Reachability for dynamic parametric processes

Anca Muscholl[1], Helmut Seidl[2], and Igor Walukiewicz[3]

[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 rendezvous 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 can effectively be reduced to reachability in the model of Hague [7,5], giving us a precise characterization of the complexity for pushdown threads 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. Section 6 considers some special instances of dynamic parametric pushdown processes. The full version of the paper is available at `http://arxiv.org/abs/1609.05385`.

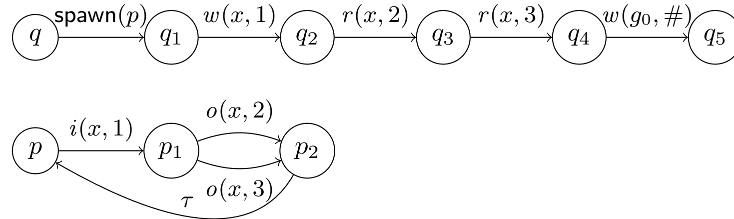
2 Basic definitions

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 dynamic parametric process can, 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 conditional if$(*)$ 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 states of the dynamic parametric process correspond to the lines in the listing, the transitions represent the control flow, they are labeled with atomic statements of the program.

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.
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.

```
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.

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.

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. The transition system is specified by a tuple $\mathcal{S} = \langle Q, G, X, V, \Delta, q_{\mathsf{init}}, v_{\mathsf{init}} \rangle$ 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 $q_{\mathsf{init}} \in Q$, and an initial value $v_{\mathsf{init}} \in V$ 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,
  - $\mathsf{spawn}(q)$, 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 $\mathsf{spawn}(q)$ 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

In this section we provide the operational semantics of dynamic parametric processes, 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, $\lambda : X \to V$ 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 $k \in \mathbb{N}$

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

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 $\bigcup_{k \in \mathbb{N}} \textit{M-trees}_k$.

We use standard notation for multisets. A multiset M over a universe U is a mapping $M : U \to \mathbb{N}_0$. It is finite if $\sum_{t \in U} M(t) < \infty$. A finite multiset M may also be represented by $M = [n_1 \cdot t_1, \ldots, n_k \cdot t_k]$ if $M(t_i) = n_i$ for $i = 1, \ldots, k$ 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 $[n_1 \cdot t_1, \ldots, n_k \cdot t_k]$ for $[n_1 \cdot t_1] + \ldots + [n_k \cdot t_k]$, 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 $[\![\mathcal{S}]\!]$. The states of $[\![S]\!]$ are m-trees, and the set of possible edge labels is:

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

Notice that we have two new kinds of labels $\overline{r}(y,v)$ and $\overline{w}(y,v)$. 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.

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 the *external variables*: these are 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 via external variables: these are 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 via external variables.

The initial state is given by $t_{\mathsf{init}} = (q_{\mathsf{init}}, \lambda_{\mathsf{init}}, [\,])$, where λ_{init} maps all locals to the initial value v_{init}. A transition between two states of $[\![\mathcal{S}]\!]$ (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 describe operations on external variables, be they global or not. For globals, these operations may come from the child sub-processes. In this case we relabel them as $\overline{r}, \overline{w}$ actions. This helps later to identify the rule that has been used for the transition (see Prop. 2). Note that the values of global variables are not part of the program state. Accordingly, these operations therefore can be considered as unconstrained input/output actions.

External transitions:

$$
\begin{array}{llll}
(q_1, \lambda, M) \overset{a}{\Longrightarrow} & (q_2, \lambda, M) & \text{if } q_1 \overset{a}{\longrightarrow} q_2 & \text{for } a \in \Sigma_{ext} \\
(q, \lambda, M_1) \overset{\overline{r}(g,v)}{\Longrightarrow} & (q, \lambda, M_2) & \text{if } M_1 \overset{r(g,v)}{\Longrightarrow} M_2 & \text{for } g \in G \\
(q, \lambda, M_1) \overset{\overline{w}(g,v)}{\Longrightarrow} & (q, \lambda, M_2) & \text{if } M_1 \overset{w(g,v)}{\Longrightarrow} M_2 & \text{for } g \in G
\end{array}
$$

Internal transitions:

$$
\begin{array}{llll}
(q_1, \lambda, M) & \overset{\tau}{\Longrightarrow} & (q_2, \lambda, M) & \text{if } q_1 \overset{\tau}{\longrightarrow} q_2 \\
(q_1, \lambda, M_1) & \overset{\mathsf{spawn}(p)}{\Longrightarrow} & (q_2, \lambda, M_2) & \text{if } q_1 \overset{\mathsf{spawn}(p)}{\longrightarrow} q_2 \text{ and } M_2 = M_1 + [n \cdot (p, \lambda_{\mathsf{init}}, [])] \text{ for some } n \geq 0 \\
(q_1, \lambda, M) & \overset{w(x,v)}{\Longrightarrow} & (q_2, \lambda', M) & \text{if } q_1 \overset{w(x,v)}{\longrightarrow} q_2 \text{ and } \lambda' = \lambda[v/x] \\
(q_1, \lambda, M) & \overset{r(x,v)}{\Longrightarrow} & (q_2, \lambda, M) & \text{if } q_1 \overset{r(x,v)}{\longrightarrow} q_2 \text{ and } v = \lambda(x) \\
(q, \lambda, M_1) & \overset{\overline{r}(x,v)}{\Longrightarrow} & (q, \lambda, M_2) & \text{if } M_1 \overset{i(x,v)}{\Longrightarrow} M_2 \text{ and } v = \lambda(x) \\
(q, \lambda, M_1) & \overset{\overline{w}(x,v)}{\Longrightarrow} & (q, \lambda', M_2) & \text{if } M_1 \overset{o(x,v)}{\Longrightarrow} M_2 \text{ and } \lambda' = \lambda[v/x]
\end{array}
$$

Here, we say that

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

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

Fig. 2. Multiset semantics of dynamic parametric processes.

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 $\lambda' : X \to V$ 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 $\overline{r}(x, v), \overline{w}(x, v)$ on the local variables of the parent.

We write $t_1 \overset{\alpha}{\Longrightarrow} t_2$ for a sequence of transitions complying with the sequence α of action labels. Note that we allow several child sub-processes to move in one step (see definition of $M_1 \overset{\alpha}{\Longrightarrow} M_2$ in Figure 2). While this makes the definition slightly more complicated, it simplifies some arguments later. Observe also 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 result in Theorem 1: 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. Recall that our semantics does not 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 $\alpha \in \Sigma^*$ is y-*consistent *if in the projection of α on operations on y, every read action $r(y, v)$ or $\overline{r}(y, v)$ which is not the first operation on y in α is immediately preceded either by $r(y, v), \overline{r}(y, v)$ or by $w(y, v)$ or $\overline{w}(y, v)$. The first operation on y in α can be either $r(y, v_{\mathsf{init}}), \overline{r}(y, v_{\mathsf{init}})$ or $w(y, v), \overline{w}(y, v)$ 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 $[\![S]\!]$ starting in the initial state, i.e., a sequence α such that $t_{\mathsf{init}} \stackrel{\alpha}{\Longrightarrow}_{\mathcal{S}} t$ 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 $[\![\mathcal{S}]\!]$ 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:*

1. 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.
2. 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 $\mathcal{A} \circlearrowleft \Gamma$ that is obtained from $\mathcal{A}$ by adding a self-loop on every state of $\mathcal{A}$ on every letter from Γ. Moreover, $\mathcal{A} \circlearrowleft \Gamma$ also belongs to $\mathcal{C}$.
3. 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, lossy channel systems, etc. The closure under alphabet extensions required by Definition 3 is only added for convenience, e.g. for building synchronized products of automata over the same alphabets of actions.

From 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 $\mathcal{S} = \langle Q, G, X, V, \Delta, q_{\mathsf{init}}, v_{\mathsf{init}} \rangle$ we set $\mathcal{A}_\mathcal{S} = \langle Q, \Sigma_{G,X,V}, \Delta, Q \rangle$, where $\Sigma_{G,X,V}$ is the alphabet of actions appearing in Δ. The automaton $\mathcal{A}_\mathcal{S}$ is called the *associated automaton* of $\mathcal{S}$.

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 of the above theorem, we obtain that the reachability problem is decidable for *pushdown dynamic parametric processes*, that is where each subprocess 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 already appeared in [9,4,11,5]. It reflects the idea that in the context of parametrization, the semantics can be chosen as accumulative, in the sense that

we are only interested in sets of reachable configurations, rather then counting their multiplicities. We adapt the set semantics to our model, and show that it is equivalent to the multiset semantics — at least as far as the reachability problem is concerned.

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

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

where $q \in Q$, $\lambda : X \to V$, 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 $[\![\mathcal{S}]\!]_s$ 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 $s_{\mathsf{init}} = (q_{\mathsf{init}}, \lambda_{\mathsf{init}}, \emptyset)$. The transitions are defined as in the multiset case but for multiset actions that become set actions:

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

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

The reachability problem for dynamic parametric processes under the set semantics asks, like in the multiset case, whether there is some consistent run of $[\![\mathcal{S}]\!]_s$ 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.*

4 External sequences and signatures

In this section we introduce the objects that will be useful for summarizing the behaviors of sub-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 6). Other actions of interest are the spawns occurring in $\mathcal{S}$:

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

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

For a sequence of actions α, let $ext(\alpha)$ be the subsequence of external actions in α, with additional renaming of $\overline{w}$, and $\overline{r}$ actions to actions without a bar, if

they refer to global variables $g \in G$:

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

Let $\overset{\alpha}{\Longrightarrow}_k$ stand for the restriction of $\overset{\alpha}{\Longrightarrow}$ 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.

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

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 $\mathsf{sig}(\alpha)$, is the subsequence of first appearances of actions in α.*

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

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

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 $\mathsf{spawn}(p)i(x,0)o(x,1)$, $\mathsf{spawn}(p)i(x,1)o(x,2)$, and $\mathsf{spawn}(p)i(x,1)o(x,0)$. 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 $b_0, \ldots, b_n$ and canonical decomposition $\alpha = b_0\alpha_1 b_1 \alpha_2 b_2 \cdots \alpha_k b_k \alpha_{k+1}$. A* lift *of α is any word $\beta = b_0\beta_1 b_1 \beta_2 b_2 \cdots \beta_k b_k \beta_{k+1}$, where β_i is obtained from α_i by inserting some number of actions $b_0, \ldots, b_{i-1}$, for $i = 1, \ldots, k+1$. We write $\mathsf{lift}(\alpha)$ for the set of all such words β. For a set $L \subseteq \Sigma_{ext}^*$ we define*

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

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

Observe that $\alpha \in \mathsf{lift}(\mathsf{sig}(\alpha))$. Another useful property is that if $\beta \in \mathsf{lift}(\alpha)$ 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. Our algorithm will analyze a process tree level by level. The main tool is an abstraction (summarization) of child sub-processes by their external behaviors. We call it *systems under hypothesis*.

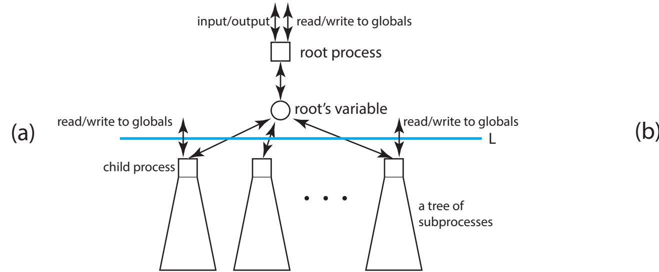


Fig. 3. Reduction to a system under hypothesis.

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 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) $L = \{\mathsf{spawn}(p)\beta : \beta \in \mathsf{pref}(L_p), \mathsf{spawn}(p) \in \Sigma_{sp}\}$ 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, ..., 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 $B \subseteq \mathsf{pref}(L)$ is a set of sequences of external actions for sets of sub-processes.

The initial state is $r_{\mathsf{init}} = (q_{\mathsf{init}}, \lambda_{\mathsf{init}}, \emptyset)$. We will use r to range over configurations of $\mathcal{S}_L$. Transitions between two states $r_1 \overset{a}{\dashrightarrow}_L r_2$ 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.

External transitions under hypothesis:

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

Internal transitions under hypothesis:

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

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

We list below two properties of $\dashrightarrow_L$. 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 $\overline{r}$ and $\overline{w}$, 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 $\mathsf{filter}(\alpha)$ is obtained by the following renaming of α:

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

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

Lemma 1. *If $(q,\lambda,\emptyset) \overset{\alpha}{\dashrightarrow}_L (q',\lambda',B)$ then $B \subseteq \mathsf{pref}(L)$, and every $\beta \in B$ is a scattered subword of $\mathsf{filter}(\alpha)$.*

Lemma 2. *If $L_1 \subseteq L_2$ and $(p,\lambda,\emptyset) \overset{\alpha}{\dashrightarrow}_{L_1} r$ then $(p,\lambda,\emptyset) \overset{\alpha}{\dashrightarrow}_{L_2} r$.*

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. The proof of the lemma is rather technical.

Lemma 3. *Suppose $L = Ext_k$. For every p, q, λ, and α we have: $(p, \lambda_{\mathsf{init}}, \emptyset) \xrightarrow{\alpha}_L (q, \lambda, B)$ for some B iff $(p, \lambda_{\mathsf{init}}, \emptyset) \overset{\alpha}{\Longrightarrow}_{k+1} (q, \lambda, S)$ for some S.*

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 $k+1$. The key observation here here is that the lift operation (cf. Definition 5) does not add new behaviours:

Lemma 4. *Assume that $L \subseteq \Sigma_{sp} \cdot \Sigma_{ext}^*$ and $L' = \mathsf{lift}(L)$. Then $(p, \lambda_{\mathsf{init}}, \emptyset) \xrightarrow{\alpha}_L (q, \lambda, B)$ for some $B \subseteq \mathsf{pref}(L)$ iff $(p, \lambda_{\mathsf{init}}, \emptyset) \xrightarrow{\alpha}_{L'} (q, \lambda, B')$ for some $B' \subseteq \mathsf{pref}(L')$.*

So Lemma 3 says that child sub-processes can be abstracted by their external behaviors. Lemmas 2 and 4 allow to abstract a set L of external behaviors by a subset $L_1 \subseteq L$, as long as $L \subseteq \mathsf{lift}(L_1)$ holds. This property suggests to use 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 $\alpha \preccurlyeq \beta$ if $\beta \in \mathsf{lift}(\alpha)$. This extends to an order on $\Sigma_{sp} \cdot \Sigma_{ext}^*$: $\mathsf{spawn}(p)\alpha \preccurlyeq \mathsf{spawn}(q)\beta$ if $p = q$ and $\alpha \preccurlyeq \beta$. For a set $L \subseteq \Sigma_{sp} \cdot \Sigma_{ext}^*$, we define $\mathsf{core}(L)$ as the set of* minimal *words in L with respect to the relation $\preccurlyeq$.*

The following lemma is easy to see:

Lemma 5. *The relation $\preccurlyeq$ is a well-quasi-order on words with equal signature. Since the number of signatures is finite, the set $\mathsf{core}(L)$ is finite for every set $L \subseteq \Sigma_{sp} \cdot \Sigma_{ext}^*$.*

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

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

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

Proof. Since $\mathsf{core}(L) \subseteq L$, the right-to-left implication follows by monotonicity. For the other direction we observe that $L \subseteq \mathsf{lift}(\mathsf{core}(L))$, so we can use Lemma 4 and monotonicity. $\square$

Now we turn to the question of computing the relation $\overset{\alpha}{\dashrightarrow}_L$ 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 $L \subseteq L' \subseteq \mathsf{lift}(L)$. Consider the set*

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

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

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

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. For this we use the last two conditions of Def. 3. The admissibility of $\mathcal{C}$ (constructively decidable emptiness) is then used to compute the core of the language of the automaton thus obtained.

Corollary 2. *The sets $\mathsf{core}(\mathit{Ext}_0)$ and $\mathsf{core}(\mathit{Ext}_0 \cap \mathit{Consistent})$ are computable.*

Corollary 3. *Under the hypothesis of Proposition 2: for every $k \geq 0$, we can compute $\mathsf{core}(\mathit{Ext}_k)$ and $\mathsf{core}(\mathit{Ext}_k \cap \mathit{Consistent})$.*

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

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

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

If we denote $L_k = \mathsf{core}(\mathit{Ext}_k)$ then by Corollary 1, the language L is equal to

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

By definition, $\mathit{Ext}_0 \subseteq \mathit{Ext}_1 \subseteq \cdots$ is an increasing sequence of sets. By Lemma 5, this means that there is some m so that $\mathsf{core}(\mathit{Ext}_m) = \mathsf{core}(\mathit{Ext}_{m+i})$, for all i. Therefore, L' is equal to

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

By Corollary 3, the set $L_m = \mathsf{core}(\mathit{Ext}_m)$ is computable and so is $\mathsf{core}(L' \cap \mathit{Consistent})$. Finally, we check if in this latter set there is a sequence starting with $\mathsf{spawn}(q_{\mathsf{init}})$ and an external write or an output of #. □

6 Simpler cases

In this section we consider pushdown dynamic parametric processes, i.e., the case where each sub-process can have its own stack. We show several restrictions of the model yielding rather reasonable complexity for deciding reachability. Proofs are omitted in this section, they can be found in the full version of the paper `http://arxiv.org/abs/1609.05385`.

We start with the case where sub-processes cannot write to their own variables, but only to the variables of the parent. This corresponds to the situation when after a parent has created child sub-processes, these child sub-processes may communicate computed results to the parent, and to other sub-processes created by the parent, but the parent cannot communicate to child sub-processes. We call such a situation *systems with generalized futures.* We have seen an example of such a system in Figure 1. Technically, systems with generalized futures are obtained by disallowing $w(x, v)$ actions in our general definition. We need two more restrictions: first, we rule out global variables altogether; second, we require that the initial value v_{init} of a register can be neither read nor written. The first of the two restrictions follows the intuition that a parent should not communicate with child sub-processes; the reason behind the second restriction is that we want to avoid tracking when an initial value has been overwritten.

The reachability problem here is defined by the occurrence of an output action $o(x, \#)$ of some special value $\#$.

For systems with generalized futures, reachability can be decided by a cheaper approach, based on signatures instead of cores.

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

A further simplification is to disallow $i(x, v)$ actions. This means that child sub-processes cannot read the value of the variable of their parents. Accordingly, they still can report results, but no longer communicate between themselves. We call this class *systems with simple futures.*

Theorem 3. *The reachability problem is* NP*-complete for pushdown dynamic parametric processes with simple futures, provided the number of variables is fixed. The complexity becomes* PTIME *if the number of values is also fixed.*

The two restrictions above disallowed global variables. The final restriction we consider is to disallow local variables and to admit global variables instead. It turns out that such systems can be *flattened* — implying that nesting of sub-process creation does no longer add to expressivity. Accordingly, the complexity is the same as for the parametrized model of [7,5]:

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

7 Conclusions

We have studied systems with parametric process creation where sub-processes may communicate via both global and local shared variables. This kind of communication can model e.g. communication with message queues of capacity one. We have shown that under mild conditions, the reachability problem for this model is decidable. The algorithm relies on abstracting 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, the presented approach avoids computing downward closures, showing that effectively computable downward closure is not needed in the general decidability results on flat systems from [11]. In particular, note that lossy channel systems form an admissible class of systems, whereas their downward closures are not effective.

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.