptiness of multi-pushdown automata is 2ETIME-Complete. In *DLT*, volume 5257 of *LNCS*, pages 121–133. Springer, 2008.
3. B. Bollig, A. Cyriac, P. Gastin, and M. Zeitoun. Temporal logics for concurrent recursive programs: Satisfiability and model checking. In *MFCS*, volume 6907 of *LNCS*, pages 132–144. Springer, 2011.
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. L. Breveglieri, A. Cherubini, Cl. Citrini, and S. Crespi-Reghizzi. Multi-pushdown languages and grammars. *Int. J. Found. Comput. Sci.*, 7(3):253–292, 1996.
6. B. Courcelle. Special tree-width and the verification of monadic second-order graph pr operties. In *FSTTCS*, volume 8 of *LIPIcs*, pages 13–29, 2010.
7. 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`.
8. 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.
9. A. Cyriac, P. Gastin, and K. Narayan Kumar. Verifying Communicating Multi-pushdown Systems. Technical report, Jan. 2014. `http://hal.archives-ouvertes.fr/hal-00943690`.
10. P. Ganty, R. Majumdar, and B. Monmege. Bounded underapproximations. *Formal Methods in System Design*, 40(2):206–231, April 2012.
11. B. Genest, D. Kuske, and A. Muscholl. A Kleene theorem and model checking algorithms for existentially bounded communicating automata. *Inf. Comput.*, 204(6):920–956, 2006.
12. S. Göller, M. Lohrey, and C. Lutz. PDL with intersection and converse: satisfiability and infinite-state model checking. *J. Symb. Log.*, 74(1):279–314, 2009.
13. 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.
14. J.G. Henriksen and P.S. Thiagarajan. Dynamic linear time temporal logic. *Ann. Pure Appl. Logic*, 96(1-3):187–207, 1999.
15. A. Heußner. Model checking communicating processes: Run graphs, graph grammars, and MSO. *ECEASST*, 47, 2012.
16. ITU-TS. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC).* ITU-TS, Geneva, February 2011.
17. S. La Torre, P. Madhusudan, and G. Parlato. A robust class of context-sensitive languages. In *LICS*, pages 161–170. IEEE Computer Society, 2007.
18. 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.
19. 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.
20. P. Madhusudan and G. Parlato. The tree width of auxiliary storage. In Thomas Ball and Mooly Sagiv, editors, *POPL*, pages 283–294. ACM, 2011.

21. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, volume 3440 of *LNCS*, pages 93–107. Springer, 2005.
22. 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.
23. W. Zielonka. Notes on finite asynchronous automata. *R.A.I.R.O. — Informatique Théorique et Applications*, 21:99–135, 1987.