

General Decidability Results for Asynchronous Shared-Memory Programs: Higher-Order and Beyond

RUPAK MAJUMDAR, Max Planck Institute for Software Systems (MPI-SWS), Germany
 RAMANATHAN S. THINNIYAM, Max Planck Institute for Software Systems (MPI-SWS), Germany
 GEORG ZETZSCHE, Max Planck Institute for Software Systems (MPI-SWS), Germany

The model of asynchronous programming arises in many contexts, from low-level systems software to high-level web programming. We take a language-theoretic perspective and show general decidability and undecidability results for asynchronous programs that capture all known results as well as show decidability of new and important classes. As a main consequence, we show decidability of safety, termination and boundedness verification for *higher-order* asynchronous programs—such as OCaml programs using Lwt—and undecidability of liveness verification already for order-2 asynchronous programs. We show that under mild assumptions, surprisingly, safety and termination verification of asynchronous programs with handlers from a language class are decidable *iff* emptiness is decidable for the underlying language class. Moreover, we show that configuration reachability and liveness (fair termination) verification are equivalent, and decidability of these problems implies decidability of the well-known “equal-letters” problem on languages. Our results close the decidability frontier for asynchronous programs.

1 INTRODUCTION

Asynchronous programming is a common way to manage concurrent requests in a system. In this style of programming, rather than waiting for a time-consuming operation to complete, the programmer can make *asynchronous* procedure calls which are stored in a *task buffer* pending later execution. Each asynchronous procedure, or *handler*, is a sequential program. When run, it can change the *global shared state* of the program, make internal synchronous procedure calls, and post further instances of handlers to the task buffer. A scheduler repeatedly and non-deterministically picks pending handler instances from the task buffer and executes their code *atomically* to completion. Asynchronous programs appear in many domains, such as operating system kernel code, web programming, or user applications on mobile platforms. This style of programming is supported natively or through libraries for most programming environments. The interleaving of different handlers hides latencies of long-running operations: the program can process a different handler while waiting for an external operation to finish. However, asynchronous scheduling of tasks introduces non-determinism in the system, making it difficult to reason about correctness.

An asynchronous program is *finite-data* if all program variables range over finite domains. Finite-data programs are still infinite state transition systems: the task buffer can contain an unbounded number of pending instances and the sequential machine implementing an individual handler can have unboundedly large state (e.g., if the handler is given as a recursive program, the stack can grow unboundedly). Nevertheless, verification problems for finite-data programs have been shown to be decidable for several kinds of handlers [Chadha and Viswanathan 2007;

Authors’ addresses: Rupak Majumdar, Max Planck Institute for Software Systems (MPI-SWS), Paul-Ehrlich-Straße, Building G26, Kaiserslautern, 67663, Germany, rupak@mpi-sws.org; Ramanathan S. Thinniyam, Max Planck Institute for Software Systems (MPI-SWS), Paul-Ehrlich-Straße, Building G26, Kaiserslautern, 67663, Germany, thinniyam@mpi-sws.org; Georg Zetzsche, Max Planck Institute for Software Systems (MPI-SWS), Paul-Ehrlich-Straße, Building G26, Kaiserslautern, 67663, Germany, georg@mpi-sws.org.

[Ganty and Majumdar 2012; Jhala and Majumdar 2007; Sen and Viswanathan 2006]. Several algorithmic approaches have been studied, which tailor to (i) the kinds of permitted handler programs and (ii) the properties that are checked.

State of the art We briefly survey the existing approaches and what is known about the decidability frontier. The *Parikh approach* applies to (first-order) recursive handler programs. Here, the decision problems for asynchronous programs are reduced to decision problems over Petri nets [Ganty and Majumdar 2012]. The key insight is that since handlers are executed atomically, the order in which a handler posts tasks to the buffer is irrelevant. Therefore, instead of considering the sequential order of posted tasks along an execution, one can equivalently consider its Parikh image. Thus, when handlers are given pushdown systems, the behaviors of an asynchronous program can be represented by a (polynomial sized) Petri net. Using the Parikh approach, safety (formulated as reachability of a global state), termination (whether all executions terminate), and boundedness (whether there is an a priori upper bound on the task buffer) are all decidable for asynchronous programs with recursive handlers, by reduction to corresponding problems on Petri nets [Ganty and Majumdar 2012; Sen and Viswanathan 2006]. Configuration reachability (reachability of a specific global state and task buffer configuration), fair termination (termination under a fair scheduler), and fair non-starvation (every pending handler instance is eventually executed) are also decidable, by separate ad hoc reductions to Petri net reachability [Ganty and Majumdar 2012]. A “reverse reduction” shows that Petri nets can be simulated by polynomial-sized asynchronous programs (already with finite-data handlers).

In the *downclosure approach*, one replaces each handler with a finite-data program that is equivalent up to “losing” handlers in the task buffer. Of course, this requires that one can compute equivalent finite-data programs for given handler programs. This has been applied to checking safety for recursive handler programs [Atig et al. 2009]. Finally, a bespoke *rank-based approach* has been applied to checking safety when handlers can perform restricted higher-order recursion [Chadha and Viswanathan 2007].

Contribution Instead of studying individual kinds of handler programs, we consider asynchronous programs in a general language-theoretic framework. The class of handler programs is given as a language class C : An asynchronous program over a language class C is one where each handler defines a language from C over the alphabet of handler names, as well as a transformer over the global state. This view leads to general results: we can obtain simple characterizations of which classes of handler programs permit decidability. For example, we do not need the technical assumptions of computability of equivalent finite-data programs from the Parikh and the downclosure approach.

Our first result shows that, under a mild language-theoretic assumption, safety and termination are decidable if and only if the underlying language class C has decidable emptiness problem.¹ Similarly, we show that boundedness is decidable iff *finiteness* is decidable for the language class C . These results are the best possible: decidability of emptiness (resp., finiteness) is a requirement for safety and termination verification already for verifying the safety or termination (resp., boundedness) of one *sequential* handler call. As corollaries, we get new decidability results for all these problems for asynchronous programs over *higher-order recursion schemes*, which form the language-theoretic basis for programming in higher-order functional languages such as OCaml [Kobayashi 2009; Ong 2015], as well as other language classes (lossy channel languages, Petri net languages, etc.).

¹The “mild language-theoretic assumption” is that the class of languages forms an effective full trio: it is closed under intersections with regular languages, homomorphisms, and inverse homomorphisms. Many language classes studied in formal language theory and verification satisfy these conditions.

Second, we show that configuration reachability, fair termination, and fair starvation are mutually reducible; thus, decidability of any one of them implies decidability of all of them. We also show decidability of these problems implies the decidability of a well-known combinatorial problem on languages: given a language over the alphabet $\{a, b\}$, decide if it contains a word with an equal number of a s and b s. Viewed contrapositively, we conclude that all these decision problems are undecidable already for asynchronous programs over order-2 pushdown languages, since the equal-letters problem is undecidable for this class.

Together, our results “close” the decidability frontier for asynchronous programs, by demonstrating reducibilities between decision problems heretofore studied separately and connecting decision problems on asynchronous programs with decision problems on the underlying language classes of their handlers.

While our algorithms do not assume that downclosures are effectively computable, we use downclosures to prove their correctness. We show that safety, termination, and boundedness problems are invariant under taking downclosures of runs; this corresponds to taking downclosures of the languages of handlers.

The observation that safety, termination, and boundedness depend only on the downclosure suggests a possible route to implementation. If there is an effective procedure to compute the downclosure for class C , then a direct verification algorithm would replace all handlers by their (regular) downclosures, and invoke existing decision procedures for this case. Thus, we get a direct algorithm based on downclosure constructions for higher order recursion schemes, using the string of celebrated recent results on effectively computing the downclosure of *word schemes* [Clemente et al. 2016; Hague et al. 2016; Zetsche 2015].

We find our general decidability result for asynchronous programs to be surprising. Already for regular languages, the complexity of safety verification jumps from NL (NFA emptiness) to EXPSPACE (Petri net coverability): asynchronous programs are far more expressive than individual handler languages. It is therefore surprising that safety and termination verification remains decidable whenever it is decidable for individual handler languages.

2 PRELIMINARIES

Basic Definitions. We assume familiarity with basic definitions of automata theory (see, e.g., [Hopcroft et al. 2007; Sipser 1997]). The projection of word w onto some alphabet Σ' , written $\text{Proj}_{\Sigma'}(w)$, is the word obtained by erasing from w each symbol which does not belong to Σ' . For a language L , define $\text{Proj}_{\Sigma'}(L) = \{\text{Proj}_{\Sigma'}(w) \mid w \in L\}$. The *subword* order \sqsubseteq on Σ^* is defined as $w \sqsubseteq w'$ for $w, w' \in \Sigma^*$ if w can be obtained from w' by deleting some letters from w' . For example, $abba \sqsubseteq bababa$ but $abba \not\sqsubseteq baaba$. The *downclosure* $\downarrow w$ with respect to the subword order of a word $w \in \Sigma^*$ is defined as $\downarrow w := \{w' \in \Sigma^* \mid w' \sqsubseteq w\}$. The downclosure $\downarrow L$ of a language $L \subseteq \Sigma^*$ is given by $\downarrow L := \{w' \in \Sigma^* \mid \exists w \in L: w' \sqsubseteq w\}$. Recall that the downclosure $\downarrow L$ of any language L is a regular language [Haines 1969].

A *multiset* $\mathbf{m}: \Sigma \rightarrow \mathbb{N}$ over Σ maps each symbol of Σ to a natural number. Let $\mathbb{M}[\Sigma]$ be the set of all multisets over Σ . We treat sets as a special case of multisets where each element is mapped onto 0 or 1. As an example, we write $\mathbf{m} = \llbracket a, a, c \rrbracket$ for the multiset $\mathbf{m} \in \mathbb{M}[\{a, b, c, d\}]$ such that $\mathbf{m}(a) = 2$, $\mathbf{m}(b) = \mathbf{m}(d) = 0$, and $\mathbf{m}(c) = 1$. We also write $|\mathbf{m}| = \sum_{\sigma \in \Sigma} \mathbf{m}(\sigma)$.

Given two multisets $\mathbf{m}, \mathbf{m}' \in \mathbb{M}[\Sigma]$ we define the multiset $\mathbf{m} \oplus \mathbf{m}' \in \mathbb{M}[\Sigma]$ for which, for all $a \in \Sigma$, we have $(\mathbf{m} \oplus \mathbf{m}')(a) = \mathbf{m}(a) + \mathbf{m}'(a)$. We also define the natural order \leq on $\mathbb{M}[\Sigma]$ as follows: $\mathbf{m} \leq \mathbf{m}'$ iff there exists $\mathbf{m}^\Delta \in \mathbb{M}[\Sigma]$ such that $\mathbf{m} \oplus \mathbf{m}^\Delta = \mathbf{m}'$. We also define $\mathbf{m}' \ominus \mathbf{m}$ for $\mathbf{m} \leq \mathbf{m}'$ analogously: for all $a \in \Sigma$, we have $(\mathbf{m}' \ominus \mathbf{m})(a) = \mathbf{m}'(a) - \mathbf{m}(a)$. For $\Sigma \subseteq \Sigma'$ we regard $\mathbf{m} \in \mathbb{M}[\Sigma]$ as a multiset of $\mathbb{M}[\Sigma']$ where undefined values are sent to 0.

Language Classes and Full Trios. A *language class* is a collection of languages, together with some finite representation. Examples are the regular (e.g. represented by finite automata) or the context-free languages (e.g. represented by pushdown automata or PDA). A relatively weak and reasonable assumption on a language class is that it is a *full trio*, that is, it is closed under each of the following operations: taking intersection with a regular language, taking homomorphic images, and taking inverse homomorphic images. Equivalently, a language class is a full trio iff it is closed under *rational transductions* [Berstel 1979].

We assume that all full trios C considered in this paper are *effective*: Given a language L from C , a regular language R , and a homomorphism h , we can compute a representation of the languages $L \cap R$, $h(L)$, and $h^{-1}(L)$ in C .

Many classes of languages studied in formal language theory form effective full trios. Examples include the regular and the context-free languages [Hopcroft et al. 2007], the indexed languages [Aho 1968; Damm and Goerdts 1986], the languages of higher-order pushdown automata [Maslov 1974], higher-order recursion schemes (HORS) [Damm 1982; Hague et al. 2008], Petri nets [Greibach 1978; Jantzen 1979], and lossy channel systems (see Section 4.1). (While HORS are usually viewed as representing a tree or collection of trees, one can also view them as representing a word language, as we explain in Section 5.)

Informally, a language class defined by non-deterministic devices with a finite-state control that allows ε -transitions and imposes no restriction between input letter and performed configuration changes (such as non-deterministic pushdown automata) is always a full trio: The three operations above can be realized by simple modifications of the finite-state control. The deterministic context-free languages are a class that is *not* a full trio.

An *asynchronous transducer* \mathcal{T} is a tuple $\mathcal{T} = (Q, \Gamma, \Sigma, E, q_0, F)$ with a set of finite states Q , finite output alphabet Γ , finite input alphabet Σ , a finite set of edges $E \subseteq Q \times \Gamma^* \times \Sigma^* \times Q$, initial state $q_0 \in Q$ and set of final states $F \subseteq Q$. We write $p \xrightarrow{v|u} q$ if $(p, v, u, q) \in E$ and the machine reads u in state p , outputs v and moves to state q . We also write $p \xrightarrow{w|w'} q$ if there are states q_0, q_1, \dots, q_n and words $u_1, u_2, \dots, u_n, v_1, v_2, \dots, v_n$ such that $p = q_0, q = q_n, w' = u_1 u_2 \dots u_n, w = v_1 v_2 \dots v_n$ and $q_i \xrightarrow{v_i|u_i} q_{i+1}$ for all $0 \leq i \leq n$.

The *transduction* $T \subseteq \Gamma^* \times \Sigma^*$ generated by the transducer \mathcal{T} is the set of tuples $(v, u) \in \Gamma^* \times \Sigma^*$ such that $q_0 \xrightarrow{v|u} q_f$ for some $q_f \in F$. Given a language $L \subseteq \Sigma^*$, we define $TL := \{v \in \Gamma^* \mid \exists u \in L (v, u) \in T\}$. A transduction $T \subseteq \Gamma^* \times \Sigma^*$ is *rational* if it is generated by some asynchronous transducer.

A language class which is closed under rational transductions is called a *full trio*.

The following equivalent characterization of full trios is well known.

THEOREM 2.1 (BERSTEL [BERSTEL 1979]). *A language class C is a full trio if and only if it is closed under each of the following operations:*

- *intersection with a regular language,*
- *taking homomorphic images, and*
- *taking inverse homomorphic images.*

Asynchronous Programs: A Language-Theoretic View. We use a language-theoretic model for asynchronous shared-memory programs.

Definition 2.2. Let C be an (effective) full trio. An *asynchronous program* (AP) over C is a tuple $\mathfrak{P} = (D, \Sigma, (L_c)_{c \in \mathfrak{C}}, d_0, \mathbf{m}_0)$, where D is a finite set of *global states*, Σ is an alphabet of *handler names*, $(L_c)_{c \in \mathfrak{C}}$ is a family of languages from C , one for each $c \in \mathfrak{C}$ where $\mathfrak{C} = D \times \Sigma \times D$ is the set of *contexts*, $d_0 \in D$ is the *initial state*, and $\mathbf{m}_0 \in \mathbb{M}[\Sigma]$ is a multiset of *initial pending handler instances*.

```

1 global var turn = ref 0 and x = ref 0;
2 let rec s1 () = if * then begin post a; s1(); post b end
3 let rec s2 () = if * then begin post a; s2(); post b end else post b
4 let a () = if !turn = 0 then begin turn := 1; x := !x + 1 end else post a
5 let b () = if !turn = 1 then begin turn := 0; x := !x - 1 end else post b
6
7 let s3 () = post s3; post s3
8
9 global var t = ref 0;
10 let c () = if !t = 0 then t := 1 else post c
11 let d () = if !t = 1 then t := 2 else post d
12 let f () = if !t = 2 then t := 0 else post f
13
14 let cc x = post c; x
15 let dd x = post d; x
16 let ff x = post f; x
17 let id x = x
18 let h g y = cc (g (dd y))
19 let rec produce g x = if * then produce (h g) (ff x) else g x
20 let s4 () = produce id ()

```

Fig. 1. Examples of asynchronous programs

A configuration $(d, \mathbf{m}) \in D \times \mathbb{M}[\Sigma]$ of \mathfrak{P} consists of a global state d and a multiset \mathbf{m} of pending handler instances. For a configuration c , we write $c.d$ and $c.\mathbf{m}$ for the global state and the multiset in the configuration respectively. The *initial* configuration c_0 of \mathfrak{P} is given by $c_0.d = d_0$ and $c_0.\mathbf{m} = \mathbf{m}_0$. The semantics of \mathfrak{P} is given as a labeled transition system over the set of configurations, with the transition relation $\xrightarrow{\sigma} \subseteq (D \times \mathbb{M}[\Sigma]) \times (D \times \mathbb{M}[\Sigma])$ given by

$$(d, \mathbf{m} \oplus \llbracket \sigma \rrbracket) \xrightarrow{\sigma} (d', \mathbf{m} \oplus \mathbf{m}') \quad \text{iff} \quad \exists w \in L_{d\sigma d'} : \text{Parikh}(w) = \mathbf{m}'$$

We use \rightarrow^* for the reflexive transitive closure of the transition relation. A configuration c is said to be *reachable* in \mathfrak{P} if $(d_0, \mathbf{m}_0) \rightarrow^* c$.

Intuitively, the set Σ of handler names specifies a finite set of procedures that can be invoked asynchronously. The shared state takes values in D . When a handler is called asynchronously, it gets added to a bag of pending handler calls (the multiset \mathbf{m} in a configuration). The language $L_{d\sigma d'}$ captures the effect of executing an instance of σ starting from the global state d , such that on termination, the global state is d' . Each word $w \in L_{d\sigma d'}$ captures a possible sequence of handlers posted during the execution.

Suppose the current configuration is (d, \mathbf{m}) . A non-deterministic scheduler picks one of the outstanding handlers $\sigma \in \mathbf{m}$ and executes it. Executing σ corresponds to picking one of the languages $L_{d\sigma d'}$ and some word $w \in L_{d\sigma d'}$. Upon execution of σ , the new configuration has global state d' and the new bag of pending calls is obtained by taking \mathbf{m} , removing an instance of σ from it, and adding the Parikh image of w to it. This reflects the current set of pending handler calls—the old ones (minus an instance of σ) together with the new ones added by executing σ . Note that a handler is executed atomically; thus, we atomically update the global state and the effect of executing the handler.

Let us see some examples of asynchronous programs. It is convenient to present these examples in a programming language syntax, and to allow each handler to have *internal actions* that perform local tests and updates to the global state. As we describe informally below, and formally in the full version, when C is a full trio, internal actions can be “compiled away” by taking an intersection with a regular language of internal actions and projecting the internal actions away. Thus, we use our simpler model throughout.

Examples. Figure 1 shows some simple examples of asynchronous programs in an OCaml-like syntax. Consider first the asynchronous program in lines 1–5. The alphabet of handlers is $s1$, $s2$, a , and b . The global states correspond to possible valuations to the global variables $turn$ and x ; assuming $turn$ is a Boolean and x takes values in \mathbb{N} , we have that $D = \{0, 1\} \times \{0, 1, \omega\}$, where ω abstracts all values other than $\{0, 1\}$. Since $s1$ and $s2$ do not touch any variables, for $d, d' \in D$, we have $L_{d,s1,d} = \{a^n b^n \mid n \geq 0\}$, $L_{d,s2,d} = \{a^n b^{n+1} \mid n \geq 0\}$, and $L_{d,s1,d'} = L_{d,s2,d'} = \emptyset$ if $d' \neq d$.

For the languages corresponding to a and b , we use syntactic sugar in the form of *internal actions*; these are local tests and updates to the global state. For our example, we have, e.g., $L_{(0,0),a,(1,1)} = \{\epsilon\}$, $L_{(1,x),a,(1,x)} = \{a\}$ for all values of x , and similarly for b . The meaning is that, starting from a global state $(0, 0)$, executing the handler will lead to the global state $(1, 1)$ and no handlers will be posted, whereas starting from a global state in which $turn$ is 1, executing the handler will keep the global state unchanged but post an instance of a . Note that all the languages are context-free.

Consider an execution of the program from the initial configuration $((0, 0), \llbracket s1 \rrbracket)$. The execution of $s1$ puts n a s and n b s into the bag, for some $n \geq 0$. The global variable $turn$ is used to ensure that the handlers a and b alternately update x . When $turn$ is 0, the handler for a increments x and sets $turn$ to 1, otherwise it re-posts itself for a future execution. Likewise, when $turn$ is 1, the handler for b decrements x and sets $turn$ back to 0, otherwise it re-posts itself for a future execution. As a result, the variable x never grows beyond 1. Thus, the program satisfies the *safety* property that no execution sets x to ω .

It is possible that the execution goes on forever: for example, if $s1$ posts an a and a b , and thereafter only b is chosen by the scheduler. This is not an “interesting” infinite execution as it is not fair to the pending a . In the case of a fair scheduler, which eventually always picks an instance of every pending task, the program terminates: eventually all the a s and b s are consumed when they are scheduled in alternation. However, if instead we started with $\llbracket s2 \rrbracket$, the program will not terminate even under a fair scheduler: the last remaining b will not be paired and will keep executing and re-posting itself forever.

Now consider the execution of $s3$. It has an infinite fair run, where the scheduler picks an instance of $s3$ at each step. However, the number of pending instances grows without bound. We shall study the *boundedness problem*, which checks if the bag can become unbounded along some run. We also study a stronger notion of fair termination, called *fair non-starvation*, which asks that every *instance* of a posted handler is executed under any fair scheduler. The execution of $s3$ is indeed fair, but there can be a specific instance of $s3$ that is never picked: we say $s3$ can *starve* an instance.

The program in lines 9–20 is *higher-order* (produce and h take functions as arguments). The language of $s4$ is the set $\{c^n d^n f^n \mid n \geq 0\}$, that is, it posts an equal number of c s, d s, and f s. It is an indexed language; we shall see (Section 5) how this and other higher-order programs can be represented using higher-order recursion schemes (HORS). Note the OCaml types of produce : $(o \rightarrow o) \rightarrow o \rightarrow o$ and $h : (o \rightarrow o) \rightarrow o \rightarrow o$ are higher-order.

The program is similar to the first: the handlers c , d , and f execute in “round robin” fashion using the global state t to find their turns. Again, we use internal actions to update the global state for readability. We ask the same decision questions as before: does the program ever reach a specific global state and does the program have an infinite (fair) run? We shall see later that safety and termination questions remain decidable, whereas fair termination does not.

3 DECISION PROBLEMS ON ASYNCHRONOUS PROGRAMS

We now describe decision problems on runs of asynchronous programs.

Runs, preruns, and downclosures. A *prerun* of an AP $\mathfrak{P} = (D, \Sigma, (L_c)_{c \in \mathbb{C}}, d_0, \mathbf{m}_0)$ is a finite or infinite sequence $\rho = (e_0, \mathbf{n}_0), \sigma_1, (e_1, \mathbf{n}_1), \sigma_2, \dots$ of alternating elements of tuples $(e_i, \mathbf{n}_i) \in D \times \mathbb{M}[\Sigma]$ and symbols $\sigma_i \in \Sigma$. The set of preruns of \mathfrak{P} will be denoted $\text{Preruns}(\mathfrak{P})$. Note that if two asynchronous programs \mathfrak{P} and \mathfrak{P}' have the same D and Σ , then $\text{Preruns}(\mathfrak{P}) = \text{Preruns}(\mathfrak{P}')$. The *length*, denoted $|\rho|$, of a finite prerun ρ is the number of configurations in ρ . The i^{th} configuration of a prerun ρ will be denoted $\rho(i)$.

We define an order \preceq on preruns as follows: For preruns $\rho = (e_0, \mathbf{n}_0), \sigma_1, (e_1, \mathbf{n}_1), \sigma_2, \dots$ and $\rho' = (e'_0, \mathbf{n}'_0), \sigma'_1, (e'_1, \mathbf{n}'_1), \sigma'_2, \dots$, we define $\rho \preceq \rho'$ if $|\rho| = |\rho'|$ and $e_i = e'_i, \sigma_i = \sigma'_i$ and $\mathbf{n}_i \leq \mathbf{n}'_i$ for each $i \geq 0$. The *downclosure* $\downarrow R$ of a set R of preruns of \mathfrak{P} is defined as $\downarrow R = \{\rho \in \text{Preruns}(\mathfrak{P}) \mid \exists \rho' \in R. \rho \preceq \rho'\}$.

A *run* of an AP $\mathfrak{P} = (D, \Sigma, (L_c)_{c \in \mathbb{C}}, d_0, \mathbf{m}_0)$ is a prerun $\rho = (d_0, \mathbf{m}_0), \sigma_1, (d_1, \mathbf{m}_1), \sigma_2, \dots$ starting with the initial configuration (d_0, \mathbf{m}_0) , where for each $i \geq 0$, we have $(d_i, \mathbf{m}_i) \xrightarrow{\sigma_{i+1}} (d_{i+1}, \mathbf{m}_{i+1})$. The set of runs of \mathfrak{P} is denoted $\text{Runs}(\mathfrak{P})$ and $\downarrow \text{Runs}(\mathfrak{P})$ is its downclosure with respect to \preceq .

An infinite run $c_0 \xrightarrow{\sigma_0} c_1 \xrightarrow{\sigma_1} \dots$ is *fair* if for all $i \geq 0$, if $\sigma \in c_i.\mathbf{m}$ then there is some $j \geq i$ such that $c_j \xrightarrow{\sigma} c_{j+1}$. That is, whenever an instance of a handler is posted, some instance of the handler is executed later. Fairness does not preclude that a specific instance of a handler is never executed. An infinite fair run *starves* handler σ if there exists an index $J \geq 0$ such that for each $j \geq J$, we have (i) $c_j.\mathbf{m}(\sigma) \geq 1$ and (ii) whenever $c_j \xrightarrow{\sigma} c_{j+1}$, we have $c_j.\mathbf{m}(\sigma) \geq 2$. In this case, even if the run is fair, a specific instance of σ may never be executed.

Now we give the definitions of the various decision problems.

Definition 3.1 (Properties of finite runs). The **Safety (Global state reachability)** problem asks, given an asynchronous program \mathfrak{P} and a global state $d_f \in D$, is there a reachable configuration c such that $c.d = d_f$? If so, d_f is said to be *reachable* (in \mathfrak{P}) and *unreachable* otherwise. The **Boundedness (of the task buffer)** problem asks, given an asynchronous program \mathfrak{P} , is there an $N \in \mathbb{N}$ such that for every reachable configuration c , we have $|c.\mathbf{m}| \leq N$? If so, the asynchronous program \mathfrak{P} is *bounded*; otherwise it is *unbounded*. The **Configuration reachability** problem asks, given an asynchronous program \mathfrak{P} and a configuration c , is c reachable?

Definition 3.2 (Properties of infinite runs). All the following problems take as input an asynchronous program \mathfrak{P} . The **Termination** problem asks if all runs of \mathfrak{P} are finite. The **Fair Non-termination** problem asks if \mathfrak{P} has some *fair* infinite run. The **Fair Starvation** problem asks if \mathfrak{P} has some fair run that starves some handler.

Our main result in this section shows that many properties of an asynchronous program \mathfrak{P} only depend on the downclosure $\downarrow \text{Runs}(\mathfrak{P})$ of the set $\text{Runs}(\mathfrak{P})$ of runs of the program \mathfrak{P} . The proof is by induction on the length of runs. Please see Appendix B for details. For any AP $\mathfrak{P} = (D, \Sigma, (L_c)_{c \in \mathbb{C}}, d_0, \mathbf{m}_0)$, we define the AP $\downarrow \mathfrak{P} = (D, \Sigma, (\downarrow L_c)_{c \in \mathbb{C}}, d_0, \mathbf{m}_0)$, where $\downarrow L_c$ is the downclosure of the language L_c under the subword order.

PROPOSITION 3.3. *Let $\mathfrak{P} = (D, \Sigma, (L_c)_{c \in \mathbb{C}}, d_0, \mathbf{m}_0)$ be an asynchronous program. Then $\downarrow \text{Runs}(\downarrow \mathfrak{P}) = \downarrow \text{Runs}(\mathfrak{P})$. In particular, the following holds. (1) For every $d \in D$, \mathfrak{P} can reach d if and only if $\downarrow \mathfrak{P}$ can reach d . (2) \mathfrak{P} is terminating if and only if $\downarrow \mathfrak{P}$ is terminating. (3) \mathfrak{P} is bounded if and only if $\downarrow \mathfrak{P}$ is bounded.*

Intuitively, safety, termination, and boundedness is preserved when the multiset of pending handler instances is “lossy”: posted handlers can get lost. This corresponds to these handlers never being scheduled by the scheduler. However, if a run demonstrates reachability of a global state, or non-termination, or unboundedness, in the lossy version, it corresponds also to a run in the original problem (and conversely).

In contrast, simple examples show that configuration reachability, fair termination, and fair non-starvation properties are not preserved under downclosures.

4 GENERAL DECIDABILITY RESULTS

In this section, we characterize those full trios C for which particular problems for asynchronous programs over C are decidable. Our decision procedures will use the following theorem, summarizing the results from [Ganty and Majumdar 2012], as a subprocedure.

THEOREM 4.1 ([GANTY AND MAJUMDAR 2012]). *Safety, boundedness, configuration reachability, termination, fair non-termination, and fair non-starvation are decidable for asynchronous programs over regular languages.*

4.1 Safety and termination

Our first main result concerns the problems of safety and termination.

THEOREM 4.2. *Let C be a full trio. The following are equivalent:*

- (i) *Safety is decidable for asynchronous programs over C .*
- (ii) *Termination is decidable for asynchronous programs over C .*
- (iii) *Emptiness is decidable for C .*

We begin with “(i) \Rightarrow (iii)”. Let $K \subseteq \Sigma^*$ be given. We construct $\mathfrak{P} = (D, \Sigma, (L_c)_{c \in \mathbb{C}}, d_0, \mathbf{m}_0)$ such that $\mathbf{m}_0 = \llbracket \sigma \rrbracket$, $D = \{d_0, d_1\}$, $L_{d_0, \sigma, d_1} = K$ and $L_c = \emptyset$ for $c \neq (d_0, \sigma, d_1)$. We see that \mathfrak{P} can reach d_1 iff K is non-empty. Next we show “(ii) \Rightarrow (iii)”. Consider the alphabet $\Gamma = (\Sigma \cup \{\varepsilon\}) \times \{0, 1\}$ and the homomorphisms $g: \Gamma^* \rightarrow \Sigma^*$ and $h: \Gamma^* \rightarrow \{\sigma\}^*$, where for $x \in \Sigma \cup \{\varepsilon\}$, we have $g((x, i)) = x$ for $i \in \{0, 1\}$, $h((x, 1)) = \sigma$, and $h((x, 0)) = \varepsilon$. If $R \subseteq \Gamma^*$ is the regular set of words in which exactly one position belongs to the subalphabet $(\Sigma \cup \{\varepsilon\}) \times \{1\}$, then the language $K' := h(g^{-1}(K) \cap R)$ belongs to C . Note that K' is \emptyset or $\{\sigma\}$, depending on whether K is empty or not. We construct $\mathfrak{P} = (D, \Sigma, (L_c)_{c \in \mathbb{C}}, d_0, \mathbf{m}_0)$ with $D = \{d_0\}$, $\mathbf{m}_0 = \llbracket \sigma \rrbracket$, $L_{d_0, \sigma, d_0} = K'$ and all languages $L_c = \emptyset$ for $c \neq (d_0, \sigma, d_0)$. Then \mathfrak{P} is terminating iff K is empty.

To prove “(iii) \Rightarrow (i)”, we design an algorithm deciding safety assuming decidability of emptiness. Given asynchronous program \mathfrak{P} and state d as input, the algorithm consists of two semi-decision procedures: one which searches for a run of \mathfrak{P} reaching the state d , and the second which enumerates regular overapproximations \mathfrak{P}' of \mathfrak{P} and checks the safety of \mathfrak{P}' using Theorem 4.1. Each \mathfrak{P}' consists of a regular language A_c overapproximating L_c for each context c of \mathfrak{P} . We use decidability of emptiness to check that $L_c \cap (\Sigma^* \setminus A_c) = \emptyset$ to ensure that \mathfrak{P}' is indeed an overapproximation.

Algorithm 1 clearly gives a correct answer if it terminates. Hence, we only have to argue that it always does terminate. Of course, if d is reachable, the first semi-decision procedure will terminate. In the other case, termination is due to the regularity of downclosures: if d is not reachable in \mathfrak{P} , then Proposition 3.3 tells us that $\downarrow \mathfrak{P}$ cannot reach d either. But $\downarrow \mathfrak{P}$ is an asynchronous program over regular languages; this means there exists a safe regular overapproximation and the second semi-decision procedure terminates.

To prove “(iii) \Rightarrow (ii)”, we adopt a similar method as for safety. Algorithm 2 for termination consists of two semi-decision procedures. By standard well-quasi-ordering arguments, an infinite run of an asynchronous program \mathfrak{P} is witnessed by a finite self-covering run. The first semi-decision procedure enumerates finite self-covering runs (trying to show non-termination). The second procedure enumerates regular asynchronous programs \mathfrak{P}' that overapproximate \mathfrak{P} . As before, to check termination of \mathfrak{P}' , it applies the procedure from Theorem 4.1. Clearly, the algorithm’s answer is always correct. Moreover, it gives an answer for every input. If \mathfrak{P} does not terminate, it will find a self-covering sequence. If \mathfrak{P} does terminate, then Proposition 3.3 tells us that $\downarrow \mathfrak{P}$ is a

Algorithm 1: Checking Safety

Input: Asynchronous program $\mathfrak{P} = (D, \Sigma, (L_c)_{c \in \mathbb{C}}, d_0, \mathbf{m}_0)$ over C , state $d \in D$

run concurrently

```

begin
    /* find a safe overapproximation */
    foreach tuple  $(A_c)_{c \in \mathbb{C}}$  of regular languages  $A_c \subseteq \Sigma^*$  do
        if  $L_c \cap (\Sigma^* \setminus A_c) = \emptyset$  for each  $c \in \mathbb{C}$  then
            if  $\mathfrak{P}' = (D, \Sigma, (A_c)_{c \in \mathbb{C}}, d_0, \mathbf{m}_0)$  does not reach  $d$  then
                return  $d$  is not reachable.
    begin
        /* find a run reaching  $d$  */
        foreach prerun  $\rho$  of  $\mathfrak{P}$  do
            if  $\rho$  is a run that reaches  $d$  then
                return  $d$  reachable.

```

Algorithm 2: Checking Termination

Input: Asynchronous program $\mathfrak{P} = (D, \Sigma, (L_c)_{c \in \mathbb{C}}, d_0, \mathbf{m}_0)$ over C

run concurrently

```

begin
    /* find a terminating overapproximation */
    foreach tuple  $(A_c)_{c \in \mathbb{C}}$  of regular languages  $A_c \subseteq \Sigma^*$  do
        if  $L_c \cap (\Sigma^* \setminus A_c) = \emptyset$  for each  $c \in \mathbb{C}$  then
            if  $\mathfrak{P}' = (D, \Sigma, (A_c)_{c \in \mathbb{C}}, d_0, \mathbf{m}_0)$  terminates then
                return  $\mathfrak{P}$  terminates.
    begin
        /* find a self-covering run */
        foreach prerun  $\rho$  of  $\mathfrak{P}$  do
            if  $\rho$  is a self-covering run then
                return  $\mathfrak{P}$  does not terminate.

```

terminating finite-state overapproximation. This implies that the second procedure will terminate in that case.

Let us point out a particular example. The class \mathcal{L} of languages of lossy channel systems is defined like the class of languages of WSTS with upward-closed sets of accepting configurations as in [Geeraerts et al. 2007], except that we only consider lossy channel systems [Abdulla et al. 1998] instead of arbitrary Well-Structured Transition Systems (WSTS). Then \mathcal{L} forms a full trio with decidable emptiness. Although downclosures of lossy channel languages are not effectively computable (an easy consequence of [Mayr 2003]), our algorithm employs Theorem 4.2 to decide safety and termination.

4.2 Boundedness

THEOREM 4.3. *Let C be a full trio. The following are equivalent:*

- (i) *Boundedness is decidable for asynchronous programs over C .*
- (ii) *Finiteness is decidable for C .*

Clearly, the construction for “(i) \Rightarrow (iii)” of Theorem 4.2 also works for “(i) \Rightarrow (ii)”: \mathfrak{P} is unbounded iff K is infinite.

Algorithm 3: Checking Boundedness

Input: Asynchronous program $\mathfrak{P} = (D, \Sigma, (L_c)_{c \in \mathfrak{C}}, d_0, \mathbf{m}_0)$ over C
 $F \leftarrow \emptyset, I \leftarrow \emptyset;$
foreach context $c = (d, \sigma, d') \in \mathfrak{C}$ **do**
 if L_c is infinite **then** /* using algorithm for safety */
 if c is reachable in \mathfrak{P} **then**
 return \mathfrak{P} is unbounded.
 $I \leftarrow I \cup \{c\}$
 else
 $F \leftarrow F \cup \{c\}$
foreach context $c \in F$ **do**
 foreach finite set $A \subseteq \Sigma^*$ **do** /* find a finite A with $L_c = A$ */
 if $L_c \cap (\Sigma^* \setminus A) = \emptyset$ and $L_c \cap \{w\} \neq \emptyset$ for each $w \in A$ **then**
 $A_c \leftarrow A;$
 break; /* end inner foreach */
foreach context $c \in I$ **do**
 $A_c \leftarrow \emptyset$
if $\mathfrak{P}' = (D, \Sigma, (A_c)_{c \in \mathfrak{C}}, d_0, \mathbf{m}_0)$ is bounded **then**
 return \mathfrak{P} is bounded.
else
 return \mathfrak{P} is unbounded.

For the converse, we first note that if finiteness is decidable for C then so is emptiness. Given $L \subseteq \Sigma^*$ from C , consider the homomorphism $h: (\Sigma \cup \{\lambda\})^* \rightarrow \Sigma^*$ with $h(a) = a$ for every $a \in \Sigma$ and $h(\lambda) = \varepsilon$. Then $h^{-1}(L)$ belongs to C and $h^{-1}(L)$ is finite if and only if L is empty: in the inverse homomorphism, λ can be arbitrarily inserted in any word. By [Theorem 4.2](#), this implies that we can also decide safety. As a consequence of considering only full trios, it is easy to see that the problem of *context reachability* reduces to safety: a context $\hat{c} = (\hat{d}, \hat{\sigma}, \hat{d}') \in \mathfrak{C}$ is *reachable in \mathfrak{P}* if there is a reachable configuration (\hat{d}, \mathbf{m}) in \mathfrak{P} with $\mathbf{m}(\hat{\sigma}) \geq 1$.

We now explain [Algorithm 3](#) for deciding boundedness of a given asynchronous program $\mathfrak{P} = (D, \Sigma, (L_c)_{c \in \mathfrak{C}}, d_0, \mathbf{m}_0)$. For every context c , we first check if L_c is infinite (feasible by assumption). This partitions the set of contexts of \mathfrak{P} into sets I and F which are the contexts for which the corresponding language L_c is infinite and finite respectively. If any context in I is reachable, then \mathfrak{P} is unbounded. Otherwise, all the reachable contexts have a finite language. For every finite language L_c for some $c \in F$, we explicitly find all the members of L_c . This is possible because any finite set A can be checked with L_c for equality. $L_c \subseteq A$ can be checked by testing whether $L_c \cap (\Sigma^* \setminus A) = \emptyset$ and $L_c \cap (\Sigma^* \setminus A)$ effectively belongs to C . On the other hand, checking $A \subseteq L_c$ just means checking whether $L_c \cap \{w\} \neq \emptyset$ for each $w \in A$, which can be done the same way. We can now construct asynchronous program \mathfrak{P}' which replaces all languages for contexts in I by \emptyset and replaces those corresponding to F by the explicit description. Clearly \mathfrak{P}' is bounded iff \mathfrak{P} is bounded (since no contexts from I are reachable) and the former can be decided by [Theorem 4.1](#).

We observe that boundedness is strictly harder than safety or termination: There are full trios for which emptiness is decidable, but finiteness is undecidable, such as the languages of reset vector addition systems [[Dufourd et al. 1998](#)] (see [[Thinniyam and Zetsche 2019](#)] for a definition of the language class) and languages of lossy channel systems.

4.3 Configuration reachability and liveness properties

Theorems 4.2 and 4.3 completely characterize for which full trios safety, termination, and boundedness are decidable. We turn to configuration reachability, fair termination, and fair starvation. We suspect that it is unlikely that there is a simple characterization of those language classes for which the latter problems are decidable. However, we show that they are decidable for a limited range of infinite-state systems. To this end, we prove that decidability of any of these problems implies decidability of the others as well, and also implies the decidability of a simple combinatorial problem that is known to be undecidable for many expressive classes of languages.

Let $Z \subseteq \{a, b\}^*$ be the language $Z = \{w \in \{a, b\}^* \mid |w|_a = |w|_b\}$. The *Z-intersection problem* for a language class C asks, given a language $K \subseteq \{a, b\}^*$ from C , whether $K \cap Z \neq \emptyset$. Informally, Z is the language of all words with an equal number of as and bs and the *Z-intersection problem* asks if there is a word in K with an equal number of as and bs.

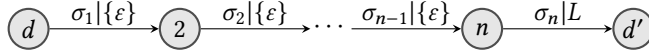
THEOREM 4.4. *Let C be a full trio. The following statements are equivalent:*

- (i) *Configuration reachability is decidable for asynchronous programs over C .*
- (ii) *Fair termination is decidable for asynchronous programs over C .*
- (iii) *Fair starvation is decidable for asynchronous programs over C .*

Moreover, if decidability holds, then *Z-intersection* is decidable for C .

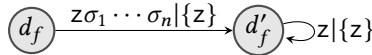
We prove **Theorem 4.4** by providing reductions among the three problems and showing that *Z-intersection* reduces to configuration reachability. We use diagrams similar to automata to describe asynchronous programs. Here, circles represent global states of the program and we draw an edge

$(d) \xrightarrow{\sigma|L} (d')$ in case we have $L_{d,\sigma,d'} = L$ in our asynchronous program \mathfrak{P} . Furthermore, we have $L_{d,\sigma,d'} = \emptyset$ whenever there is no edge that specifies otherwise. To simplify notation, we draw an edge $d \xrightarrow{w|L} d'$ in an asynchronous program for a word $w \in \Sigma^*$, $w = \sigma_1 \dots \sigma_n$ with $\sigma_1, \dots, \sigma_n \in \Sigma$, to symbolize a sequence of states



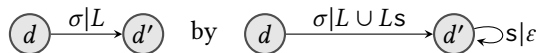
which removes $\llbracket \sigma_1, \dots, \sigma_n \rrbracket$ from the task buffer and posts a multiset of handlers specified by L .

Proof of “(ii) \Rightarrow (i)” Given an asynchronous program $\mathfrak{P} = (D, \Sigma, (L_c)_{c \in \mathbb{C}}, d_0, \mathbf{m}_0)$ and a configuration $(d_f, \mathbf{m}_f) \in D \times \mathbb{M}[\Sigma]$, we construct asynchronous program \mathfrak{P}' as follows. Let z be a fresh letter and let $\mathbf{m}_f = \llbracket \sigma_1, \dots, \sigma_n \rrbracket$. We obtain \mathfrak{P}' from \mathfrak{P} by adding a new state d'_f and including the following edges:



Starting from $(d_0, \mathbf{m}_0 \oplus \llbracket z \rrbracket)$, the program \mathfrak{P}' has a fair infinite run iff (d_f, \mathbf{m}_f) is reachable in \mathfrak{P} . The ‘if’ direction is obvious. Conversely, z has to be executed in any fair run ρ of \mathfrak{P}' which implies that d'_f is reached by \mathfrak{P}' in ρ . Since only z can be executed at d'_f in ρ , this means that the multiset is exactly \mathbf{m}_f when d_f is reached during ρ . Clearly this initial segment of ρ corresponds to a run of \mathfrak{P} which reaches the target configuration.

Proof of “(iii) \Rightarrow (ii)” We construct $\mathfrak{P}' = (D, \Sigma', (L'_c)_{c \in \mathbb{C}}, d_0, \mathbf{m}'_0)$ given $\mathfrak{P} = (D, \Sigma, (L_c)_{c \in \mathbb{C}}, d_0, \mathbf{m}_0)$ over C as follows. Let $\Sigma' = \Sigma \cup \{s\}$, where s is a fresh handler. Replace each edge



at every state $d \in D$. Moreover, we set $\mathbf{m}'_0 = \mathbf{m}_0 \oplus \llbracket s, s \rrbracket$. Then \mathfrak{P}' has an infinite fair run that starves some handler if and only if \mathfrak{P} has an infinite fair run. From an infinite fair run ρ of \mathfrak{P} , we

obtain an infinite fair run of \mathfrak{P}' which starves s , by producing s while simulating ρ and consuming it in the loop. Conversely, from an infinite fair run ρ' of \mathfrak{P}' which starves some τ , we obtain an infinite fair run ρ of \mathfrak{P} by omitting all productions and consumptions of s and removing two extra instances of s from all configurations.

Proof of “(i) \Rightarrow (iii)” From $\mathfrak{P} = (D, \Sigma, (L_c)_{c \in \mathbb{C}}, d_0, \mathbf{m}_0)$ over C , for each subset $\Gamma \subseteq \Sigma$ and $\tau \in \Sigma$, we construct an asynchronous program $\mathfrak{P}_{\Gamma, \tau} = (D', \Sigma', (L_c)_{c \in \mathbb{C}'}, d'_0, \mathbf{m}'_0)$ over C such that a particular configuration is reachable in $\mathfrak{P}_{\Gamma, \tau}$ if and only if \mathfrak{P} has a fair infinite run $\rho_{\Gamma, \tau}$, where Γ is the set of handlers that is executed infinitely often in $\rho_{\Gamma, \tau}$ and $\rho_{\Gamma, \tau}$ starves τ . Since there are only finitely many choices for Γ and τ , decidability of configuration reachability implies decidability of fair starvation. The idea is that run $\rho_{\Gamma, \tau}$ exists if and only if there exists a run

$$(d_0, \mathbf{m}_0) \xrightarrow{\sigma_1} \dots \xrightarrow{\sigma_n} (d_n, \mathbf{m}_n) = (e_0, \mathbf{n}_0) \xrightarrow{\gamma_1} (e_1, \mathbf{n}_1) \xrightarrow{\gamma_2} \dots \xrightarrow{\gamma_k} (e_k, \mathbf{n}_k), \quad (1)$$

where $\bigcup_{i=1}^k \{\gamma_i\} = \Gamma$, for each $1 \leq i \leq k$ $\mathbf{n}_i \in \mathbb{M}[\Gamma]$, $\mathbf{m}_n \leq \mathbf{n}_k$, and for each $i \in \{1, \dots, k\}$ with $\gamma_i = \tau$, we have $\mathbf{n}_{i-1}(\tau) \geq 2$. In such a run, we call $(d_0, \mathbf{m}_0) \xrightarrow{\sigma_1} \dots \xrightarrow{\sigma_n} (d_n, \mathbf{m}_n)$ its *first phase* and $(e_0, \mathbf{n}_0) \xrightarrow{\gamma_1} \dots \xrightarrow{\gamma_k} (e_k, \mathbf{n}_k)$ its *second phase*.

Let us explain how $\mathfrak{P}_{\Gamma, \tau}$ reflects the existence of a run as in Eq. (1). The set Σ' of handlers of $\mathfrak{P}_{\Gamma, \tau}$ includes Σ , $\bar{\Sigma}$ and $\hat{\Sigma}$, where $\bar{\Sigma} = \{\bar{\sigma} \mid \sigma \in \Sigma\}$ and $\hat{\Sigma} = \{\hat{\sigma} \mid \sigma \in \Sigma\}$ are disjoint copies of Σ . This means, a multiset $\mathbb{M}[\Sigma']$ contains multisets $\mathbf{m}' = \mathbf{m} \oplus \bar{\mathbf{m}} \oplus \hat{\mathbf{m}}$ with $\mathbf{m} \in \mathbb{M}[\Sigma]$, $\bar{\mathbf{m}} \in \mathbb{M}[\bar{\Sigma}]$, and $\hat{\mathbf{m}} \in \mathbb{M}[\hat{\Sigma}]$. A run of $\mathfrak{P}_{\Gamma, \tau}$ simulates the two phases of ρ . While simulating the first phase, $\mathfrak{P}_{\Gamma, \tau}$ keeps two copies of the task buffer, \mathbf{m} and $\bar{\mathbf{m}}$. The copying is easily accomplished by a homomorphism with $\sigma \mapsto \sigma\bar{\sigma}$ for each $\sigma \in \Sigma$. At some point, $\mathfrak{P}_{\Gamma, \tau}$ switches into simulating the second phase. There, $\bar{\mathbf{m}}$ remains unchanged, so that it stores the value of \mathbf{m}_n in Eq. (1) and can be used in the end to make sure that $\mathbf{m}_n \leq \mathbf{n}_k$.

Hence, in the second phase, $\mathfrak{P}_{\Gamma, \tau}$ works, like \mathfrak{P} , only with Σ . However, whenever a handler $\sigma \in \Sigma$ is executed, it also produces a task $\hat{\sigma}$. These handlers are used at the end to make sure that every $\gamma \in \Gamma$ has been executed at least once in the second phase. Also, whenever τ is executed, $\mathfrak{P}_{\Gamma, \tau}$ checks that at least two instances of τ are present in the task buffer, thereby ensuring that τ is starved.

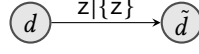
In the end, a distinguished final state allows $\mathfrak{P}_{\Gamma, \tau}$ to execute handlers in Γ and $\bar{\Gamma}$ simultaneously to make sure that $\mathbf{m}_n \leq \mathbf{n}_k$. In its final state, $\mathfrak{P}_{\Gamma, \tau}$ can execute handlers $\hat{\gamma} \in \hat{\Gamma}$ and $\gamma \in \Gamma$ (without creating new handlers). In the final configuration, there can be no $\hat{\sigma}$ with $\sigma \in \Sigma \setminus \Gamma$, and there has to be exactly one $\hat{\gamma}$ for each $\gamma \in \Gamma$. This guarantees that (i) each handler in Γ is executed at least once during the second phase, (ii) every handler executed in the second phase is from Γ , and (iii) \mathbf{m}_n contains only handlers from Γ (because handlers from $\bar{\Sigma}$ cannot be executed in the second phase).

Let us now describe $\mathfrak{P}_{\Gamma, \tau}$ in detail. We have $\Sigma' = \Sigma \cup \bar{\Sigma} \cup \hat{\Sigma} \cup \{z\}$, where z is a fresh letter. The set of states is $D' = D \cup \tilde{D} \cup \{d_f\}$, where $\tilde{D} = \{\tilde{d} \mid d \in D\}$ is a disjoint copy of D for simulating the second phase, and d_f is a fresh state. Moreover, we change the languages L_c in the following way:

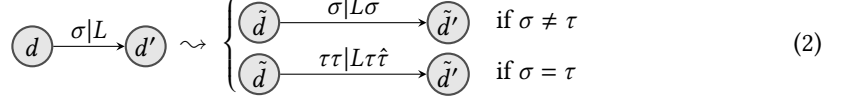
$$\textcircled{d} \xrightarrow{\sigma|L} \textcircled{d'} \rightsquigarrow \textcircled{d} \xrightarrow{\sigma\bar{\sigma}|h^\pm(L)} \textcircled{d'}$$

where $h^\pm: \Sigma^* \rightarrow (\Sigma \cup \bar{\Sigma})^*$ is the homomorphism with $\sigma \mapsto \sigma\bar{\sigma}$ for every $\sigma \in \Sigma$. This means, for every context $c = (d, \sigma, d')$, we have $d \xrightarrow{\sigma\bar{\sigma}|h^\pm(L)} d'$ in $\mathfrak{P}_{\Gamma, \tau}$. Note that since C is a full trio, it is in particular closed under homomorphisms. Hence, $h^\pm(L)$ belongs to C . Moreover, $\mathfrak{P}_{\Gamma, \tau}$ can

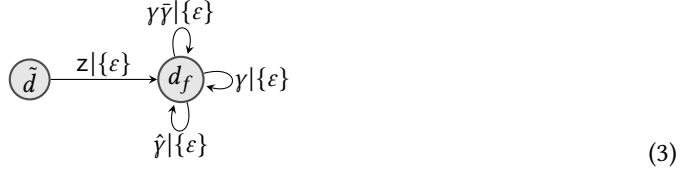
spontaneously switch to simulating the second phase: For each $d \in D$, we have



Here, the handler z merely allows us to move to state \tilde{d} without interfering with the other handlers. In the second phase, $\mathfrak{P}_{\Gamma, \tau}$ simulates \mathfrak{P} slightly differently. We perform the following replacement:

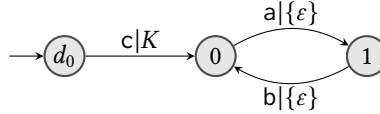


Note that since C is a full trio, the languages $Lv = \{uv \mid u \in L\}$ belong to C for every word v . Finally, $\mathfrak{P}_{\Gamma, \tau}$ can spontaneously switch to its distinguished state d_f , so that we have for every $\tilde{d} \in \tilde{D}$ and every $\gamma \in \Gamma$:



The initial configuration of $\mathfrak{P}_{\Gamma, \tau}$ is (d_0, \mathbf{m}'_0) , where $\mathbf{m}'_0 = \mathbf{m}_0 \oplus \bar{\mathbf{m}}_0 \oplus \llbracket z \rrbracket$, where $\bar{\mathbf{m}}_0$ is obtained from \mathbf{m}_0 by replacing every occurrence of $\sigma \in \Sigma$ in \mathbf{m}_0 with $\bar{\sigma}$. The final configuration is (d_f, \mathbf{m}_f) , where $\mathbf{m}_f \in \mathbb{M}[\hat{\Gamma}]$ is the multiset with $\mathbf{m}_f(\hat{\gamma}) = 1$ for every $\gamma \in \Gamma$. It is now straightforward to check that (d_f, \mathbf{m}_f) is reachable in $\mathfrak{P}_{\Gamma, \tau}$ if and only if \mathfrak{P} has an infinite fair run that starves τ .

Decidability of Z-intersection To complete the proof of [Theorem 4.4](#), we reduce Z-intersection to configuration reachability. Given $K \subseteq \{a, b\}^*$ from C , we construct the asynchronous program $\mathfrak{P} = (D, \Sigma, (L_c)_{c \in \mathbb{C}}, d_0, \mathbf{m}_0)$ over C where $D = \{d_0, 0, 1\}$, $\Sigma = \{a, b, c\}$, by including the following edges:



The initial task buffer is $\mathbf{m}_0 = \llbracket c \rrbracket$. Then clearly, the configuration $(0, \llbracket \rrbracket)$ is reachable in \mathfrak{P} if and only if $K \cap Z \neq \emptyset$.

If the construction seems abstract, recall the example from [Section 2](#): the procedure $s1$ plays the role of K and generates strings from its language in $\{a, b\}^*$; procedures a and b take turns to ensure there is an equal number of them; the states 0 and 1 are values of turn.

[Theorem 4.4](#) is useful in the contrapositive to show undecidability. For example, one can show undecidability of Z-intersection for languages of lossy channel systems (see [Section 4.1](#)): One expresses reachability in a non-lossy FIFO system by making sure that the numbers of enqueue- and dequeue-operations match. Thus, for asynchronous programs over lossy channel systems, the problems of [Theorem 4.4](#) are undecidable. We also use [Theorem 4.4](#) in [Section 5](#) to conclude undecidability for higher-order asynchronous programs, already at order 2.

5 HIGHER-ORDER ASYNCHRONOUS PROGRAMS

We apply our general decidability results to asynchronous programs over (deterministic) higher-order recursion schemes (HORS). Kobayashi [[Kobayashi 2009](#)] has shown how higher-order functional programs can be modeled using HORS. In his setting, a program contains instructions that access certain resources. The path language of the HORS produced by Kobayashi is the set of

possible sequences of instructions. For us, the input program contains post instructions and we translate higher-order programs with post instructions into a HORS whose path language is used as the language of handlers.

We recall some definitions from [Kobayashi 2009]. The set of *types* is defined by the grammar $A := o \mid A \rightarrow A$. The *order* $\text{ord}(A)$ of a type A is inductively defined as $\text{ord}(o) = 0$ and $\text{ord}(A \rightarrow B) := \max(\text{ord}(A) + 1, \text{ord}(B))$. The *arity* of a type is inductively defined by $\text{arity}(o) = 0$ and $\text{arity}(A \rightarrow B) = \text{arity}(B) + 1$. We assume a countably infinite set Var of typed variables $x : A$. For a set Θ of typed symbols, the set $\tilde{\Theta}$ of *terms* generated from Θ is the least set which contains Θ such that whenever $s : A \rightarrow B$ and $t : A$ belong to $\tilde{\Theta}$, then also $st : B$ belongs to $\tilde{\Theta}$. By convention the type $o \rightarrow \dots (o \rightarrow (o \rightarrow o))$ is written $o \rightarrow \dots \rightarrow o \rightarrow o$ and the term $((t_1 t_2) t_3 \dots) t_n$ is written $t_1 t_2 \dots t_n$. We write \bar{x} for a sequence (x_1, x_2, \dots, x_n) of variables.

A higher-order recursion scheme (HORS) is a tuple $\mathcal{S} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$ where Σ is a set of typed *terminal* symbols of types of order 0 or 1, \mathcal{N} is a set of typed *non-terminal* symbols (disjoint from terminal symbols), $S : o$ is the start non-terminal symbol and \mathcal{R} is a set of rewrite rules $Fx_1x_2 \dots x_n \rightarrow t$ where $F : A_1 \rightarrow \dots \rightarrow A_n \rightarrow o$ is a non-terminal in \mathcal{N} , $x_i : A_i$ for all i are variables and $t : o$ is a term generated from $\Sigma \cup \mathcal{N} \cup \text{Var}$. The order of a HORS is the maximum order of a non-terminal symbol. We define a rewrite relation \rightarrow on terms over $\Sigma \cup \mathcal{N}$ as follows: $F\bar{a} \rightarrow t[\bar{x}/\bar{a}]$ if $F\bar{x} \rightarrow t \in \mathcal{R}$, and if $t \rightarrow t'$ then $ts \rightarrow t's$ and $st \rightarrow st'$. The reflexive, transitive closure of \rightarrow is denoted \rightarrow^* . A *sentential form* t of \mathcal{S} is a term over $\Sigma \cup \mathcal{N}$ such that $S \rightarrow^* t$.

If N is the maximum arity of a symbol in Σ , then a (possibly infinite) tree over Σ is a partial function tr from $\{0, 1, \dots, N-1\}^*$ to Σ that fulfills the following conditions: $\varepsilon \in \text{dom}(tr)$, $\text{dom}(tr)$ is closed under prefixes, and if $tr(w) = a$ and $\text{arity}(a) = k$ then $\{j \mid wj \in \text{dom}(tr)\} = \{0, 1, \dots, k-1\}$.

A *deterministic* HORS is one where there is exactly one rule of the form $Fx_1x_2 \dots x_n \rightarrow t$ for every non-terminal F . Following [Kobayashi 2009], we show how a deterministic HORS can be used to represent a higher-order pushdown language arising from a higher-order functional program.

Sentential forms can be seen as ranked trees over $\Sigma \cup \mathcal{N} \cup \text{Var}$. A sequence Π over $\{0, 1, \dots, n-1\}$ is a *path* of tr if every finite prefix of $\Pi \in \text{dom}(tr)$. The set of paths in a tree tr will be denoted $\text{Paths}(tr)$. Note that we are only interested in finite paths in our context. Associated with any path $\Pi = n_1, n_2, \dots, n_k$ is the word $w_\Pi = tr(n_1)tr(n_1n_2) \dots tr(n_1n_2 \dots n_k)$.

We associate a *value tree* $\mathcal{T}_\mathcal{S}$ associated with a deterministic HORS \mathcal{S} in the following way. For a sentential form t define the finite tree t^\perp by case analysis: $t^\perp = f$ if f is a terminal symbol and $t^\perp = t_1^\perp t_2^\perp$ if $t = t_1 t_2$ and $t_1^\perp \neq \perp$, and $t^\perp = \perp$ otherwise.

Intuitively, the tree t^\perp is obtained by replacing any subterm whose root is a non-terminal by the nullary symbol \perp . We define the partial order \leq_t on $\text{dom}(\Sigma) \cup \{\perp\}$ by $\perp \leq_t a$ for all $a \in \text{dom}(\Sigma)$, which is extended to trees by

$$t \leq_t s \iff \forall \bar{n} \in \text{dom}(t) : \bar{n} \in \text{dom}(s) \wedge t(\bar{n}) \leq_t s(\bar{n})$$

The value tree generated by a HORS \mathcal{S} is denoted $\mathcal{T}_\mathcal{S}$ and is obtained by repeated application of the rewrite rules to the start symbol S . To make this formal, we write $\text{lub}(T)$ for the least upper bound of a collection T of trees with respect to the order \leq_t . Then we set $\mathcal{T}_\mathcal{S} := \text{lub}(\{t^\perp \mid S \rightarrow^* t\})$. Any HORS for which the value tree $\mathcal{T}_\mathcal{S}$ is well-defined is called *productive*. All HORS dealt with in this paper are assumed to be productive. Checking if a given HORS is productive is decidable, see, e.g., [Grellois 2016].

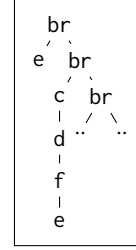
Let $\Sigma_1 := \{a \in \Sigma \mid \text{arity}(a) = 1\}$. The *path language* $\mathcal{L}_p(\mathcal{S})$ of a deterministic HORS \mathcal{S} is defined as $\{\text{Proj}_{\Sigma_1}(w_\Pi) \mid \Pi \in \text{Paths}(\mathcal{T}_\mathcal{S})\}$. The *tree language* $\mathcal{L}_t(\mathcal{S})$ associated with a HORS is the set of finite trees over Σ generated by \mathcal{S} .

The deterministic HORS corresponding to the higher-order function `s3` from Figure 1 is given by $\mathcal{S} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$, where

$$\begin{aligned}\Sigma &= \{\text{br} : \mathbf{o} \rightarrow \mathbf{o} \rightarrow \mathbf{o}, \text{c}, \text{d}, \text{f} : \mathbf{o} \rightarrow \mathbf{o}, \text{e} : \mathbf{o}\} \\ \mathcal{N} &= \{S : \mathbf{o}, F : (\mathbf{o} \rightarrow \mathbf{o}) \rightarrow \mathbf{o} \rightarrow \mathbf{o}, H : (\mathbf{o} \rightarrow \mathbf{o}) \rightarrow \mathbf{o} \rightarrow \mathbf{o}, I : \mathbf{o} \rightarrow \mathbf{o}\} \\ \mathcal{R} &= \{S \rightarrow F I e, I x \rightarrow x, F G x \rightarrow \text{br}(F(H G)(fx))(G x), H G x \rightarrow c(G(dx))\}\end{aligned}$$

The path language $\mathcal{L}_p(\mathcal{S}) = \{c^n d^n f^n \mid n \geq 0\}$. To see this, apply the reduction rules to get the value tree $\mathcal{T}_{\mathcal{S}}$ shown on the right:

$$\begin{aligned}S &\rightarrow F I e \rightarrow \text{br}(F(HI)(fe))(Ie) \\ &\rightarrow \text{br}(F(HI)(fe))e \\ &\rightarrow \text{br}(\text{br}(F(H^2I)(f^2e))(HI)(fe))e \\ &\rightarrow \text{br}(\text{br}(F(H^2I)(f^2e))c(I(dfe))e) \\ &\rightarrow \text{br}(\text{br}(F(H^2I)(f^2e))cdf e)e \\ &\rightarrow \dots\end{aligned}$$



A HORS \mathcal{S} is called a *word scheme* if it has exactly one nullary terminal symbol e and all other terminal symbols $\tilde{\Sigma}$ are of arity one. The *word language* $\mathcal{L}_w(\mathcal{S}) \subseteq \tilde{\Sigma}^*$ defined by \mathcal{S} is $\mathcal{L}_w(\mathcal{S}) = \{a_1 a_2 \dots a_n \mid (a_1(a_2 \dots (a_n(e)) \dots)) \in \mathcal{L}_t(\mathcal{S})\}$. We denote by \mathcal{H} the class of languages $\mathcal{L}_w(\mathcal{S})$ that occur as the word language of a higher-order recursion scheme \mathcal{S} . Note that path languages and languages of word schemes are both word languages over the set $\tilde{\Sigma}$ of unary symbols considered as letters. They are connected by the following proposition, a proof of which is given in Appendix C.²

PROPOSITION 5.1. *For every order- n HORS $\mathcal{S} = (\Sigma, \mathcal{N}, S, \mathcal{R})$ there exists an order- n word scheme $\mathcal{S}' = (\Sigma', \mathcal{N}', S', \mathcal{R}')$ such that $\mathcal{L}_p(\mathcal{S}) = \mathcal{L}_w(\mathcal{S}')$.*

A consequence of [Kobayashi 2009] and Prop. 5.1 is that the “post” language of higher-order functional programs can be modeled as the language of a word scheme. Hence, we define an *asynchronous program* over HORS as an asynchronous program over the language class \mathcal{H} and we can use the following results on word schemes.

THEOREM 5.2. *HORS and word schemes form effective full trios [Clemente et al. 2016]. Emptiness [Kobayashi and Ong 2011] and finiteness [Parys 2018] of order- n word schemes are $(n - 1)$ -EXPTIME-complete.*

Now Theorems 4.2 and 4.3, together with Proposition 5.1 imply the decidability results in Corollary 5.3. The undecidability result is a consequence of Theorem 4.4 and the undecidability of the Z-intersection problem for indexed languages or equivalently, order-2 pushdown automata as shown in [Zetzsch 2015]. Order-2 pushdown automata can be effectively turned into order-2 OI grammars [Damm and Goerdts 1986], which in turn can be translated into order-2 word schemes [Damm 1982]. See also [Kobayashi 2019, Theorem 4].

COROLLARY 5.3. *For asynchronous programs over HORS: (1) Safety, termination, and boundedness are decidable. (2) Configuration reachability, fair termination, and fair starvation are undecidable already at order-2.*

²The models of HORS (used in model checking higher order programs [Kobayashi 2009]) and word schemes (used in language-theoretic exploration of downclosures [Clemente et al. 2016; Hague et al. 2016]) are somewhat different. Thus, we show an explicit reduction between the two formalisms.

6 A DIRECT ALGORITHM AND COMPLEXITY ANALYSIS

We say that *downclosures are computable* for a language class C if for a given description of a language L in C , one can compute an automaton for the regular language $\downarrow L$. A consequence of Proposition 3.3 and Theorem 4.1 is that if one can compute downclosures for some language class, then one can avoid the enumerative approaches of Section 4 and get a “direct algorithm.” The direct algorithm replaces each handler by its downclosure and then invokes the decision procedure summarized in Theorem 4.1.

6.1 Higher Order Recursion Schemes

The direct algorithm for asynchronous programs over HORS relies on the recent breakthrough results on computing downclosures.

THEOREM 6.1 ([CLEMENTE ET AL. 2016; HAGUE ET AL. 2016; ZETZSCHE 2015]). *Downclosures are effectively computable for \mathcal{H} .*

Unfortunately, current techniques do not yet provide a complexity upper bound based on the above theorem. To understand why, we describe the current algorithm for computing downclosures. In [Zetsche 2015], it was shown that in a full trio, downclosures are computable if and only if the *diagonal problem* for C is decidable. The latter asks, given a language $L \subseteq \Sigma^*$, whether for every $k \in \mathbb{N}$, there is a word $w \in L$ with $|w|_\sigma \geq k$ for every $\sigma \in \Sigma$. The diagonal problem was then shown to be decidable for higher-order pushdown automata [Hague et al. 2016] and then for word schemes [Clemente et al. 2016]. Unfortunately, the algorithm from [Zetsche 2015] to compute downclosures using an oracle for the diagonal problem employs enumeration to compute a downclosure automaton, thus we have hidden the enumeration into the downclosure computation.

We conjecture that downclosures can in fact be computed in elementary time (for word schemes of fixed order). This would imply an elementary time procedure for asynchronous programs over HORS as well.

6.2 Context Free Languages

For handlers over context-free languages, given e.g., as pushdown automata, Ganty and Majumdar show an EXPSPACE upper bound. Precisely, the algorithm of [Ganty and Majumdar 2012] constructs for each handler a polynomial-size Petri net with certain guarantees (forming so-called *adequate family of Petri nets*) that accepts a Parikh equivalent language. These Petri nets are then used to construct a larger Petri net, polynomial in the size of the asynchronous program and the adequate family of Petri nets, in which the respective property (safety, boundedness, or termination) can be phrased as a query decidable in EXPSPACE. A natural question is whether there is a downclosure-based algorithm with the same asymptotic complexity.

Our goal is to replace the Parikh-equivalent Petri nets with Petri nets recognizing the downclosure of a language. It is an easy consequence of Proposition 3.3 that the resulting Petri nets can be used in place of the adequate families of Petri nets in the procedures for safety, termination, and boundedness of [Ganty and Majumdar 2012]. Thus, if we can ensure these Petri nets are polynomial in the size of the handler, we get an EXPSPACE upper bound. Unfortunately, a finite automaton for $\downarrow L$ may require exponentially many states in the pushdown automaton [Bachmeier et al. 2015]. Thus a naive approach gives a 2EXPSPACE upper bound.

We show here that for each context-free language L , one can construct in polynomial time a 1-bounded Petri net accepting $\downarrow L$. (Recall that a 1-bounded Petri net if every reachable marking has at most one token in each place.) This is a language theoretic result of independent interest. When used in the construction of [Ganty and Majumdar 2012], this matches the EXPSPACE upper bound.

As a byproduct, our translation yields a simple direct construction of a finite automaton for $\downarrow L$ when L is given as a pushdown automaton. This is of independent interest because earlier constructions of $\downarrow L$ always start from a context-free grammar and produce (of necessity!) exponentially large NFAs [Bachmeier et al. 2015; Courcelle 1991; van Leeuwen 1978].

We begin with some preliminary definitions.

Pushdown automata. If Γ is an alphabet, we write $\bar{\Gamma} = \{\bar{\gamma} \mid \gamma \in \Gamma\}$. Moreover, if $x = \bar{y}$, then we define $\bar{x} = y$. For a word $v \in (\Gamma \cup \bar{\Gamma})^*$, $v = v_1 \cdots v_n$, $v_1, \dots, v_n \in \Gamma \cup \bar{\Gamma}$, we set $\bar{v} = \bar{v}_n \cdots \bar{v}_1$. A *pushdown automaton* is a tuple $\mathcal{A} = (Q, \Sigma, \Gamma, E, q_0, q_f)$, where Q is a finite set of *states*, Σ is its *input alphabet*, Γ is its *stack alphabet*, E is a finite set of *edges*, $q_0 \in Q$ is its *initial state*, and $F \subseteq Q$ is its set of *final states*. An edge is a four-tuple (p, R, v, q) , where $p, q \in Q$, $R \subseteq \Sigma^*$ is a regular language, and $v \in \Gamma \cup \bar{\Gamma} \cup \{\varepsilon\}$. A *configuration* of \mathcal{A} is a pair (q, w) with $q \in Q$ and $w \in \Gamma^*$. For configurations (q, w) and (q', w') , we write $(q, w) \xrightarrow{u} (q', w')$ if there is an edge (q, R, v, q') in \mathcal{A} such that $u \in R$ and (i) if $v = \varepsilon$, then $w' = w$, (ii) if $v \in \Gamma$, then $w' = wv$ and (iii) if $v = \bar{\gamma}$ for $\gamma \in \Gamma$, then $w = w'\gamma$.

A *run* in \mathcal{A} is a sequence $(q_0, w_0), \dots, (q_n, w_n)$ of configurations and words $u_1, \dots, u_n \in \Sigma^*$ such that $(q_{i-1}, w_{i-1}) \xrightarrow{u_i} (q_i, w_i)$ for $1 \leq i \leq n$. Its *length* is n and its *initial* and *final configuration* are (q_0, w_0) and (q_n, w_n) , respectively. The run is said to *read* the word $u_1 \cdots u_n$. The *stack height* of the run is defined as $\max\{|w_i| \mid 0 \leq i \leq n\}$. We call the run *positive* (resp. *negative*) if $|w_i| \geq |w_0|$ (resp. $|w_i| < |w_0|$) for every $1 \leq i \leq n$, i.e. if the stack never drops below its initial height (resp. is always below its initial height).

We write $(q, w) \xRightarrow{u} (q', w')$ for configurations $(q, w), (q', w')$ if there is a run with initial configuration (q, w) and final configuration (q', w') that reads u . If there is such a run with stack height $\leq h$, then we write $(q, w) \xRightarrow{u}_h (q', w')$. The *language accepted by* \mathcal{A} is

$$L(\mathcal{A}) = \{u \in \Sigma^* \mid \exists (q_0, \varepsilon) \xRightarrow{u} (q_f, \varepsilon)\}.$$

There is also a language accepted with bounded stack height. For $h \in \mathbb{N}$, we define

$$L_h(\mathcal{A}) = \{u \in \Sigma^* \mid (q_0, \varepsilon) \xRightarrow{u}_h (q_f, \varepsilon)\}.$$

In order to exploit the symmetry between forward and backward computations in pushdown automata, we will consider dual pushdown automata. The *dual automaton* of \mathcal{A} , denoted $\bar{\mathcal{A}}$, is obtained from \mathcal{A} by changing each edge $p \xrightarrow{R|v} q$ into $q \xrightarrow{R^{\text{rev}}|\bar{v}} p$. Then $L(\bar{\mathcal{A}}) = L(\mathcal{A})^{\text{rev}}$. We will sometimes argue by *duality*, which is the principle that every statement that is true for any \mathcal{A} is also true for any $\bar{\mathcal{A}}$.

Petri nets. A (labeled) *Petri net* is a tuple $N = (\Sigma, S, T, \partial_0, \partial_1, \lambda, \mathbf{m}_0, \mathbf{m}_f)$ where Σ is its *input alphabet*, S is a finite set of *places*, T is a finite set of *transitions*, $\partial_0, \partial_1 : T \rightarrow \mathbb{M}[S]$ are maps that specify an *input marking* $\partial_0(t)$ and an *output marking* $\partial_1(t)$ for each transition $t \in T$, $\lambda : T \rightarrow \Sigma \cup \{\varepsilon\}$ assigns labels to transitions, and $\mathbf{m}_0, \mathbf{m}_f$ are its *initial* and *final marking*. More generally, multisets $\mathbf{m} \in \mathbb{M}[S]$ are called *markings* of N .

For markings $\mathbf{m}_1, \mathbf{m}_2 \in \mathbb{M}[S]$ and $a \in \Sigma \cup \{\varepsilon\}$, we write $\mathbf{m}_1 \xrightarrow{a} \mathbf{m}_2$ if there is a transition $t \in T$ with $\lambda(t) = a$, $\mathbf{m}_1 \geq \partial_0(t)$, and $\mathbf{m}_2 = \mathbf{m}_1 \ominus \partial_0(t) \oplus \partial_1(t)$. Moreover, we write $\mathbf{m}_1 \xRightarrow{w} \mathbf{m}_2$ if there are $n \in \mathbb{N}$, $a_1, \dots, a_n \in \Sigma \cup \{\varepsilon\}$, and markings $\mathbf{m}'_0, \dots, \mathbf{m}'_n$ such that $w = a_1 \cdots a_n$ and $\mathbf{m}_1 = \mathbf{m}'_0 \xrightarrow{a_1} \mathbf{m}'_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} \mathbf{m}'_n = \mathbf{m}_2$. Furthermore, we write $\mathbf{m}_1 \Rightarrow \mathbf{m}_2$ if there exists a $w \in \Sigma^*$ with $\mathbf{m}_1 \xRightarrow{w} \mathbf{m}_2$. The *language accepted by* N is $L(N) = \{w \in \Sigma^* \mid \mathbf{m}_0 \xRightarrow{w} \mathbf{m}_f\}$.

For $k \in \mathbb{N}$, a Petri net N is k -bounded if for every marking $\mathbf{m} \in \mathbb{M}[S]$ with $\mathbf{m}_0 \Rightarrow \mathbf{m}$, we have $|\mathbf{m}| \leq k$.

Our main results are as follows.

THEOREM 6.2 (SUCCINCT DOWNCLOSURES FOR PDAs). *Given a pushdown automaton \mathcal{A} , one can construct in polynomial time a pushdown automaton $\hat{\mathcal{A}}$ so that $\downarrow L(\hat{\mathcal{A}}) = \downarrow L(\mathcal{A})$. Moreover, if h is a bound on the stack height and denoting by $L_h(\mathcal{A})$ the language of words accepted by \mathcal{A} using a run bounded by h , $L(\hat{\mathcal{A}}) = L_h(\hat{\mathcal{A}})$ where h is polynomial in the size of \mathcal{A} .*

As a pushdown automaton with bounded stack can be simulated by a 1-bounded Petri net (essentially, by keeping places for each position in the stack), we get the following corollary and also the promised EXPSPACE upper bound.

COROLLARY 6.3. *Given a pushdown automaton \mathcal{A} , one can construct in polynomial time a 1-bounded Petri net N with $L(N) = \downarrow L(\mathcal{A})$.*

The augmented automaton $\hat{\mathcal{A}} = (Q, \Sigma, \hat{\Gamma}, \hat{E}, q_0, q_f)$ is defined as follows. We first compute the set

$$\Delta_{p,q}(\mathcal{A}) = \{a \in \Sigma \mid \exists u \in M_{p,q}(\mathcal{A}), |u|_a \geq 1\},$$

where $M_{p,q}(\mathcal{A}) = \{u \in \Sigma^* \mid \exists v \in \Gamma^*: (p, \varepsilon) \xRightarrow{u} (p, v), (q, v) \Rightarrow (q, \varepsilon)\}$. Note that it is easy to construct in polynomial time a pushdown automaton $\mathcal{A}_{p,q}$ for the language $M_{p,q}(\mathcal{A})$:

$\mathcal{A}_{p,q}$ has a set $Q \cup Q_p \cup Q_q$ of states consisting of three disjoint copies of the states of \mathcal{A} . The transitions between two states in Q_p is inherited from \mathcal{A} while for two states in Q_q we replace the input on any transition by ε but retain the stack operations from \mathcal{A} . The start state is $p \in Q_p$ and the final state is $q \in Q_q$.

There is an epsilon transition from $p \in Q_p$ to the corresponding copy $p \in Q$ which places a new symbol $\#$ (i.e. $\# \notin \Gamma$) on the stack. Similarly, there is an ε transition from $q \in Q$ to $q \in Q_q$ which pops $\#$. The new symbol $\#$ is used to ensure that the stack contents when leaving $p \in Q_p$ is the same as that when entering $q \in Q_q$. This concludes the construction of $\mathcal{A}_{p,q}$.

Since $a \in \Delta_{p,q}(\mathcal{A})$ iff $M_{p,q}(\mathcal{A}) \cap \Sigma^* a \Sigma^* \neq \emptyset$, we can decide in polynomial time whether a given $a \in \Sigma$ belongs to $\Delta_{p,q}(\mathcal{A})$ by checking the PDA for $M_{p,q}(\mathcal{A}) \cap \Sigma^* a \Sigma^* \neq \emptyset$ obtained by product construction for emptiness. Thus, we can compute $\Delta_{p,q}(\mathcal{A})$ in polynomial time. We construct $\hat{\mathcal{A}}$ as follows. For any $p, q \in Q$, we introduce a fresh stack symbol $[p, q]$ and then we add edges

$$p \xrightarrow{\Delta_{p,q}(\mathcal{A})^* |[p,q]} p, \quad q \xrightarrow{\Delta_{q,p}(\mathcal{A})^* |[p,q]} q. \quad (4)$$

The following lemma tells us that $L(\hat{\mathcal{A}})$ has the same downward closure as \mathcal{A} .

LEMMA 6.4. $L(\mathcal{A}) \subseteq L(\hat{\mathcal{A}}) \subseteq \downarrow L(\mathcal{A})$.

Since the inclusion $L(\mathcal{A}) \subseteq L(\hat{\mathcal{A}})$ is obvious, we prove $L(\hat{\mathcal{A}}) \subseteq \downarrow L(\mathcal{A})$. We proceed by induction on the number m of executions of new edges (i.e. those from Eq. (4)).

More specifically, we show that if $u \in L(\hat{\mathcal{A}})$ is accepted using m executions of new edges, then there is a $u' \in L(\mathcal{A})$ that is accepted using $m' < m$ executions of new edges and we have $u \sqsubseteq u'$.

Suppose u is accepted using a run ρ with $m > 0$ executions of new edges. Then ρ must apply one edge $p \xrightarrow{\Delta_{p,q}(\mathcal{A})^* |[p,q]} p$ and thus also the edge $q \xrightarrow{\Delta_{q,p}(\mathcal{A})^* |[p,q]} q$ to remove the letter $[p, q]$ from the stack. Thus, ρ can be decomposed as $\rho = \rho_1 \rho_2 \rho_3 \rho_4 \rho_5$, where ρ_2 and ρ_4 are the executions of the new edges. Let $u = u_1 u_2 u_3 u_4 u_5$ be the corresponding decomposition of u .

The run ρ_1 must end in state p and with some stack content $w \in \Gamma^*$. Then, ρ_3 is a run from $(p, w[p, q])$ to $(q, w[p, q])$ and ρ_5 is a run from (q, w) to (q_f, ε) with $q_f \in F$.

Since u_2 and u_4 are read while executing the new edges, we have $u_2 \in \Delta_{p,q}(\mathcal{A})^*$ and $u_4 \in \Delta_{q,p}(\tilde{\mathcal{A}})^*$. We can therefore write $u_2 = r_1 \cdots r_k$ and $u_4 = s_1 \cdots s_\ell$ with $r_1, \dots, r_k \in \Delta_{p,q}(\mathcal{A})$ and $s_1, \dots, s_\ell \in \Delta_{q,p}(\tilde{\mathcal{A}})$. By definition, this means for each $1 \leq i \leq k$, there is a word $\tilde{r}_i \in M_{p,q}(\mathcal{A})$ that contains the letter r_i . Likewise, for every $1 \leq i \leq \ell$, there is a $\tilde{s}_i \in M_{q,p}(\tilde{\mathcal{A}})$ that contains s_i .

Since $\tilde{r}_i \in M_{p,q}(\mathcal{A})$ and $\tilde{s}_j \in M_{q,p}(\tilde{\mathcal{A}})$ for $1 \leq i \leq k$ and $1 \leq j \leq \ell$, there are words x_i and y_j in Γ^* so that

$$\begin{aligned} (p, \varepsilon) &\xRightarrow{\tilde{r}_i} (p, x_i) & \text{and} & & (q, x_i) &\Rightarrow (q, \varepsilon) \\ (p, \varepsilon) &\Rightarrow (p, y_j) & \text{and} & & (q, y_j) &\xRightarrow{\tilde{s}_j} (q, \varepsilon) \end{aligned}$$

We can therefore construct a new run $\rho' = \rho_1 \rho'_2 \rho'_3 \rho'_4 \rho_5$, where

$$\begin{aligned} \rho'_2 : (p, w) &\xRightarrow{\tilde{r}_1} \cdots \xRightarrow{\tilde{r}_k} (p, wx_1 \cdots x_k) \Rightarrow \cdots \Rightarrow (p, wx_1 \cdots x_k y_\ell \cdots y_1) \\ \rho'_4 : (q, wx_1 \cdots x_k y_\ell \cdots y_1) &\xRightarrow{\tilde{s}_1} \cdots \xRightarrow{\tilde{s}_\ell} (q, wx_1 \cdots x_k) \Rightarrow \cdots \Rightarrow (q, w). \end{aligned}$$

Moreover, since ρ_3 is a positive run from (p, w) to (q, w) , we obtain ρ'_3 from ρ_3 by replacing the prefix w of every stack by $wx_1 \cdots x_k y_1 \cdots y_\ell$.

Then ρ' reads some word $u_1 \tilde{r}_1 \cdots \tilde{r}_k f u_3 \tilde{s}_1 \cdots \tilde{s}_\ell g u_5$ for $f, g \in \Sigma^*$. Note that since r_i occurs in \tilde{r}_i and s_i occurs in \tilde{s}_j , we have $u = u_1 u_2 u_3 u_4 u_5 \sqsubseteq u_1 \tilde{r}_1 \cdots \tilde{r}_k f u_3 \tilde{s}_1 \cdots \tilde{s}_\ell g u_5$.

We now show that every word in $\tilde{\mathcal{A}}$ is accepted by a run with bounded stack height.

LEMMA 6.5. $L(\tilde{\mathcal{A}}) = L_h(\hat{\mathcal{A}})$, where $h = 2|Q|^2$.

Before we prove Lemma 6.5, we need another observation. Just like Lemma 6.4, one can show that for any $p, q \in Q$, we have $M_{p,q}(\mathcal{A}) \subseteq M_{p,q}(\hat{\mathcal{A}}) \subseteq \downarrow M_{p,q}(\mathcal{A})$ and in particular

$$\Delta_{p,q}(\mathcal{A}) = \Delta_{p,q}(\hat{\mathcal{A}}) \quad \Delta_{q,p}(\tilde{\mathcal{A}}) = \Delta_{q,p}(\tilde{\hat{\mathcal{A}}}), \quad (5)$$

where the second identity follows from the first: Duality yields $\Delta_{q,p}(\tilde{\mathcal{A}}) = \Delta_{q,p}(\tilde{\hat{\mathcal{A}}})$ and since $\hat{\mathcal{A}}$ and $\tilde{\hat{\mathcal{A}}}$ are isomorphic (i.e. they are the same up to a renaming of stack symbols), we have $\Delta_{q,p}(\tilde{\hat{\mathcal{A}}}) = \Delta_{q,p}(\hat{\mathcal{A}})$.

We now prove Lemma 6.5. Let $u \in L(\tilde{\mathcal{A}})$. We show that any minimal length accepting run ρ reading u must have stack height $\leq h$ and hence $u \in L_h(\hat{\mathcal{A}})$.

Suppose the stack height of ρ is larger than $h = 2|Q|^2$.

Claim: ρ decomposes into runs $\rho_1, \rho_2, \rho_3, \rho_4, \rho_5$ reading u_1, u_2, u_3, u_4, u_5 , respectively, so that there are $p, q \in Q$ and $w \in \Gamma^*$ with

- ρ_2 is a positive run from (p, w) to (p, wv) of length ≥ 2
- ρ_3 is a positive run from (p, wv) to (q, wv)
- ρ_4 is a negative run from (q, wv) to (q, w)

Let c_h be a configuration along ρ with stack height at least $2|Q|^2 + 1$. Then there exist $|Q|^2$ configurations $c_2 \xrightarrow{*} c_4 \xrightarrow{*} \cdots \xrightarrow{*} c_{2|Q|^2} \xrightarrow{*} c_h$ along ρ such that c_{2i} is the last time that the stack height is $2i$. Symmetrically, we have $c_h \xrightarrow{*} c'_{2|Q|^2} \xrightarrow{*} \cdots \xrightarrow{*} c'_4 \xrightarrow{*} c'_2$ where c'_{2i} is the last occurrence of stack height $2i$. Clearly by definition the runs between consecutive c_{2i} (resp. c'_{2i}) configurations is positive (resp. negative). Additionally, the length of the run between them must be at least 2. Considering the pair of states at each c_{2i}, c'_{2i} , there are $|Q|^2$ possibilities. Hence there must exist indices $2i < 2j$ such that the c_{2i} and c_{2j} have the same state p and c'_{2i} and c'_{2j} have the same state

q . It is now clear that $\rho_2 = c_{2i} \rightarrow^* c_{2j}$, $\rho_3 = c_{2j} \rightarrow^* c'_{2j}$ and $\rho_4 = c'_{2j} \rightarrow^* c'_{2i}$ satisfy the conditions of the claim.

These conditions imply that $u_2 \in M_{p,q}(\hat{\mathcal{A}})$ and $u_4 \in M_{q,p}(\tilde{\mathcal{A}})$. Therefore, we have $u_2 \in \Delta_{p,q}(\hat{\mathcal{A}})^* = \Delta_{p,q}(\mathcal{A})^*$ and $u_4 \in \Delta_{q,p}(\tilde{\mathcal{A}})^* = \Delta_{q,p}(\bar{\mathcal{A}})^*$ by Eq. (5).

We obtain the run ρ' from ρ as follows. We replace ρ_2 by a single execution of the edge $p \xrightarrow{\Delta_{p,q}(\mathcal{A})^* \parallel [p,q]} p$ reading u_2 . Moreover, we replace ρ_4 by a single execution of the edge $q \xrightarrow{\Delta_{q,p}(\bar{\mathcal{A}})^* \parallel [p,q]} q$. Then ρ' is clearly a run reading $u = u_1 u_2 u_3 u_4 u_5$. Furthermore, since ρ_2 has length ≥ 2 , but the single edge used instead in ρ' only incurs length 1, ρ' is strictly shorter than ρ . This is in contradiction to the minimal length of ρ .

REMARK. The augmented automaton $\hat{\mathcal{A}}$ yields a very simple construction of a finite automaton (of exponential size) for $\downarrow L(\mathcal{A})$. First, it is easy to construct a finite automaton for $L_h(\hat{\mathcal{A}})$. Then, by introducing ε -edges, we get a finite automaton for $\downarrow L_h(\hat{\mathcal{A}})$, which, by Lemmas 6.4 and 6.5, equals $\downarrow L(\mathcal{A})$.

It is now straightforward to construct a polynomial size 1-bounded Petri net $N = (\Sigma, S, T, \partial_0, \partial_0, \mathbf{m}_0, \mathbf{m}_f)$ with $L(N) = L_h(\hat{\mathcal{A}})$. First, by adding states, we turn $\hat{\mathcal{A}}$ into a pushdown automaton $\mathcal{A}' = (Q', \Sigma, \hat{\Gamma}, E', q_0, q_f)$, where every edge reads at most one letter, i.e. every edge $p \xrightarrow{R|v} q$ in \mathcal{A}' has $R = \{x\}$ for some $x \in \Sigma \cup \{\varepsilon\}$ (this is done by ‘pasting’ the automaton for R in place of the edge). Moreover, we add ε -edges, so that for every edge $p \xrightarrow{\{x\}|v} q$, we also have an edge $p \xrightarrow{\{\varepsilon\}|v} q$. Then clearly $L_h(\mathcal{A}') = \downarrow L_h(\hat{\mathcal{A}}) = \downarrow L(\mathcal{A})$.

The net N has a place p for each state p of \mathcal{A}' and for each $i \in \{1, \dots, h\}$ and $\gamma \in \hat{\Gamma}$, it has a place (i, γ) . Moreover, for each $i \in \{0, \dots, h\}$, it has a place s_i . Here, the idea is that a configuration $c = (p, \gamma_1 \dots \gamma_n)$ of \mathcal{A}' with $\gamma_1, \dots, \gamma_n \in \hat{\Gamma}$ is represented as a marking $\mathbf{m}_c = \llbracket p, (1, \gamma_1), \dots, (n, \gamma_n), s_n \rrbracket$. We call a marking of this form a *stack marking* and will argue that every reachable marking in N is a stack marking. The transitions in N correspond to edges in \mathcal{A}' . For each edge $p \xrightarrow{\{x\}|v} q$ in $\hat{\mathcal{A}}$, we add the following transitions:

- if $v = \bar{\gamma}$ for some $\gamma \in \hat{\Gamma}$, then we have for every $1 \leq n \leq h$ a transition t with $\partial_0(t) = \llbracket p, (n, \gamma), s_n \rrbracket$, $\partial_1(t) = \llbracket q, s_{n-1} \rrbracket$, and $\lambda(t) = x$.
- if $v \in \hat{\Gamma}$, then for every $0 \leq n < h$, we add a transition t with $\partial_0(t) = \llbracket p, s_n \rrbracket$, $\partial_1(t) = \llbracket q, (n+1, v), s_{n+1} \rrbracket$, and $\lambda(t) = x$.
- if $v = \varepsilon$, then we add a transition t with $\partial_0(t) = \llbracket p \rrbracket$, $\partial_1(t) = \llbracket q \rrbracket$, and $\lambda(t) = x$.

Then clearly every reachable marking is a stack marking and we have $c \xrightarrow{x} c'$ for configurations c, c' of \mathcal{A}' of stack height $\leq h$ if and only if $\mathbf{m}_c \xrightarrow{x} \mathbf{m}_{c'}$. Therefore, if we set $\mathbf{m}_0 = \llbracket q_0, s_0 \rrbracket$ and $\mathbf{m}_f = \llbracket q_f, s_0 \rrbracket$ as initial and final marking, we have $L(N) = L_h(\mathcal{A}') = \downarrow L(\mathcal{A})$. This completes the proof of Corollary 6.3.

REFERENCES

- Parosh Aziz Abdulla, Ahmed Bouajjani, and Bengt Jonsson. 1998. On-the-Fly Analysis of Systems with Unbounded, Lossy FIFO Channels. In *Proceedings of the 10th International Conference on Computer Aided Verification (CAV 1998)*. 305–318. <https://doi.org/10.1007/BFb0028754>
- Alfred V. Aho. 1968. Indexed Grammars - An Extension of Context-Free Grammars. *J. ACM* 15, 4 (1968), 647–671. <https://doi.org/10.1145/321479.321488>
- Mohamed Faouzi Atig, Ahmed Bouajjani, and Shaz Qadeer. 2009. Context-Bounded Analysis for Concurrent Programs with Dynamic Creation of Threads. In *Proceedings of TACAS 2009*. 107–123.
- Georg Bachmeier, Michael Luttenberger, and Maximilian Schlund. 2015. Finite Automata for the Sub- and Superword Closure of CFLs: Descriptive and Computational Complexity. In *Proceedings of LATA 2015*. 473–485.
- Jean Berstel. 1979. *Transductions and context-free languages*. Teubner-Verlag.
- Rohit Chadha and Mahesh Viswanathan. 2007. Decidability Results for Well-Structured Transition Systems with Auxiliary Storage. In *CONCUR '07: Proc. 18th Int. Conf. on Concurrency Theory (LNCS)*, Vol. 4703. Springer, 136–150.
- Lorenzo Clemente, Pawel Parys, Sylvain Salvati, and Igor Walukiewicz. 2016. The Diagonal Problem for Higher-Order Recursion Schemes is Decidable. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*. ACM, 96–105. <https://doi.org/10.1145/2933575.2934527>
- Bruno Courcelle. 1991. On constructing obstruction sets of words. *Bulletin of the EATCS* 44 (1991), 178–186.
- Werner Damm. 1982. The IO-and OI-hierarchies. *Theoretical Computer Science* 20, 2 (1982), 95–207.
- Werner Damm and Andreas Goerdt. 1986. An automata-theoretical characterization of the OI-hierarchy. *Information and Control* 71, 1 (1986), 1–32.
- Catherine Dufourd, Alain Finkel, and Philippe Schnoebelen. 1998. Reset Nets Between Decidability and Undecidability. In *Proceedings of ICALP 1998*. 103–115.
- Pierre Ganty and Rupak Majumdar. 2012. Algorithmic verification of asynchronous programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 34, 1 (2012), 6.
- Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin. 2007. Well-structured languages. *Acta Inf.* 44, 3-4 (2007), 249–288. <https://doi.org/10.1007/s00236-007-0050-3>
- Sheila A. Greibach. 1978. Remarks on blind and partially blind one-way multicounter machines. *Theoretical Computer Science* 7, 3 (1978), 311 – 324. [https://doi.org/10.1016/0304-3975\(78\)90020-8](https://doi.org/10.1016/0304-3975(78)90020-8)
- Charles Grellois. 2016. *Semantics of linear logic and higher-order model-checking*. Ph.D. Dissertation. Université Denis Diderot Paris 7.
- Matthew Hague, Jonathan Kochems, and C.-H. Luke Ong. 2016. Unboundedness and downward closures of higher-order pushdown automata. In *POPL 2016: Principles of Programming Languages*. ACM, 151–163.
- Matthew Hague, Andrzej S. Murawski, C.-H. Luke Ong, and Olivier Serre. 2008. Collapsible Pushdown Automata and Recursion Schemes. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*. 452–461. <https://doi.org/10.1109/LICS.2008.34>
- Leonard H Haines. 1969. On free monoids partially ordered by embedding. *Journal of Combinatorial Theory* 6, 1 (1969), 94–98.
- John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2007. *Introduction to automata theory, languages, and computation, 3rd Edition*. Addison-Wesley.
- Matthias Jantzen. 1979. On the hierarchy of Petri net languages. *RAIRO - Theoretical Informatics and Applications - Informatique Théorique et Applications* 13, 1 (1979), 19–30. http://www.numdam.org/item?id=ITA_1979__13_1_19_0
- Ranjit Jhala and Rupak Majumdar. 2007. Interprocedural Analysis of Asynchronous Programs. In *POPL '07: Proc. 34th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages*. ACM Press, 339–350.
- Naoki Kobayashi. 2009. Types and higher-order recursion schemes for verification of higher-order programs. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*. 416–428. <https://doi.org/10.1145/1480881.1480933>
- Naoki Kobayashi. 2019. Inclusion between the frontier language of a non-deterministic recursive program scheme and the Dyck language is undecidable. *Theoretical Computer Science* 777 (2019), 409–416.
- Naoki Kobayashi and C.-H. Luke Ong. 2011. Complexity of Model Checking Recursion Schemes for Fragments of the Modal Mu-Calculus. *Logical Methods in Computer Science* 7, 4 (2011).
- AN Maslov. 1974. The hierarchy of indexed languages of an arbitrary level. *Doklady Akademii Nauk* 217, 5 (1974), 1013–1016.
- Richard Mayr. 2003. Undecidable problems in unreliable computations. *Theoretical Computer Science* 297, 1-3 (2003), 337–354.
- Luke Ong. 2015. Higher-Order Model Checking: An Overview. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*. 1–15. <https://doi.org/10.1109/LICS.2015.9>

- Pawel Parys. 2018. The Complexity of the Diagonal Problem for Recursion Schemes. In *Proceedings of FSTTCS 2017 (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 93. 45:1–45:14.
- Koushik Sen and Mahesh Viswanathan. 2006. Model Checking Multithreaded Programs with Asynchronous Atomic Methods. In *CAV '06: Proc. 18th Int. Conf. on Computer Aided Verification (LNCS)*, Vol. 4144. Springer, 300–314.
- Michael Sipser. 1997. *Introduction to the theory of computation*. PWS Publishing Company.
- Ramanathan S. Thinniyam and Georg Zetsche. 2019. Regular separability and intersection emptiness are independent problems. In *Proceedings of FSTTCS 2019 (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 150. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 51:1–51:15. <https://doi.org/10.4230/LIPIcs.FSTTCS.2019.51>
- Jan van Leeuwen. 1978. Effective constructions in well-partially-ordered free monoids. *Discrete Mathematics* 21, 3 (1978), 237–252. [https://doi.org/10.1016/0012-365X\(78\)90156-5](https://doi.org/10.1016/0012-365X(78)90156-5)
- Georg Zetsche. 2015. An Approach to Computing Downward Closures. In *ICALP 2015*, Vol. 9135. Springer, 440–451. The undecidability of Z intersection is shown in the full version: <http://arxiv.org/abs/1503.01068>.

A COMPILING AWAY INTERNAL ACTIONS

We have commented in the examples of Section 2 that internal actions and internal updates of the global state are useful in modeling asynchronous programs from their programming language syntax. Indeed, we note that the definition of asynchronous programs in [Ganty and Majumdar 2012] additionally uses a separate alphabet of *internal actions*, in addition to the alphabet of handler posts.

We end the section by showing how a model of asynchronous programs with internal actions can be reduced to our, simpler, model.

Let C be a language class over an alphabet Σ of handler names. The definition of asynchronous programs with internal actions, as used by [Ganty and Majumdar 2012; Jhala and Majumdar 2007; Sen and Viswanathan 2006], is as follows.³ An *asynchronous program over C with internal actions* (aka AP over C with internal actions) is a tuple $\mathfrak{P} = (D, \Sigma, \Sigma_i, \mathcal{L}, R, d_0, \mathbf{m}_0)$, where $D, \Sigma, d_0, \mathbf{m}_0$ are as in Definition 2.2, Σ_i is an alphabet of *internal actions* disjoint from Σ , the set $\mathcal{L} = (L_\sigma)_{\sigma \in \Sigma}$ consists of languages from C (one for each handler $\sigma \in \Sigma$), and an automaton $R = (D, \Sigma \cup \Sigma_i, \delta)$ where D is the set of states, $\Sigma \cup \Sigma_i$ the alphabet and δ the transition relation specifying the effect of each internal action on the global state D . We will write $d \xRightarrow[R]{w}^* d'$ to mean that there is a sequence of transitions with labels $w_1 w_2 \dots w_n = w$ in the automaton R using which we can reach d' from d .

For an alphabet, Σ , the *Parikh map* Parikh: $\Sigma^* \rightarrow \mathbb{M}[\Sigma]$ maps a word $w \in \Sigma^*$ to a multiset Parikh(w) such that Parikh(w)(a) is the number of occurrences of a in w . For example, Parikh($abbab$)(a) = 2, Parikh($abbab$)(b) = 3 and Parikh(ε) = $\llbracket \rrbracket$. For a language L , we define Parikh(L) = {Parikh(w) | $w \in L$ }. If the alphabet Σ is not clear from the context, we write Parikh $_\Sigma$.

The semantics of a \mathfrak{P} is given as a labeled transition system over the set of configurations, with a transition relation $\rightarrow \subseteq (D \times \mathbb{M}[\Sigma]) \times \Sigma \times (D \times \mathbb{M}[\Sigma])$ defined as follows: let $\mathbf{m}, \mathbf{m}' \in \mathbb{M}[\Sigma]$, $d, d' \in D$ and $\sigma \in \Sigma$

$$\begin{aligned} (d, \mathbf{m} \oplus \llbracket \sigma \rrbracket) &\xrightarrow[\mathfrak{P}]{\sigma} (d', \mathbf{m} \oplus \mathbf{m}') \\ \text{iff} & \\ \exists w \in (\Sigma \cup \Sigma_i)^* : d &\xRightarrow[R]{w}^* d' \wedge w \in L_\sigma \wedge \mathbf{m}' = \text{Parikh}_\Sigma(w) . \end{aligned}$$

We now show that internal actions can be compiled away. Thus, for the algorithms that follow, we use the more abstract, language-theoretic version of Definition 2.2, while we use internal actions as syntactic sugar in examples.

³The language class C in [Ganty and Majumdar 2012] is fixed to be the class of context free languages. Their definition generalizes to any language class.

LEMMA A.1. *Let C be a full trio. Given an AP \mathfrak{P}_i with internal actions over C , one can construct an AP \mathfrak{P} over C such that both have identical sets of runs.*

PROOF. The proof is along similar lines to that of Lemmas 4.3, 4.5 in [Ganty and Majumdar 2012]. Given $\mathfrak{P}_i = (D, \Sigma, \Sigma_i, \mathcal{L}, R, d_0, \mathbf{m}_0)$ we construct $\mathfrak{P} = (D, \Sigma, (L_c)_{c \in \mathbb{C}}, d_0, \mathbf{m}_0)$ such that

$$(d, \mathbf{m} \oplus \llbracket \sigma \rrbracket) \xrightarrow[\mathfrak{P}]{\sigma} (d', \mathbf{m} \oplus \mathbf{m}') \quad \text{iff} \quad (d, \mathbf{m} \oplus \llbracket \sigma \rrbracket) \xrightarrow[\mathfrak{P}_i]{\sigma} (d', \mathbf{m} \oplus \mathbf{m}')$$

Let $L(R_{d,d'})$ be the language of the automaton R when d is the initial state and d' is the accepting state. We define $L_{d\sigma d'}$ as:

$$L_{d\sigma d'} := \text{Proj}_{\Sigma}(L_{\sigma} \cap L(R_{d,d'}))$$

First observe that the projection operation is a homomorphism and $L(R_{d,d'})$ is a regular language; hence by virtue of C being a full trio $L_{d\sigma d'}$ as defined above is in C . The conditions $\exists w \in (\Sigma \cup \Sigma_i)^* : d \xRightarrow[R]{w} d' \wedge w \in L_{\sigma} \wedge \mathbf{m}' = \text{Parikh}_{\Sigma}(w)$ and $\exists w \in L_{d\sigma d'} : \text{Parikh}(w) = \mathbf{m}'$ are seen to be equivalent from the definition of $L_{d\sigma d'}$, concluding the proof of the lemma. ■

B PROOF OF PROPOSITION 3.3

We prove the following proposition.

Let $\mathfrak{P} = (D, \Sigma, (L_c)_{c \in \mathbb{C}}, d_0, \mathbf{m}_0)$ be an asynchronous program. Then $\downarrow \text{Runs}(\downarrow \mathfrak{P}) = \downarrow \text{Runs}(\mathfrak{P})$. In particular,

- (1) For every $d \in D$, \mathfrak{P} can reach d if and only if $\downarrow \mathfrak{P}$ can reach d .
- (2) \mathfrak{P} is terminating if and only if $\downarrow \mathfrak{P}$ is terminating.
- (3) \mathfrak{P} is bounded if and only if $\downarrow \mathfrak{P}$ is bounded.

PROOF. A run of the asynchronous program \mathfrak{P} is defined as a sequence $c_0, \sigma_1, c_1, \sigma_2, \dots$ containing alternating elements of configurations c_i and letters σ_i beginning with the configuration $c_0 = (d_0, \mathbf{m}_0)$. First we observe that

$$\text{Runs}(\mathfrak{P}) \subseteq \text{Runs}(\downarrow \mathfrak{P}) \tag{6}$$

because every transition enabled in \mathfrak{P} is also enabled in $\downarrow \mathfrak{P}$. Next, we claim:

$$\forall \rho \in \text{Runs}(\downarrow \mathfrak{P}) \exists \rho' \in \text{Runs}(\mathfrak{P}) \rho \preceq \rho' \tag{7}$$

Let $\rho|_k = (d_0, \mathbf{m}_0), \sigma_1, (d_1, \mathbf{m}_1), \sigma_2, \dots, \sigma_k, (d_k, \mathbf{m}_k)$ be the $2k+1$ -length prefix of ρ . We show that for each $\rho|_k$ there exists $\rho'_k \in \text{Runs}(\mathfrak{P})$ such that $\rho|_k \preceq \rho'_k$ and in addition, $\forall k \forall i \leq k \rho'_k(i) = \rho'_{k+1}(i)$. We can then define $\rho'(i) := \rho'_i(i)$ and clearly $\rho \preceq \rho'$.

We prove by induction on k .

Base Case: $\rho|_0 = \rho'|_0 = (d_0, \mathbf{m}_0)$.

Induction Step: Let $\rho|_k = (d_0, \mathbf{m}_0), \sigma_1, (d_1, \mathbf{m}_1), \sigma_2, \dots, (d_k, \mathbf{m}_k) \in \text{Runs}(\downarrow \mathfrak{P})$. By induction hypothesis there is $\rho'_{k-1} = (d_0, \mathbf{m}_0), \sigma_1, (d_1, \mathbf{m}'_1), \sigma_2, \dots, (d_{k-1}, \mathbf{m}'_{k-1}) \in \text{Runs}(\mathfrak{P})$ such that $\rho_{k-1} \preceq \rho'_{k-1}$.

$$\begin{aligned}
& (d_{k-1}, \mathbf{m}_{k-1}) \xrightarrow[\downarrow \mathfrak{P}]{\sigma_k} (d_k, \mathbf{m}_k) \\
\Rightarrow & \exists \mathbf{m}_{k-1}'': \mathbf{m}_{k-1} = \mathbf{m}_{k-1}'' \oplus \llbracket \sigma_k \rrbracket \wedge (d_{k-1}, \mathbf{m}_{k-1}'' \oplus \llbracket \sigma_k \rrbracket) \xrightarrow[\downarrow \mathfrak{P}]{\sigma_k} (d_k, \mathbf{m}_k) \\
\Rightarrow & \exists w \in \Sigma^*: w \in \downarrow L_{d_{k-1}\sigma_k d_k} \wedge \text{Parikh}(w) \oplus \mathbf{m}_{k-1}'' = \mathbf{m}_k \\
\Rightarrow & \exists w' \in \Sigma^*: w \sqsubseteq w' \wedge w' \in L_{d_{k-1}\sigma_k d_k} \\
\Rightarrow & (d_{k-1}, \mathbf{m}_{k-1}) \xrightarrow[\mathfrak{P}]{\sigma_k} (d_k, \mathbf{m}_k \oplus \mathbf{m}_\Delta) \text{ where } \mathbf{m}_\Delta \oplus \text{Parikh}(w) = \text{Parikh}(w') \\
\Rightarrow & (d_{k-1}, \mathbf{m}_{k-1}') \xrightarrow[\mathfrak{P}]{\sigma_k} (d_k, \mathbf{m}_k') \text{ where } \mathbf{m}_k' = \mathbf{m}_k \oplus \mathbf{m}_\Delta \oplus (\mathbf{m}_{k-1}' \ominus \mathbf{m}_{k-1})
\end{aligned}$$

Defining $\rho'_k := \rho'_{k-1}, \sigma_k, (d_k, \mathbf{m}_k')$ we see that $\rho|_k \preceq \rho'_k$, completing the proof of Equation 7. We are now ready to show that $\downarrow \text{Runs}(\downarrow \mathfrak{P}) = \downarrow \text{Runs}(\mathfrak{P})$. The direction $\downarrow \text{Runs}(\mathfrak{P}) \subseteq \downarrow \text{Runs}(\downarrow \mathfrak{P})$ follows immediately from Equation 6. Conversely, let

$$\begin{aligned}
& \rho = (s_0, \mathbf{n}_0), \sigma_1, (s_1, \mathbf{n}_1), \sigma_2, \dots \in \downarrow \text{Runs}(\downarrow \mathfrak{P}) \\
\Rightarrow & \exists \rho' \in \text{Runs}(\downarrow \mathfrak{P}) \quad \rho \preceq \rho' \\
\Rightarrow & \exists \rho'' \in \text{Runs}(\mathfrak{P}) \quad \rho' \preceq \rho'' \quad \text{by Equation 7} \\
\Rightarrow & \rho \in \downarrow \text{Runs}(\mathfrak{P})
\end{aligned}$$

We have proved that $\downarrow \text{Runs}(\downarrow \mathfrak{P}) = \downarrow \text{Runs}(\mathfrak{P})$. We now show that each of the three properties i.e. safety, termination and boundedness only depend on the downclosure of the runs.

Safety:

d is reachable in \mathfrak{P}

$$\text{iff } \exists \rho = (d_0, \mathbf{m}_0), \sigma_1, (d_1, \mathbf{m}_1), \sigma_2, \dots, \sigma_k, (d_k, \mathbf{m}_k) \in \text{Runs}(\mathfrak{P}) : d_k = d$$

By Equation 6, we know $\rho \in \text{Runs}(\mathfrak{P})$ implies $\rho \in \downarrow \text{Runs}(\mathfrak{P})$. Conversely, if there is $\rho' = (s_0, \mathbf{n}_0), \sigma_1, (s_1, \mathbf{n}_1), \sigma_2, \dots, \sigma_k, (s_k, \mathbf{n}_k) \in \downarrow \text{Runs}(\mathfrak{P})$ with $s_k = d$, then by Equation 7, there is $\rho = (d_0, \mathbf{m}_0), \sigma_1, (d_1, \mathbf{m}_1), \sigma_1, \dots, \sigma_k, (d_k, \mathbf{m}_k) \in \text{Runs}(\mathfrak{P})$ with $\rho' \preceq \rho$ which implies $d_k = d$. Hence we have

d is reachable in \mathfrak{P}

$$\text{iff } \exists \rho = (s_0, \mathbf{n}_0), \sigma_1, (s_1, \mathbf{n}_1), \sigma_2, \dots, \sigma_k, (s_k, \mathbf{n}_k) \in \downarrow \text{Runs}(\mathfrak{P}) : s_k = d$$

By a similar argument as above we also have:

Termination:

\mathfrak{P} does not terminate

$$\text{iff } \exists \rho \in \text{Runs}(\mathfrak{P}) : \rho \text{ is infinite}$$

$$\text{iff } \exists \rho \in \downarrow \text{Runs}(\mathfrak{P}) : \rho \text{ is infinite}$$

Boundedness:

\mathfrak{P} is bounded

$$\text{iff } \exists N \in \mathbb{N} \forall \rho \in \text{Runs}(\mathfrak{P}) \forall i |\rho(2i).m| < N$$

$$\text{iff } \exists N \in \mathbb{N} \forall \rho \in \downarrow \text{Runs}(\mathfrak{P}) \forall i |\rho(2i).m| < N$$

In each of the three cases, the property only depends on the downclosure and hence one may equivalently replace \mathfrak{P} by $\downarrow\mathfrak{P}$ since $\downarrow\text{Runs}(\downarrow\mathfrak{P}) = \downarrow\text{Runs}(\mathfrak{P})$. ■

C PROOF OF PROPOSITION 5.1

We begin with a simple observation. For every HORS $\mathcal{S} = (\Sigma, \mathcal{N}, S, \mathcal{R})$, there exists another HORS $\mathcal{S}' = (\Sigma', \mathcal{N}', S, \mathcal{R}')$ where $\Sigma' = \{\text{br}, e\} \cup \tilde{\Sigma}$ with br of arity 2, e of arity 0 and all symbols in $\tilde{\Sigma}$ of arity 1; such that $\mathcal{L}_p(\mathcal{S}) = \mathcal{L}_p(\mathcal{S}')$.

By rewriting every terminal symbol A of arity n using the rule $Ax_1x_2\cdots x_n \rightarrow \text{br}(x_1\text{br}(x_2\text{br}(\cdots \text{br}(x_{n-1}x_n)))$, we get \mathcal{S}' , which has the same path language.

We assume due to the above observation that $\Sigma = \{\text{br}, e\} \cup \tilde{\Sigma}$ where br is binary, e is nullary and all letters in $\tilde{\Sigma}$ are unary. Define $\mathcal{S}' = (\Sigma', \mathcal{N}', S', \mathcal{R}')$ as $\Sigma' = \tilde{\Sigma} \cup \{e\}$, $\mathcal{N}' = \mathcal{N} \cup \{B : o \rightarrow (o \rightarrow o)\}$, $\mathcal{R}' = \{r[\text{br}/B] \mid r \in \mathcal{R}\} \cup \{Bxy \rightarrow x, Bxy \rightarrow y\}$, where by $r[\text{br}/B]$ we mean the rule r with br uniformly replaced by B . Note that the only new non-terminal symbol introduced is B , which is of order 1. Hence the obtained word scheme \mathcal{S}' is of the same order as \mathcal{S} .

$\mathcal{L}_p(\mathcal{S}) \subseteq \mathcal{L}_w(\mathcal{S}')$: Let $w \in \mathcal{L}_p(\mathcal{S})$. Therefore there exists a finite path Π and a sentential form t such that $\forall \bar{n} \in \text{dom}(\Pi) \ t(\bar{n}) = \Pi(\bar{n}) \wedge \Pi(\bar{n}) \in \Sigma$. We derive $t' := t[\text{br}/B]$ using the corresponding rules in \mathcal{R}' . Note that the corresponding path Π' in t' satisfies $\forall \bar{n} \in \text{dom}(\Pi') \ \Pi'(\bar{n}) \in \Sigma \cup \{B\}$. We then apply either $Bxy \rightarrow x$ or $Bxy \rightarrow y$ to each B in t' according to the path Π' to obtain w in the word scheme.

$\mathcal{L}_w(\mathcal{S}') \subseteq \mathcal{L}_p(\mathcal{S})$: We define the order \leq_{pre} on sequences of natural numbers \bar{n}, \bar{m} as $\bar{n} \leq_{\text{pre}} \bar{m}$ if $\bar{m} = \bar{n}\bar{k}$ for some sequence \bar{k} .

Suppose that for given a sentential form t' of \mathcal{S}' there exists a sentential form t of \mathcal{S} and a map $\alpha : \text{dom}(t') \rightarrow \text{dom}(t)$ (simply called *embedding* henceforth) satisfying the following conditions:

- $\forall \bar{n} \in \text{dom}(t') \ \alpha(\bar{n}) \in \Sigma$

$$t'(\bar{n}) = \begin{cases} B & \text{if } t(\alpha(\bar{n})) = \text{br} \\ t(\alpha(\bar{n})) & \text{otherwise.} \end{cases} \quad (8)$$

- $\forall \bar{n}, \bar{m} \in \text{dom}(t'), \bar{n} \leq_{\text{pre}} \bar{m}$ implies

$$\alpha(\bar{n}) \leq_{\text{pre}} \alpha(\bar{m}) \quad (9)$$

$$(\forall \bar{l} \ (\bar{n} <_{\text{pre}} \bar{l} <_{\text{pre}} \bar{m}) \text{ implies } t'(\bar{l}) \notin \tilde{\Sigma}) \text{ implies } \quad (10)$$

$$(\forall \bar{k} \ \alpha(\bar{n}) <_{\text{pre}} \bar{k} <_{\text{pre}} \alpha(\bar{m}) \text{ implies } t(\bar{k}) \notin \tilde{\Sigma})$$

Informally, Equations 8, 9 and 10 state the following: α preserves labels, except for the case of br where it maps to B , α preserves the order \leq_{pre} and the images of any two nodes with node labels from $\tilde{\Sigma}$ with no node label from $\tilde{\Sigma}$ in between are mapped to two such labels with the same property.

We will show by induction on the length of the derivation that such a pair (t, α) always exists given some t' . Let us see how the existence of such t and α gives us the proposition. Consider a word $w = w_1w_2\cdots w_n \in \mathcal{L}_w(\mathcal{S}')$. In other words, there is a term $t' = w_1(w_2\cdots(w_n(e)))\cdots$ such that $S' \xrightarrow[\mathcal{S}]{*} t'$. Corresponding to this, we have a sentential form $S \xrightarrow[\mathcal{S}]{*} t$ and α satisfying the given conditions. In particular, there exists a path Π with $\text{dom}(\Pi) \subseteq \text{dom}(t)$ which is the path in t connecting the $\alpha(e)$ to $\alpha(0^n)$. It is immediate that $\text{Proj}_{\tilde{\Sigma}}(\Pi) = w$. It remains to show the existence of t and α by induction on the length of derivations.

Base case: This is trivial since $t = t' = S$.

Induction step: Suppose $t'_0 \xrightarrow[r \in \mathcal{R}']{*} t'$ where t'_0 is a sentential form. By induction hypothesis, there is a sentential form t_0 of \mathcal{S} and t'_0 embeds into t_0 via map α_0 . Assume that the rule r is applied at

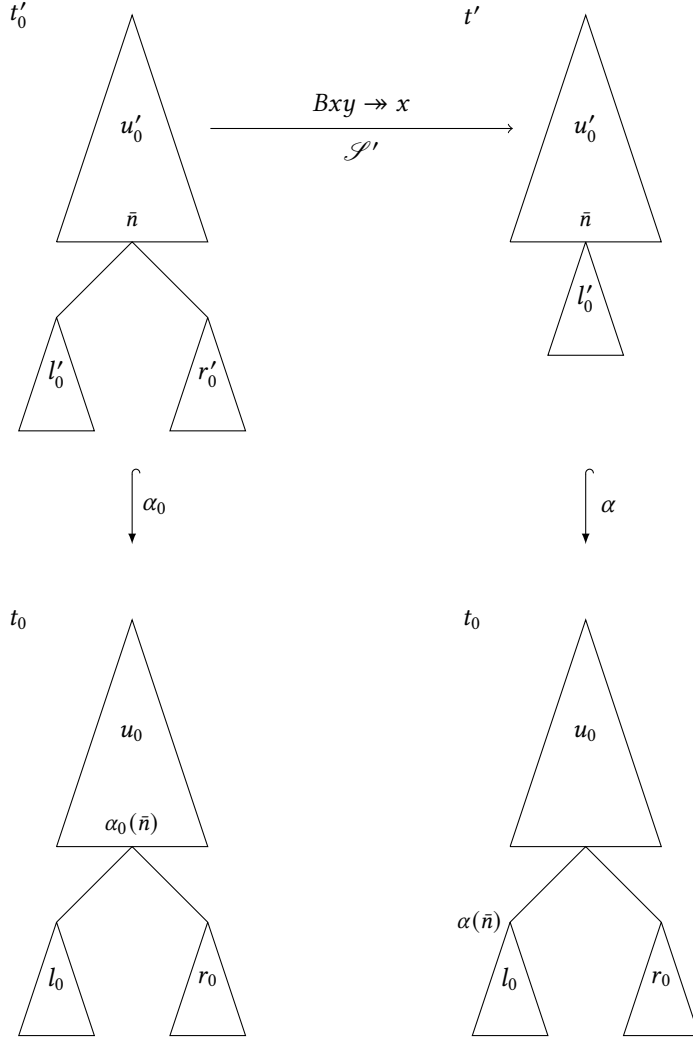


Fig. 2. Construction of embedding α from α_0 in the case of the rule $Bxy \rightarrow x$.

position \bar{n} on t'_0 . We now have two cases to consider:

Case 1: The rule r is $Bxy \rightarrow x$ (the case $Bxy \rightarrow y$ being symmetric). By induction hypothesis, we have t_0 and an embedding α_0 of t'_0 into t_0 . Referring to Figure 2, we see that t can be taken to be t_0 and α maps all nodes in the subtree u'_0 into u_0 as before, while the subtree l'_0 rooted at \bar{n} is mapped into l_0 . It is immediate that α preserves the order and since by induction hypothesis $\alpha_0(\bar{n})$ has label br , Equation 10 is also satisfied by α since no new $\tilde{\Sigma}$ labelled nodes have been added.

Case 2: We demonstrate for the case when a non-terminal of arity two is replaced for an easier reading of the proof: the rule r is $Nxy \rightarrow t_1$ for some nonterminal $N \neq B$. Referring to Figure 3, the rule replaces the subtree rooted at \bar{n} in t'_0 with $t_1[x/l'_0, y/r'_0]$ where l'_0, r'_0 are respectively the left and right subtrees of the subtree rooted at \bar{n} in u'_0 . For nodes in the subtree u'_0 of t' , α mimics α_0 . By induction we know that the subtrees l'_0, r'_0 of t'_0 embed respectively into l_0, r_0 of t_0 . Thus α maps

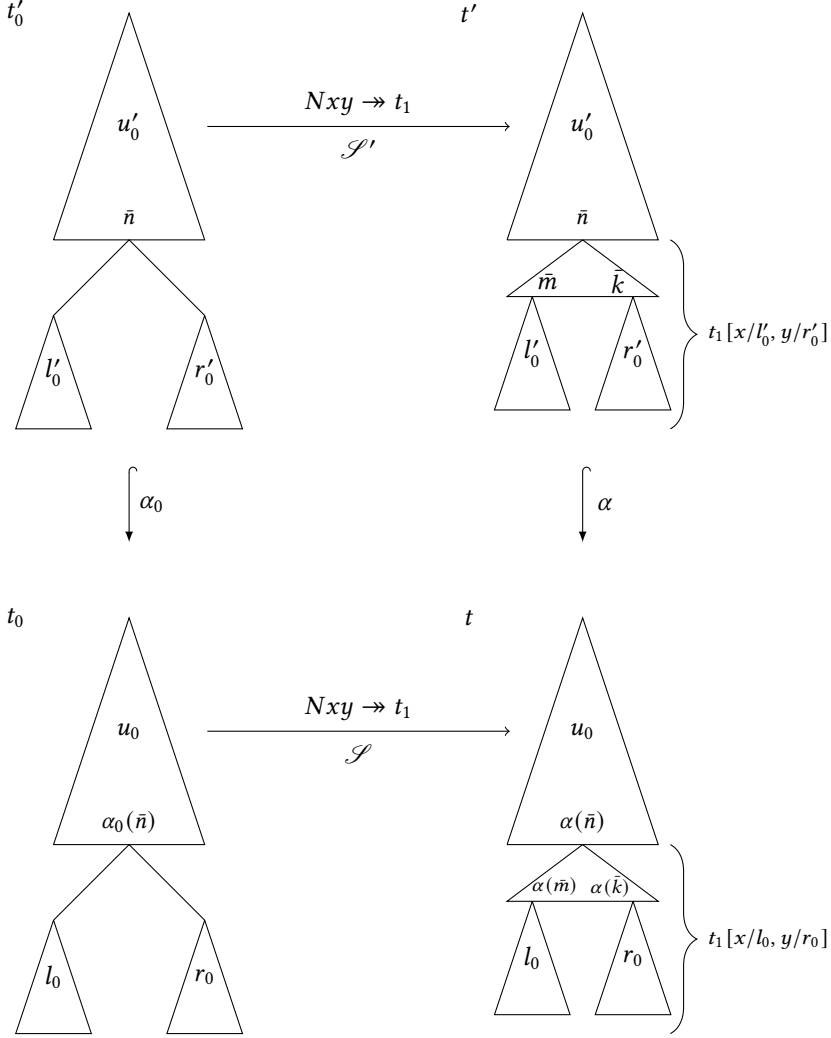


Fig. 3. Construction of embedding α from α_0 when the rule is $Nxy \rightarrow t_1$.

every subtree l'_0 (resp. r'_0) rooted at \bar{m} (resp. \bar{k}) in $t_1[x/l'_0, y/r'_0]$ into the corresponding subtree l_0 (resp. r_0) rooted at $\alpha(\bar{m})$ (resp. $\alpha(\bar{k})$). Nodes in $t_1[x/l'_0, y/r'_0]$ which are not in any of the l'_0 (or r'_0) subtrees (i.e.) between \bar{n} and \bar{m} (or \bar{k}) have corresponding nodes in $t_1[x/l_0, y/r_0]$ to which they can be mapped. Label preservation and order preservation immediately follow by appeal to the induction hypothesis. In order to see that Equation 10 holds, consider consecutive $\tilde{\Sigma}$ nodes \bar{n}' , \bar{m}' in t' (i.e. $\bar{n}' \leq \bar{m}'$ and there are no $\tilde{\Sigma}$ labels in between). If both \bar{n}' and \bar{m}' are in u'_0 or in one of the l'_0 (or r'_0) then the induction hypothesis applies. In the case where $\bar{n}' \in u'_0$ and $\bar{m}' \in l'_0$ (resp. r'_0) this means that no $\tilde{\Sigma}$ labels are present in the path from \bar{n} to \bar{m} (resp. \bar{k}), \bar{n}' to \bar{n} or \bar{m} (resp. \bar{k}) to \bar{m}' . The path from $\alpha(\bar{n})$ to $\alpha(\bar{m})$ is identical to that from \bar{n} to \bar{m} (resp. \bar{k}). The induction hypothesis also implies that the first a label for some $s \in \tilde{\Sigma}$ in l'_0 must be mapped to first a label in l_0 (or u'_0 does not contain any $\tilde{\Sigma}$ labels). Hence there are no $\tilde{\Sigma}$ labels between $\alpha(\bar{m})$ (resp. $\alpha(\bar{k})$) and $\alpha(\bar{m}')$

and similarly between $\alpha(\bar{n}')$ and $\alpha(\bar{n})$. The final case is when either \bar{n}' or \bar{m}' lies between $\alpha(\bar{n})$ and $\alpha(\bar{m})$ (resp. $\alpha(\bar{k})$). There are subcases here to consider when the other point lies in u'_0, l_0 or also between \bar{n} and \bar{m} . In all of these subcases, it easily follows that there are no extra $\tilde{\Sigma}$ labels introduced in between two consecutive nodes which are in the image of α .