Constrained Dynamic Tree Networks

Matthew Hague[1] and Vincent Penelle[2]

[1] Royal Holloway, University of London matthew.hague@rhul.ac.uk
[2] Université de Bordeaux, LaBRI, UMR 5800, Talence, France
vincent.penelle@labri.fr

Abstract. We generalise Constrained Dynamic Pushdown Networks, introduced by Bouajjani *et al.*, to Constrained Dynamic Tree Networks. In this model, we have trees of processes which may monitor their children. We allow the processes to be defined by any computation model for which the alternating reachability problem is decidable. We address the problem of symbolic reachability analysis for this model. More precisely, we consider the problem of computing an effective representation of their reachability sets using finite state automata. We show that backwards reachability sets starting from regular sets of configurations are always regular. We provide an algorithm for computing backwards reachability sets using tree automata.

Keywords: Model-checking, Dynamic Networks, Concurrency, Pushdown Systems, Alternation, Higher-order, Collapsible Pushdown Systems

1 Introduction

Bouajjani *et al.* [2] defined Constrained Dynamic Networks of Pushdown Systems: a model of concurrent computation where configurations of processes are tree structures, and each process is given by a pushdown system. During an execution, new child processes can be created, and a parent can test the states of its children before performing an execution step. They considered the global backwards reachability problem for these systems. That is, given a regular set of target configurations, compute the set of configurations that can reach the target set. They showed that, under a *stability* constraint, this backwards reachability set is regular and computable.

The stability constraint requires that once a test a parent may make on its children is satisfied, then it will remain satisfied, even if the children continue their execution. In the simplest case, this allows a parent to test for termination in a given state of its children. In general, this constraint allows a parent to (repeatedly) test whether its children have passed certain stages of execution (and their state in doing so).

We show here that Bouajjani *et al.*'s result is not dependent on the processes in the tree being modelled by pushdown systems. In fact, all that is required is that the *alternating reachability problem* is decidable for the systems labelling the nodes in the tree. Intuitively, in the alternating reachability problem, some

steps during the run of the system may be required to split into separate paths. From the initial state, we ask whether all paths of the execution reach a given final state.

Thus, we introduce *Constrained Dynamic Tree Networks*, which are tree networks of processes as before, but the individual processes can be labelled by any system for which the alternating reachability problem is decidable.

One particular instance of interest is the case of networks of *collapsible pushdown systems* [14]. Collapsible pushdown systems are a generalisation of pushdown systems that are known to be equi-expressive with *Higher-Order Recursion Schemes*. The alternating reachability problem is known to be decidable for these systems [32]. In fact, the backwards reachability sets of alternating collapsible pushdown systems are also known to be computable and regular [4]. Thus, we obtain a new model of concurrent higher-order programs for which the backwards reachability sets are also computable and regular. An advantage of our approach is that we do not need to consider the technical difficulties of reasoning about collapsible pushdown systems. The proof presented here only needs to take care of the concurrent aspects of the computations. Thus, we obtain results for quite complex systems with a relatively modest proof.

Modern day programming increasingly embraces higher-order programming, both via the inclusion of higher-order constructs in languages such as C++, JavaScript and Python, but also via the importance of *callbacks* in highly popular technologies such as jQuery and Node.js. For example, to read a file in Node.js, one would write

```
fs.readFile('f.txt', function (err, data) { ..use data.. });
```

In this code, the call to `readFile` spawns a new thread that asynchronously reads `f.txt` and sends the `data` to the function argument. This function will have access to, and frequently use, the closure information of the scope in which it appears (for example, variables defined before the `readFile` statement). The rest of the program runs *in parallel* with this call. This style of programming is fundamental to both jQuery and Node.js programming, as well as being a popular for programs handling input events or slow IO operations such as fetching remote data or querying databases (e.g. HTML5's indexedDB).

Analysing such programs is a challenge for verification tools which usually do not model higher-order recursion, or closures, accurately. However, several higher-order model-checking tools have been recently developed. This trend was pioneered by Kobayashi *et al.* [18]. The feasibility of higher-order model-checking in practice has been demonstrated by numerous higher-order model-checking tools [17,19,21,30,5,6,36]. Since all of these tools can handle the alternating reachability problem, it is possible that our techniques may be used to provide model checking tools for concurrent higher-order programs.

Our construction follows Bouajjani *et al.* and uses a *saturation* method to construct a regular representation of the backwards reachability set. However, our automaton representation is different: it separates the representation of the system states from the tree structure. We also use different techniques to prove

correctness of the construction. In particular, our soundness proof works by defining and showing soundness of each transition of the automaton, rather than dissecting complete runs. This is an application of a technique first used for a saturation technique for solving parity games over pushdown systems [15].

The full version of this article with appendices is available online [16].

1.1 Related Work

Dynamic pushdown networks have been studied without the process tree structure or constraints allowing a parent to inspect its children [26,39]. Various decidability-preserving locking techniques have also been investigated [24]. Some of these works also allow the tree structure to be taken into account [23,31]. Touili and Atig have also considered communication structures that are not necessarily trees [41]. However, these works consider pushdown networks only.

There has been some work studying concurrent variants of recursion scheme model checking, including a context-bounded algorithm for recursion schemes [20], and further underapproximation methods such as phase-bounded, ordered, and scope-bounding [12,38]. These works allow only a fixed number of threads.

Dynamic thread creation is permitted by both Yasukata *et al.* [42,43] and by Chadha and Viswanathan [7]. In Yasukata *et al.*'s model, recursion schemes may spawn and join threads. Communication is permitted only via nested locks. Their work is a generalisation of results for order-1 pushdown systems [11]. Chadha and Viswanathan allow threads to be spawned, but only one thread runs at a time, and must run to completion. Moreover, the tree structure is not maintained.

The saturation technique was popularised by Bouajjani *et al.* [1] for the analysis of pushdown systems, which was implemented in the successful Moped tool [37,40]. Saturation methods also exist for *ground tree rewrite systems* and related systems [25,3,27], though use different techniques.

Ground tree rewrite systems may also be generalised to trees where the nodes are labelled by higher-order stacks. Penelle proves decidability of first order logic with reachability over such systems [34]. However, this result does not allow nodes to have an unbounded number of direct children, and does not consider collapsible stacks in their full generality.

A related model of tree rewriting was introduced by Clemente *et al.* [8]. This model allows more powerful rewriting rules than ground tree rewrite systems while still enjoying decidability of alternating reachability. It is shown that reachability over alternating variants of a number of pushdown system models can be reduced to this model. In particular, this includes (ordered) annotated pushdown systems which are tightly related to (concurrent) collapsible pushdown systems. We believe it is likely that constrained dynamic networks of annotated pushdown systems could also be encoded in this model. However, our result applies to any system for which alternating reachability is decidable, and does not require an encoding of the underlying model into any particular form.

There is various research into meta-results on the analysis of concurrent systems, where the concurrent structure is the object of the research. Recent work by La Torre *et al.* has shown that parameterised safety analysis is possible of asynchronous networks of shared-memory systems [22], provided, amongst other

constraints, the downwards closure of the system is computable. Muscholl *et al.* also consider a parameterised model where processes may spawn an arbitrary (uncontrolled) amount of identical child processes [29]. Collapsible pushdown systems are known to have these properties [9,13,44,33]. Other works have studied multi-stack pushdown systems and offered either bounded tree-width [28], or split-width [10], as explanations for decidability. However, these results have not been extended to higher orders.

2 Alternating transition system

We define the notion of alternating transition system. An alternating transition system accepts labels γ to which operations θ can be applied. Transitions of the system are of the form $s \xrightarrow{\theta} S$. where S is a set of states. A label γ is accepted from s whenever $\theta(\gamma)$ is accepted from every state in S. If a state s is final, it accepts all labels.

We will consider Γ to be a set of labels, and Ops a set of operations $\theta : \Gamma \nrightarrow \Gamma$ over Γ. Note, we do not require that θ is defined over all elements of Γ. We also define the special operation Id such that $\mathrm{Id}(\gamma) = \gamma$ for all $\gamma \in \Gamma$.

Definition 1 (Alternating transition systems over Γ, Ops). *An alternating transition system over Γ, Ops is a tuple $\mathcal{N} = (\mathbb{S}, \mathbb{F}, \eta)$, where $\mathbb{S}$ is a finite set of states, $\mathbb{F} \subseteq \mathbb{S}$ is the set of final states, $\eta \subseteq \mathbb{S} \times \mathrm{Ops} \times 2^{\mathbb{S}}$ is the set of transitions.*

Given $\gamma \in \Gamma$ and $s \in \mathbb{S}$, we inductively define acceptance of γ from s, denoted $\gamma \vdash s$. We have $\gamma \vdash s$ if s is final, or if there is a transition $\nu = (s, \theta, S)$ such that $\theta(\gamma)$ is defined and $\theta(\gamma) \vdash s'$ for every $s' \in S$.

Requirement In all the following, we suppose that for every alternating transition system $\mathcal{N}$, given $\gamma \in \Gamma$ and $s \in \mathbb{S}$, we can decide whether $\gamma \vdash s$.

Example 1. An alternating pushdown system with stack alphabet Σ is an alternating transition system with $\Gamma = \Sigma^*$ and the set of operations $\mathrm{Ops} = \{(a, u) \mid a \in \Sigma, u \in \Sigma^*\}$. where for all $w \in \Sigma^*$

$$(a,u)(w) = \begin{cases} uv & w = av \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Here we represent a stack as a word, and the top of the stack appears leftmost. A transition $s \xrightarrow{(a,u)} S$ represents an alternating transition from a configuration (s, aw) of the pushdown system (with control state s and stack aw) to a set of configurations containing (s', uw) for each $s' \in S$. We will have $aw \vdash s$ if we can show $s' \vdash uw$ for each $s' \in S$.

3 Constrained Dynamic Tree Networks

We define constrained dynamic tree networks (CDTNs), which allow process trees with dynamic thread creation and parents to inspect their children.

Definition 2 (Constrained Dynamic Tree Network over Γ, Ops). *A constrained dynamic network over Γ, Ops is a tuple $\mathcal{M} = (\mathrm{P}, \mathrm{F}, \delta)$ with:*

- P *is a finite set of states and* $\mathrm{F} \subseteq \mathrm{P}$ *is the set of final states,*
- δ *a finite set of transitions of the following form:*

  C1 $\phi : p \xrightarrow{\theta} p_a$*, with* $\theta \in \mathrm{Ops}, p, p_a \in \mathrm{P}$*, and* ϕ *is a regular language over* P^**.*

  C2 $\phi : p \to p_a \rhd p_b$*, with* $p, p_a, p_b \in \mathrm{P}$*, and* ϕ *is a regular language over* P^**.*

An $\mathcal{M}$-configuration is a tree labelled by $\mathrm{P} \times \Gamma$. Let $\mathcal{T}(\mathrm{P} \times \Gamma)$ denote the set of these configurations. More explicitly, a configuration is either a leaf node $(p, \gamma)(\emptyset)$ or a tree $(p, \gamma)(t_1, \cdots, t_m)$ with root (p, γ) and children $t_1, \cdots, t_m$ where $p \in \mathrm{P}$, $\gamma \in \Gamma$, and for each $1 \leq i \leq m$ we have that t_i is an $\mathcal{M}$-configuration. A *context* C is a tree labelled by $(\mathrm{P} \times \Gamma) \cup \{\Box\}$ containing exactly one node labelled by $\Box$, which is a leaf. We write $C[t]$ to denote the configuration obtained by replacing $\Box$ by t in C. Furthermore, let

$$S((p, \gamma)(t_1, \cdots, t_m)) = p$$

extract the *internal* state of the root node of a configuration.

Transitions of the form C1 apply θ to a node, while transitions of the form C2 create a new child process. That is, the application of a transition of the form C1 to a configuration $C[(p, \gamma)(t_1, \cdots, t_m)]$ yields $C[(p_a, \theta(\gamma))(t_1, \cdots, t_m)]$, if $\theta(\gamma)$ is defined and $S(t_1) \cdots S(t_m) \in \phi$. The application of a transition of the form C2 to a configuration $C[(p, \gamma)(t_1, \cdots, t_m)]$ yields the configuration $C[(p_a, \gamma)(t_1, \cdots, t_m, (p_b, \gamma)(\emptyset))]$ if $S(t_1) \cdots S(t_m) \in \phi$.

3.1 Stability constraint

We give the restriction on child constraints ϕ that allows us to preserve decidability of reachability for CDTNs. Intuitively, this constraint asserts that once a constraint ϕ is satisfied, it will remain satisfied even if its children progress.

Definition 3 (Stability relation [2]). *Given an alphabet* Σ *and a binary relation* ρ *over* Σ*, we say that a subset* $\mathfrak{S}$ *of* Σ *is* ρ*-stable if for every* $a, b \in \Sigma$*,* $\rho(a, b) \wedge a \in \mathfrak{S} \Rightarrow b \in \mathfrak{S}$*.*

A language L *is* ρ*-stable if it is defined by a regular expression of the form*

$$e ::= \mathfrak{S}\text{, } \rho\text{-stable set} \mid e + e \mid e.e \mid e^*$$

In [2], it is shown that if a language L is ρ-stable, for every $a, b \in \Sigma$, $u, v \in \Sigma^*$, $uav \in L \wedge \rho(a, b) \Rightarrow ubv \in L$. Given a CDTN $\mathcal{M} = (\mathrm{P}, \mathrm{F}, \delta)$ we define

$$\rho_\delta = \{(p, p') \mid \exists \phi : p \xrightarrow{\theta} p' \in \delta \vee \exists \phi : p \to p' \rhd p'' \in \delta\} \ .$$

We say $\mathcal{M}$ is ρ_δ-stable iff for all $\phi : p \xrightarrow{\theta} p_a \in \delta$ and $\phi : p \to p_a \rhd p_b \in \delta$ we have ϕ is ρ_δ-stable (can be checked looking at regular expressions defining each ϕ).

3.2 Automaton

We now define a notion of tree automata over the configurations of a constrained dynamic tree network. As these configurations can have an unbounded arity, we need to have an automaton model which can deal with unbounded arity, thus we use an adapted version of hedge automata. Transitions of our automata are of the form $p(L) \to q$, meaning they can rewrite a tree to a state q, if

- the internal state of its root is p,
- the i^{th} son of its root can be rewritten to the state q_i, and
- $q_1 \cdots q_m$ is in the regular language L (if the node has m sons).

Moreover, the automaton checks that the element of the root is accepted by an alternating transition system which is bound to the transition (more precisely, we will use a single alternating transition system for the whole automaton, which has a unique initial state for each rule of the automaton). In the following definition, let $\mathrm{Reg}(\mathcal{Q})$ be the set of regular languages over alphabet $\mathcal{Q}$.

Definition 4 ($\mathcal{M}$-automaton). *Given a CDTN $\mathcal{M} = (\mathrm{P}, \mathrm{F}, \delta)$, an $\mathcal{M}$-automaton is a tuple $\mathcal{A} = (\mathcal{Q}, \mathcal{F}, \Delta, \mathcal{N})$, where:*

- *$\mathcal{Q}$ is a finite set of states and $\mathcal{F} \subseteq \mathcal{Q}$ the set of final states,*
- *$\Delta \subseteq \mathrm{P} \times \mathrm{Reg}(\mathcal{Q}) \times \mathcal{Q}$ a finite set of transitions of the form $p(L) \to q$,*
- *$\mathcal{N} = (\mathbb{S}, \mathbb{F}, \eta)$ an alternating transition system over Γ, Ops, such that for every $r \in \Delta$, there is a unique state $s_r \in \mathbb{S}$. Without loss of generality, we suppose that these states have no incoming transition[3] and that these states are not final[4]. Intuitively, s_r accepts the set of elements of Γ that allow r to fire. A $\mathcal{M}$-automaton is analogous to a tree automaton, with the difference that letters are replaced with sets of labels accepted from a state of an alternating transition system.*

An $\mathcal{A}$-configuration is a tree labelled by $(\mathrm{P} \times \Gamma) \cup \mathcal{Q}$, such that only leaves can be labelled by $\mathcal{Q}$. Given a transition $r = p(L) \to q$ and two $\mathcal{A}$-configurations t and t', we have $t \xrightarrow{r} t'$ if and only if $t = C[(p, \gamma)(q_1, \cdots, q_m)]$, $t' = C[q]$, $q_1 \cdots q_m \in L$ and $\gamma \vdash s_r$.

Let $\xrightarrow[\Delta]{*}$ be the transitive closure of $\left(\bigcup_{r \in \Delta} \xrightarrow{r}\right)$. The set of $\mathcal{M}$-configurations recognised by $\mathcal{A}$ from the state q is $\mathcal{L}_q(\mathcal{A}) = \{t \in \mathcal{T}(\mathrm{P} \times \Gamma) \mid t \xrightarrow[\Delta]{*} q\}$.

Note, the membership problem for $\mathcal{M}$-automata is decidable whenever it is decidable whether $\gamma \vdash s$ for a given γ and s. Similarly, emptiness is decidable whenever it is decidable if $\exists \gamma . \gamma \vdash s$ for a given s.

[3] If it is not the case, we create a copy of these states on which we conserve all the transition as an "internal state", and remove the incoming transitions to these states.

[4] If so, for a state s_r, we create a new final state s and add the transition $s_r \xrightarrow{\mathrm{Id}} s$, and remove s_r from the set of final states.

Example 2. We can accept regular sets of pushdown networks as defined by Bouajjani *et al.* [2] by defining the word automata used to recognise pushdown stacks as alternating transition systems with operations of the form (a, ε), where ε is the empty word, and operations have the same semantics as in Example 1. That is, each operation consumes the leftmost character of the word representation of the stack. For this we will need an explicit end-of-stack marker.

4 Backwards Reachability

In this section, we show that we can compute the backwards reachability set of CDTNs. That is, if a CDTN $\mathcal{M}$ is ρ_δ-stable, then the set of predecessors of a regular set is regular. Here, by regular we mean the set is accepted by an $\mathcal{M}$-automaton. We remark in the conclusion how this notion of regularity may be related to a more conventional one.

Given S a set of $\mathcal{M}$-configurations, we denote $\mathrm{pre}^*_{\mathcal{M}}(S)$ the set of *predecessors* of elements of S, i.e., $\mathrm{pre}^*_{\mathcal{M}}(S) = \{s \mid \exists s' \in S, s \xrightarrow[\mathcal{M}]{*} s'\}$.

Theorem 1. *Given $\mathcal{M}$ a ρ_δ-stable CDTN and $\mathcal{A}$ an $\mathcal{M}$-automaton, it is possible to compute a $\mathcal{M}$-automaton $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}') = \mathrm{pre}^*_{\mathcal{M}}(\mathcal{L}(\mathcal{A}))$.*

For the proof, we construct the automaton $\mathcal{A}'$ from $\mathcal{A}$ and $\mathcal{M}$ in two steps.

4.1 The automaton $\mathcal{A}_p$.

First we add to the states of the automaton the internal state of the root of the $\mathcal{M}$-configuration that was reduced to this state. Informally, we replace every transition $p(L) \to q$ with $p(L) \to (q, p)$, so given an $\mathcal{M}$-configuration t such that if $t \xrightarrow[\Delta]{*} q$, we have $t \xrightarrow[\Delta_p]{*} (q, S(t))$. This will be useful in the actual construction of $\mathcal{A}'$, as to inversely apply $\mathcal{M}$-rules, we will need to check if the constraint of the rule is satisfied, which will be given by this information (using the stability property, as we remember the final state of the root of each son). More formally, we will also need to adapt the constraint L and to add states to the inner alternating transition system. For notational convenience, let $\mathcal{Q}_p = \mathcal{Q} \times \mathrm{P}$.

We define $\mathcal{A}_p = (\mathcal{Q}_p, \mathcal{F} \times \mathrm{P}, \Delta_p, \mathcal{N}_p)$, where

$$\Delta_p = \left\{ p(L_{\mathrm{P}}) \to (q,p) \;\middle|\; L_{\mathrm{P}} = \left\{ (q_1,p_1)\cdots(q_m,p_m) \;\middle|\; \begin{array}{c} p(L) \to q \in \Delta, \\ q_1 \cdots q_m \in L, \\ p_1, \cdots, p_m \in \mathrm{P} \end{array} \right\} \right\}$$

and $\mathcal{N}_p = (\mathbb{S}_p, \mathbb{F}_p, \eta_p)$, with

- $\mathbb{S}_p = \mathbb{S} \backslash \{s_r \mid r \in \Delta\} \cup \{s_r \mid r \in \Delta_p\}$,
- $\mathbb{F}_p = \mathbb{F} \cap \mathbb{S}_p \cup \{s_r \mid r = p(L_{\mathrm{P}}) \to (q,p), s_{r'} \in \mathbb{F}, r' = p(L) \to q\}$,
- $\eta_p = \eta \cup \{s_r \xrightarrow{\theta} S \mid s_{r'} \xrightarrow{\theta} S \in \eta, r = p(L_{\mathrm{P}}) \to (q,p), r' = p(L) \to q\}$.

Lemma 1. $\mathcal{L}(\mathcal{A}_p) = \mathcal{L}(\mathcal{A})$.

Proof. We only have to observe that for every t, $t \xrightarrow[\Delta_p]{*} (q, S(t))$ if and only if $t \xrightarrow[\Delta]{*} q$, and that (q,p) is final if and only if q is final.

4.2 From constraints over P to constraints over $\mathcal{Q}_p$.

In order to faithfully compute the automaton $\mathcal{A}'$, we need to be able to transfer the constraint of $\mathcal{M}$ to the states of $\mathcal{A}'$. Indeed, we need to recognise only valid predecessors of the configurations recognised by $\mathcal{A}$, i.e. those which satisfy the constraints ϕ. Given a regular language $\phi \subseteq \mathrm{P}^*$, we thus define $\langle\phi\rangle = \{(q_1,p_1)\cdots(q_m,p_m) \mid p_1\cdots p_m \in \phi, q_1,\cdots,q_m \in \mathcal{Q}\}$. It is straightforward to see that this language is also regular.

4.3 Closed set of constraints

During the construction of $\mathcal{A}'$, we add new transitions of the form $p(L) \to (q',p')$. The constraints L will be constructed from those already appearing in $\mathcal{A}_p$ and the constraints ϕ used in $\mathcal{M}$, using intersection and right-quotient operations. Intersection $L \cap \langle\phi\rangle$ allows us to check that the guarding constraint of an $\mathcal{M}$-rule is satisfied at the considered position in the configuration. The right-quotient

$$L(q,p)^{-1} = \{(q_1,p_1)\cdots(q_m,p_m) \mid (q_1,p_1)\cdots(q_m,p_m)(q,p) \in L\}$$

allows us to get immediate predecessors by an operation of the form C2. We define Λ to be the smallest family of languages over $\mathcal{Q}_p$ such that:

- If $r = p(L) \to (q,p) \in \Delta_p$, then $L \in \Lambda$,
- If $L \in \Lambda$ and $\tau = \phi : p \xrightarrow{\theta} p_a \in \delta$, or $\tau = \phi : p \to p_a \rhd p_b \in \delta$, then $L \cap \langle\phi\rangle \in \Lambda$,
- If $L \in \Lambda$ and $(q,p) \in \mathcal{Q}_p$, then $L(q,p)^{-1} \in \Lambda$.

Finiteness of Λ was shown by Bouajjani *et al.* [2]. To prove it, observe that as the L and ϕ are regular, there are automata recognising them. Moreover there is a finite number of such constraints. We can take the product of all these automata to get a finite automaton, and associate each constraint with a set of final states of the product. Indeed, each $L \in \Lambda$ can be associated with a set of final states (as taking the right-product is equivalent to moving backward by one transition, and as we already have a product automaton, we don't have to introduce new states for the intersection). Thus, only a finite number of automata can be generated.

Lemma 2. *[2, Lemma 3] Λ is finite.*

4.4 Constructing $\mathcal{A}'$.

We now actually describe our saturation algorithm constructing $\mathcal{A}'$. To do so we start from $\mathcal{A}_p$ and only add new transitions: we will never add new states, so this process terminates. The main idea is, for every $\mathcal{M}$-rule $r = \phi : p \xrightarrow{\theta} p_a$ and every transition $p_a(L) \to (q',p')$ starting with p_a, to add a new transition starting with p and ending in the same states (q',p'). Moreover, we ensure the sons of the node we apply the rule to satisfy ϕ by setting the constraint of the rule to $L \cap \langle\phi\rangle$. We also ensure that the elements recognised from the state associated with the new rule are predecessors by θ of those recognised from the one associated with the old rule. For the spawning rule $\phi : p \to p_a \rhd p_b$, we moreover ensure that there is exactly one son less and the label was also accepted by the last son. We

need that the label was also accepted by the last son since the spawn operation creates a copy of the parent process's label. Hence, the label of the parent must also be the label of the last son.

We construct $\mathcal{A}' = (\mathcal{Q} \times \mathrm{P}, \mathcal{F} \times \mathrm{P}, \Delta', \mathcal{N}')$, with $\mathcal{N}' = (\mathbb{S}_p, \mathbb{F}_p, \eta')$. We give the formal definition of the construction first, and then informally explain the two rules R1. and R2..

We define Δ' and η' inductively as the fixed point of the following sequence. We begin with $\Delta'_0 = \Delta_p$ and $\eta'_0 = \eta_p$. Now, suppose Δ'_{i-1} and η'_{i-1} are defined. We construct Δ'_i and η'_i to be at least Δ'_{i-1} and η'_{i-1} plus transitions added with one of the following rules:

R1. if we have:
- $\tau = \phi : p \xrightarrow{\theta} p_a \in \delta$,
- $r = p_a(L) \to (q', p') \in \Delta'_{i-1}$,

we add
- $r' = p(L \cap \langle\phi\rangle) \to (q', p')$ to Δ'_i,
- $\nu' = s_{r'} \xrightarrow{\theta} \{s_r\}$ to η'_i.

R2. if we have:
- $\tau = \phi : p \to p_a \rhd p_b \in \delta$,
- $r_1 = p_a(L_1) \to (q', p') \in \Delta'_{i-1}$,
- $r_2 = p_b(L_2) \to (q'', p'') \in \Delta'_{i-1}$, with $\varepsilon \in L_2$,

we add
- $r' = p(L_1(q'', p'')^{-1} \cap \langle\phi\rangle) \to (q', p')$ to Δ'_i,
- $\nu' = s_{r'} \xrightarrow{\mathrm{Id}} \{s_{r_1}, s_{r_2}\}$ to η'_i.

This process terminates when $\Delta'_{i-1} = \Delta'_i$ and $\eta'_{i-1} = \eta'_i$. As the set of states is fixed, there is a finite number of possible rules, thus we terminate and $\mathcal{A}'$ exists.

Intuitively, R1. works as follows. We want to extend the automaton to recognise the result of a reverse application of $\phi : p \xrightarrow{\theta} p_a$. That is, whenever a configuration t' with the root node having internal state p_a is accepted, we should now accept a configuration t with root internal state p. Hence, we look for a transition (r) that will read and accept the root node of t' and introduce a new transition (r') that will read and accept the root of t. In addition, we need to take care of the children of the root. In particular, to be able to apply τ the children must satisfy ϕ. This is why we intersect with $\langle\phi\rangle$. Furthermore, to simulate the (reverse) update to γ, we add the transition $s_{r'} \xrightarrow{\theta} \{s_r\}$ to assert that the label accepted by r' would be accepted by r after an application of θ.

The rule R2. works similarly to R1., except we need to deal with the addition of a new child in the transition from t to t'. This is a removal when applied in reverse, hence the introduced transition performs a right-quotient on the language of children. In addition, we have to ensure that the spawned child has the same label as the parent. To do this, we look at the transition r_2 used to accept the final child. Note, the right quotient removes the target (q'', p'') of this transition. When applying this transition is reverse, the label γ of the root of t must be the same as the label of the root of t' and its final child. This explains the transition $s_{r'} \xrightarrow{\mathrm{Id}} \{s_{r_1}, s_{r_2}\}$ which ensures γ is accepted at both the root and its final child.

5 Correctness

We show that $\mathcal{A}'$ accepts $\text{pre}^*_{\mathcal{M}}(\mathcal{L}(\mathcal{A}))$. It is sufficient to prove the following property, which we discuss in the following subsections.

Proposition 1. *Given* $(q,p) \in \mathcal{Q}_p$*, we have* $\mathcal{L}_{(q,p)}(\mathcal{A}') = \text{pre}^*_{\mathcal{M}}(\mathcal{L}_{(q,p)}(\mathcal{A}_p))$.

5.1 Soundness

Proposition 2. *Given* $(q,p) \in \mathcal{Q}_p$*, we have* $\mathcal{L}_{(q,p)}(\mathcal{A}') \subseteq \text{pre}^*_{\mathcal{M}}(\mathcal{L}_{(q,p)}(\mathcal{A}_p))$.

We give the complete proof of this proposition in the full version. Intuitively, to prove this proposition, we associate to each state of an automaton (and the inner alternating transition system as well) a *meaning* that is intimately connected to the backwards reachability set we want to construct. We consider a transition $r = p(L) \to (q,p')$ to be sound under the following condition: if we take elements satisfying the meaning of each state appearing at the left of the transition, then the configuration including all these elements satisfies the meaning of the right state of the transition. Intuitively this says that, assuming all actions taken by other transitions in the automaton are correct, the current transition does nothing wrong. We inductively show that every transition appearing in $\mathcal{A}'$ is sound. Finally, we show that if an automaton is sound and contains $\mathcal{A}_p$, it satisfies the proposition, showing that it is the case for $\mathcal{A}'$.

5.2 Completeness

The proof of completeness of $\mathcal{A}'$ is conceptually simpler than the soundness proof. It proceeds by a straightforward induction over the length of the run showing a configuration is in the backwards reachability set. In the base case we have the configuration is accepted by $\mathcal{A}_p$ and the proof is immediate. In the inductive case, we have t reaches t' by a single transition, and an accepting run of $\mathcal{A}'$ over t'. We then inspect the transition from t to t' and show that our construction of $\mathcal{A}'$ ensures that we can modify the accepting run of t' to obtain an accepting run of t. For space reasons, we give the proof in the full version.

Proposition 3. *Given* $(q,p) \in \mathcal{Q}_p$*, we have* $\text{pre}^*_{\mathcal{M}}(\mathcal{L}_{(q,p)}(\mathcal{A}_p)) \subseteq \mathcal{L}_{(q,p)}(\mathcal{A}')$.

6 Conclusion

We have shown that the saturation algorithm for constrained dynamic pushdown networks introduced by Bouajjani *et al.* [2] can be generalised to not only pushdown networks, but networks of any system for which the alternating reachability problem is decidable. In particular, this includes collapsible pushdown systems, or higher-order recursion schemes, which thus allows the analysis of a kind of concurrent higher-order programs.

We showed that, given a target set of configurations represented by an $\mathcal{M}$-automata, the backwards reachability set is computable and also representable by an $\mathcal{M}$-automaton. We make some remarks on $\mathcal{M}$-automata as a notion of regularity. In order to accept a configuration, an automaton must perform several alternating reachability checks. This is not regular in the conventional sense. However, for alternating pushdown systems, and indeed alternating collapsible

pushdown systems, the backwards reachability set of a regular set of stacks is known to have a regular representation [1,4]. Thus, we can replace the alternating reachability tests with regular automata which run over the stack contents labelling each node. Thus we obtain a truly regular representation of the backwards reachability sets of CDTNs over these systems.

A natural avenue of future work is to attempt to generalise our model further, to permit more intricate communication between processes. One option is to allow the child nodes to inspect the internal state of their parent processes. In general this leads to an undecidable model. It is an open problem to discover a form of interesting upwards communication that is decidable. Similarly, we may seek to relax the stability constraint. One such option is to use the stability constraint defined by Touili and Atig [41] where internal states are grouped into mutually reachable equivalence classes. Thus, any run moves through a bounded number of equivalence classes. We can then insist that constraints are over the equivalence classes rather than individual states. This is reminiscent of *context-bounded* analysis [35]. We can adapt our construction to allow downwards and upwards communication of this form, but it is not clear whether Λ remains finite.

Acknowledgments We thank the anonymous reviewers for their remarks. This work was supported by the Engineering and Physical Sciences Research Council [EP/K009907/1].

References

1. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR*, 1997.
2. A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *CONCUR*, 2005.
3. W. S. Brainerd. Tree generating regular systems. *Information and Control*, 14(2):217–231, 1969.
4. C. H. Broadbent, A. Carayol, M. Hague, and O. Serre. A saturation method for collapsible pushdown systems. In *ICALP*, 2012.
5. C. H. Broadbent, A. Carayol, M. Hague, and O. Serre. C-shore: a collapsible approach to higher-order verification. In *ICFP*, 2013.
6. C. H. Broadbent and N. Kobayashi. Saturation-based model checking of higher-order recursion schemes. In *CSL*, 2013.
7. R. Chadha and M. Viswanathan. Decidability results for well-structured transition systems with auxiliary storage. In *CONCUR*, 2007.
8. L. Clemente, P. Parys, S. Salvati, and I. Walukiewicz. Ordered tree-pushdown systems. In *FSTTCS*, 2015.
9. L. Clemente, P. Parys, S. Salvati, and I. Walukiewicz. The diagonal problem for higher-order recursive schemes is decidable. In *LiCS*, 2016.
10. A. Cyriac, P. Gastin, and K. N. Kumar. MSO decidability of multi-pushdown systems via split-width. In *CONCUR*, 2012.
11. 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 *VMCAI*, 2011.
12. M. Hague. Saturation of concurrent collapsible pushdown systems. In *FSTTCS*, 2013.

13. M. Hague, J. Kochems, and C.-H. L. Ong. Unboundedness and downward closures of higher-order pushdown automata. In *POPL*, 2016.
14. M. Hague, A. S. Murawski, C.-H. Luke Ong, and O. Serre. Collapsible pushdown automata and recursion schemes. In *LiCS*, 2008.
15. M. Hague and C.-H. L. Ong. Winning regions of pushdown parity games: A saturation method. In *CONCUR*, 2009.
16. M. Hague and V. Penelle. Constrained dynamic tree networks, 2018. URL: `https://figshare.com/articles/main_pdf/6850508`, `doi:10.17637/rh.6850508`.
17. N. Kobayashi. Model-checking higher-order functions. In *PPDP*, 2009.
18. N. Kobayashi. Higher-order model checking: From theory to practice. In *LiCS*, 2011.
19. N. Kobayashi. A practical l inear time algorithm for trivial automata model checking of higher-order recursion schemes. In *FoSSaCS*, 2011.
20. N. Kobayashi and A. Igarashi. Model-checking higher-order programs with recursive types. In *ESOP*, 2013.
21. Naoki Kobayashi. GTRecS2: A model checker for recursion schemes based on games and types. A tool available at `http://www-kb.is.s.u-tokyo.ac.jp/~koba/gtrecs2/`, 2012.
22. S. La Torre, A. Muscholl, and I. Walukiewicz. Safety of parametrized asynchronous shared-memory systems is almost always decidable. In *CONCUR*, 2015.
23. P. Lammich, M. Müller-Olm, and A. Wenner. Predecessor sets of dynamic pushdown networks with tree-regular constraints. In *CAV*, 2009.
24. Peter Lammich, Markus Müller-Olm, Helmut Seidl, and Alexander Wenner. Contextual locking for dynamic pushdown networks. In *SAS*, 2013.
25. C. Löding. *Infinite Graphs Generated by Tree Rewriting*. PhD thesis, RWTH Aachen, 2003.
26. D. Lugiez. Forward analysis of dynamic network of pushdown systems is easier without order. *Int. J. Found. Comput. Sci.*, 22(4):843–862, 2011.
27. D. Lugiez and P. Schnoebelen. The regular viewpoint on pa-processes. In *CONCUR*, 1998.
28. P. Madhusudan and G. Parlato. The tree width of auxiliary storage. In *POPL*, 2011.
29. A. Muscholl, H. Seidl, and I. Walukiewicz. Reachability for dynamic parametric processes. In *VMCAI*, 2017.
30. R. P. Neatherway, S. J. Ramsay, and C.-H. L. Ong. A traversal-based algorithm for higher-order model checking. In *ICFP*, 2012.
31. Benedikt Nordhoff, Markus Müller-Olm, and Peter Lammich. Iterable forward reachability analysis of monitor-dpns. In *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday*, 2013.
32. C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LiCS*, 2006.
33. P. Parys. The complexity of the diagonal problem for recursion schemes. In *FSTTCS*, 2018.
34. V. Penelle. Rewriting higher-order stack trees. In *CSR*, 2015.
35. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, 2005.
36. S. J. Ramsay, R. P. Neatherway, and C.-H. Luke Ong. A type-directed abstraction refinement approach to higher-order model checking. In *POPL*, 2014.
37. S. Schwoon. *Model-checking Pushdown Systems*. PhD thesis, Technical University of Munich, 2002.

38. A. Seth. Games on higher order multi-stack pushdown systems. In *RP*, 2009.
39. F. Song and T. Touili. Model checking dynamic pushdown networks. *Formal Asp. Comput.*, 27(2):397–421, 2015.
40. D. Suwimonteerabuth, F. Berger, S. Schwoon, and J. Esparza. jmoped: A test environment for java programs. In *CAV*, 2007.
41. T. Touili and M. Faouzi Atig. Verifying parallel programs with dynamic communication structures. *Theor. Comput. Sci.*, 411(38-39):3460–3468, 2010.
42. K. Yasukata, N. Kobayashi, and K. Matsuda. Pairwise reachability analysis for higher order concurrent programs by higher-order model checking. In *CONCUR*, 2014.
43. K. Yasukata, T. Tsukada, and N. Kobayashi. Verification of higher-order concurrent programs with dynamic resource creation. In *APLAS*, 2016.
44. G. Zetzsche. An approach to computing downward closures. In *ICALP*, 2015.