

Safety of parametrized asynchronous shared-memory systems is almost always decidable

Salvatore La Torre¹, Anca Muscholl², and Igor Walukiewicz²

¹ Università degli Studi di Salerno

² LaBRI, Bordeaux University, CNRS

Abstract

Verification of concurrent systems is a difficult problem in general, and this is the case even more in a parametrized setting where unboundedly many concurrent components are considered. Recently, Hague proposed an architecture with a leader process and unboundedly many copies of a contributor process interacting over a shared memory for which safety properties can be effectively verified. All processes in Hague’s setting are pushdown automata. Here, we extend it by considering other formal models and, as a main contribution, find very liberal conditions on the individual processes under which the safety problem is decidable: the only substantial condition we require is the effective computability of the downward closure for the class of the leader processes. Furthermore, our result allows for a hierarchical approach to constructing models of concurrent systems with decidable safety problem: networks with tree-like architecture, where each process shares a register with its children processes (and another register with its parent). Nodes in such networks can be for instance pushdown automata, Petri nets, or multi-pushdown systems with decidable reachability problem.

1998 ACM Subject Classification D.2.4 Software/Program Verification, C.2.4 Distributed Systems

Keywords and phrases Verification, parametrized systems, shared memory

1 Introduction

Parametrized concurrent systems, i.e., systems composed of an arbitrary number of concurrent components of finitely many kinds, are the natural models of many concrete systems such as distributed network protocols, operating system drivers, or multi-threaded applications for multi-core hardware, just to mention a few. In some cases, they are algorithmically easier to analyze than the corresponding non-parametrized models which makes them a suitable abstractions for concurrency (see [20]).

Verification of shared-memory systems is a notoriously difficult problem, even for simple properties as safety. As a classical example, two pushdown systems communicating via a shared variable can directly simulate a Turing machine. In order to gain decidability, it is crucial to limit the synchronization power of such systems, for example by placing restrictions on the policies of the synchronization primitives, or bounding the number of interactions. For parametrized systems, assuming that the components have no identities is helpful besides being appropriate for many concurrent systems of interest (see also [13] for a survey).

In this paper, we revisit the verification problem for safety properties of *parametrized asynchronous shared-memory systems*. These systems consist of a *leader* process \mathcal{D} and an arbitrary number of identical *contributor* processes \mathcal{C} . The processes communicate via shared memory modelled by read/write registers. There are two important features of such $(\mathcal{C}, \mathcal{D})$ -systems: first, there are no locking mechanisms on the shared memory, and second, contributors do not have identities. This setting has been proposed by Hague [18],



© Salvatore La Torre and Anca Muscholl and Igor Walukiewicz;
licensed under Creative Commons License CC-BY

Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

$Class_{\mathcal{D}}$: possible instances for the leader	$Class_{\mathcal{C}}$: possible instances for contributors
<ul style="list-style-type: none"> ■ pushdown automata, ■ Petri nets, ■ decidable subclasses of multi-stack pushdown automata, ■ stacked counter automata, ■ order-2 pushdown automata. 	<ul style="list-style-type: none"> ■ anything in $Class_{\mathcal{D}}$, ■ higher-order pushdown automata with collapse, ■ lossy channel systems, ■ hierarchical composition of $(\mathcal{C}, \mathcal{D})$-systems with $Class_{\mathcal{D}}$ and $Class_{\mathcal{C}}$ from this table.

■ **Figure 1** Examples of models that fit our general decidability result for $(\mathcal{C}, \mathcal{D})$ -systems.

who studied the case when leaders and contributors are pushdown automata and showed an EXPSPACE upper bound. Esparza et al. [15] settled the complexity of the problem for pushdown automata proving it PSPACE-complete.

In this paper, we prove a general decidability result for verifying safety properties of $(\mathcal{C}, \mathcal{D})$ -systems. It gives conditions on leaders and contributors, expressed in terms of basic language theoretic closure and effectiveness properties, under which the problem is decidable. The main requirement is that the downward closure of the language of the leader should be effectively computable. This requirement is interesting in itself, and we remark that, in our setting, it is weaker than having effective semilinear Parikh images.

Our work shows that the setting of $(\mathcal{C}, \mathcal{D})$ -systems can be instantiated in many different ways, while preserving the decidability of safety properties. Figure 1 lists some examples of types of systems for leaders and for contributors that our theorem covers. For example, the leader and contributors can be themselves Petri nets, or restrictions of multi-pushdown processes with decidable reachability problem.

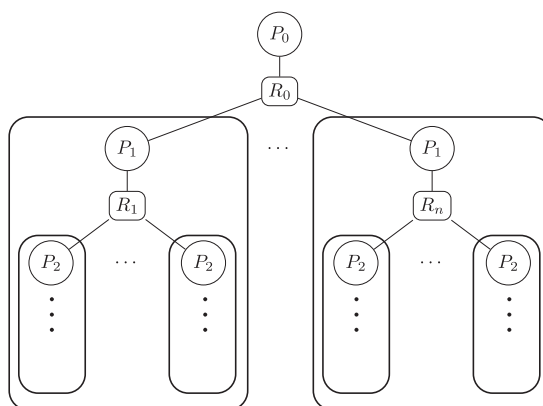
One interesting consequence of our main result is that it can be applied recursively, by instantiating a contributor \mathcal{C} by another $(\mathcal{C}, \mathcal{D})$ -system (see Figure 2). This implies that safety properties can be verified for networks that have a tree-like architecture: each process shares a register with its children processes (and another register with its parent). Nodes in such networks can belong to one of the formal models listed above.

Finally, as a byproduct, our construction allows to reprove in a different way known complexity results for $(\mathcal{C}, \mathcal{D})$ -systems over pushdown automata [15]. We believe that our approach is simpler thanks to the use of downward closures.

Related work. Parametrized verification of shared-memory multi-threaded programs has been studied for finite-state threads in [7, 21] and for threads modeled as pushdown systems in [6, 9, 23, 24, 8]. The main difference with our setting is that in those models the synchronization primitives are allowed, and thus for pushdown threads, reachability becomes undecidable even if we restrict to finite data domains. The decidability results in [6, 9, 23, 24, 8] concern the reachability analysis up to a bounded number of execution contexts, and in [6, 9], dynamic thread creation is allowed.

There is a rich literature concerning the verification of asynchronously-communicating parametrized programs that is mostly related to the verification of distributed protocols and varies for approaches and models (see e.g. [16, 10, 14, 22] for some early work, and [12, 5, 28] and references therein).

Parametrized tree systems, i.e., systems formed of an arbitrary number of processes operating on a tree-like architecture, have been studied by tree rewritings (see [3, 2] and



■ **Figure 2** Example of a hierarchical composition of $(\mathcal{C}, \mathcal{D})$ -systems, P_i are process types and R_i are R/W registers. P_0 is the leader \mathcal{D} of a $(\mathcal{C}, \mathcal{D})$ -system whose contributors are themselves $(\mathcal{C}, \mathcal{D})$ -systems each one with leader from P_1 and contributors from P_2 .

references therein). Our hierarchical composition of $(\mathcal{C}, \mathcal{D})$ -systems is quite different from the models studied there. Namely, each process shares a finite memory with its children processes and its parent process: all the interactions with the network neighbours are through asynchronous accesses to such memories. As processes, we allow several classes of systems, not just finite-state systems. On the other side, in our model there is no notion of global transitions.

Organization of the paper. In Section 2, we give some basic definitions and introduce the notion of $(\mathcal{C}, \mathcal{D})$ -system. In Section 3, the accumulator semantics is introduced and shown equivalent to the standard semantics of $(\mathcal{C}, \mathcal{D})$ -systems. In Section 4, we give two constructions that allow to decompose the semantics of $(\mathcal{C}, \mathcal{D})$ -systems into the parts concerning respectively the leader and the contributors. In Section 5, these constructions are used to give a decision algorithm that shows the main result of the paper. In Section 6, we use our approach to study the computational complexity of the reachability for $(\mathcal{C}, \mathcal{D})$ -systems for the classes of finite automata and pushdown automata. We conclude in Section 7 with a few remarks.

2 Preliminaries

We first define the parametrized systems that we consider, and their reachability problem. These systems consist of one instance of a leader process \mathcal{D} , and an arbitrary number of instances of a contributor process \mathcal{C} . Both \mathcal{C} and \mathcal{D} can be arbitrary, potentially infinite, transition systems. One can think of them as transition systems generated by, for example, pushdown automata, Petri nets, or lossy channel systems. Our decidability result will refer to the closure properties of classes of transition systems over which \mathcal{C} and \mathcal{D} range.

A *transition system* is a graph with states and labelled edges. The labels of edges are called *actions*. There may be infinitely many states in a transition system, but we will assume that the set of actions is finite. A transition system will come with an initial state. A *trace* is a sequence of actions labelling a path starting in the initial state. A word v is a *subword* of u if it can be obtained from u by erasing letters.

The *synchronized product* of two transition systems is a system whose state set is the product of the state sets of the two systems, and whose transitions are defined according to the rule: for actions common to the two systems the transition should be synchronized,

whereas actions of only one of the two systems affect only the relevant component of the pair. In particular, the synchronized product of two transition systems over the same alphabet is just the standard product of the two.

For our decidability results we will assume implicitly that transition systems are given by some finite description. For example, when we will talk about the class of pushdown transition systems we will assume that they are given by pushdown automata.

We say that a class of transition systems is *effectively closed* under some operation if from a description of a transition system in the class we can effectively construct a description of the image of the transition system under the operation. Our decidability result will use a couple of abstract properties of classes of transition systems. For a class \mathcal{C} of transitions systems we say that:

- \mathcal{C} is *effectively closed under synchronized products with finite automata* if for every description of a system in \mathcal{C} , and every finite automaton, one can effectively find a description in \mathcal{C} of the synchronized product of the transition system with the automaton.
- \mathcal{C} has *decidable reachability problem* if there is an algorithm deciding for a given action and a description of a transition system in \mathcal{C} if from the initial state of \mathcal{C} there is a trace containing this action.
- \mathcal{C} has an *effective downward closure* if there is an algorithm calculating for a given description of a transition system in \mathcal{C} the finite automaton accepting all subwords of the traces of \mathcal{C} from the initial state.

Observe that having effective downward closure implies having decidable reachability problem. We will give an example of the use of these notions in a simple result on page 6 (Corollary 2).

We proceed to the formal definition of $(\mathcal{C}, \mathcal{D})$ -systems. These systems are composed of arbitrary many instances of a contributor process \mathcal{C} and one instance of a leader process \mathcal{D} . The processes communicate through a shared register. We write G for the finite set of register values, and use g, h to range over elements of G . The initial value of the register is denoted g_{init} . The alphabets of both \mathcal{C} and \mathcal{D} contain actions representing reads and writes to the register:

$$\Sigma_{\mathcal{C}} = \{r(g), w(g) : g \in G\} , \quad \Sigma_{\mathcal{D}} = \{\bar{r}(g), \bar{w}(g) : g \in G\} .$$

Both \mathcal{C} and \mathcal{D} are, possibly infinite, transition systems over these alphabets:

$$\mathcal{C} = \langle S, \delta \subseteq S \times \Sigma_{\mathcal{C}} \times S, s_{init} \rangle \quad \mathcal{D} = \langle T, \Delta \subseteq T \times \Sigma_{\mathcal{D}} \times T, t_{init} \rangle .$$

The transition systems do not have internal actions. Adding them to the alphabets would not modify our results, so for simplicity we prefer not to deal with them here. Internal actions will be useful though when we consider hierarchical composition of $(\mathcal{C}, \mathcal{D})$ -systems.

A $(\mathcal{C}, \mathcal{D})$ -system consists of an unspecified number of copies of \mathcal{C} , one copy of \mathcal{D} , and a shared register. A configuration of such a system can be represented by a triple $(f : \mathbb{N} \rightarrow S, t \in T, g \in G)$, consisting of a function f counting the number of instances of \mathcal{C} in a given state, the state t of \mathcal{D} and the current register value g .

In principle we would expect that f is a partial function defined only on an initial interval of \mathbb{N} . This would represent that indeed there are only finitely many copies of \mathcal{C} . Given the questions we are interested in, this is irrelevant, so we prefer not to put this condition.

The transitions of the $(\mathcal{C}, \mathcal{D})$ -system are presented below. We write $f \xrightarrow{a} f'$ for the extension from S to $\mathbb{N} \rightarrow S$ of the transition relation δ of \mathcal{C} : there should be an index i such

that $f(i) \xrightarrow{a} f'(i)$ in \mathcal{C} and $f(j) = f'(j)$ for $j \neq i$.

$$\begin{array}{ll}
(f, t, g) \xrightarrow{\bar{w}(h)} (f, t', h) & \text{if } t \xrightarrow{\bar{w}(h)} t' \text{ in } \Delta \\
(f, t, g) \xrightarrow{\bar{r}(h)} (f, t', h) & \text{if } t \xrightarrow{\bar{r}(h)} t' \text{ in } \Delta \text{ and } h = g \\
(f, t, g) \xrightarrow{w(h)} (f', t, h) & \text{if } f \xrightarrow{w(h)} f' \text{ in } \delta \\
(f, t, g) \xrightarrow{r(h)} (f', t, h) & \text{if } f \xrightarrow{r(h)} f' \text{ in } \delta \text{ and } h = g
\end{array}$$

The *reachability problem* is to decide if in a given $(\mathcal{C}, \mathcal{D})$ -system the register can contain some error value that we denote by $\#$. Observe that it may be assumed w.l.o.g. that this value is written by the leader \mathcal{D} . This means that we are asking if there is a trace of the $(\mathcal{C}, \mathcal{D})$ -system from the initial configuration $(f_{init}, t_{init}, g_{init})$, with label from $(\Sigma_{\mathcal{C}} \cup \Sigma_{\mathcal{D}})^* \bar{w}(\#)$. Here, f_{init} is a constant function assigning the initial state of \mathcal{C} to every $i \in \mathbb{N}$. In the following we will simply say that we want to decide if there is a $\#$ -trace in the $(\mathcal{C}, \mathcal{D})$ -system.

3 Accumulator semantics

The semantics we have presented above, although natural, is not that easy to work with. Here we formulate a different semantics, called accumulator semantics, that is equivalent if the reachability problem is considered. As an example of the advantage offered by the accumulator semantics we give a very simple argument for the decidability in the case when \mathcal{C} ranges over finite state systems.

In the accumulator semantics, instead of a function $f : \mathbb{N} \rightarrow S$ we use a set $A \subseteq S$ that we call accumulator to reflect the fact that it can only grow. The idea is that since we reason in parametrized setting, we do not need to count precisely how many copies of \mathcal{C} have reached a given state. Once that a state is reached, it can be “duplicated” an arbitrary number of times. So in the accumulator semantics configurations are of the form $(A \subseteq S, t \in T, g \in G)$, and the transitions are:

$$\begin{array}{ll}
(A, t, g) \xrightarrow{\bar{w}(h)} (A, t', h) & \text{if } t \xrightarrow{\bar{w}(h)} t' \text{ in } \Delta \\
(A, t, g) \xrightarrow{\bar{r}(h)} (A, t', h) & \text{if } t \xrightarrow{\bar{r}(h)} t' \text{ in } \Delta \text{ and } h = g \\
(A, t, g) \xrightarrow{w(h)} (A \cup \{s'\}, t, h) & \text{if } s \xrightarrow{w(h)} s' \text{ in } \delta \text{ for some } s \in A \\
(A, t, g) \xrightarrow{r(h)} (A \cup \{s'\}, t, h) & \text{if } h = g \text{ and } s \xrightarrow{r(h)} s' \text{ in } \delta \text{ for some } s \in A
\end{array}$$

► **Proposition 1.** *There is a $\#$ -trace from $(f_{init}, t_{init}, g_{init})$ in the $(\mathcal{C}, \mathcal{D})$ -system iff there is one from $(\{s_{init}\}, t_{init}, g_{init})$ in the accumulator semantics.*

Proof. For the left to right direction, we take a run $\{(f_i, t_i, g_i)\}_{i=1, \dots, n}$ and show that $\{(A_i, t_i, g_i)\}_{i=1, \dots, n}$ is a run, where A_i is the set states of \mathcal{C} that have appeared as a value of one of f_1, \dots, f_i .

For the right to left direction we prove a more general statement. Suppose that σ is a $\#$ -trace in the accumulator semantics from a state (A, t, g) and f is such that $|\{i : f(i) = s\}| \geq 2^{|\sigma|}$ for all $s \in A$. Then we show that there is a $\#$ -trace from (f, t, g) in $(\mathcal{C}, \mathcal{D})$ -system. The lemma then follows since in f_{init} we have arbitrarily many copies of s_{init} . The proof of this statement is by induction on the length of σ . One step in the accumulator semantics is simulated by letting either \mathcal{D} take the step, or half of the copies of \mathcal{C} take the step. ◀

Note that the standard and the accumulator semantics do not generate the same traces. In order to simulate a step in the accumulator semantics, the $(\mathcal{C}, \mathcal{D})$ -system may need to perform several steps.

If \mathcal{C} is a finite state automaton then the A -part in the accumulator semantics is of bounded size. This gives a simple decidability result:

► **Corollary 2.** *Suppose that $\text{Class}_{\mathcal{D}}$ is closed under synchronized products with finite automata. The reachability problem for $(\mathcal{C}, \mathcal{D})$ -systems where \mathcal{C} is a finite-state automaton, and \mathcal{D} is from $\text{Class}_{\mathcal{D}}$, effectively reduces to the reachability problem in $\text{Class}_{\mathcal{D}}$.*

Proposition 1 allows us to use the accumulator semantics as the semantics of $(\mathcal{C}, \mathcal{D})$ -systems, and this will be our implicit assumption in the following sections.

4 Capacities and downward closures

Our objective is a decidability result for $(\mathcal{C}, \mathcal{D})$ -systems. It will be obtained by combining two reductions that we describe in this section. First, we will decompose the semantics of a $(\mathcal{C}, \mathcal{D})$ -system into the part concerning \mathcal{C} and the one concerning \mathcal{D} . Lemma 4 reduces our problem to that of finding an input on which we can run separately two parts. The second step starts from the observation that instead of \mathcal{D} we can work with the downward closure of \mathcal{D} (Lemma 5). Then we can rely on the well-known fact that the downward closure of *any* language is regular. Using a decomposition technique similar to that of Lemma 4 we obtain our main technical result, Lemma 7. This will be turned into a decision procedure in the next section.

We start by defining the transition system \mathcal{D}^{κ} , that captures the part of the $(\mathcal{C}, \mathcal{D})$ -system concerning \mathcal{D} . The system \mathcal{D}^{κ} is obtained by abstracting the register contributions of \mathcal{C} by a set $K \subseteq G$ of possible values. Let $\mathcal{D}^{\kappa} = \langle \mathcal{P}(G) \times T \times G, \delta, (\emptyset, t_{init}, g_{init}) \rangle$. So a configuration of \mathcal{D}^{κ} has the form

$$(K \subseteq G, t \in T, g \in G).$$

Intuitively, K represents a *capacity*: the values that contributors have already written into the register up to the present point of the execution of the system. State $t \in T$ is the current state of \mathcal{D} , and $g \in G$ is the current content of the register.

To update the K -component of a configuration we introduce a new alphabet

$$\Sigma_{\nu} = \{\nu(g) : g \in G\},$$

and let the alphabet of \mathcal{D}^{κ} be $\Sigma_{\mathcal{D}} \cup \Sigma_{\nu}$. The intuition behind the transitions of \mathcal{D}^{κ} presented below is the following. Since an arbitrary number of copies of \mathcal{C} can be started, whenever a value g is written in the register by a contributor we can construct a different run where this instance of the contributor is duplicated some number of times. In this new run, any time in the future of the computation when the value g is needed in the register we can use one of these duplicates. We capture this phenomenon in the transitions of \mathcal{D}^{κ} by enabling a read action $\bar{r}(h)$ whenever $h \in K$.

Precisely, the transitions of \mathcal{D}^{κ} are:

$$\begin{aligned} (K, t, g) &\xrightarrow{\bar{w}(h)} (K, t', h) && \text{if } t \xrightarrow{\bar{w}(h)} t' \text{ in } \Delta, \\ (K, t, g) &\xrightarrow{\bar{r}(h)} (K, t', h) && \text{if } t \xrightarrow{\bar{r}(h)} t' \text{ in } \Delta \text{ and } h \in K \cup \{g\}, \\ (K, t, g) &\xrightarrow{\nu(h)} (K \cup \{h\}, t, h) && \text{if } h \notin K. \end{aligned}$$

It is not difficult to see that if there is a $\#$ -trace in the $(\mathcal{C}, \mathcal{D})$ -system then there is one in \mathcal{D}^κ . The opposite is clearly not true because \mathcal{D}^κ ignores the form of \mathcal{C} : there is no check that $\nu(h)$ actions can indeed come from writes of \mathcal{C} . We will recover the equivalence by putting an additional condition on traces of \mathcal{D}^κ (cf. Lemma 4 below).

In order to obtain a sufficient condition on traces of \mathcal{D}^κ we construct a ‘‘capacity aware’’ version of \mathcal{C} . This is the transition system $\mathcal{C}^\kappa = \langle \mathcal{P}(G) \times S \times G, \delta^\kappa, (\emptyset, s_{init}, g_{init}) \rangle$ where δ^κ is:

$$\begin{aligned} (K, s, g) &\xrightarrow{\bar{w}(h)} (K, s, h) & (K, s, g) &\xrightarrow{\bar{r}(h)} (K, s, h) & (K, s, g) &\xrightarrow{\nu(h)} (K \cup \{h\}, s, h) \\ (K, s, g) &\xrightarrow{w(h)} (K, s', h) & \text{if } s &\xrightarrow{w(h)} s' \text{ in } \delta \text{ and } h \in K \\ (K, s, g) &\xrightarrow{r(h)} (K, s', h) & \text{if } s &\xrightarrow{r(h)} s' \text{ in } \delta \text{ and } h \in K \cup \{g\}. \end{aligned}$$

This automaton follows the actions of \mathcal{D}^κ (first line above) in order to be aware of the current contents of the register and the capacity. At the same time, \mathcal{C}^κ can also do the $w(h)$ actions provided they are declared in the capacity K , and the $r(h)$ actions when h is either in the capacity K or in the register. So the capacity restricts the write actions of contributors and allows for more read actions.

We stress the following:

1. Both for \mathcal{C}^κ and \mathcal{D}^κ , the content of the register after a transition is determined by the executed action.
2. Both systems have two kinds of reads: from the register g , and from the capacity K . We refer to actions $\bar{r}(h)$ and $r(h)$ as *capacity reads* whenever $h \in K$. Note that these reads simulate a read of a value written by a copy of \mathcal{C} .
3. The K -component of \mathcal{C}^κ and \mathcal{D}^κ is determined by the sequence of $\nu(h)$ -moves. An execution of \mathcal{D}^κ can have at most one $\nu(h)$ action for every $h \in G$.

Lemma 4 below formulates the condition on traces of the transition system \mathcal{D}^κ that correspond to traces in the $(\mathcal{C}, \mathcal{D})$ -system.

Notation. We will use the convention of writing $\Sigma_{\mathcal{D}, \nu}$ for $\Sigma_{\mathcal{D}} \cup \Sigma_{\nu}$. Similarly for $\Sigma_{\mathcal{C}, \nu}$ and $\Sigma_{\mathcal{C}, \mathcal{D}, \nu}$. By $v|_{\Sigma}$ we will denote the subword of v obtained by erasing the symbols not in Σ .

► **Definition 3.** A trace $v \nu(h) \in \Sigma_{\mathcal{D}, \nu}^*$ is *\mathcal{C} -supported* if there exists a word $u \in \Sigma_{\mathcal{C}, \mathcal{D}, \nu}^*$ such that

$$u|_{\Sigma_{\mathcal{D}, \nu}} = v \quad \text{and} \quad u \nu(h) w(h) \in L(\mathcal{C}^\kappa).$$

A trace $v \in \Sigma_{\mathcal{D}, \nu}^*$ is *totally \mathcal{C} -supported* if every prefix $v' \nu(h)$ of v is supported.

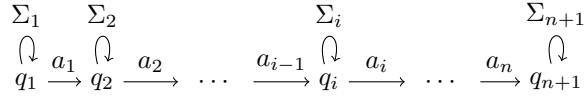
The intuition behind this definition is that every $\nu(h)$ in a trace of \mathcal{D}^κ should be supported by a run of contributors witnessing that the write action $w(h)$ is indeed possible.

► **Lemma 4.** *There is a $\#$ -trace in a $(\mathcal{C}, \mathcal{D})$ -system iff there is a totally \mathcal{C} -supported $\#$ -trace in \mathcal{D}^κ .*

Lemma 4 tells us that in order to find a $\#$ -trace in $(\mathcal{C}, \mathcal{D})$ -system we need to find a $\#$ -trace in \mathcal{D}^κ and verify that it is supported. We will now see that we can actually work with the set of subwords of \mathcal{D}^κ . This is important, as for every language the set of its subwords is a regular language. Moreover, the minimal automaton for the downward closure has a very simple form. The main technical result of this section says that our initial problem reduces to finding a particular pattern in this minimal automaton.

► **Lemma 5.** *If $v_1\nu(h_1)\dots v_i\nu(h_i)$ is a \mathcal{C} -supported trace and v_j is a subword of $v'_j \in \Sigma_{\mathcal{D}}^*$ for every $j = 1, \dots, i$, then $v'_1\nu(h_1)\dots v'_i\nu(h_i)$ is \mathcal{C} -supported.*

The *downward closure* of language L , denoted $L\downarrow$, is the set of subwords of the words in L . In the following, we denote by $\mathcal{D}^\kappa\downarrow$ the minimal automaton accepting the downward closure of the set of traces from \mathcal{D}^κ . Every minimal automaton accepting a downward closed language is a graph where the only cycles are self-loops on some states (cf. Lemma 15 in the appendix). So every word accepted by $\mathcal{D}^\kappa\downarrow$ comes from a pattern of the form in Figure 3, where on each q_i there is a self-loop on letters from some, possibly empty, alphabet $\Sigma_i \subseteq \Sigma_{\mathcal{D}}$. Note that actions of the form $\nu(h)$ do not occur on self-loops, since by observation 3 on page 7 their number is bounded in any trace from \mathcal{D} . As $\mathcal{D}^\kappa\downarrow$ is a finite automaton, there



■ **Figure 3** A pattern in $\mathcal{D}^\kappa\downarrow$.

are finitely many such patterns. We can thus take patterns one by one, and check if there is one that determines a fully \mathcal{C} -supported trace. The problem is that to check this we need to fix a trace in advance, and it is not clear how to do this since we have no bound on the length of a fully supported trace. The definition of compatible patterns and the lemma that follows go around this problem.

► **Definition 6.** Consider a pattern as in Figure 3. The pattern is *\mathcal{C} -compatible* up to position i if for every $j = 1, \dots, i$ there are words $v_j \in \Sigma_j^*$ such that $v_0a_1\dots v_ia_i$ is \mathcal{C} -supported. The pattern is *fully \mathcal{C} -compatible* if for every $i = 1, \dots, n$ such that $a_i = \nu(h)$ for some h , the pattern is \mathcal{C} -compatible up to position i . A *#-pattern* is one ending with $a_n = \bar{w}(\#)$.

The difference between the above definition and Definition 3 is that in the latter we work with a single trace that is \mathcal{C} -supported. In Definition 6 we may have to consider different \mathcal{C} -supported traces for distinct positions of the pattern. This is necessary, because we cannot fix in advance a trace for all positions of the pattern.

► **Lemma 7.** *There is a fully \mathcal{C} -supported #-trace in \mathcal{D}^κ iff there is a fully \mathcal{C} -compatible #-pattern in $\mathcal{D}^\kappa\downarrow$.*

Lemma 7 together with Lemma 4 reduces our reachability problem to the problem of finding a fully \mathcal{C} -compatible #-pattern in a finite automaton $\mathcal{D}^\kappa\downarrow$.

5 A general decidability result

In this section we present the main result of the paper giving conditions under which the reachability problem for $(\mathcal{C}, \mathcal{D})$ -systems is decidable. The theorem refers to the properties of classes of transition systems defined on page 4. We also discuss the hypotheses of the theorem as well its applicability referring back to examples from Figure 1.

► **Theorem 8.** *Suppose $\text{Class}_{\mathcal{C}}$, $\text{Class}_{\mathcal{D}}$ are two classes of transition systems closed under synchronized products with finite automata. If $\text{Class}_{\mathcal{C}}$ has a decidable reachability problem, and $\text{Class}_{\mathcal{D}}$ has effective downward closure then the reachability problem for $(\mathcal{C}, \mathcal{D})$ -systems, with \mathcal{C} from $\text{Class}_{\mathcal{C}}$ and \mathcal{D} from $\text{Class}_{\mathcal{D}}$, is decidable.*

Proof. We describe an algorithm deciding the reachability problem for $(\mathcal{C}, \mathcal{D})$ -systems. Given \mathcal{C} and \mathcal{D} , the algorithm first computes \mathcal{C}^κ and \mathcal{D}^κ as defined on page 6. The definition of \mathcal{D}^κ tells us that it can be obtained by first extending \mathcal{D} with actions in Σ_ν and then making a product with a finite automaton that takes care of the capacity set and the register content (similar for \mathcal{C}^κ). Then the algorithm computes the finite automaton $\mathcal{D}^\kappa \downarrow$ for the downward closure of \mathcal{D}^κ . These operations are effective since \mathcal{D} is in $\text{Class}_{\mathcal{D}}$.

In the next step the algorithm examines all $\#$ -patterns in $\mathcal{D}^\kappa \downarrow$ of the form:

$$\begin{array}{ccccccc} \Sigma_1 & & \Sigma_2 & & & & \Sigma_{n+1} \\ \downarrow & a_1 & \downarrow & a_2 & \cdots & a_n & \downarrow & \# \\ q_1 & \longrightarrow & q_2 & \longrightarrow & \cdots & \longrightarrow & q_{n+1} & \longrightarrow & q_{n+2} \end{array}$$

and checks if there is one that is fully \mathcal{C} -compatible. The algorithm answers yes if and only if it finds a $\#$ -pattern in $\mathcal{D}^\kappa \downarrow$ that is fully \mathcal{C} -compatible. Observe that by Lemma 7 this holds iff \mathcal{D}^κ has a fully \mathcal{C} -supported $\#$ -trace, and thus by Lemma 4 iff there exists a $\#$ -trace in the $(\mathcal{C}, \mathcal{D})$ -system.

To complete the proof, we need to show how to check that a pattern as above is fully \mathcal{C} -compatible. Let k_1, \dots, k_l be the indices such that a_{k_i} is of the form $\nu(h_i)$. The algorithm checks if the prefix of the pattern up to a_{k_i} is \mathcal{C} -compatible for $i = 1, \dots, l$.

For each $i = 1, \dots, l$, the check proceeds as follows. First, starting from the pattern up to a_{k_i} it constructs the finite automaton accepting $\Gamma_1^* a_1 \dots \Gamma_{k_i}^* \nu(h_i) w(h_i)$ where $\Gamma_j = \Sigma_j \cup \Sigma_{\mathcal{C}}$ for $j = 1, \dots, k_i$. Then, it takes the synchronized product of the resulting automaton with \mathcal{C}^κ . Denote it with $(\mathcal{C}^\kappa)_i$. The final step of the check is to test for reachability of $w(h_i)$ in $(\mathcal{C}^\kappa)_i$. Note that this can be done by hypothesis since from the properties of $\text{Class}_{\mathcal{C}}$, $(\mathcal{C}^\kappa)_i$ is still in $\text{Class}_{\mathcal{C}}$. In fact, the test succeeds iff the pattern up to a_{k_i} is \mathcal{C} -compatible. This concludes the proof. \blacktriangleleft

In Figure 1 we have listed some concrete instances of $(\mathcal{C}, \mathcal{D})$ -systems for which the reachability problem is decidable thanks to Theorem 8. For all the listed classes the closure under synchronized products required by the theorem is immediate, since all of them have finite control.

The effective downward closure, and thus effective reachability problem, holds for pushdown automata [11], Petri net languages [17], stacked counter automata [30], and higher-order pushdown automata of order 2 [29].

Multi-stack pushdown automata (MPA) are Turing powerful already with two stacks. There are though subclasses of MPA with a decidable reachability problem such as path-tree MPA, and scope-bounded MPA [25, 26]. The former include bounded-phase MPA and ordered MPA. For all these classes it is known that visibly multi-pushdown languages have effective semilinear Parikh images. In the Appendix we point out that this implies that even for non visibly-pushdown languages, all these classes have effective Parikh images. Then Corollary 9 below implies decidability for these classes of MPA.

Reachability is decidable for lossy channel systems [4] and higher-order pushdown automata with collapse [19]. Lossy channel systems do not have effective downward closure: this can be seen by a rather direct reduction from the problem of deciding boundedness of the set of reachable configurations [27]. For higher-order pushdown automata it is not known if the downward closure is effective.

Theorem 8 makes several assumptions about the classes $\text{Class}_{\mathcal{C}}$ and $\text{Class}_{\mathcal{D}}$. It is worth to examine them closer.

The closure property of the theorem, closure under synchronized product, is implied by closure under rational transductions. Given two alphabets Σ and Γ , a *rational transduction* from Σ to Γ is the subset of $\Sigma^* \times \Gamma^*$ generated by a finite-state transducer. A class of languages is *closed under rational transductions* if for every language L in the class, and every finite-state transducer T the image of L under T is in the class. Observe that the closure under synchronized products with finite automata does not imply closure under projections, and more generally under homomorphisms. So the closure requirements our theorem are weaker than the closure under rational transductions.

Having an effective downward closure is an interesting condition in itself that probably deserves to be better understood. Zetsche [29] has recently shown that a sufficient condition for a class to have an effective downward closure is to be closed under rational transductions and to have *effective semilinear Parikh images*. The latter means that there is an algorithm that given a description of a transition system calculates a semilinear representation of the Parikh image of the language of the transition system. A closer examination of his argument shows that our closure under synchronized product with finite automata, together with effective semilinear Parikh images, already implies effective downward closure. Thus in our theorem we can replace the requirement that $Class_{\mathcal{D}}$ has effective downward closure by effective semilinear Parikh images.

► **Corollary 9.** *Suppose that $Class_{\mathcal{C}}, Class_{\mathcal{D}}$ are two classes of transition systems closed under synchronized products with finite automata. If $Class_{\mathcal{C}}$ has a decidable reachability problem and $Class_{\mathcal{D}}$ has effective semilinear Parikh images then the reachability problem for $(\mathcal{C}, \mathcal{D})$ -systems, with \mathcal{C} from $Class_{\mathcal{C}}$ and \mathcal{D} from $Class_{\mathcal{D}}$, is decidable.*

Some requirements of the theorem, as the closure under products with finite automata seem rather unavoidable. Observe that if, for example, we take $Class_{\mathcal{D}}$ to be the class of process algebra processes, then the register can act as a common state making the reachability problem undecidable even for the case when $Class_{\mathcal{C}}$ is a trivial class containing one process that does nothing. Clearly, the same holds for more general rewriting as process rewrite and term rewriting systems.

Another example of a class that is not closed under products with finite automata is the class of context-free FIFO rewriting systems (that has an effective downward closure though [1]). Our theorem cannot be applied with this class as $Class_{\mathcal{C}}$ or $Class_{\mathcal{D}}$, and we do not know if the reachability problem becomes undecidable in this case.

6 Complexity issues

We have not yet discussed complexity issues. One of the reasons why the algorithm from the proof of Theorem 8 may not be optimal is that it requires to generate the downward closure explicitly. Here we consider two instances where the downward closure can be generated on-the-fly. The two results of this section are already known [15]. Our purpose is to indicate that our approach is algorithmically interesting, and gives arguably more transparent proofs.

The first result is quite immediate thanks to the accumulator semantics.

► **Proposition 10.** *The reachability problem for $(\mathcal{C}, \mathcal{D})$ -systems is in NP when \mathcal{C} ranges over finite state systems, and \mathcal{D} over pushdown systems.*

Proof. The claim is obvious if \mathcal{D} is also finite state, since the first component in the accumulator grows monotonically. Thus, a $\#$ -trace from $(\{s_{init}\}, t_{init}, g_{init})$ in the accumulator semantics can be guessed in polynomial time. If \mathcal{D} is a pushdown system, then a trace in the

accumulator semantics splits in $\leq |S|$ phases, where the accumulator component is constant in each phase. So this reduces to (1) guessing the sequence of A -values and (2) a reachability question for a pushdown system executing in $\leq |S|$ phases; each phase corresponding to a particular value of the accumulator. In a phase with value A , the value of the register can change via ϵ -transitions corresponding to contributor writes from states in A . ◀

The second result solves the case when both \mathcal{D} and \mathcal{C} are pushdown systems.

► **Theorem 11.** *The reachability problem for $(\mathcal{C}, \mathcal{D})$ -systems is in PSPACE when both \mathcal{C} and \mathcal{D} range over pushdown systems.*

The proof of this result starts with a construction by Courcelle [11] that provides an exponential size NFA for $\mathcal{D}^\kappa \downarrow$; moreover the transitions of the automaton can be computed on-the-fly in PSPACE.

Let $\sigma = h_1, \dots, h_i$ be a sequence of pairwise distinct values from G and $1 \leq j \leq i$. Let \mathcal{F}_j be a pushdown automaton accepting \mathcal{C} -supported words over $\Sigma_{\mathcal{D}, \nu}$ that contain exactly the prefix of length j of σ as the occurrences of Σ_ν -symbols. So \mathcal{F}_j accepts words of the form

$$v_1\nu(h_1) \dots v_{j-1}\nu(h_{j-1})v_j\nu(h_j), \quad v_k \in \Sigma_{\mathcal{D}}^* \quad (1)$$

and can be obtained essentially as the projection of \mathcal{C}^κ on $\Sigma_{\mathcal{D}, \nu}$ augmented with a check that the occurrences from Σ_ν correspond to h_1, \dots, h_j (we also need to check for a transition on $w(h_j)$ from the state entered after reading the last Σ_ν). Note that once we fix σ , the size of each pushdown \mathcal{F}_j is polynomial in the size of \mathcal{C} .

► **Lemma 12.** *An NFA \mathcal{B}_j can be effectively constructed such that:*

1. $L(\mathcal{B}_j) \subseteq L(\mathcal{F}_j)$;
2. for every $u \in L(\mathcal{F}_j)$ there is a subword v of u with $v \in L(\mathcal{B}_j)$.

The NFA \mathcal{B}_j is of size exponential in the size of \mathcal{C} and its transitions can be computed on-the-fly in PSPACE.

For the proof of the above lemma we refer e.g. to Theorem 7 in [15]. A alternative proof is to take the NFA that accepts words generated by a CFG equivalent to \mathcal{F}_σ with derivation trees where no variable occurs more than once on any path. Note also that since $L(\mathcal{B}_j) \subseteq L(\mathcal{F}_j)$, the words accepted by this automaton are of the form (1) as well.

We use now Lemmas 5 and 12 in order to replace both \mathcal{D} and \mathcal{C} by NFAs. First we guess a sequence $\sigma = h_1, \dots, h_i$ of distinct register values.

For every $1 \leq j \leq i$ let $\widehat{\mathcal{F}}_j$ be an NFA accepting extensions of the words accepted by \mathcal{B}_j , more precisely the words $u_1\nu(h_1) \dots u_{j-1}\nu(h_{j-1})u_j\nu(h_j)$ with $u_k \in \Sigma_{\mathcal{D}}^*$ such that there are subwords v_k of u_k with $v_1\nu(h_1) \dots v_{j-1}\nu(h_{j-1})v_j\nu(h_j) \in L(\mathcal{B}_j)$. Observe that $\widehat{\mathcal{F}}_j$ is of the same size as \mathcal{B}_j and its transitions can also be generated in PSPACE. The next lemma shows that $L(\mathcal{F}_j) = L(\widehat{\mathcal{F}}_j)$ for every $1 \leq j \leq i$.

► **Lemma 13.** $L(\mathcal{F}_j) = L(\widehat{\mathcal{F}}_j)$.

Proof. From item (2) of Lemma 12 and the definition of $\widehat{\mathcal{F}}_j$, we get $L(\mathcal{F}_j) \subseteq L(\widehat{\mathcal{F}}_j)$.

For the other direction take a word $u \in \widehat{\mathcal{F}}_j$. By definition, it is necessarily of the form $u = u_1\nu(h_1) \dots u_{j-1}\nu(h_{j-1})u_j\nu(h_j)$. Moreover there is a word v accepted by \mathcal{B}_j of the form $v = v_1\nu(h_1) \dots v_{j-1}\nu(h_{j-1})v_j\nu(h_j)$, with v_k subword of u_k for every k . Since by Lemma 12, $L(\mathcal{B}_j) \subseteq L(\mathcal{F}_j)$, we have that v is accepted by \mathcal{F}_j . Thus, by Lemma 5, u is \mathcal{C} -supported and contains the prefix of length j of σ as sequence of Σ_ν symbols, therefore it is accepted by \mathcal{F}_j . ◀

The algorithm required in Theorem 11 nondeterministically guesses on-the-fly a trace in $\mathcal{D}^\omega \downarrow$, and runs simultaneously $\widehat{\mathcal{F}}_1, \dots, \widehat{\mathcal{F}}_i$ on this trace in order to check if it is fully supported according to Lemma 4. Since all the automata can be generated in PSPACE the whole algorithm is in PSPACE.

7 Conclusions

Parametrized models with decidable reachability problem are relatively rare. We have studied parametrized systems where processes have no identities, and there are no locking mechanisms on the shared memory [18]. The model has turned out to have interesting algorithmic properties: safety analysis is decidable when its components are chosen from a wide range of formal models. Technically, there are two novelties of in our approach: the accumulator semantics, and the use of downward closures. Our result allows for a compositional construction of a formal model of a distributed system, as schematically presented in Figure 2.

This work puts a spotlight on the effective downward closure property. It would be interesting to investigate this property for other models, as for example for higher-order pushdowns. Among other important issues raised by this work are the questions of the complexity of computing downward closures, and in particular of computing them on-the-fly.

It is not clear if there is an elegant characterization of classes $Class_{\mathcal{C}}$ and $Class_{\mathcal{D}}$ for which the reachability problem for $(\mathcal{C}, \mathcal{D})$ -systems is decidable. The differences between Corollary 2 and Theorem 8 appear difficult to bridge.

References

- 1 P. A. Abdulla, L. Boasson, and A. Bouajjani. Effective lossy queue languages. In *ICALP'01*, LNCS, pages 639–651. Springer, 2001.
- 2 P. A. Abdulla, F. Haziza, and L. Holík. All for the price of few. In *VMCAI'13*, LNCS, pages 476–495. Springer, 2013.
- 3 P. A. Abdulla, N. B. Henda, G. Delzanno, F. Haziza, and A. Rezine. Parameterized tree systems. In *FORTE'08*, LNCS, pages 69–83. Springer, 2008.
- 4 P. A. Abdulla and B. Jonsson. Verifying programs with unreliable channels. *Information and Computation*, 127(2):91–101, 1996.
- 5 B. Aminof, T. Kotek, S. Rubin, F. Spegni, and H. Veith. Parameterized model checking of rendezvous systems. In *CONCUR'14*, LNCS, pages 109–124. Springer, 2014.
- 6 M. F. Atig, A. Bouajjani, and S. Qadeer. Context-bounded analysis for concurrent programs with dynamic creation of threads. *Logical Methods in Computer Science*, 7(4):1–48, 2011.
- 7 T. Ball, S. Chaki, and S. K. Rajamani. Parameterized verification of multithreaded software libraries. In *TACAS'01*, LNCS, pages 158–173. Springer, 2001.
- 8 B. Bollig, P. Gastin, and J. Schubert. Parameterized verification of communicating automata under context bounds. In *Reachability problems - International Workshop*, LNCS, pages 45–57. Springer, 2014.
- 9 A. Bouajjani, J. Esparza, S. Schwoon, and J. Strejcek. Reachability analysis of multithreaded software with asynchronous communication. In *FSTTCS'05*, LNCS, pages 348–359. Springer, 2005.
- 10 E. M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks. *ACM Trans. Program. Lang. Syst.*, 19(5):726–750, 1997.
- 11 B. Courcelle. On constructing obstruction sets of words. *Bulletin of EATCS*, 1991.
- 12 G. Delzanno. Parameterized verification and model checking for distributed broadcast protocols. In *ICGT'14*, LNCS, pages 1–16. Springer, 2014.

- 13 J. Esparza. Keeping a crowd safe: On the complexity of parameterized verification. In *STACS'14*, LIPIcs, pages 1–10. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014.
- 14 J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *LICS'99*, pages 352–359. IEEE, 1999.
- 15 J. Esparza, P. Ganty, and R. Majumdar. Parameterized verification of asynchronous shared-memory systems. In *CAV'13*, LNCS, pages 124–140. Springer, 2013.
- 16 S. A. German and P. A. Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992.
- 17 P. Habermehl, R. Meyer, and H. Wimmel. The downward-closure of Petri net languages. In *ICALP'10*, LNCS, pages 466–477. Springer, 2010.
- 18 M. Hague. Parameterised pushdown systems with non-atomic writes. In *FSTTCS'11*, LIPIcs, pages 457–468. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.
- 19 M. Hague, A. S. Murawski, C. L. Ong, and O. Serre. Collapsible pushdown automata and recursion schemes. In *LICS'08*, pages 452–461. IEEE, 2008.
- 20 V. Kahlon. Parameterization as abstraction: A tractable approach to the dataflow analysis of concurrent programs. In *LICS'08*, pages 181–192. IEEE, 2008.
- 21 A. Kaiser, D. Kroening, and T. Wahl. Dynamic cutoff detection in parameterized concurrent programs. In *CAV'10*, LNCS, pages 645–659. Springer, 2010.
- 22 Y. Kesten, A. Pnueli, E. Shahar, and L. D. Zuck. Network invariants in action. In *CONCUR'02*, LNCS, pages 101–115. Springer, 2002.
- 23 S. La Torre, P. Madhusudan, and G. Parlato. Model-checking parameterized concurrent programs using linear interfaces. In *CAV'10*, LNCS, pages 629–644. Springer, 2010.
- 24 S. La Torre, P. Madhusudan, and G. Parlato. Sequentializing parameterized programs. In *FIT'12*, EPTCS, pages 34–47, 2012.
- 25 S. La Torre, M. Napoli, and G. Parlato. Scope-bounded pushdown languages. In *DLT'14*, LNCS, pages 116–128. Springer, 2014.
- 26 S. La Torre, M. Napoli, and G. Parlato. A unifying approach for multistack pushdown automata. In *MFCs'14*, LNCS, pages 377–389. Springer, 2014.
- 27 R. Mayr. Undecidable problems in unreliable computations. *TCS*, 1-3(297):337–354, 2003.
- 28 K. S. Namjoshi and R. J. Treffer. Analysis of dynamic process networks. In *TACAS'15*, LNCS, pages 164–178. Springer, 2015.
- 29 G. Zetsche. An approach to computing downward closures. *CoRR*, abs/1503.01068, 2015.
- 30 G. Zetsche. Computing downward closures for stacked counter automata. In *STACS'15*, LIPIcs, pages 743–756. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

A Proof of Lemma 4

► **Lemma 14.** *There is a #-trace in a $(\mathcal{C}, \mathcal{D})$ -system iff there is a totally \mathcal{C} -supported #-trace in \mathcal{D}^κ .*

Proof. For the left to right direction take a #-trace $a_1 \dots a_n$ and a run ρ of the $(\mathcal{C}, \mathcal{D})$ -system (with $A_{init} = \{s_{init}\}$):

$$(A_{init}, t_{init}, g_{init}) \xrightarrow{a_1} \dots \xrightarrow{a_n} (A_n, t_n, g_n) .$$

Let $\theta(1) < \dots < \theta(k)$ be the indices of the first occurrence of a write $w(g) \in \Sigma_{\mathcal{C}}$ in the word $a_1 \dots a_n$, and denote $w(h_i) = a_{\theta(i)}$ for $i = 1, \dots, k$. By definition, for every $l < \theta(i)$ we have $a_l \neq w(h_i)$. For convenience we also put $\theta(0) = 0$ and $\theta(k+1) = n+1$.

We show that the desired #-trace of \mathcal{D}^κ is

$$v = v_1 \nu(h_1) \dots v_k \nu(h_k) v_{k+1} \quad \text{where} \quad v_j = (a_{\theta(j-1)} \dots a_{\theta(j-1)})|_{\Sigma_{\mathcal{D}}} .$$

To see that v is a trace in \mathcal{D}^κ , recall that reads $\bar{r}(g)$ of \mathcal{D}^κ are enabled either because g is the current register value or because g is in the K -component. If an occurrence of some $\bar{r}(g)$ in v is such that g is not the current register value, then it must be the case that the last write before $\bar{r}(g)$ in the original trace $a_1 \dots a_n$ is of the form $w(g)$, thus the K -component of the state of \mathcal{D}^κ before reading $\bar{r}(g)$ contains g .

To show that v is totally \mathcal{C} -supported, consider an arbitrary prefix $u\nu(h_i)$ of v . We describe how to construct a word u' s.t. $u'|_{\Sigma_{\mathcal{D}, \nu}} = u$ and a run of \mathcal{C}^κ on $u'\nu(h_i)w(h_i)$.

From the transition $(A_{\theta(i)-1}, t_{\theta(i)-1}, g_{\theta(i)-1}) \xrightarrow{w(h_i)} (A_{\theta(i)}, t_{\theta(i)}, g_{\theta(i)})$ of ρ , we get that there are $s \in A_{\theta(i)-1}$ and $s' \in A_{\theta(i)}$ s.t. $s \xrightarrow{w(h_i)} s'$ is a transition of \mathcal{C} . Moreover, there exist a sequence of positions $0 \leq j_1 < \dots < j_l < \theta(i)$ and states $s_0, \dots, s_{l+1} \in S$ with $s_m \in A_{j_m}$ for every m , s.t. $s_0 = s_{init}$, $s_{l+1} = s$ and $s_{m-1} \xrightarrow{a_{j_m}} s_m$ is a transition of \mathcal{C} for $m = 1, \dots, l$.

We take u' as the restriction of $a_1 \dots a_n$ to the positions corresponding to $u \in \Sigma_{\mathcal{D}}^*$ and $\{j_1, \dots, j_l\}$.

Clearly, $u'|_{\Sigma_{\mathcal{D}}} = u$. We show that $u'\nu(h_i)w(h_i) \in L(\mathcal{C}^\kappa)$. By definition of the positions $\theta(j)$, each occurrence of a write $w(h)$ must have $\nu(h)$ to its left in u' . Thus on reading $w(h)$ in u' , if (K, s, g) is the current state of \mathcal{C}^κ , h must already belong to the capacity set K (recall that \mathcal{C}^κ transitions on $\nu(h)$ add h to the capacity set and once in the capacity set a symbol is never removed from it). We can then take the transition that updates the state $s \in S$ in (K, s, g) according to the transitions $s_{m-1} \xrightarrow{a_{j_m}} s_m$ above.

In the remaining case, i.e., on reading $r(h)$ in u' , we just need to argue that the transition is enabled, that is, either h is the current register value or it belongs to the current capacity set K . Assume that h is not the current register value in u' . In other words, the last write before this occurrence of $r(h)$ in u' is either $w(g)$ or $\bar{w}(g)$, with $g \neq h$. Since this $r(h)$ was enabled in $a_1 \dots a_n$, it must be the case that there was some occurrence of $w(h)$ between $w(g)$, or $\bar{w}(g)$, and $r(h)$ in $a_1 \dots a_n$. But then h belongs to the current capacity set K .

For the right to left direction, we take a trace of \mathcal{D}^κ

$$v = v_1 \nu(h_1) \dots v_m \nu(h_m) v_{m+1}$$

of \mathcal{D}^κ , and assume that it is totally \mathcal{C} -supported. This gives us words $u_1^i, \dots, u_i^i \in \Sigma_{\mathcal{C}, \mathcal{D}}^*$ such that for all $i \leq m$:

$$u^i = u_1^i \nu(h_1) \dots u_i^i \nu(h_i) w(h_i) \in L(\mathcal{C}^\kappa) \quad \text{and} \quad u_j^i|_{\Sigma_{\mathcal{D}}} = v_j .$$

We shuffle them together to obtain a trace of the $(\mathcal{C}, \mathcal{D})$ -system. When shuffling we will put reads $r(h)$ in u_j^i that require writes from v_j close to those writes, the other parts of u 's can go arbitrarily. A read $r(h)$ with h in the current capacity set K (see last rule of \mathcal{C}^κ) will be simulated by first inserting a transition of the form $(A, t, g) \xrightarrow{w(h)} (A \cup \{s'\}, t, h)$ for $s \xrightarrow{w(h)} s'$ in δ and $s \in A$. Such a transition always exists since $h \in K$ implies that $\nu(h)$ precedes this occurrence of $r(h)$ and thus the first write $w(h)$ has been already simulated (hence some s'' s.t. $s'' \xrightarrow{w(h)} s'''$ has been already added to the accumulated set A).

Here is a more detailed description. We consider $v_k \in \Sigma_{\mathcal{D}}^*$, for some $k = 1, \dots, m$ and suppose that it is of the form

$$v_k = a_{k,1} \dots a_{k,l}.$$

Then u_k^i can be written as

$$u_k^i = x_{k,1}^i a_{k,1} \dots x_{k,l}^i a_{k,l} x_{k,l+1}^i.$$

Recall now that the value of the register in \mathcal{C}^κ is determined by the last action executed. Let $g_{k,j}$ be the value of the register after $a_{k,j}$ is read (this means that $g_{k,j} = h$ when $a_{k,j}$ is $\bar{w}(h)$ or $\bar{r}(h)$). Then $x_{k,j+1}^i$ may start with a sequence of $r(g_{k,j})$ followed by an action that changes the contents of the register. The latter transition does not depend on the contents of the register. So we have $x_{k,j+1}^i = y_{k,j+1}^i z_{k,j+1}^i$ with $y_{k,j+1}^i$ a sequence of $r(g_{k,j})$, and $z_{k,j+1}^i$ starts either with a write or a capacity read. Also note that $x_{k,1}^i$ follows after $\nu(h_{k-1})$ in u^i , so we can set $g_{k,1} = h_{j-1}$ and the same decomposition as before applies to $x_{k,1}^i = y_{k,1}^i z_{k,1}^i$.

With these preparations we consider the word

$$t_k = (y_{k,1}^k \dots y_{k,1}^m z_{k,1}^k \dots z_{k,1}^m a_{k,1}) \dots (y_{k,l}^k \dots y_{k,l}^m z_{k,l}^k \dots z_{k,l}^m a_{k,l}) \\ (y_{k,l+1}^k \dots y_{k,l+1}^m z_{k,l+1}^k \dots z_{k,l+1}^m)$$

Parentheses above are only added for readability. Let also $t_{m+1} = v_{m+1}$. We construct \bar{t}_j from t_j as follows:

1. replace each occurrence of $r(h)$ by $w(h)r(h)$ when $r(h)$ is a capacity read in the corresponding u^i word;
2. replace each occurrence of $\bar{r}(h)$ by $w(h)\bar{r}(h)$ when $\bar{r}(h)$ is a capacity read in v .

Note that since \mathcal{D}^κ and \mathcal{C}^κ are synchronized on ν -actions, the capacity component evolves in the same way in a run of \mathcal{C}^κ on u^i and one of \mathcal{D}^κ on v .

We claim that $t = \bar{t}_1 \dots \bar{t}_m \bar{t}_{m+1}$ is a $\#$ -trace of the $(\mathcal{C}, \mathcal{D})$ -system in the accumulator semantics. We show this by induction on $k = 1, \dots, m$. Actually we show a stronger assertion: after reading $\bar{t}_1 \dots \bar{t}_k$, the current value A_k of the accumulator contains for each register value $h_j \in G$ with $1 \leq j \leq k$, some state s with $s \xrightarrow{w(h_j)}$.

Let us first consider $k = 1$. By definition of \mathcal{C}^κ , before $\nu(h_1)$ there can be no write action $w(h)$ nor any capacity read. Therefore, all z^1 -words are empty. All reads in y^1 are enabled since they read either the initial value or values from v_1 . Similarly, none of the actions of v_1 is a capacity read, and since the y^1 sequences do not modify the register content, each read $\bar{r}(g)$ of v_1 is still enabled in w_1 . Since $u^1 = u_1^1 \nu(h_1) w(h_1)$ is a trace of \mathcal{C}^κ we know that the accumulator value A_1 after \bar{t}_1 contains some state s with $s \xrightarrow{w(h_1)}$.

By induction we assume that $\bar{t}_1 \dots \bar{t}_k$ is a trace of the $(\mathcal{C}, \mathcal{D})$ -system and that the accumulator value A_k contains for each h_j with $j \in \{1, \dots, k\}$, some state s with $s \xrightarrow{w(h_j)}$. In particular, the writes $w(h)$ that we added when replacing a capacity read $r(h)$ by $w(h)r(h)$

(case 1), or a $\bar{r}(h)$ by $w(h)\bar{r}(h)$ (case 2), are enabled in the $(\mathcal{C}, \mathcal{D})$ -system because $h = h_j$ with $j \leq k$.

We can show now that $\bar{t}_1 \dots \bar{t}_k \bar{t}_{k+1}$ is a trace of the $(\mathcal{C}, \mathcal{D})$ -system. By construction (case 1 above) the prefix of \bar{t}_{k+1} obtained from $y_{k+1,1}^{k+1} \dots y_{k+1,1}^m$ is enabled: recall that all these reads are capacity reads of value h_k . The sequences of reads $y_{k+1,j}^{k+1} \dots y_{k+1,j}^m$ with $j > 1$ are enabled because the value read is determined by $a_{k+1,j-1}$. All sequences of t_{k+1} obtained from some $z_{k+1,j}^{k+1} \dots z_{k+1,j}^m$ are enabled because each non-empty $z_{k+1,j}^{k+1}$ starts with a write or a capacity read. Finally, each $a_{k+1,j}$ is enabled in t_{k+1} if it is a write, or a capacity read (case 2 above). Let us assume that $a_{k+1,j} = \bar{r}(h)$ but h is not in the capacity. Then necessarily, $z_{k+1,j}^{k+1} \dots z_{k+1,j}^m = \varepsilon$ since all u^i are traces of \mathcal{C}^κ and each non-empty $z_{k+1,j}^{k+1}$ starts with a write or a capacity read. Since all reads in $y_{k+1,j}^{k+1} \dots y_{k+1,j}^m$ are $r(g)$ with g the register value after $a_{k+1,j-1}$, and since $a_{k+1,j} = \bar{r}(h)$ is not a capacity read in v , it holds that $g = h$. Thus every $a_{k+1,j}$ is enabled in t for $j > 1$. For $j = 1$ we note the following: since v is a trace of \mathcal{D}^κ , $a_{k+1,1} = \bar{r}(h)$ means that $h = h_j$, so h is in the capacity.

To conclude the induction step, note that since u^{k+1} is a trace in \mathcal{C}^κ there is some state $s \in A_{k+1}$ with $s \xrightarrow{w(h_{k+1})}$. \blacktriangleleft

B Proof of Lemma 5

Proof. Take $v_1\nu(h_1) \dots v_i\nu(h_i)$. As the trace is \mathcal{C} -supported, the definition gives us u_1, \dots, u_i such that $u_1\nu(h_1) \dots u_i\nu(h_i)$ is a trace in \mathcal{C}^κ . We need to find u'_1, \dots, u'_i s.t. $u'_1\nu(h_1) \dots u'_i\nu(h_i)$ is a trace in \mathcal{C}^κ and $u'_i|_{\Sigma_{\mathcal{D}}^*} = v'_i$ for $i = 1, \dots, i$. For this let us look first at $v_1 = a_1 \dots a_l$. We have $u_1 = u_0^1 a_1 u_1^1 \dots a_l u_l^1$. Then since v_1 is a subword of v'_1 , the latter can be presented as $v'_1 = x_0 a_1 x_1 \dots a_l x_l$, for some words $x_j \in \Sigma_{\mathcal{D}}^*$. We take $u'_1 = u_0^1 x_0 a_1 u_1^1 x_1 \dots a_l u_l^1 x_l$. Similarly for v_2, \dots, v_i . Consider the words

$$\begin{aligned} w &= u_1\nu(h_1) \dots u_i\nu(h_i)w(h_i) \\ w' &= u'_1\nu(h_1) \dots u'_i\nu(h_i)w(h_i) \end{aligned}$$

We know that w is a trace of \mathcal{C}^κ and we need to show that w' is, too. Observe that the difference between the two words is only in actions from $\Sigma_{\mathcal{D}}$. These actions are enabled from each state and modify only the register component of \mathcal{C}^κ , neither the K -component nor the state of \mathcal{C} . Moreover, every subword u_i^j is preceded in w' by the same action as in w . Since \mathcal{C}^κ has the property that an action determines the value of the register component, \mathcal{C}^k will be exactly in the same state when starting reading u_i^j in both w and w' . The x parts do not matter since actions from $\Sigma_{\mathcal{D}}$ are always executable in \mathcal{C}^κ . \blacktriangleleft

C Downward closure lemma

► Lemma 15. *Every minimal DFA \mathcal{A} such that $L(\mathcal{A}) = L \downarrow$ for some language L is a DAG with self-loops.*

Proof. Consider \mathcal{A} and L as in the statement. The complement of $L \downarrow$ is an upward closed language, say $K \uparrow$. Moreover, we can assume that K is finite, say $K = \{u_1, \dots, u_k\}$, since the subword order is a well-quasi-order.

A DFA for $K \uparrow$ has as states tuples $\langle v_1, \dots, v_k \rangle$ with v_i prefix of u_i for each i . From state $\langle v_1, \dots, v_k \rangle$ a transition by letter a goes to $\langle v'_1, \dots, v'_k \rangle$, where $v'_i = v_i a$ if it is a prefix of u_i , and $v'_i = v_i$ otherwise. Final states are of the form $\langle v_1, \dots, v_k \rangle$ with $v_i = u_i$ for some i . The DFA has the claimed form and minimizing it preserves the form of a DAG with self-loops, since equivalent states must be unordered in the DAG. \blacktriangleleft

D Proof of Lemma 7

► **Lemma 16.** *There is a totally \mathcal{C} -supported #-trace in \mathcal{D}^κ a totally \mathcal{C} -compatible #-pattern in $\mathcal{D}^\kappa \downarrow$.*

Proof. For the left to right direction, take a totally \mathcal{C} -supported trace in \mathcal{D}^κ . By unrolling the definitions one can check that the pattern is totally \mathcal{C} -compatible.

For the right to left direction we take a totally \mathcal{C} -compatible #-pattern in $\mathcal{D}^\kappa \downarrow$ as in Figure 3.

For $i = 1, \dots, l$ let k_i be the index such that $a_{k_i} = \nu(h_i)$. The definition of compatibility tells us that for every $i = 1, \dots, l$ there is a word

$$w^i = v_1^i a_1 \dots v_{k_i}^i \nu(h_i)$$

that is \mathcal{C} -supported and such that $v_j^i \in \Sigma_j^*$. Consider

$$w = (v_1^1 \dots v_{k_1}^1 a_1) \dots (v_{k_1}^1 \dots v_{k_1}^l \nu(h_1)) (v_{k_1+1}^2 \dots v_{k_1+1}^l a_{k_1+1}) \dots (v_{k_l}^l \nu(h_l))$$

where parenthesis have been added for readability. We have that for every $i = 1, \dots, l$, the word w^i is a subword of the prefix of w up to $\nu(h_i)$ (as in Lemma 5). So by Lemma 5 the prefix of w up to $\nu(h_i)$ is \mathcal{C} -supported for every $i = 1, \dots, l$. Since $v_j^i \in \Sigma_j^*$ for all i, j , the word w is accepted by our fixed pattern. Hence $w \in \mathcal{D}^\kappa \downarrow$. This means that there is a word $w' \in \mathcal{D}^\kappa$ such that w is a subword of w' in the sense of Lemma 5. In consequence, w' is a totally \mathcal{C} -supported trace in \mathcal{D}^κ , and moreover a #-trace. ◀

E Proof of Corollary 9

Proof. We show that under the assumptions of the corollary, $Class_{\mathcal{D}}$ has effective downward closure. Let M be an effective description of a transition system from \mathcal{D} . The algorithm will enumerate all patterns as in Figure 3, and check if $\Sigma_0^* w_1 \dots \Sigma_n^* w_n \subseteq L(M) \downarrow$. If so then it will add the pattern to the set X . Each time a pattern is added to X the algorithm checks if $L(M)$ is contained in the sum of patterns from X . Since each pattern is captured by a deterministic finite automaton, we can easily complement each of them and then construct a deterministic finite automaton for the conjunction of the complement automaton with M . Thus, we take the product with M , and then check the above containment by checking for the emptiness of the resulting transition system (that is still in \mathcal{D}).

So it remains to show how to decide $\Sigma_0^* w_1 \dots \Sigma_n^* w_n \subseteq L(M) \downarrow$. We make a product of the pattern with $L(M)$, and duplicate the letters in the Σ_i . By the closure properties of $Class_{\mathcal{D}}$, the resulting system is still in $Class_{\mathcal{D}}$. Then we take the Parikh image of the result and check that we can have an arbitrary number of occurrences for each of the letters from $\cup_{i=1}^n \Sigma_i$. ◀

F Multistack Pushdown Automata

Multi-stack pushdown automata (MPA) extend pushdown automata by allowing multiple pushdown stores. The general model is Turing powerful (already with two stacks). Recent research on MPA, also motivated by recent developments in the verification of multithreaded programs, has focused on restricted forms of MPA that define robust class of languages that include non-context-free context-sensitive languages (see [25, 26]).

We show the following lemma for MPA.

► **Lemma 17.** *Bounded context-switching, bounded phase, ordered, path-tree and scope-bounded MPA have semilinear Parikh images. Moreover, the class of MPA is closed under rational transductions.*

Proof. The closure under rational transductions can be shown as for pushdown systems: the finite control is modified such that it keeps track of the transduction.

To show the semi-linearity of the Parikh-image we first transform the input alphabet Σ into a visible one s.t. for each $a \in \Sigma$ we add the internal symbol a_t , the push symbols $a_{c,s}$ and the pop symbols $a_{r,s}$ for each stack s . Then we modify the MPA M by replacing each internal move (q, a, q') with (q, a_t, q') , each push move onto stack s (q, a, q', γ) with $(q, a_{c,s}, q', \gamma)$, and each pop move from stack s (q, a, γ, q') with $(q, a_{r,s}, \gamma, q')$. Denote with M' the resulting visibly MPA. By the results from [25, 26], we can construct a context-free grammar G' generating the context-free language that is Parikh equivalent to $L(M')$, then we replace in the productions of G' each occurrence of $a_{r,s}$, $a_{c,s}$ and a_t with a . The resulting grammar G clearly generates a language that is Parikh equivalent to $L(M)$. Thus, the stated result follows from the semi-linearity of Parikh images for context-free languages. ◀

From [29], each class of transition systems that is effectively closed under rational transductions and with semi-linear Parikh-images has an effective subword problem. Therefore, by Lemma 17, we get:

► **Lemma 18.** *Bounded context-switching, bounded phase, ordered, path-tree and scope-bounded MPA have an effective subword problem.*