

Reachability for dynamic parametric processes

Anca Muscholl¹, Helmut Seidl², and Igor Walukiewicz³

¹ LaBRI, Univ. Bordeaux and TUM-IAS

² Fakultät für Informatik, TU München

³ LaBRI, CNRS, Univ. Bordeaux

Abstract. In a dynamic parametric process every subprocess may spawn arbitrarily many, identical child processes, that may communicate either over global variables, or over local variables that are shared with their parent. We show that reachability for dynamic parametric processes is decidable under mild assumptions. These assumptions are e.g. met if individual processes are realized by pushdown systems, or even higher-order pushdown systems. We also provide algorithms for subclasses of pushdown dynamic parametric processes, with complexity ranging between NP and DEXPTIME.

1 Introduction

Programming languages such as Java, Erlang, Scala offer the possibility to generate recursively new threads (or processes, actors,...). Threads may exchange data through globally accessible data structures, e.g. via static attributes of classes like in Java, Scala. In addition, newly created threads may locally communicate with their parent threads, in Java, e.g., via the corresponding thread objects, or via messages like in Erlang.

Various attempts have been made to analyze systems with recursion and dynamic creation of threads that may or may not exchange data. A single thread executing a possibly recursive program operating on finitely many local data, can conveniently be modeled by a *pushdown system*. Intuitively, the pushdown formalizes the call stack of the program while the finite set of states allows to formalize the current program state together with the current values of the local variables. For such systems reachability of a bad state or a regular set of bad configurations is decidable [17, 1]. The situation becomes more intricate if multiple threads are allowed. Already for two pushdown threads reachability is undecidable if communication via a 2-bit global is allowed. In absence of global variables, reachability becomes undecidable already for two pushdown threads if a rendez-vous primitive is available [16]. A similar result holds if finitely many locks are allowed [10]. Interestingly, decidability is retained if locking is performed in a disciplined way. This is, e.g., the case for nested [10] and contextual locking [3]. These decidability results have been extended to dynamic pushdown networks as introduced by Bouajjani et al. [2]. This model combines pushdown threads with dynamic thread creation by means of a *spawn* operation, while it ignores any exchange of data between threads. Indeed, reachability of dedicated states

or even regular sets of configurations stays decidable in this model, if finitely many global locks together with nested locking [12, 14] or contextual locking [13] are allowed. Such regular sets allow, e.g., to describe undesirable situations such as concurrent execution of conflicting operations.

Here, we follow another line of research where models of multi-threading are sought which allow exchange of data via shared variables while still being decidable. The general idea goes back to Kahlon, who observed that various verification problems become decidable for multi-pushdown systems that are *parametric* [9], i.e., systems consisting of an arbitrary number of indistinguishable pushdown threads. Later, Hague extended this result by showing that an extra designated leader thread can be added without sacrificing decidability [7]. All threads communicate here over a shared, bounded register *without* locking. It is crucial for decidability that only one thread has an identity, and that the operations on the shared variable do not allow to elect a second leader. Later, Esparza et al. clarified the complexity of deciding reachability in that model [5]. La Torre et al. generalized these results to hierarchically nested models [11]. Still, the question whether reachability is decidable for *dynamically evolving* parametric pushdown processes, remained open.

We show that reachability is decidable for a very general class of dynamic processes with parametric spawn. We require some very basic properties from the class of transitions systems that underlies the model, like e.g. effective non-emptiness check. In our model every sub-process can maintain e.g. a pushdown store, or even a higher-order pushdown store, and can communicate over global variables, as well as via local variables with its sub-processes and with its parent. As in [7, 5, 11], all variables have bounded domains and no locks are allowed.

Since the algorithm is rather expensive, we also present meaningful instances where reachability can be decided by simpler means. As one such instance we consider the situation where communication between sub-processes is through global variables only. We show that reachability for this model with pushdowns can effectively be reduced to reachability in the parametric model of Hague [7, 5], called (C, D) -systems — giving us a precise characterization of the complexity as PSPACE. As another instance, we consider a parametric variant of *generalized futures* where spawned sub-processes may not only return a single result but create a stream of answers. For that model, we obtain complexities between NP and DEXPTIME. This opens the venue to apply e.g. SAT-solving to check safety properties of such programs.

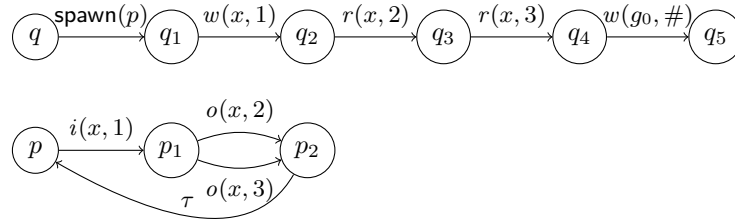
Overview. Section 2 provides basic definitions, and the semantics of our model. In Section 3 we show a simpler semantics, that is equivalent w.r.t. reachability. Section 4 introduces some prerequisites for Section 5, which is the core of the proof of our main result. Section 6 considers the complexity for some special instances of dynamic parametric pushdown processes.

2 Basic definitions

In this section we introduce our model of dynamic parametric processes. We refrain from using some particular program syntax; instead we use potentially infinite state transition systems with actions on transitions. Actions may manipulate local or global variables, or spawn *parametrically* some sub-processes: this means that an unspecified number of sub-processes is created — all with the same designated initial state. Making the spawn operation parametric is the main abstraction step that allows us to obtain decidability results.

Before giving formal definitions we present two examples in order to give an intuitive understanding of the kind of processes we are interested in.

Example 1. A parametric system could, e.g., be defined by an explicitly given finite transition system:



In this example, the root starts in state q by spawning a number of sub-processes, each starting in state p . Then the root writes the value 1 into the local variable x , and waits for some child to change the value of x first to 2, and subsequently to 3. Only then, the root will write value $\#$ into the global variable g_0 . Every child on the other hand, when starting execution at state p , waits for value 1 in the variable x of the parent and then chooses either to write 2 or 3 into x , then returns to the initial state. The read/write operations of the children are denoted as input/output operations $i(x, v)$, $o(x, v)$, because they act on the parent's local. Note that at least two children are required to write $\#$.

More interesting examples require more program states. Here, it is convenient to adopt a programming-like notation as in the next example.

Example 2. Consider the program from Figure 1. The states of the system correspond to the lines in the listing, and $\text{if}(\ast)$ denotes non-deterministic choice. There is a single global variable which is written to by the call `write(#)`, and a single local variable x per sub-process, with initial value 0. The corresponding local of the parent is accessed via the keyword `parent`.

The question is whether the root can eventually write $\#$? This would be the case if the value of the root's local variable becomes 2. This in turn may occur once the variable x of some descendant is set to 1. In order to achieve this, cooperation of several sub-processes is needed. Here is one possible execution.

1. The root spawns two sub-processes in state p , say T_1 and T_2 .

```

1 root() {
    spawn(p);
    switch (x) {
4     case 2:    write(#);
    }
7 }

p() {
    switch (parent.x) {
10    case 0 :    spawn(p);
                if (*) parent.x = 1
                else switch (x) {
13                case 1 : parent.x = 1; break;
                case 2 : break;
                }; break;
16    case 1 :    spawn(p);
                if (*) parent.x = 0
                else switch (x) {
19                case 1: parent.x = 2; break;
                case 2: parent.x = 2; break;
                };
22    }
}

```

Fig. 1. A program defining a dynamic parametric process.

2. T_1 changes the value of the local variable of the root to 1 (line 11).
3. T_2 then can take the **case 1** branch and first spawn T_3 .
4. T_3 takes the **case 0** branch, spawns a new process and changes the value of **parent.x** to 1.
5. As the variable **parent.x** of T_3 is the local variable of T_2 , the latter can now take the second branch of the nondeterministic choice and change **parent.x** to 2 (line 19) — which is the local variable of the root.

□

In the following sections we present a formal definition of our parametric model, state the reachability problem, and the main results. This is done in three steps. In the first subsection, we introduce the syntax that will be given in a form of a transition system, as the one from the first example. Next, we give the formal operational semantics that captures the behavior described in the above examples. Finally, we formulate general requirements on a class of systems and state the result saying that for every class satisfying these requirements, the reachability problem for the associated dynamic parametric processes is decidable.

2.1 Transition systems

A *dynamic parametric process* \mathcal{S} is a transition system over a dedicated set of action names. One can think of it as a control flow graph of a program. In this transition system the action names are uninterpreted. In Section 2.2 we will define their semantics. Such a transition system can be obtained by symbolically executing a program, say, expanding while loops and procedure calls. Another possibility is that the control flow of a program is given by a pushdown automaton; in this case the transition system will have configurations of the pushdown automaton as states.

The transition system is specified by a tuple $\mathcal{S} = \langle Q, G, X, V, \Delta, q_{\text{init}}, v_{\text{init}} \rangle$ consisting of:

- a (possibly infinite) set Q of states,
- finite sets G and X of global and local variables, respectively, and a finite set V of values for variables; these are used to define the set of labels,
- an initial state $q_{\text{init}} \in Q$, and an initial value $v_{\text{init}} \in V$ for variables,
- a set of rules Δ of the form $q \xrightarrow{a} q'$, where the label a is one of the following:
 - τ , that will be later interpreted as a silent action,
 - $r(x, v)$, $w(x, v)$, will be interpreted as a read or a write of value $v \in V$ from or to a local or global variable $x \in X \cup G$ of the process,
 - $i(x, v)$, $o(x, v)$, will be interpreted as a read or a write of value $v \in V$ to or from a local variable $x \in X$ of the parent process,
 - $\text{spawn}(q)$, will be interpreted as a spawn of an arbitrary number (possibly zero) of new sub-processes, all starting in state $q \in Q$. We assume that the number of different $\text{spawn}(q)$ operations appearing in Δ is finite.

Observe that the above definition ensures that the set of labels of transitions is finite.

We are particularly interested in classes of systems when Q is not finite. This is the case when, for example, individual sub-processes execute recursive procedures. For that purpose, the transition system \mathcal{S} may be chosen as a configuration graph of a *pushdown system*. In this case the set Q of states is $Q_l \cdot \Gamma^*$ where Q_l is a finite set of control states, and Γ is a finite set of pushdown symbols. The (infinite) transition relation Δ between states is specified by a finite set of rewriting rules of the form $qv \xrightarrow{a} q'w$ for suitable $q, q' \in Q_l, v \in \Gamma^*, w \in \Gamma^*$.

Instead of plain recursive programs, we could also allow *higher-order* recursive procedures, realized by higher-order pushdown systems or even collapsible pushdown systems as considered, e.g., in [15, 8]. Here, procedures may take other procedures as arguments.

2.2 Multiset semantics

A dynamic parametric process is a transition system with labels of a special form. As we have seen from the examples, such a transition system can be provided either directly, or as the configuration graph of a machine, or as the flow-graph of a program with procedure calls. In this subsection we provide the

operational semantics of programs given by such transition systems, where we interpret the operations on variables as expected, and the spawns as creation of sub-processes. The latter operation will not create one sub-process, but rather an arbitrary number of sub-processes. There will be also a set of global variables to which every sub-process has access by means of reads and writes.

As a dynamic parametric process executes, sub-processes may change the values of local and global variables and spawn new children. The global state of the entire process can be thus represented as a tree of sub-processes with the initial process at the root. Nodes at depth 1 are the sub-processes spawned by the root; these children can also spawn sub-processes that become nodes at depth 2, etc, see e.g., Figure 3(a). Every sub-process has a set of local variables, that can be read and written by itself, as well as by its children.

A global state of a dynamic parametric process \mathcal{S} has the form of a *multiset configuration tree*, or *m-tree* for short. An m-tree is defined recursively by

$$t ::= (q, \lambda, M)$$

where $q \in Q$ is a sub-process state, $\lambda : X \rightarrow V$ is a valuation of (local) variables, and M is a *finite multiset* of m-trees. We consider only m-trees of finite depth. Another way to say this is to define m-trees of depth at most k , for every $k \in \mathbb{N}$

$$\begin{aligned} M\text{-trees}_0 &= Q \times (X \rightarrow V) \times [] \\ M\text{-trees}_k &= Q \times (X \rightarrow V) \times \mathcal{M}(M\text{-trees}_{k-1}) \quad \text{for } k > 0 \end{aligned}$$

where for any U , $\mathcal{M}(U)$ is the set of all finite multisubsets of U . Then the set of all m-trees is given by $\bigcup_{k \in \mathbb{N}} M\text{-trees}_k$.

We use standard notation for multisets. A multiset M over a universe U is a mapping $M : U \rightarrow \mathbb{N}_0$. It is finite if $\sum_{t \in U} M(t) < \infty$. A finite multiset M may also be represented by $M = [n_1 \cdot t_1, \dots, n_k \cdot t_k]$ if $M(t_i) = n_i$ for $i = 1, \dots, k$ and $M(t) = 0$ otherwise. In particular, the empty multiset is denoted by $[]$. For convenience we may omit multiplicities $n_i = 1$. We say that $t \in M$ whenever $M(t) \geq 1$, and $M \subseteq M'$ whenever $M(t) \leq M'(t)$ for all $t \in M'$. Finally, $M + M'$ is the mapping with $(M + M')(t) = M(t) + M'(t)$ for all $t \in U$. For convenience, we also allow the short-cut $[n_1 \cdot t_1, \dots, n_k \cdot t_k]$ for $[n_1 \cdot t_1] + \dots + [n_k \cdot t_k]$, i.e., we allow also multiple occurrences of the same tree in the list. Thus, e.g., $[3 \cdot t_1, 5 \cdot t_2, 1 \cdot t_1] = [4 \cdot t_1, 5 \cdot t_2]$.

The *semantics* of a dynamic parametric process \mathcal{S} is a transition system denoted $\llbracket \mathcal{S} \rrbracket$. The states of $\llbracket \mathcal{S} \rrbracket$ are m-trees, and the set of possible edge labels is:

$$\begin{aligned} \Sigma &= \{\tau\} \cup \{\text{spawn}\} \times Q \cup \\ &\quad \{i(x, v), o(x, v), r(y, v), w(y, v), \bar{r}(y, v), \bar{w}(y, v) : \\ &\quad x \in X, y \in X \cup G, v \in V\}. \end{aligned}$$

Notice that we have two new kinds of labels $\bar{r}(y, v)$ and $\bar{w}(y, v)$. These represent the actions of child sub-processes on global variables $y \in G$, or on the local variables $x \in X$ shared with the parent.

External transitions:

$$\begin{aligned}
(q_1, \lambda, M) &\xrightarrow{a} (q_2, \lambda, M) \text{ if } q_1 \xrightarrow{a} q_2 \quad \text{for } a \in \Sigma_{ext} \\
(q, \lambda, M_1) &\xrightarrow{\bar{r}(g,v)} (q, \lambda, M_2) \text{ if } M_1 \xrightarrow{r(g,v)} M_2 \text{ for } g \in G \\
(q, \lambda, M_1) &\xrightarrow{\bar{w}(g,v)} (q, \lambda, M_2) \text{ if } M_1 \xrightarrow{w(g,v)} M_2 \text{ for } g \in G
\end{aligned}$$

Internal transitions:

$$\begin{aligned}
(q_1, \lambda, M) &\xrightarrow{\tau} (q_2, \lambda, M) \text{ if } q_1 \xrightarrow{\tau} q_2 \\
(q_1, \lambda, M_1) &\xrightarrow{\text{spawn}(p)} (q_2, \lambda, M_2) \text{ if } q_1 \xrightarrow{\text{spawn}(p)} q_2 \text{ and } M_2 = M_1 + [n \cdot (p, \lambda_{\text{init}}, [])] \text{ for some } m \geq 0 \\
(q_1, \lambda, M) &\xrightarrow{w(x,v)} (q_2, \lambda', M) \text{ if } q_1 \xrightarrow{w(x,v)} q_2 \text{ and } \lambda' = \lambda[v/x] \\
(q_1, \lambda, M) &\xrightarrow{r(x,v)} (q_2, \lambda, M) \text{ if } q_1 \xrightarrow{r(x,v)} q_2 \text{ and } v = \lambda(x) \\
(q, \lambda, M_1) &\xrightarrow{\bar{r}(x,v)} (q, \lambda, M_2) \text{ if } M_1 \xrightarrow{i(x,v)} M_2 \text{ and } v = \lambda(x) \\
(q, \lambda, M_1) &\xrightarrow{\bar{w}(x,v)} (q, \lambda', M_2) \text{ if } M_1 \xrightarrow{o(x,v)} M_2 \text{ and } \lambda' = \lambda[v/x]
\end{aligned}$$

Here, we say that

$$M_1 \xrightarrow{a} M_2 \quad \text{for } a \in \Sigma_{ext}$$

if there is a multi-subset $M_1 = M' + [n_1 \cdot t_1, \dots, n_r \cdot t_r]$ (where the t_i need not necessarily be distinct) and executions $t_i \xrightarrow{\alpha_i a} t'_i$ for $i = 1, \dots, r$ for sequences $\alpha_i \in (\Sigma \setminus \Sigma_{ext})^*$ and $M_2 = M' + [n_1 \cdot t'_1, \dots, n_r \cdot t'_r]$.

Fig. 2. Multiset semantics of dynamic parametric processes.

Throughout the paper we will use the notation

$$\Sigma_{ext} = \{i(x, v), o(x, v), r(g, v), w(g, v) : x \in X, g \in G, v \in V\}$$

for the set of so-called *external actions*. They are called external because they concern either the global variables, or the local variables of the parent of the sub-process. Words in Σ_{ext}^* will describe the *external behaviors* of a sub-process, i.e., the interactions with the external world.

The initial state is given by $t_{\text{init}} = (q_{\text{init}}, \lambda_{\text{init}}, [])$, where λ_{init} maps all locals to the initial value v_{init} . A transition between two states of $\llbracket \mathcal{S} \rrbracket$ (m-trees) $t_1 \xrightarrow{a}_{\mathcal{S}} t_2$ is defined by induction on the depth of m-trees. We will omit the subscript \mathcal{S} for better readability. The definition is given in Figure 2.

External transitions (cf. Figure 2) describe operations on external variables, be they global or local. If the actions come from child sub-processes then for technical convenience we add a bar to them. Thanks to adding a bar, a label determines the rule that has been used for the transition (This is important in Prop. 2). The values of global variables are not part of the program state. Accordingly, these operations therefore can be considered as unconstrained input/output actions.

Internal transitions may silently change the current state, spawn new sub-processes or update or read the topmost local variables of the process. The

expression $\lambda[v/x]$ denotes the function $\lambda' : X \rightarrow V$ defined by $\lambda'(x') = \lambda(x')$ for $x' \neq x$ and $\lambda'(x) = v$. In the case of **spawn**, the initial state of the new sub-processes is given by the argument, while the fresh local variables are initialized with the default value. In the last two cases (cf. Figure 2) the external actions $i(x, v)$, $o(x, v)$ of the child sub-processes get relabeled as the corresponding internal actions $\bar{r}(x, v)$, $\bar{w}(x, v)$ on the local variables of the parent.

We write $t_1 \xrightarrow{\alpha} t_2$ for a sequence of transitions complying with the sequence α of action labels. We have chosen the option to allow several child sub-processes to move in one step. While this makes the definition slightly more complicated, it simplifies some arguments later. Observe that the semantics makes the actions (labels) at the top level explicit, while the actions of child sub-processes are explicit only if they refer to globals or affect the local variables of the parent.

2.3 Problem statement and main result

In this section we define the reachability problem and state our main theorem: it says that the reachability problem is decidable for dynamic parametric processes built upon an *admissible* class of systems. The notion of admissible class will be introduced later in this section. Before we do so, we introduce a *consistency* requirement for runs of parametric processes. In our semantics we have chosen not to constrain the operations on global variables. Their values are not stored in the overall state. At some moment, though, we must require that sequences of read/write actions on some global variable $y \in G$ can indeed be realized via reading from and writing to y .

Definition 1 (Consistency). *Let $y \in G$ be a global variable. A sequence $\alpha \in \Sigma_{ext}^*$ is y -consistent if in the projection of α on operations on y , every read action $r(y, v)$ or $\bar{r}(y, v)$ which is not the first operation on y in α is immediately preceded either by $r(y, v)$, $\bar{r}(y, v)$ or by $w(y, v)$ or $\bar{w}(y, v)$. The first operation on y in α can be either $r(y, v_{init})$, $\bar{r}(y, v_{init})$ or $w(y, v)$, $\bar{w}(y, v)$ for some v .*

A sequence α is consistent if it is y -consistent for every variable $y \in G$. Let $Consistent$ be the set of all consistent sequences. As we assume both G and V to be finite, this is a regular language.

Our goal is to decide reachability for dynamic parametric processes.

Definition 2 (Consistent run, reachability). *A run of a dynamic parametric process \mathcal{S} is a path in $\llbracket \mathcal{S} \rrbracket$ starting in the initial state, i.e., a sequence α such that $t_{init} \xrightarrow{\alpha}_{\mathcal{S}} t$ holds. If α is consistent, it is called a consistent run.*

The reachability problem is to decide if for a given \mathcal{S} , there is a consistent run of $\llbracket \mathcal{S} \rrbracket$ containing an external write or an output action of some distinguished value $\#$.

Our definition of reachability talks about a particular value of some variable, and not about a particular state of the process. This choice is common, e.g., reaching a bad state may be simulated by writing a particular value, that is only

possible from bad states. The definition admits not only external writes but also output actions because we will also consider processes without external writes.

We cannot expect the reachability problem to be decidable without any restriction on \mathcal{S} . Instead of considering a particular class of dynamic parametric processes, like those build upon pushdown systems, we will formulate mild conditions on a class of such systems that turn out to be sufficient for deciding the reachability problem. These conditions will be satisfied by the class of pushdown systems, that is our primary motivation. Still we prefer this more abstract approach for two reasons. First, it simplifies notations. Second, it makes our results applicable to other cases as, for example, configuration graphs of higher-order pushdown systems with collapse.

In order to formulate our conditions, we require the notion of *automata*, with possibly infinitely many states. An *automaton* is a tuple:

$$\mathcal{A} = \langle Q, \Sigma, \Delta \subseteq Q \times \Sigma \times Q, F \subseteq Q \rangle$$

where Q is a set of states, Σ is a finite alphabet, Δ is a transition relation, and F is a set of accepting states. Observe that we do not single out an initial state. Apart from the alphabet, all other components may be infinite sets.

We now define what it means for a class of automata to have sufficiently good decidability and closure properties.

Definition 3 (Admissible class of automata). *We call a class \mathcal{C} of automata admissible if it has the following properties:*

- Constructively decidable emptiness check: *For every automaton \mathcal{A} from \mathcal{C} and every state q of \mathcal{A} , it is decidable if \mathcal{A} has some path from q to an accepting state, and if the answer is positive then the sequence of labels of one such path can be computed.*
- Alphabet extension: *There is an effective construction that given an automaton \mathcal{A} from \mathcal{C} , and an alphabet Γ disjoint from the alphabet of \mathcal{A} , produces the automaton $\mathcal{A} \circ \Gamma$ that is obtained from \mathcal{A} by adding a self-loop on every state of \mathcal{A} on every letter from Γ . Moreover, $\mathcal{A} \circ \Gamma$ also belongs to \mathcal{C} .*
- Synchronized product with finite-state systems: *There is an algorithm that from a given automaton \mathcal{A} from \mathcal{C} and a finite-state automaton \mathcal{A}' over the same alphabet, constructs the synchronous product $\mathcal{A} \times \mathcal{A}'$, that belongs to \mathcal{C} , too. The states of the product are pairs of states of \mathcal{A} and \mathcal{A}' ; there is a transition on some letter from such a pair if there is one from both states in the pair. A pair of states (q, q') is accepting in the synchronous product iff q is an accepting state of \mathcal{A} and q' is an accepting state of \mathcal{A}' .*

There are many examples of admissible classes of automata. The simplest is the class of finite automata. Other examples are (configuration graphs of) pushdown automata, higher-order pushdown automata with collapse, VASS with action labels, communicating automata, etc.

Given a dynamic parametric process \mathcal{S} , we obtain an automaton $\mathcal{A}_{\mathcal{S}}$ by declaring all states final. That is, given the transition system $\mathcal{S} = \langle Q, G, X, V, \Delta, q_{\text{init}}, v_{\text{init}} \rangle$

we set $\mathcal{A}_S = \langle Q, \Sigma_{G,X,V}, \Delta, Q \rangle$, where $\Sigma_{G,X,V}$ is the alphabet of actions appearing in Δ . The automaton \mathcal{A}_S is referred to as the *associated automaton* of \mathcal{S} . The main result of this paper is:

Theorem 1. *Let \mathcal{C} be an admissible class of automata. The reachability problem for dynamic parametric processes with associated automata in \mathcal{C} , is decidable.*

As a corollary, we obtain that the reachability problem is decidable for *push-down dynamic parametric processes*, that is where each sub-process is a pushdown automaton. Indeed, in this case \mathcal{C} is the class of pushdown automata. Similarly, we get decidability for dynamic parametric processes with subprocesses being higher-order pushdown automata with collapse, and the other classes listed above.

3 Set semantics

The first step towards deciding reachability for dynamic parametric processes is to simplify the semantics. The idea of using a set semantics instead of a multiset semantics has already been suggested in [9, 4, 11, 5]. We adapt it to our model, and show that the resulting set semantics is equivalent to the multiset semantics — at least as far as the reachability problem is concerned. We conclude this section with several useful properties of runs of our systems that are easy to deduce from the set semantics.

Set configuration trees or *s-trees* for short, are of the form

$$s ::= (q, \lambda, S)$$

where $q \in Q$, $\lambda : X \rightarrow V$, and S is a finite set of s-trees. As in the case of m-trees, we consider only *finite* s-trees. In particular, this means that s-trees necessarily have finite depth. Configuration trees of depth 0 are those where S is empty. The set $S\text{-trees}_k$ of s-trees of depth $k \geq 0$ is defined in a similar way as the set $M\text{-trees}_k$ of multiset configuration trees of depth k .

With a given dynamic parametric process \mathcal{S} , the set semantics associates a transition system $\llbracket \mathcal{S} \rrbracket_s$ with s-trees as states. Its transitions have the same labels as in the case of multiset semantics. Moreover, we will use the same notation as for multiset transitions. It should be clear which semantics we are referring to, as we use t for m-trees and s for s-trees.

As expected, the initial s-tree is $s_{\text{init}} = (q_{\text{init}}, \lambda_{\text{init}}, \emptyset)$.

The transitions are defined as in the multiset case but for multiset actions that become set actions:

$$S \xrightarrow{\text{spawn}(p)} S \cup \{(p, \lambda_{\text{init}}, \emptyset)\} \quad \text{and} \quad S_1 \xrightarrow{a} S_2 \quad \text{if } a \in \Sigma_{\text{ext}}$$

for $S_2 = S_1 \cup B$ where for each $s_2 \in B$ there is some $s_1 \in S_1$ so that $s_1 \xrightarrow{\alpha a} s_2$ for some sequence $\alpha \in (\Sigma \setminus \Sigma_{\text{ext}})^*$.

The reachability problem for dynamic parametric processes under the set semantics asks, like in the multiset case, whether there is some consistent run of $\llbracket \mathcal{S} \rrbracket_s$ that contains an external write or an output of a special value $\#$.

Proposition 1. *The reachability problems of dynamic parametric processes under the multiset and the set semantics, respectively, are equivalent.*

We proceed to show that the set and the multiset semantics are equivalent in the context of reachability.

On s-trees and sets of s-trees, we define inductively the preorder \sqsubseteq by

- $s \sqsubseteq s$;
- if $S \sqsubseteq S'$ then $(q, \lambda, S) \sqsubseteq (q, \lambda, S')$;
- if for all $s \in S$ there is some $s' \in S'$ with $s \sqsubseteq s'$, then $S \sqsubseteq S'$.

The relation \sqsubseteq is reflexive and transitive, but not necessarily anti-symmetric. Thus, it defines an equivalence relation on s-trees.

Every m-tree determines an s-tree by changing multisets to sets:

$$\text{set}((q, \lambda, M)) = (q, \lambda, \{\text{set}(t) : t \in M\})$$

The next two lemmas state a correspondence between multiset and set semantics.

Lemma 1. *For all m-trees t_1, t_2 , multisets M_1, M_2 , s-tree s_1 , and set of s-trees S_1 :*

- If $t_1 \xRightarrow{a} t_2$ and $\text{set}(t_1) \sqsubseteq s_1$ then $s_1 \xRightarrow{a} s_2$ for some s_2 with $\text{set}(t_2) \sqsubseteq s_2$.
- If $M_1 \xRightarrow{a} M_2$ and $\text{set}(M_1) \sqsubseteq S_1$ then $S_1 \xRightarrow{a} S_2$ for some S_2 with $\text{set}(M_2) \sqsubseteq S_2$.

Proof. We will show only the most involved case of multiset transitions. Suppose $M_1 \xRightarrow{a} M_2$. Then we have by definition some subset $B = [n_1 \cdot t_1, \dots, n_r \cdot t_r]$ of M_1 where for $i = 1, \dots, r$, $t_i \xRightarrow{\alpha_i a} t'_i$ holds for a sequence $\alpha_i \in (\Sigma \setminus \Sigma_{ext})^*$, and $M_2 = M_1 \dot{-} B + [n_1 \cdot t'_1, \dots, n_r \cdot t'_r]$. Since $\text{set}(M_1) \sqsubseteq S_1$, we have for all i , some $s_i \in S_1$ with $\text{set}(t_i) \sqsubseteq s_i$. Then by induction assumption $s_i \xRightarrow{\alpha_i a} s'_i$ with $\text{set}(t'_i) \sqsubseteq s'_i$. Taking $S_2 = S_1 \cup \{s'_1, \dots, s'_r\}$ we obtain $\text{set}(M_2) \sqsubseteq S_2$ and $S_1 \xRightarrow{a} S_2$. \square

For the next lemma we introduce the auxiliary notions of n -thick multisets and n -thick m-trees for $n \in \mathbb{N}$. They are defined by mutual recursion. A multiset is n -thick if every element in M is n -thick and appears with the multiplicity at least n (note that $M = []$ is n -thick for every n). An m-tree $t = (q, \lambda, M)$ is n -thick if M is n -thick.

Lemma 2. – If $s_1 \xRightarrow{a} s_2$ for s-trees s_1, s_2 , then there is some factor $m \geq 1$ so that for every $n \geq 1$ and $(m \cdot n)$ -thick t_1 with $\text{set}(t_1) = s_1$, $t_1 \xRightarrow{a} t_2$ holds for some n -thick t_2 with $\text{set}(t_2) = s_2$.

– If $S_1 \xRightarrow{a} S_2$ for sets of s-trees S_1, S_2 , then there is some factor $m \geq 1$ so that for every $n \geq 1$ and $(m \cdot n)$ -thick M_1 with $\text{set}(M_1) = S_1$, $M_1 \xRightarrow{a} M_2$ holds for some n -thick multiset M_2 with $\text{set}(M_2) = S_2$.

Proof. We consider only set transitions. If $a = \text{spawn}(p)$, then $S_2 = S_1 \cup \{(p, \lambda_{\text{init}}, \emptyset)\}$ and we can choose m as 1. Now assume that $S_2 = S_1 \cup \{s'_1, \dots, s'_{m'}\}$ where for each i , there is some $s_i \in S_1$ with $s_i \xrightarrow{\alpha_i a} s'_i$ for some sequence $\alpha_i \in (\Sigma \setminus \Sigma_{\text{ext}})^*$ of actions with corresponding factor m_i . Then define m as the maximum of $m' + 1$ and $m_i, i = 1, \dots, m'$. Consider some M_1 with $\text{set}(M_1) = S_1$ which is $(m \cdot n)$ -thick. Thus, for each i , there is some t_i with $\text{set}(t_i) = s_i$ with $M_1(t_i) \geq mn \geq m_i n$. The induction hypothesis gives us some t'_i with $\text{set}(t'_i) = s'_i$ which is n -thick so that $t_i \xrightarrow{\alpha_i a} t'_i$. We define $M_2 = M_1 \dot{-} [n \cdot t_1, \dots, n \cdot t_{m'}] + [n \cdot t'_1, \dots, n \cdot t'_{m'}]$. Then M_2 is n -thick and $M_1 \xrightarrow{a} M_2$. \square

Corollary 1. *The reachability problems for the multiset and set semantics are equivalent.*

Proof. Lemma 1 implies that if $t_{\text{init}} \xrightarrow{\alpha} t$ for some sequence α and some t then $s_{\text{init}} \xrightarrow{\alpha} s$ for some s .

For the opposite direction take an execution $s_{\text{init}} \xrightarrow{\alpha} s$ for some s . By definition, t_{init} is m -thick for every $m \geq 1$. Then Lemma 2 gives us an execution $t_{\text{init}} \xrightarrow{\alpha} t$. \square

4 External sequences and signatures

In this section we define some useful languages describing the behavior of dynamic parametric processes. Since our constructions and proofs will proceed by induction on the depth of s-trees, we will be particularly interested in sequences of external actions of subtrees of processes, and in signatures of such sequences, as defined below. Recall the definition of the alphabet of external actions Σ_{ext} (see page 7). Other actions of interest are the spawns occurring in \mathcal{S} :

$$\Sigma_{sp} = \{\text{spawn}(p) : \text{spawn}(p) \text{ is a label of a transition in } \mathcal{S}\}$$

Recall that according to our definitions, Σ_{sp} is finite.

For a sequence of actions α , let $\text{ext}(\alpha)$ be the subsequence of external actions in α , with additional renaming of \bar{w} , and \bar{r} actions to actions without a bar, if they refer to global variables $g \in G$:

$$\text{ext}(a) = \begin{cases} r(g, v) & \text{if } a = r(g, v) \text{ or } a = \bar{r}(g, v) \\ w(g, v) & \text{if } a = w(g, v) \text{ or } a = \bar{w}(g, v) \\ a & \text{if } a = i(x, v) \text{ or } a = o(x, v) \\ \epsilon & \text{otherwise} \end{cases}$$

Let $\xrightarrow{\alpha}_k$ stand for the restriction of $\xrightarrow{\alpha}$ to s-trees of depth at most k (the trees of depth 0 have only the root). This allows to define a family of languages of *external behaviors* of trees of processes of height k . This family will be the main object of our study.

$$\text{Ext}_k = \{\text{spawn}(p) \text{ ext}(\alpha) : (p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha}_k s \text{ for some } s, \text{spawn}(p) \in \Sigma_{sp}\}$$

The following definitions introduce abstraction and concretization operations on (sets of) sequences of external actions. The abstraction operation extracts from a sequence its *signature*, that is, the subsequence of first occurrences of external actions:

Definition 4 (Signature, canonical decomposition). *The signature of a word $\alpha \in \Sigma_{ext}^*$, denoted $\text{sig}(\alpha)$, is the subsequence of first appearances of actions in α .*

For a word α with signature $\text{sig}(\alpha) = b_0 b_1 \dots b_k$, the (canonical) decomposition is $\alpha = b_0 \alpha_1 b_1 \alpha_2 b_2 \dots \alpha_k b_k \alpha_{k+1}$, where b_i does not appear in $\alpha_1 \dots \alpha_i$, for all i .

For words $\beta \in \Sigma_{sp} \cdot \Sigma_{ext}^$ the signature is defined by $\text{sig}(\text{spawn}(p)\alpha) = \text{spawn}(p) \cdot \text{sig}(\alpha)$.*

The above definition implies that α_1 consists solely of repetitions of b_0 . In Example 2 the signatures of the executions at level 1 are $\text{spawn}(p)i(x,0)o(x,1)$, $\text{spawn}(p)i(x,1)o(x,2)$, and $\text{spawn}(p)i(x,1)o(x,0)$. Observe that all signatures at level 1 in this example are prefixes of the above signatures.

While the signature operation removes actions from a sequence, the concretization operation *lift* inserts them in all possible ways.

Definition 5 (lift). *Let $\alpha \in \Sigma_{ext}^*$ be a word with signature b_0, \dots, b_n and canonical decomposition $\alpha = b_0 \alpha_1 b_1 \alpha_2 b_2 \dots \alpha_k b_k \alpha_{k+1}$. A lift of α is any word $\beta = b_0 \beta_1 b_1 \beta_2 b_2 \dots \beta_k b_k \beta_{k+1}$, where β_i is obtained from α_i by inserting some number of actions b_0, \dots, b_{i-1} , for $i = 1, \dots, k+1$. We write $\text{lift}(\alpha)$ for the set of all such words β . For a set $L \subseteq \Sigma_{ext}^*$ we define*

$$\text{lift}(L) = \bigcup \{\text{lift}(\alpha) : \alpha \in L\}$$

We also define $\text{lift}(\text{spawn}(p) \cdot \alpha)$ as the set $\text{spawn}(p) \cdot \text{lift}(\alpha)$, and similarly $\text{lift}(L)$ for $L \subseteq \Sigma_{sp} \cdot \Sigma_{ext}^$.*

Observe that $\alpha \in \text{lift}(\text{sig}(\alpha))$. Another useful property is that if $\beta \in \text{lift}(\alpha)$ then α, β agree in their signatures.

5 Systems under hypothesis

This section presents the proof of our main result, namely, Theorem 1 stating that the reachability problem for dynamic parametric processes is decidable for an admissible class of systems. The corresponding algorithm will analyze a process tree level by level. The main tool is an abstraction of child sub-processes by their external behaviors. We call it *systems under hypothesis*.

Let us briefly outline this idea. A configuration of a dynamic parametric process is a tree of sub-processes, Figure 3(a). The root performs (1) input/output external operations, (2) read/writes to global variables, and (3) internal operations in form of reads/writes to its local variables, that are also accessible to the

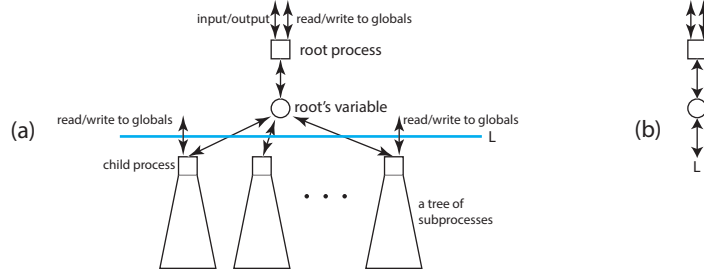


Fig. 3. Reduction to a system under hypothesis.

child sub-processes. We are now interested in possible sequences of operations on the global variables and the local variables of the root, that can be done by the child sub-processes. If somebody provided us with the set L_p of all such possible sequences, for child sub-processes starting at state p , for all p , we could simplify our system as illustrated in Figure 3(b). We would replace the set of all sub-trees of the root by (a subset of) $L = \{\text{spawn}(p)\beta : \beta \in \text{pref}(L_p), \text{spawn}(p) \in \Sigma_{sp}\}$ summarizing the possible behaviors of child sub-processes.

A set $L \subseteq \Sigma_{sp} \cdot \Sigma_{ext}^*$ is called a *hypothesis*, as it represents a guess about the possible behaviors of child sub-processes.

Let us now formalize the notion of *execution of the system under hypothesis*. For that, we define a system \mathcal{S}_L that cannot spawn child sub-processes, but instead may use the hypothesis L . We will show that if L correctly describes the behavior of child sub-processes then the set of runs of \mathcal{S}_L equals the set of runs of \mathcal{S} with child sub-processes. This approach provides a way to compute the set of possible external behaviors of the original process tree level-wise: first for systems restricted to s-trees of height at most 1, then 2, \dots , until a fixpoint is reached.

The configurations of \mathcal{S}_L are of the form (q, λ, B) , where λ is as before a valuation of local variables, and $B \subseteq \text{pref}(L)$ is a set of sequences of external actions for sets of sub-processes.

The initial state is $r_{\text{init}} = (q_{\text{init}}, \lambda_{\text{init}}, \emptyset)$. We will use r to range over configurations of \mathcal{S}_L . Transitions between two states $r_1 \xrightarrow{a}_L r_2$ are listed in Figure 4. Notice that transitions on actions of child sub-processes are modified so that now L is used to test if an action of a simulated child sub-process is possible.

We list below two properties of $\xrightarrow{\cdot}_L$. In order to state them in a convenient way, we introduce a *filtering* operation filter on sequences. The point is that external actions of child sub-processes are changed to \bar{r} and \bar{w} , when they are exposed at the root of a configuration tree. In the definition below we rename them back; additionally, we remove irrelevant actions. So $\text{filter}(\alpha)$ is obtained by

External transitions under hypothesis:

$$\begin{aligned}
(q_1, \lambda, B) &\xrightarrow{a}_L (q_2, \lambda, B) && \text{if } q_1 \xrightarrow{a} q_2 \text{ if } a \in \Sigma_{ext} \\
(q, \lambda, B) &\xrightarrow{\overline{w}(g,v)}_L (q, \lambda, B \cup B' \cdot \{w(g, v)\}) && \text{if } \emptyset \neq B' \subseteq B, B' \cdot \{w(g, v)\} \subseteq \text{pref}(L) \\
(q, \lambda, B) &\xrightarrow{\overline{r}(g,v)}_L (q, \lambda, B \cup B' \cdot \{r(g, v)\}) && \text{if } \emptyset \neq B' \subseteq B, B' \cdot \{r(g, v)\} \subseteq \text{pref}(L)
\end{aligned}$$

Internal transitions under hypothesis:

$$\begin{aligned}
(q_1, \lambda, B) &\xrightarrow{\tau}_L (q_2, \lambda, B) && \text{if } q_1 \xrightarrow{\tau} q_2 \\
(q_1, \lambda, B) &\xrightarrow{\text{spawn}(p)}_L (q_2, \lambda, B \cup \{\text{spawn}(p)\}) && \text{if } q_1 \xrightarrow{\text{spawn}(p)} q_2 \text{ and } \text{spawn}(p) \in \text{pref}(L) \\
(q_1, \lambda, B) &\xrightarrow{w(x,v)}_L (q_2, \lambda', B) && \text{if } q_1 \xrightarrow{w(x,v)} q_2 \text{ and } \lambda' = \lambda[v/x] \\
(q_1, \lambda, B) &\xrightarrow{r(x,v)}_L (q_2, \lambda, B) && \text{if } q_1 \xrightarrow{r(x,v)} q_2 \text{ and } \lambda(x) = v \\
(q, \lambda, B) &\xrightarrow{\overline{w}(x,v)}_L (q, \lambda', B \cup B' \cdot \{o(x, v)\}) && \text{if } \emptyset \neq B' \subseteq B, B' \cdot \{o(x, v)\} \subseteq \text{pref}(L), \\
&&& \lambda' = \lambda[v/x] \\
(q, \lambda, B) &\xrightarrow{\overline{r}(x,v)}_L (q, \lambda, B \cup B' \cdot \{i(x, v)\}) && \text{if } \emptyset \neq B' \subseteq B, B' \cdot \{i(x, v)\} \subseteq \text{pref}(L), \\
&&& \lambda(x) = v
\end{aligned}$$

Fig. 4. Transitions under hypothesis ($g \in G, x \in X$).

the following renaming of α :

$$\begin{array}{ll}
\text{filter :} & \begin{array}{ll} \overline{r}(x, v) \rightarrow i(x, v), & \overline{r}(g, v) \rightarrow r(g, v), \\ \overline{w}(x, v) \rightarrow o(x, v), & \overline{w}(g, v) \rightarrow w(g, v), \\ a \rightarrow a & \text{if } a \in \Sigma_{sp}, \\ a \rightarrow \epsilon & \text{otherwise} \end{array}
\end{array}$$

The next two lemmas follow directly from the definition of $\xrightarrow{\alpha}_L$.

Lemma 3. *If $(q, \lambda, \emptyset) \xrightarrow{\alpha}_L (q', \lambda', B)$ then $B \subseteq \text{pref}(L)$, and every $\beta \in B$ is a scattered subword of $\text{filter}(\alpha)$.*

Lemma 4. *If $L_1 \subseteq L_2$ and $(p, \lambda, \emptyset) \xrightarrow{\alpha}_{L_1} r$ then $(p, \lambda, \emptyset) \xrightarrow{\alpha}_{L_2} r$.*

The next lemma states a basic property of the relation $\xrightarrow{\alpha}_L$. If we take for L the set of all possible behaviors of child sub-processes with s-trees of height at most k , then $\xrightarrow{\alpha}_L$ gives us all possible behaviors of a system with s-trees of height at most $k + 1$. This corresponds exactly to the situation depicted in Figure 3.

Lemma 5. *Suppose $L = \text{Ext}_k$. For every p, q, λ , and α we have: $(p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha}_L (q, \lambda, B)$ for some B iff $(p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha}_{k+1} (q, \lambda, S)$ for some S .*

Proof. The statement may look almost tautological, but is not. We prove two directions:

1. Whenever $(p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha}_{k+1} (q, \lambda, S)$, then there is some B so that $(p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha}_L (q, \lambda, B)$ holds.

2. Whenever $(p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha}_L (q, \lambda, B)$, then there is some finite set S of s -trees of depth at most k so that $(p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha}_{k+1} (q, \lambda, S)$.

Statement 1 follows by induction on the length of α where the set B satisfies the invariant that for every $s \in S$, there is some $\beta = \text{spawn}(p')\beta'$ in B such that $(p', \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha'}_k s$ holds for some α' , with $\text{ext}(\alpha') = \beta'$.

The proof of statement 2 is more technical. Assume that $\alpha = a_1 \cdots a_n$ and $(p_i, \lambda_i, B_i) \xrightarrow{a_i}_L (p_{i+1}, \lambda_{i+1}, B_{i+1})$ for $i = 1, \dots, n$, where $(p_1, \lambda_1, B_1) = (p, \lambda_{\text{init}}, \emptyset)$ and $(p_{n+1}, \lambda_{n+1}, B_{n+1}) = (q, \lambda, B)$. We construct below a corresponding sequence of sets of s -trees S_1, \dots, S_{n+1} with $(p_i, \lambda_i, S_i) \xrightarrow{a_i}_k (p_{i+1}, \lambda_{i+1}, S_{i+1})$, $i = 1, \dots, n$.

Since $B_{n+1} \subseteq \text{pref}(\text{Ext}_k)$, we have for every $\beta = \text{spawn}(p_\beta)\beta' \in B_{n+1}$ that $(p_\beta, \lambda_{\text{init}}, \emptyset) \xrightarrow{\gamma_\beta}_k (q_\beta, \lambda, S_\beta)$, for some γ_β with $\text{ext}(\gamma_\beta) = \beta'$. This means for $\beta' = b_{\beta,1} \cdots b_{\beta,n_\beta}$ that γ_β has the form: $\gamma_\beta = \gamma_{\beta,1}b_{\beta,1} \cdots \gamma_{\beta,n_\beta}b_{\beta,n_\beta}$ with $\text{ext}(\gamma_{\beta,j}) = \varepsilon$ for all j . Moreover, there are s -trees $s_{\beta,j}$ of depth at most k so that $s_{\beta,j} \xrightarrow{\gamma_{\beta,j}b_{\beta,j}}_k s_{\beta,j+1}$ for $j = 1, \dots, n_\beta$ with $s_{\beta,1} = (p_\beta, \lambda_{\text{init}}, \emptyset)$ and $s_{\beta,n_\beta+1} = (q_\beta, \lambda, S_\beta)$. Then we define S_i as the set of all $s_{\beta,j}$ such that $\text{spawn}(p_\beta)b_{\beta,1} \cdots b_{\beta,j-1}$ is a scattered subword of $\text{filter}(a_1 \cdots a_{i-1})$.

It remains to prove that $(p_i, \lambda_i, S_i) \xrightarrow{a_i}_k (p_{i+1}, \lambda_{i+1}, S_{i+1})$ holds for every $i = 1, \dots, n$. For that we perform a case distinction on action a_i . If a_i is of the form $\tau, r(g, v), w(g, v), i(x, v)$ or $o(x, v)$, then $B_{i+1} = B_i$, $S_{i+1} = S_i$ and the assertion holds. If a_i is the action $\text{spawn}(p')$, then $B_{i+1} = B_i \cup \{\text{spawn}(p')\}$. Likewise, $S_{i+1} = S_i \cup \{(p', \lambda_{\text{init}}, \emptyset)\}$ in accordance with our claim.

The most complicated case is when a_i is of one of the forms $\bar{r}(y, v), \bar{w}(y, v)$ for $y \in X \cup G$ and $v \in V$. Then $B_{i+1} = B_i \cup B' \cdot \{b\}$ for $b = \text{filter}(a_i)$ and some $B' \subseteq B_i$ with $B' \cdot \{b_i\} \subseteq B_{n+1} \subseteq \text{pref}(L)$. Let S be the set consisting of all $s_{\beta,j+1}$ such that $\text{spawn}(p_\beta)b_{\beta,1} \cdots b_{\beta,j}$ is a scattered subword of $\text{filter}(a_1 \cdots a_{i-1})$ and $b_{\beta,j} = b$. This set S is not empty since B' is not empty. We have $S_{i+1} = S_i \cup S$ by the definition of S_i and S_{i+1} . Now for every $s_{\beta,j+1} \in S$ we have by definition $s_{\beta,j} \xrightarrow{\gamma_{\beta,j}b_{\beta,j}}_k s_{\beta,j+1}$, and $b_{\beta,j} = b$. Since $\text{spawn}(p_\beta)b_{\beta,1} \cdots b_{\beta,j-1}$ is a scattered subword of $\text{filter}(a_1 \cdots a_{i-1})$, we have $s_{\beta,j} \in S_i$. This shows that we have indeed $(p_i, \lambda_i, S_i) \xrightarrow{a_i}_k (p_{i+1}, \lambda_{i+1}, S_i \cup S)$, with $S_i \cup S = S_{i+1}$. \square

The question we will pursue now is whether in the lemma above, we may replace Ext_k with some simpler set and still get all computations of the system of height $k+1$. Of central importance here is the following lemma saying that the lift operation (cf. Definition 5) does not add new behaviours.

Lemma 6. *Assume that $L \subseteq \Sigma_{sp} \cdot \Sigma_{ext}^*$ and $L' = \text{lift}(L)$. Then $(p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha}_L (q, \lambda, B)$ for some $B \subseteq \text{pref}(L)$ iff $(p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha}_{L'} (q, \lambda, B')$ for some $B' \subseteq \text{pref}(L')$.*

Proof. The left-to-right direction is obvious by monotonicity, since $L \subseteq L'$.

We focus on the right-to-left direction. The main idea is that the first occurrences of actions in sequences from B suffice to simulate any sequence from B' .

Assume that $\alpha = a_1 \cdots a_n$, and $(p_i, \lambda_i, B'_i) \xrightarrow{a_i}_{L'} (p_{i+1}, \lambda_{i+1}, B'_{i+1})$ for $i = 1, \dots, n$ where $(p_1, \lambda_1, B'_1) = (p, \lambda_{\text{init}}, \emptyset)$ and $(p_{n+1}, \lambda_{n+1}, B'_{n+1}) = (q, \lambda, B')$. For $i = 1, \dots, n+1$, we define a subset $B_i \subseteq \text{pref}(L)$ by $B_i = \{\beta \in \text{pref}(L) : \text{lift}(\beta) \cap B'_i \neq \emptyset\}$. By case distinction, we verify that indeed $(p_i, \lambda_i, B_i) \xrightarrow{a_i}_L (p_{i+1}, \lambda_{i+1}, B_{i+1})$ holds for $i = 1, \dots, n$. If a_i is either $\tau, r(x, v), w(x, v)$ or in Σ_{ext} , then $B'_{i+1} = B'_i$, $B_{i+1} = B_i$, and the assertion holds inductively. If a_i is the action $\text{spawn}(p')$, then $B'_{i+1} = B'_i \cup \{\text{spawn}(p')\}$, $B_{i+1} = B_i \cup \{\text{spawn}(p')\}$, and the assertion holds, again by induction.

It remains to consider the case where a_i is $\bar{r}(y, v)$ or $\bar{w}(y, v)$, where $y \in X \cup G$, $v \in V$. Let $b = \text{filter}(a_i)$, and $B'_{i+1} = B'_i \cup B' \cdot \{b\}$ for some $\emptyset \neq B' \subseteq B'_i$ with $B' \cdot \{b\} \subseteq \text{pref}(L')$. We need to find some $B \subseteq B_i$ such that $B_{i+1} = B_i \cup B \cdot \{b\}$.

Let $B = \{\beta \in B_i : \text{lift}(\beta) \cap B' \neq \emptyset\}$. We claim that $B_i \cup B \cdot \{b\} = B_{i+1}$. To show $B_i \cup B \cdot \{b\} \subseteq B_{i+1}$ we argue that for every $\beta \in B$, $\text{lift}(\beta b) \cap B' b \neq \emptyset$. Conversely, let $\beta' b \in B' \{b\}$, and $\beta \in B_{i+1}$ such that $\beta' b \in \text{lift}(\beta)$. We need to show that $\beta \in B_i \cup B \{b\}$. In particular, we know that $\beta \in \text{pref}(L)$. Either β is of the form $\beta = \beta_1 b$ and β_1 has no b . Thus, $\beta' \in \text{lift}(\beta_1)$, so $\beta_1 \in B$ and $\beta \in B \{b\}$. Or $\beta = \beta_1 b \beta_2$ with $\beta' \in \text{lift}(\beta)$. Since $\beta' \in B'_i$ we have in this case $\beta \in B_i$ by definition. \square

So Lemma 5 says that child sub-processes can be abstracted by their external behaviors. Lemmas 4 and 6 allow to abstract a set L of external behaviors by a subset $L_1 \subseteq L$, as long as $L \subseteq \text{lift}(L_1)$ holds. In the following, we introduce a *well-quasi-order* to characterize a smallest such subset, which we call *core*.

Definition 6 (Order, core). We define an order on Σ_{ext}^* by $\alpha \preceq \beta$ if $\beta \in \text{lift}(\alpha)$. This extends to an order on $\Sigma_{\text{sp}} \cdot \Sigma_{\text{ext}}^*$: $\text{spawn}(p)\alpha \preceq \text{spawn}(q)\beta$ if $p = q$ and $\alpha \preceq \beta$. For a set $L \subseteq \Sigma_{\text{sp}} \cdot \Sigma_{\text{ext}}^*$, we define $\text{core}(L)$ as the set of minimal words in L with respect to the relation \preceq .

The following lemma states the most important property of the order \preceq :

Lemma 7. The relation \preceq is a well-quasi-order on words with equal signature. Since the number of signatures is finite, the set $\text{core}(L)$ is finite for every set $L \subseteq \Sigma_{\text{sp}} \cdot \Sigma_{\text{ext}}^*$.

Proof. We spell out what it means that $\alpha \preceq \beta$, by expanding the definition of $\text{lift}(\alpha)$. First recall that if $\alpha \preceq \beta$ then the two sequences have the same signatures. Let $\alpha = \text{spawn}(p)\alpha'$ and $\beta = \text{spawn}(p)\beta'$, $\text{sig}(\alpha') = \text{sig}(\beta') = b_1 \cdots b_k$ for some p . Consider the canonical decompositions of α', β' :

$$\alpha' = b_1 \alpha'_1 b_2 \alpha'_2 \cdots b_k \alpha'_k, \quad \beta' = b_1 \beta'_1 b_2 \beta'_2 \cdots b_k \beta'_k.$$

We have $\alpha \preceq \beta$ iff α'_i is a scattered subword of β'_i , for every $i = 1, \dots, k+1$. Since being a scattered subword is a well-quasi-order relation, the lemma follows. \square

Consider, e.g., the set $L = Ext_1$ of all external behaviors of depth 1 in Example 1. Then $core(L)$ consists of the sequences:

$spawn(q) w(g_0, \#), spawn(p) i(x, 1) o(x, 2) o(x, 3), spawn(p) i(x, 1) o(x, 3) o(x, 2)$

together with all their prefixes (recall that k in Ext_k refers to s-trees of depth *at most* k).

The development till now can be summarized by the following:

Corollary 2. *For a set $L \subseteq \Sigma_{sp} \cdot \Sigma_{ext}^*$, and $L' = core(L)$: $(p, \lambda_{init}, \emptyset) \xrightarrow{\alpha}_L (q, \lambda, B)$ for some $B \subseteq L$ iff $(p, \lambda_{init}, \emptyset) \xrightarrow{\alpha}_{L'} (q, \lambda, B')$ for some $B' \subseteq L'$.*

Proof. Since $core(L) \subseteq L$, the right-to-left implication follows by monotonicity. For the other direction we observe that $L \subseteq lift(core(L))$, so we can use Lemma 6 and monotonicity. \square

Now we turn to the question of computing the relation $\xrightarrow{\alpha}_L$ for a *finite* set L . For this we need our admissibility assumptions from page 9.

Proposition 2. *Let \mathcal{C} be an admissible class of automata, and let \mathcal{S} be a transition system whose associated automaton is in \mathcal{C} . Suppose we have two sets $L, L' \subseteq \Sigma_{sp} \cdot \Sigma_{ext}^*$ with $L \subseteq L' \subseteq lift(L)$. Consider the set*

$$K = \{spawn(p)ext(\alpha) : spawn(p) \in \Sigma_{sp} \text{ and } (p, \lambda_{init}, \emptyset) \xrightarrow{\alpha}_{L'} r', \text{ for some } r'\}$$

determined by \mathcal{S} and L' . If L is finite then we can compute the sets

$$core(K) \quad \text{and} \quad core(\{\alpha \in K : \alpha \text{ consistent}\}) .$$

The proof of the above proposition works by augmenting the transition system \mathcal{S} by a finite-state component taking care of the valuation of local variables and of prefixes of L that were used in the hypothesis. The admissibility of \mathcal{C} is then used to compute the core of the language of the automaton thus obtained.

Proof. By Lemmas 4 and 6, the relations $\xrightarrow{\alpha}_L$ and $\xrightarrow{\alpha}_{L'}$ are the same. Since L is finite, there are only finitely many sets $B \subseteq L$. Moreover, the number of valuations λ is finite by our initial definitions. This allows to construct a finite automaton \mathcal{A} , whose states are pairs (λ, B) and transitions are as those of $\xrightarrow{\alpha}_L$ but without the first component:

$$\begin{aligned} (\lambda, B) &\xrightarrow{\overline{w}(g,v)} (\lambda, B \cup B' \cdot \{w(g, v)\}) \text{ if } \emptyset \neq B' \subseteq B, B' \cdot \{w(g, v)\} \subseteq \text{pref}(L), \\ (\lambda, B) &\xrightarrow{\overline{r}(g,v)} (\lambda, B \cup B' \cdot \{r(g, v)\}) \text{ if } \emptyset \neq B' \subseteq B, B' \cdot \{r(g, v)\} \subseteq \text{pref}(L) \\ (\lambda, B) &\xrightarrow{spawn(p)} (\lambda, B \cup \{spawn(p)\}) \quad \text{if } spawn(p) \in \text{pref}(L) \\ (\lambda, B) &\xrightarrow{w(x,v)} (\lambda[v/x], B) \\ (\lambda, B) &\xrightarrow{r(x,v)} (\lambda, B) \quad \text{if } \lambda(x) = v \\ (\lambda, B) &\xrightarrow{\overline{w}(x,v)} (\lambda[v/x], B \cup B' \cdot \{o(x, v)\}) \text{ if } \emptyset \neq B' \subseteq B, B' \cdot \{o(x, v)\} \subseteq \text{pref}(L) \\ (\lambda, B) &\xrightarrow{\overline{r}(x,v)} (\lambda, B \cup B' \cdot \{i(x, v)\}) \text{ if } \lambda(x) = v, \emptyset \neq B' \subseteq B, B' \cdot \{i(x, v)\} \subseteq \text{pref}(L) \\ (\lambda, B) &\xrightarrow{a} (\lambda, B) \quad \text{for all } a \in \Sigma_{ext} \end{aligned}$$

Now consider the automaton $\mathcal{A}_{\mathcal{S}}$ associated to \mathcal{S} . This automaton belongs to our admissible class \mathcal{C} , so its alphabet extension is also in \mathcal{C} :

$$\mathcal{A}'_{\mathcal{S}} = \mathcal{A}_{\mathcal{S}} \odot \{\bar{r}(y, v), \bar{w}(y, v) : y \in X \cup G, v \in V\}.$$

Intuitively, we add to $\mathcal{A}_{\mathcal{S}}$ self-loops on actions that are in \mathcal{A}_L but not in $\mathcal{A}_{\mathcal{S}}$. Finally, consider the product $\mathcal{A}^K = \mathcal{A}'_{\mathcal{S}} \times \mathcal{A}_L$. We have that for every pair of states q, q' , valuations λ, λ' , and sets B, B' : $(q, \lambda, B) \xrightarrow{-\alpha}_{\rightarrow L} (q', \lambda, B')$ iff there is a path labeled α from (q, λ, B) to (q', λ, B') in \mathcal{A}^K . Then $\text{spawn}(p)\alpha \in K$ iff there is path in \mathcal{A}^K from $(p, \lambda_{\text{init}}, \emptyset)$ labeled by some α' with $\text{ext}(\alpha') = \alpha$.

The above paragraph says that in order to compute $\text{core}(K)$ it is enough to compute $\text{core}(\text{ext}(K_p))$ where K_p is the set of labels of runs of \mathcal{A}^K from $(p, \lambda_{\text{init}}, \emptyset)$. Since \mathcal{A}^K belongs to our admissible class \mathcal{C} , we can use the effective emptiness test. If the language of \mathcal{A}^K is not empty then we can find a word α_1 that is accepted from $(p, \lambda_{\text{init}}, \emptyset)$. Next we look for $\beta \in \text{core}(\text{ext}(K_p))$ with $\beta \preceq \text{ext}(\alpha_1)$. To this end, for every $\beta \preceq \text{ext}(\alpha_1)$ we consider an automaton \mathcal{A}^β accepting all the words α' such that $\text{ext}(\alpha') = \beta$. We build the product of \mathcal{A}^K with \mathcal{A}^β and check for emptiness. Then we choose one minimal β_1 for which this product is non-empty.

To find a next word from $\text{core}(\text{ext}(K_p))$ we construct a finite automaton \mathcal{N}_{β_1} accepting all words α' such that $\beta_1 \not\preceq \text{ext}(\alpha')$. Then we consider $\mathcal{A}^K \times \mathcal{N}_{\beta_1}$ instead of \mathcal{A}^K . If the language accepted by $\mathcal{A}^K \times \mathcal{N}_{\beta_1}$ is not empty then we get a word α_2 in the language. We apply the above procedure to α_2 , iterating through all words $\beta \preceq \text{ext}(\alpha_2)$ and checking if the language of $\mathcal{A}^K \times \mathcal{A}_{\beta_1}^\beta$ is empty; here $\mathcal{A}_{\beta_1}^\beta$ is a finite automaton accepting all words α' such that $\text{ext}(\alpha') = \beta$ and $\alpha' \in \mathcal{N}_{\beta_1}$. We choose one minimal β_2 for which such a product is non-empty. For the following iteration we construct \mathcal{N}_{β_2} accepting all words α' such that $\beta_2 \not\preceq \text{ext}(\alpha')$. So $\mathcal{N}_{\beta_1} \times \mathcal{N}_{\beta_2}$ accepts all words α' that $\beta_1 \not\preceq \text{ext}(\alpha')$ and $\beta_2 \not\preceq \text{ext}(\alpha')$. We continue this way, finding words $\beta_1, \dots, \beta_k \in \text{core}(\text{ext}(K_p))$ till $\mathcal{A}^K \times \mathcal{N}_{\beta_1} \times \dots \times \mathcal{N}_{\beta_k}$ is empty. At that point we know that $\{\beta_1, \dots, \beta_k\} = \text{core}(\text{ext}(K_p))$.

This procedure works also for the second statement by observing that the set of all consistent sequences, let us call it *Consistent*, is a regular language. So instead of starting with K in the above argument we start with $K \cap \text{Consistent}$. \square

In the next two corollaries, \mathcal{S} is such that its associated automaton $\mathcal{A}_{\mathcal{S}}$ belongs to an admissible class.

Corollary 3. *The sets $\text{core}(\text{Ext}_0)$ and $\text{core}(\text{Ext}_0 \cap \text{Consistent})$ are computable.*

Proof. As we are concerned with s-trees of depth 0, all occurring configurations are of the form (q, λ, \emptyset) . This means that $\text{spawn}(p)\alpha \in \text{Ext}_0$ iff $(p, \lambda_{\text{init}}, \emptyset) \xrightarrow{-\alpha}_{\rightarrow \emptyset} (q, \lambda, \emptyset)$ for some q and λ . We can then use Proposition 2 with $L = L' = \emptyset$. \square

Corollary 4. *Under the hypothesis of Proposition 2: for every $k \geq 0$, we can compute $\text{core}(\text{Ext}_k)$ and $\text{core}(\text{Ext}_k \cap \text{Consistent})$.*

Proof. We start with $L_0 = \text{core}(\text{Ext}_0)$ that we can compute by Corollary 3. Now assume that $L_i = \text{core}(\text{Ext}_i)$ has already been computed. By Lemma 5, L_{i+1} equals the core of $\{\text{spawn}(p)\text{ext}(\alpha) : (p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha}_{L_i} r, \text{ some } r\}$ which, by Proposition 2, is effectively computable. \square

Now we have all ingredients to prove Theorem 1.

Proof (of Theorem 1). Take a process \mathcal{S} as in the statement of the theorem. The external behaviors of \mathcal{S} are described by the language

$$L = \bigcup_{k \in \mathbb{N}} \{\text{spawn}(p)\text{ext}(\alpha) : (p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha}_{\text{Ext}_k} r, \text{ for some } r\}$$

If we denote $L_k = \text{core}(\text{Ext}_k)$ then by Corollary 2, the language L is equal to

$$L' = \bigcup_{k \in \mathbb{N}} \{\text{spawn}(p)\text{ext}(\alpha) : (p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha}_{L_k} r, \text{ for some } r\}$$

By definition, $\text{Ext}_0 \subseteq \text{Ext}_1 \subseteq \dots$ is an increasing sequence of sets. By Lemma 7, this means that there is some m so that $\text{core}(\text{Ext}_m) = \text{core}(\text{Ext}_{m+i})$, for all i . Therefore, L' is equal to

$$\{\text{spawn}(p)\text{ext}(\alpha) : (p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha}_{L_m} r, \text{ for some } r\}$$

By Corollary 4, the set $L_m = \text{core}(\text{Ext}_m)$ is computable and so is $\text{core}(L' \cap \text{Consistent})$. Finally, we check if in this latter set there is a sequence starting with $\text{spawn}(q_{\text{init}})$ and an external write or an output of $\#$. \square

6 Processes with generalized features

In this section we consider *pushdown* dynamic parametric processes where a sub-process cannot write to its own variables, but only to the variables of its parent. This corresponds to the situation when after a parent has created sub-processes, the latter may communicate computed results to the parent and to their siblings, but the parent cannot communicate to child sub-processes. We call such a model a *pushdown dynamic parametric process with generalized futures*. We have seen an example of such a system in Figure 1. Technically, processes with generalized futures are obtained by disallowing $w(x, v)$ actions in our general definition. Additionally, we rule out global variables, i.e., $G = \emptyset$. Accordingly, we may no longer define reachability via reachability of a write action to some global variable, but as reachability of an output action $o(x, \#)$ of some special value $\#$ to some variable x of the root process.

For processes with generalized futures, reachability can be decided by a somewhat simpler approach. In particular, we present an EXPTIME algorithm to decide reachability.

In this section we need an additional assumption concerning the initial value of variables. Since this initial value causes problems as it is the one that cannot

be reproduced once overwritten, we require:

Proviso: We consider systems where the initial value v_{init} of a variable can be neither read nor written.

Since in the case that we consider in this section there are no global variables, the external alphabet simplifies to:

$$\Sigma_{\text{ext}} = \{i(x, v), o(x, v) : x \in X, v \in V\}.$$

From the definition of transitions $r_1 \xrightarrow{a}_L r_2$ of sub-processes modulo hypothesis we can see that the label a can be either an external action, or internal action of the form: $\tau, \text{spawn}(p), r(x, v), \overline{w}(x, v), \overline{r}(x, v)$.

Disallowing $w(x, v)$ operations has an important impact on the $\xrightarrow{\alpha}_L$ semantics. By inspecting the rules, we notice that there is only one remaining rule, namely $\overline{w}(x, v)$, that changes the value of the component λ , and this rule does not change the state component.

The next lemma is the main technical step in this section. It says that for processes with generalized futures, $\text{sig}(\text{Ext}_k)$ as hypothesis yields the same behaviors as Ext_k .

Lemma 8. *For a dynamic parametric process \mathcal{S} with generalized futures, let $L = \text{sig}(\text{Ext}_k)$ and $L' = \text{Ext}_k$. We have: $(p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha}_L (q, \lambda, B)$ for some λ , $B \subseteq L$ iff $(p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha'}_{L'} (q, \lambda', B')$ for some λ' , $B' \subseteq L'$, and some α' with $\text{ext}(\alpha) = \text{ext}(\alpha')$.*

Proof. For the right-to-left implication observe that $\text{Ext}_k \subseteq \text{lift}(\text{sig}(\text{Ext}_k))$. So, if $(p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha}_{L'} (q, \lambda, B)$ then $(p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha}_{L''} (q, \lambda, B)$ for $L'' = \text{lift}(\text{sig}(\text{Ext}_k))$. But then Lemma 6 gives us $(p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha}_L (q, \lambda, B')$ for some B' .

For the left-to-right implication note first that it does not follow from monotonicity, since $\text{sig}(\text{Ext}_k)$ may contain sequences that are not in Ext_k . So assume that $\alpha = a_1 \cdots a_n$, and $(p_i, \lambda_i, B_i) \xrightarrow{a_i}_L (p_{i+1}, \lambda_{i+1}, B_{i+1})$ for $i = 1, \dots, n$ where $(p_1, \lambda_1, B_1) = (p, \lambda_{\text{init}}, \emptyset)$ and $(p_{n+1}, \lambda_{n+1}, B_{n+1}) = (q, \lambda, B)$. We look for subsets B'_i of Ext_k , for $i = 1, \dots, n+1$, such that $(p_i, \lambda'_i, B'_i) \xrightarrow{\delta_i a_i}_{L'} (p_{i+1}, \lambda'_{i+1}, B'_{i+1})$ for some sequence δ_i of internal actions. The additional δ 's are needed since the elements of L are subsequences of those from L' .

For every maximal signature $\beta \in B_{n+1} \subseteq \text{pref}(L)$, we fix a sequence $\beta' \in \text{pref}(L')$ such that $\text{sig}(\beta') = \beta$: for $\beta = \text{spawn}(p_\beta) b_{\beta,1} \cdots b_{\beta,n_\beta}$ we will write $\beta' = \text{spawn}(p_\beta) b_{\beta,1} \gamma_{\beta,1} \cdots b_{\beta,n_\beta} \gamma_{\beta,n_\beta}$, for some $\gamma_{\beta,j} \in \Sigma_{\text{ext}}^*$.

We define B'_i as the smallest prefix closed set that satisfies the following property for every maximal sequence $\beta \in B_{n+1}$: if the prefix $\text{spawn}(\beta) b_{\beta,1} \cdots b_{\beta,j}$ is in B_i (for some $j \leq n_\beta$) then $\text{spawn}(\beta) b_{\beta,1} \gamma_{\beta,2} \cdots b_{\beta,j-1} \gamma_{\beta,j-1} b_{\beta,j}$ is in B'_i .

It remains to prove that $(p_i, \lambda'_i, B'_i) \xrightarrow{\delta_i a_i}_{L'} (p_{i+1}, \lambda'_{i+1}, B'_{i+1})$ for some sequence δ_i of internal actions. For that we perform a case distinction on the action a_i . If a_i is τ , $i(x, v)$ or $o(x, v)$, then only the state changes, so the assertion holds by

induction. If a_i is $\text{spawn}(p')$, then $B_{i+1} = B_i \cup \{\text{spawn}(p')\}$. Likewise, $B'_{i+1} = B'_i \cup \{\text{spawn}(p')\}$ according to our claim.

The case $a_i = r(x, v)$ is a bit more tricky. If $\lambda'_i(x) = v$ then we can do $(p_i, \lambda'_i, B'_i) \xrightarrow{a_i}_{L'} (p_{i+1}, \lambda'_{i+1}, B'_{i+1})$ immediately. If not, we need to re-establish the value v for x . Since by our assumption the initial value is neither read nor written, we know that there must be some $j < i$ with $a_j = \overline{w}(x, v)$. So B'_i contains some sequence ending with $o(x, v)$. But then we can “replay” this output and do $a_i: (p_i, \lambda'_i, B'_i) \xrightarrow{\overline{w}(x, v)}_{L'} (p_i, \lambda''_i, B'_i) \xrightarrow{r(x, v)}_{L'} (p_{i+1}, \lambda''_{i+1}, B'_{i+1})$.

The last, more involved case is when $a_i \in \{\overline{r}(x, v), \overline{w}(x, v) : x \in X, v \in V\}$. Then $B_{i+1} = B_i \cup B \cdot \{b_i\}$ for $b_i = \text{filter}(a_i)$ and some non-empty $B \subseteq B_i$ with $B \cdot \{b_i\} \subseteq B_{n+1}$. Consider the set

$$B' = \{\text{spawn}(p_\beta) b_{\beta,1} \gamma_{\beta,2} \cdots b_{\beta,j} : \text{spawn}(p_\beta) b_{\beta,1} \cdots b_{\beta,j} \in B\}.$$

By definition, $B' \subseteq B'_i$, and B' is not empty since B is not empty. Take some element of B' , say $\beta' = \text{spawn}(p_\beta) b_{\beta,1} \gamma_{\beta,2} \cdots b_{\beta,j}$. For $\gamma = \gamma_{\beta,j+1}$ we have $\beta' \gamma \in \text{pref}(L')$ and $\text{sig}(\beta' \gamma) = \text{sig}(\beta')$. We claim that we can construct a run $(p_i, \lambda'_i, \overline{B}_1) \xrightarrow{\delta}_{L'} (p_i, \lambda''_i, \overline{B}_2)$, for every \overline{B}_1 containing β' , and \overline{B}_2 consisting of \overline{B}_1 and all prefixes of $\beta' \gamma$. The sequence δ consists of internal actions.

Assuming this claim, that we prove in the next paragraph, we proceed as above for each sequence in B' , one after the other. Since B' is finite, say with k elements, at the end we get a computation $(p_i, \lambda'_i, B'_i) \xrightarrow{\delta_1 \cdots \delta_k}_{L'} (p_i, \lambda''_i, B''_i)$ with B''_i consisting of B'_i and all prefixes of the set

$$B'' = \{\text{spawn}(p_\beta) b_{\beta,1} \gamma_{\beta,2} \cdots b_{\beta,j-1} \gamma_{\beta,j} : \text{spawn}(p_\beta) b_{\beta,1} \cdots b_{\beta,j-1} \in B\}.$$

Observe that $B''_i \cup B'' \cdot \{b_i\} = B'_{i+1}$. Then we can do $(p_i, \lambda'_i, B''_i) \xrightarrow{a_i}_{L'} (p_{i+1}, \lambda'_{i+1}, B'_{i+1})$ as claimed at the beginning.

It remains to prove the claim from the above paragraph. Consider a configuration $(q, \lambda', \overline{B})$ with $\overline{B} \subseteq \text{pref}(L')$ a prefix closed set, and a sequence $\beta' \in \overline{B}$. We want to show that for every sequence γ of external actions such that $\beta' \gamma \in \text{pref}(L')$ and $\text{sig}(\beta') = \text{sig}(\beta' \gamma)$, there is a sequence δ of internal actions such that $(q, \lambda', \overline{B}) \xrightarrow{\delta}_L (q, \lambda'', \overline{B}_1)$, with \overline{B}_1 consisting of \overline{B} and all prefixes of $\beta' \gamma$.

Let $\gamma = c_1 \cdots c_k$. Since $\text{sig}(\beta') = \text{sig}(\beta' \gamma)$, for every c_i there is a prefix of β' ending in c_i , say $\beta'_i c_i$. If c_i is an input action, say $i(x, v)$, then we must have that at the moment when $\beta'_i c_i$ was added into the \overline{B} component, the valuation λ was such that $\lambda(x) = v$. Thanks to our assumption that the initial value cannot be read, v is not an initial value. So the only way to have $\lambda(x) = v$ was to execute an output $o(x, v)$ before. This action gives a sequence $\beta''_i o(x, v) \in \overline{B}$, for some β''_i ; in particular $\beta''_i o(x, v) \in \text{pref}(L')$.

After these preparations we show how to execute the sequence γ . If c_i is an output action then we just execute it since this is always possible as $\beta'_i c_i \in \overline{B}$, and so $\beta'_i \in \overline{B}$, by prefix closure. If c_i is an input action, say $i(x, v)$, then we first execute $o(x, v)$, that is possible since $\beta'_i o(x, v) \in \overline{B}$. Then we execute $i(x, v)$. \square

Next we turn the question how to compute the set of signatures of executions efficiently.

Lemma 9. *Let \mathcal{S} be a pushdown dynamic parametric process with generalized futures. For any state p of \mathcal{S} , and any prefix closed set $L \subseteq \Sigma_{sp} \Sigma_{ext}^*$ of signatures, $L = \text{sig}(L)$, consider the language*

$$K_p = \{ \text{ext}(\alpha) : (p, \lambda_{\text{init}}, \emptyset) \xrightarrow{-\alpha}_L (q, \lambda, B) \text{ for some } B \} .$$

We can compute in EXPTIME the sets

$$\{ \text{sig}(\alpha) : \alpha \in K_p \} , \quad \{ \text{sig}(\alpha) : \alpha \in K_p \cap \text{Consistent} \} .$$

The computation time is exponential in $|V|$, $|X|$, and polynomial in the size of p and the pushdown automaton defining \mathcal{S} .

Proof. Assume that $\alpha = a_1 \cdots a_n$, and $(p_i, \lambda_i, B_i) \xrightarrow{-a_i}_L (p_{i+1}, \lambda_{i+1}, B_{i+1})$ for $i = 1, \dots, n$ where $(p_1, \lambda_1, B_1) = (p, \lambda_{\text{init}}, \emptyset)$ and $(p_{n+1}, \lambda_{n+1}, B_{n+1}) = (q, \lambda, B)$. Recall that the configurations of $\xrightarrow{-\alpha}_L$ are (q, λ, B) where q is the state of the system, $\lambda : X \rightarrow V$ is a valuation of the internal variables, and $B \subseteq \text{pref}(L)$ is a set of words in $\Sigma_{sp} \Sigma_{ext}^*$. Since L is a set of signatures its size is at most exponential in $|V|$ and $|X|$. So there are at most doubly exponentially many B 's. This is too much, as we are after a single exponential complexity. We will show that the same sequence can be executed with B 's of polynomial size.

Let $b_1 \cdots b_k$ be the subsequence of first occurrences of actions of the form $\bar{r}(x, v)$, $\bar{w}(x, v)$ in $a_1 \cdots a_n$. For every b_j we choose a sequence $\beta_j \in B$ that is used to perform b_j . In other words, if the first occurrence of b_j is a_l then $\beta_j \in B_l$ and $\beta_j b_j \in B_{l+1}$. We define B'_i as the subset of B_i consisting all the prefixes of words β_j that are in B_i ; more precisely for every $i = 1, \dots, n$ we set:

$$B'_i = \{ \beta \in B : \beta \text{ prefix of some } \beta_j, j = 1, \dots, k \}$$

By induction on i we show that $(p_i, \lambda_i, B'_i) \xrightarrow{-a_i}_L (p_{i+1}, \lambda_{i+1}, B'_{i+1})$, for $i = 1, \dots, n$. So every sequence of actions can be executed using configurations where the B -component is of polynomial size.

For every $\text{spawn}(p) \in \Sigma_{sp}$, from the pushdown automaton defining \mathcal{S} we can construct a pushdown automaton \mathcal{P}^p for the language

$$K^p = \{ \alpha : (p, \lambda_{\text{init}}, \emptyset) \xrightarrow{-\alpha}_L (q, \lambda, B) \} .$$

For this we take the product of the automaton for \mathcal{S} with a finite-state automaton taking care of the λ -component and the B -component. Note that thanks to the previous paragraph, this finite state automaton is of exponential size (the λ -component is exponential in $N = \max(|X|, |V|)$, and the sets B consist of at most N^2 sequences and their prefixes).

The statement of the lemma requires us to compute the set $\text{sig}(\text{ext}(K^p))$. This can be done by the following general algorithm starting with $i = 0$ and $K_0 = K^p$:

1. Find a word $\alpha_i \in K_i$. Let $\beta_i = \text{sig}(\text{ext}(\alpha_i))$.
2. Consider $K_{i+1} = K_i \cap N_i$ where N_i is the set of words α such that $\text{sig}(\text{ext}(\alpha))$ is not β_i (N_i is a regular language).
3. if K_{i+1} not empty, go to the first step.

This iteration terminates in exponential number of iterations, since there are exponentially many signatures, and after every iteration we find one new signature. Each iteration of the above algorithm takes exponential time: this is because the pushdown automaton for \mathcal{S} is of exponential-size, and so is every pushdown automaton obtained by taking consecutive intersections with the finite automata for N_i . Accordingly, the set $\text{sig}(\text{exp}(K^p))$ can be computed in EXPTIME for \mathcal{S} given by a pushdown system.

The statement concerning the consistent sequences follows by the same argument using the fact that the set *Consistent* of consistent sequences is regular. So it is enough to start the iteration from $K^p \cap \text{Consistent}$ instead of K^p . \square

Lemma 10. *Let \mathcal{S} be a pushdown dynamic parametric process with generalized futures. Then for every $k \geq 0$, the sets $\text{sig}(\text{Ext}_k)$ as well as $\text{sig}(\text{Ext}_k \cap \text{Consistent})$ are computable in EXPTIME (exponential in $|V|$, $|X|$, polynomial in the pushdown automaton defining \mathcal{S} , and independent of k).*

Proof. We start with $L_0 = \text{sig}(\text{Ext}_0)$. We can compute it in EXPTIME thanks to Lemma 9 by taking $L = \emptyset$.

Assume now that $L_k = \text{sig}(\text{Ext}_k)$ is already computed, and set $L'_k = \text{Ext}_k$. By Lemmas 5 and 8, Ext_{k+1} is equal to

$$\begin{aligned} \{\text{spawn}(p)\text{ext}(\alpha) : (p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha}_{L'_k} (q, \lambda, B) \text{ for some } B\} = \\ \{\text{spawn}(p)\text{ext}(\alpha) : (p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha}_{L_k} (q, \lambda, B) \text{ for some } B\} \end{aligned}$$

We can now apply Lemma 9 with $L = L_k$ to compute $L_{k+1} = \text{sig}(\text{Ext}_{k+1})$.

Every step can be done in deterministic exponential time. The complexity bound follows since the number of steps is at most exponential: the sets increase after each iteration and the number of signatures is exponential in $|X|, |V|$ (and polynomial in the number of states of the automaton defining \mathcal{S}). \square

Theorem 2. *The reachability problem for pushdown dynamic parametric processes with generalized futures is in DEXPTIME (exponential in $|V|$, $|X|$, and polynomial in the size of the pushdown automaton defining \mathcal{S} .)*

Proof. Let \mathcal{S} be a pushdown dynamic parametric process with generalized futures. The external behaviors of \mathcal{S} are described by the language

$$L = \bigcup_{k \in \mathbb{N}} \{\text{spawn}(p)\text{ext}(\alpha) : (p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha}_{\text{Ext}_k} r, \text{ for some } r\}.$$

If we denote $L_k = \text{sig}(\text{Ext}_k)$ then by Lemma 8 the language L is equal to

$$L' = \bigcup_{k \in \mathbb{N}} \{\text{spawn}(p)\text{ext}(\alpha) : (p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha}_{L_k} r, \text{ for some } r\}.$$

By definition, $Ext_0 \subseteq Ext_1 \subseteq \dots$ is an increasing sequence of sets. As there are finitely many signatures, this means that there is some m so that $\text{sig}(Ext_m) = \text{sig}(Ext_{m+i})$, for all i . Therefore, L' is equal to

$$L'' = \{\text{spawn}(p)\text{ext}(\alpha) : (p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha}_{L_m} r, \text{ for some } r\}$$

By Lemma 10, the set $L_m = \text{sig}(Ext_m)$ can be computed in EXPTIME, and so can be $\text{sig}(L'' \cap \text{Consistent})$. Finally we check if in the latter language there is a sequence starting with $\text{spawn}(q_{\text{init}})$ and containing $o(x, \#)$. \square

6.1 Simple futures

Here we consider pushdown processes with *simple futures*, where every subprocess communicates only with its parent, by writing values into the registers shared with the parent. So there is no communication between siblings, therefore we work with $\Sigma_{\text{ext}} = \{o(x, v) : x \in X, v \in V\}$. Internal transitions are with actions of the form $\tau, \text{spawn}(p), r(x, v), \bar{w}(x, v)$.

For $\beta \in \Sigma_{\text{ext}}^*$ let $\text{out}(\beta) = \{o(x, v) : \beta = \text{spawn}(p')\beta_1 o(x, v)\beta_2\}$, and let $\text{out}(\text{spawn}(p)\alpha) = \{\text{spawn}(p)\} \cup \text{spawn}(p)\text{out}(\alpha)$. For a set $L \subseteq \Sigma_{\text{sp}} \cdot \Sigma_{\text{ext}}^*$ let $\text{out}(L) = \{\text{out}(\beta) : \beta \in L\}$.

Lemma 11. *Let $L \subseteq \Sigma_{\text{ext}}^*$ be a prefix-closed set of signatures and $L' = \text{out}(L)$.*

Then $(p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha}_L (q, \lambda, B)$ for some λ and $B \subseteq L$ iff $(p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha'}_{L'} (q, \lambda', B')$ for some λ' and $B' \subseteq L'$, with $\text{ext}(\alpha) = \text{ext}(\alpha')$.

Proof. For the left-to-right direction we show by induction that $(p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha}_L (q, \lambda, B)$ implies $(p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha}_{L'} (q, \lambda, B')$ for $B' = \text{out}(B)$. Let us assume that $(p_1, \lambda_1, B_1) \xrightarrow{a}_L (p_2, \lambda_2, B_2)$. We show that $(p_1, \lambda_1, \text{out}(B_1)) \xrightarrow{a}_{L'} (p_2, \lambda_2, \text{out}(B_2))$. If a is either $\tau, r(x, v)$ or $o(x, v)$, then the transition changes only the state. If a is $\text{spawn}(p')$ then $B_2 = B_1 \cup \{\text{spawn}(p')\}$ and also $\text{out}(B_2) = \text{out}(B_1) \cup \{\text{spawn}(p')\}$. If $a = \bar{w}(x, v)$, then $B_2 = B_1 \cup B\{o(x, v)\}$ for some $B \subseteq B_1$ with $B\{o(x, v)\} \subseteq L$. Let $B' = \{\text{spawn}(p') : \text{spawn}(p')\beta \in B, \text{ for some } \beta\}$. We have $B' \subseteq \text{out}(B_1)$ and $\text{out}(B_2) = \text{out}(B_1) \cup B'\{o(x, v)\}$, because B_1 is prefix closed.

For the right-to-left direction we show by induction on $|\alpha'|$ that $(p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha'}_{L'} (q, \lambda', B')$ for some λ' and $B' \subseteq L'$, then $(p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha}_L (q, \lambda, B)$ for some λ and $B \subseteq L$, such that $B' \subseteq \text{out}(B)$, and α' with $\text{ext}(\alpha') = \text{ext}(\alpha)$.

Assume $(p, \lambda_{\text{init}}, \emptyset) = (p_1, \lambda_1, B'_1) \xrightarrow{a_1}_{L'} (p_2, \lambda'_2, B'_2) \xrightarrow{a_2}_{L'} \dots \xrightarrow{a_{n-1}}_{L'} (p_n, \lambda'_n, B'_n) = (q, \lambda', B')$. We look for some λ_i and $B_i \subseteq L$ such that $B'_i \subseteq \text{out}(B_i)$ and $(p_i, \lambda_i, B_i) \xrightarrow{\delta_i a_i}_L (p_{i+1}, \lambda_{i+1}, B_{i+1})$, for every i , for some sequences of internal actions δ_i . The case where a_i is τ or $o(x, v)$ is immediate, since only the state changes. Same holds for $a_i = \text{spawn}(p')$, since then $B'_{i+1} = B'_i \cup \{\text{spawn}(p')\}$ and $B_{i+1} = B_i \cup \{\text{spawn}(p')\}$. In all three cases δ_i is empty.

Let us assume that $a_i = r(x, v)$, so $\lambda'_i(x) = v$. If $\lambda_i(x) = v$ then we can do a_i immediately, and δ_i is empty. If not, since the initial value v_{init} can be neither

read nor written, there must be some $j < i$ such that $a_j = \bar{w}(x, v)$. So B'_i contains $\text{spawn}(p')$ and $\text{spawn}(p')o(x, v)$, for some p' . By inductive assumption we have $B'_i \subseteq \text{out}(B_i)$, thus there is some sequence $\text{spawn}(p')\beta o(x, v) \in B_i$, for some $\beta \in \Sigma_{\text{ext}}^*$. So we can “replay” this output and do $(p_i, \lambda_i, B_i) \xrightarrow{\bar{w}(x, v)}_L (p_i, \lambda_{i+1}, B_{i+1}) \xrightarrow{r(x, v)}_L (p_{i+1}, \lambda_{i+1}, B_{i+1})$ (here we have $B_i = B_{i+1}$).

The last case is when $a_i = \bar{w}(x, v)$, so $B'_{i+1} = B'_i \cup B'\{o(x, v)\}$, with $B' \subseteq B'_i$, $B'\{o(x, v)\} \subseteq L'$. In particular, $B' \subseteq \Sigma_{\text{sp}}$. Since $B' \subseteq \text{out}(B_i)$ we also have $B' \subseteq B_i$. Since $B'\{o(x, v)\} \subseteq \text{out}(L)$ we can choose some subset $B \subseteq L$ with spawn 's from B' , and such that $B\{o(x, v)\} \subseteq L$. So every sequence in $B\{o(x, v)\}$ is of the form $\text{spawn}(q_j)\beta_j o(x, v)$, with β_j consisting only of outputs $o(x', v')$, $j = 1, \dots, k$. In addition, $B' = \{\text{spawn}(q_j) : j\}$. Let δ^j be obtained from β_j by renaming $o(x', v')$ into $\bar{w}(x', v')$, and $\delta_i = \delta^1 \dots \delta^k$. We have $(p_i, \lambda_i, B_i) \xrightarrow{\delta_i}_{L'} (p_i, \hat{\lambda}_i, \hat{B}_i) \xrightarrow{\bar{w}(x, v)}_L (p_{i+1}, \lambda_{i+1}, B_{i+1})$, where $B'\{o(x, v)\} \subseteq \text{out}(B_{i+1})$ (here we have $p_i = p_{i+1}$). \square

Together with Lemma 8 we obtain:

Corollary 5. *Let $L = \text{Ext}_k$ and $L' = \text{out}(L)$. Then $(p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha}_L (q, \lambda, B)$ for some λ and $B \subseteq L$ iff $(p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha'}_{L'} (q, \lambda', B')$ for some λ' and $B' \subseteq L'$, with $\text{ext}(\alpha) = \text{ext}(\alpha')$.*

The next lemma computes the set of outputs at level 0:

Lemma 12. *Let \mathcal{S} be a pushdown dynamic parametric process with simple futures. For any state p of \mathcal{S} , and any prefix-closed set $L \subseteq \Sigma_{\text{sp}} \cup \Sigma_{\text{sp}}\Sigma_{\text{ext}}$ of outputs, consider the language*

$$K_p = \{\text{ext}(\alpha) : (p, \lambda_{\text{init}}, \emptyset) \xrightarrow{\alpha}_L (q, \lambda, B) \text{ for some } B\}.$$

If the number of variables is fixed, then we can determine whether $o(x, v) \in \text{out}(K_p)$ in NP (resp., PTIME if $|V|$ is fixed).

Proof. From the pushdown system of \mathcal{S} we can construct a pushdown system \mathcal{P}^p for K_p , by adding the λ and B component to the finite control. Doing this yields a pushdown automaton of exponential size, because there are exponentially many B . However, since the B component grows monotonically, we can guess beforehand the polynomially many changes and check reachability on the new pushdown of polynomial size. \square

As a consequence we obtain with similar arguments as for signatures:

Lemma 13. *Let \mathcal{S} be a pushdown dynamic parametric process with simple futures and a fixed number of variables. Then for every $k \geq 0$, we can check membership in $\text{out}(\text{Ext}_k)$ in NP (resp., PTIME if $|V|$ is fixed).*

Finally we obtain:

Theorem 3. *The reachability problem for pushdown dynamic parametric processes with simple futures and a fixed number of variables is NP-complete (resp., PTIME if $|V|$ is fixed).*

Proof. We only sketch the NP lower bound, that already holds for finite-state systems, by a reduction from SAT. Let $C_1 \wedge \dots \wedge C_m$ be a CNF formula with clauses C_j over variables x_1, \dots, x_n . The root process first spawns sub-processes with initial state 1, then 2, and so on up to n . Then it guesses a valuation $(b_1, \dots, b_n) \in \{0, 1\}^n$ by writing into the (unique) register $(1, b_1), \dots, (n, b_n)$. A sub-process with initial state i will read the value (i, b_i) and then write into the register C_{j_1}, \dots, C_{j_k} , where $j_1 < \dots < j_k$ are the indices of those clauses that become true if x_i is set to b_i . The root process needs to read C_1, \dots, C_n from the register in order to output $\#$.

7 Dynamic parametric processes without local variables

In this section we consider another restriction of dynamic parametric pushdown processes, that is incomparable to the previous ones: we allow only communication over global variables. We show that in this case the power of the **spawn** operation is quite limited, and that the hierarchical structure can be *flattened*. More precisely, we give a reduction to the reachability problem of parametric pushdown processes without local variables, where additionally only the root can do the **spawn** and will spawn just once, as its first action. A system with this property can be simulated by a (C, D) -system as in [7]. We would actually need a slight extension of (C, D) -systems since the literature considers only the variant with a single variable. Yet the same methods give decision algorithms for any fixed finite number of variables. In consequence, this reduction implies that the reachability problem when each sub-process is realized as a pushdown system, is PSPACE-complete.

Given a dynamic parametric process $\mathcal{S} = (Q, G, \emptyset, V, q_{\text{init}}, v_{\text{init}}, \Delta)$ without local variables we construct a new process $\text{flat}(\mathcal{S})$ with one more global variable g_{sp} , that we call *spawn variable*. This variable will store **spawn** demands, that is the state p when some process has performed **spawn**(p):

$$V_{sp} = \{p \in Q : \text{spawn}(p) \text{ occurs in } \Delta\}.$$

Recall that V_{sp} is finite since the number of **spawn** transitions in Δ is finite. The main property of $\text{flat}(\mathcal{S})$ will be its restricted use of **spawn**.

The process $\text{flat}(\mathcal{S})$ is given by $(Q', G \cup \{g_{sp}\}, \emptyset, V \cup V_{sp}, q'_{\text{init}}, v_{\text{init}}, \Delta')$ where $Q' = Q \cup \{q'_{\text{init}}, q''_{\text{init}}\}$ and Δ' is obtained from Δ as follows. We replace every **spawn** transition $q \xrightarrow{\text{spawn}(p)} q'$ in Δ by a write $q \xrightarrow{w(g_{sp}, p)} q'$ into the spawn variable. In addition, from q'_{init} we add to Δ' the transition $q'_{\text{init}} \xrightarrow{\text{spawn}(q''_{\text{init}})} q_{\text{init}}$, and from q''_{init} we add $q''_{\text{init}} \xrightarrow{r(g_{sp}, p)} p$ for every state $p \in Q$. The result is that in $\text{flat}(\mathcal{S})$ the only **spawn** operation happens from the new initial state q'_{init} , this operation creates a number of sub-processes all starting in state q''_{init} . Subsequently, the sub-process

started in q'_{init} goes to the state q_{init} and proceeds as in \mathcal{S} , but instead of doing $\text{spawn}(p)$ it just writes p into the spawn variable g_{sp} . The sub-processes that have been spawned are all in state q''_{init} from which they may proceed by reading p from the spawn variable. As we can see, every reachable configuration of $\text{flat}(\mathcal{S})$ is a tree of height 1, consisting only of a root and immediate children of the root created by the single spawn operation. In order to proceed, the children of the root need to read some p value from the spawn variable.

Lemma 14. *A dynamic parametric process \mathcal{S} without local variables has a consistent run containing some external write of $\#$ if and only if $\text{flat}(\mathcal{S})$ has one.*

Proof. Since we consider processes without local variables the valuation of the local variables is the empty function. In general, a configuration is a triple (q, λ, S) , but now we can omit λ . So in this proof s-trees will have the form (q, S) , where S is a finite set of s-trees. The configurations of $\text{flat}(\mathcal{S})$ have one more property as only the root can do a spawn : the occurring s-trees all are of the form (q, S) where S is a set of pairs (q', \emptyset) for some state q' — implying that S is essentially a set of states.

Assume that the process \mathcal{S} has a consistent run in the set semantics containing some external write of $\#$. To simulate this run, let $\text{flat}(\mathcal{S})$ start with a transition $q'_{\text{init}} \xrightarrow{\text{spawn}(q'_{\text{init}})} q_{\text{init}}$ spawning sub-processes in state q''_{init} . A spawn operation $\text{spawn}(p)$ by some sub-process in \mathcal{S} is replaced in $\text{flat}(\mathcal{S})$ by a write $w(g_{sp}, p)$ or $\bar{w}(g_{sp}, p)$ into the spawn variable. This allows a child sub-process to wake up by moving from state q''_{init} to p . So when simulating the run of \mathcal{S} , we keep the invariant that the root processes in \mathcal{S} and $\text{flat}(\mathcal{S})$ are in the same state, and for every state q but q''_{init} : state q appears in a configuration of \mathcal{S} iff it appears in the corresponding configuration of $\text{flat}(\mathcal{S})$.

Conversely, consider a consistent run of $\text{flat}(\mathcal{S})$ reaching an external write of $\#$. It must start with $(q'_{\text{init}}, \emptyset) \xrightarrow{\text{spawn}(q'_{\text{init}})} (q_{\text{init}}, \{q''_{\text{init}}\})$. We show that for every consistent run $(q_{\text{init}}, \{q''_{\text{init}}\}) \xrightarrow{\alpha'} (q, S')$ of $\text{flat}(\mathcal{S})$ we can find a consistent sequence α such that $(q_{\text{init}}, \emptyset) \xrightarrow{\alpha'} (q, S)$ where S consists of states from S' apart q''_{init} plus additionally the state p from the last write $w(g_{sp}, p)$ to the spawn variable in α' ; if there is such a write.

The proof of this claim is by induction on the length of α' considering possible transitions from (q, S') one by one:

- Root process read or write: if $(q, S') \xrightarrow{w(g, v)} (q_1, S')$ with $g \neq g_{sp}$ then $(q, S) \xrightarrow{w(g, v)} (q_1, S)$. Same for $r(g, v)$.
- Root process spawn : if $(q, S') \xrightarrow{w(g_{sp}, p)} (q_1, S')$ then $(q, S) \xrightarrow{\text{spawn}(p)} (q_1, S \cup \{(p, \emptyset)\})$.
- Sub-process read or write: let $(q, S') \xrightarrow{\bar{w}(g, v)} (q, S' \cup \{(q_2, \emptyset)\})$ for $g \neq g_{sp}$, and $(q_1, \emptyset) \xrightarrow{w(g, v)} (q_2, \emptyset)$ for some $(q_1, \emptyset) \in S'$. Since q_1 occurs in S by our induction hypothesis we get $(q, S) \xrightarrow{\bar{w}(g, v)} (q, S_2)$ for some suitable S_2 . Same for $\bar{r}(g, v)$.

- Sub-process **spawn**: let $(q, S') \xrightarrow{w(g_{sp}, p)} (q, S' \cup \{(q_2, \emptyset)\})$ for some $q_1 \in S'$ with $(q_1, \emptyset) \xrightarrow{w(g_{sp}, p)} (q_2, \emptyset)$. Since q_1 occurs in S , by induction we can pick an s-tree (q_1, S_1) within S and add a sibling $(q_1, S_1) \xrightarrow{\text{spawn}((p, \emptyset))} (q_2, S_1 \cup \{(p, \emptyset)\})$. This gives us $(q, S) \xrightarrow{\tau} (q, S')$ with S' containing $(q_2, S_1 \cup \{(p, \emptyset)\})$.
- Sub-process “wake-up”: let $(q, S') \xrightarrow{r(g_{sp}, p)} (q, S' \cup \{p\})$. This transition is not simulated by any transition in \mathcal{S} (note that p already occurs in S by our induction hypothesis and the fact that α is consistent).

□

Theorem 4. *Let the number of variables be fixed. The reachability problem for pushdown dynamic parametric processes without local variables is PSPACE-complete.*

Proof. If \mathcal{S} is given by a pushdown automaton, then so is $\text{flat}(\mathcal{S})$. Now $\text{flat}(\mathcal{S})$ has only one **spawn** at the very beginning of its execution. Such a system can be simulated by a (C, D) -system [7, 4, 5] of the same size: the leader system D is $\text{flat}(\mathcal{S})$ with the initial state q_{init} , and the contributor system C is $\text{flat}(\mathcal{S})$ with the initial state q''_{init} . A small obstacle is that in the literature (C, D) -systems were defined with only one variable, while we need at least two. It can be checked that the reachability problem for (C, D) -systems given by pushdown automata is PSPACE-complete [4, 5] even for systems with more than one variable as long as the number of variables is fixed. Our reduction then shows that our reachability problem is in PSPACE. Since (C, D) -systems can be directly simulated by our model we also get the PSPACE lower bound. □

8 Conclusions

We have studied systems with parametric process creation where sub-processes may communicate via both global and local shared variables. We have shown that under mild conditions, reachability for this model is decidable. The algorithm relies on the abstraction of the behavior of the created child sub-processes by means of finitely many minimal behaviors. This set of minimal behaviors is obtained by a fixpoint computation whose termination relies on well-quasi-orderings. This bottom-up approach is different from the ones used before [7, 5, 11]. In particular, it avoids computing a downward closure, thus showing that computability of the downward closure is not needed in the general decidability results on flat systems from [11].

We have also considered special cases for pushdown dynamic parametric processes where we obtained solutions of a relatively low complexity. In absence of local variables we have shown that reachability can be reduced to reachability for systems without dynamic sub-process creation, implying that reachability is PSPACE-complete. For the (incomparable) case where communication is restricted to child sub-processes reporting their results to siblings and their parents, we have also provided a dedicated method with DEXPTIME complexity.

We conjecture that this bound is tight. Finally, when sub-processes can report results only to their parents, the problem becomes just NP-complete.

An interesting problem for further research is to study the reachability of a particular *set* of configurations as considered, e.g., for dynamic pushdown networks [2]. One such set could, e.g., specify that all children of a given sub-process have terminated. For dynamic pushdown networks with nested or contextual locking, such kinds of barriers have been considered in [6, 13]. It remains as an intriguing question whether or not similar concepts can be handled also for dynamic parametric processes.

References

1. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In A. W. Mazurkiewicz and J. Winkowski, editors, *CONCUR '97: Concurrency Theory, 8th International Conference, Warsaw, Poland, July 1-4, 1997, Proceedings*, pages 135–150. Springer, LNCS 1243, 1997.
2. A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *Concurrency Theory. 16th Int. Conf. (CONCUR)*, pages 473–487. Springer, LNCS 3653, 2005.
3. R. Chadha, P. Madhusudan, and M. Viswanathan. Reachability under contextual locking. In C. Flanagan and B. König, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, pages 437–450. Springer, LNCS 7214, 2012.
4. A. Durand-Gasselin, J. Esparza, P. Ganty, and R. Majumdar. Model checking parameterized asynchronous shared-memory systems. In D. Kroening and C. S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 67–84. Springer, LNCS 9206, 2015.
5. J. Esparza, P. Ganty, and R. Majumdar. Parameterized verification of asynchronous shared-memory systems. *J. ACM*, 63(1):10, 2016.
6. T. M. Gawlitza, P. Lammich, M. Müller-Olm, H. Seidl, and A. Wenner. Join-lock-sensitive forward reachability analysis for concurrent programs with dynamic process creation. In R. Jhala and D. A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, pages 199–213. Springer, LNCS 6538, 2011.
7. M. Hague. Parameterised pushdown systems with non-atomic writes. In S. Chakraborty and A. Kumar, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2011, December 12-14, 2011, Mumbai, India*, volume 13 of *LIPICs*, pages 457–468. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011.
8. M. Hague, A. S. Murawski, C.-H. L. Ong, and O. Serre. Collapsible pushdown automata and recursion schemes. In *LICS'08*, pages 452–461. IEEE Computer Society, 2008.

9. V. Kahlon. Parameterization as abstraction: A tractable approach to the dataflow analysis of concurrent programs. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*, pages 181–192. IEEE Computer Society, 2008.
10. V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, pages 505–518. Springer, LNCS 3576, 2005.
11. S. La Torre, A. Muscholl, and I. Walukiewicz. Safety of parametrized asynchronous shared-memory systems is almost always decidable. In L. Aceto and D. de Frutos-Escrig, editors, *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1-4, 2015*, volume 42 of *LIPICs*, pages 72–84. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.
12. P. Lammich and M. Müller-Olm. Conflict analysis of programs with procedures, dynamic thread creation, and monitors. In M. Alpuente and G. Vidal, editors, *SAS*, volume 5079 of *Lecture Notes in Computer Science*, pages 205–220. Springer, LNCS, 2008.
13. P. Lammich, M. Müller-Olm, H. Seidl, and A. Wenner. Contextual locking for dynamic pushdown networks. In F. Logozzo and M. Fähndrich, editors, *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, pages 477–498. Springer, LNCS 7935, 2013.
14. P. Lammich, M. Müller-Olm, and A. Wenner. Predecessor sets of dynamic pushdown networks with tree-regular constraints. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, pages 525–539. Springer, LNCS 5643, 2009.
15. C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS'06*, pages 81–90, 2006.
16. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, 2000.
17. I. Walukiewicz. Pushdown processes: Games and model checking. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification, 8th International Conference, CAV '96, New Brunswick, NJ, USA, July 31 - August 3, 1996, Proceedings*, pages 62–74. Springer, LNCS 1102, 1996.