Algorithmic Verification of Asynchronous Programs

PIERRE GANTY, IMDEA Software, Madrid, Spain
RUPAK MAJUMDAR, MPI-SWS, Kaiserslautern, Germany

Asynchronous programming is a ubiquitous systems programming idiom to manage concurrent interactions with the environment. In this style, instead of waiting for time-consuming operations to complete, the programmer makes a non-blocking call to the operation and posts a callback task to a task buffer that is executed later when the time-consuming operation completes. A co-operative scheduler mediates the interaction by picking and executing callback tasks from the task buffer to completion (and these callbacks can post further callbacks to be executed later). Writing correct asynchronous programs is hard because the use of callbacks, while efficient, obscures program control flow.

We provide a formal model underlying asynchronous programs and study verification problems for this model. We show that the safety verification problem for finite-data asynchronous programs is EXPSPACE-complete. We show that liveness verification for finite-data asynchronous programs is decidable and polynomial-time equivalent to Petri Net reachability. Decidability is not obvious, since even if the data is finite-state, asynchronous programs constitute infinite-state transition systems: both the program stack and the task buffer of pending asynchronous calls can be potentially unbounded.

Our main technical construction is a polynomial-time semantics-preserving reduction from asynchronous programs to Petri Nets and conversely. The reduction allows the use of algorithmic techniques on Petri Nets to the verification of asynchronous programs.

We also study several extensions to the basic models of asynchronous programs that are inspired by additional capabilities provided by implementations of asynchronous libraries, and classify the decidability and undecidability of verification questions on these extensions.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms: Languages, Verification, Reliability

Additional Key Words and Phrases: Asynchronous (event-driven) programming, liveness, fair termination, Petri nets

ACM Reference Format:
Ganty, P. and Majumdar, R. 2011. Algorithmic verification of asynchronous programs. ACM Trans. Program. Lang. Syst. V, N, Article A (January YYYY), 46 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Asynchronous programming is a ubiquitous idiom to manage concurrent interactions with the environment with low overhead. In this style of programming, rather than waiting for a time-consuming operation to complete, the programmer can make *asynchronous* proce-

This work is supported by the National Science Foundation, under grants CCF-0546170, CCF-0702743, and CNS-0720881. Pierre Ganty was sponsored by the Comunidad de Madrid's Program PROMETIDOS-CM (S2009TIC-1465), by the PEOPLE-COFUND's program AMAROUT (PCOFUND-2008-229599), and by the Spanish Ministry of Science and Innovation (TIN2010-20639). A preliminary version of this paper appeared in the ACM-SIGPLAN Symposium on the Principles of Programming Languages, 2009. Author's addresses: Pierre Ganty, IMDEA Software, Madrid, Spain, pierre.ganty@imdea.org; and Rupak Majumdar, MPI-SWS, Kaiserslautern, Germany, rupak@mpi-sws.org.

dure calls which are stored in a *task buffer* pending for later execution, instead of being executed right away. We call *handlers* those procedures that are asynchronously called by the program. In addition, the programmer can also make the usual *synchronous* procedure calls where the caller blocks until the callee finishes. A co-operative *scheduler* repeatedly picks *pending handler instances* from the task buffer and executes them atomically to completion. Execution of the handler instance can lead to further handler being *posted*. We say that handler p is posted whenever an instance of p is added to the task buffer. The posting of a handler is done using the asynchronous call mechanism. The interleaving of different picks-and-executes of pending handler instances (a pick-and-execute is often referred to as a *dispatch*) hides latency in the system. Asynchronous programming has been used to build fast servers and routers [Pai et al. 1999; Kohler et al. 2000], embedded systems and sensor networks [Hill et al. 2000], and forms the basis of web programming using Ajax.

Writing correct asynchronous programs is hard. The loose coupling between asynchronous calls obscures the control and data flow, and makes it harder to reason about them. The programmer must keep track of concurrent interactions, manage data flow between posted handlers (including saving and passing appropriate state between dispatches), and ensure progress. Since the scheduling and resource management is co-operative and performed by the programmer, one mis-behaving procedure (e.g., one that does not terminate, or takes up too many system resources) can bring down the entire system.

We study the problem of algorithmic verification of *safety* and *liveness* properties of asynchronous programs. Informally, safety properties specify that “something bad never happens,” and liveness properties specify that “something good eventually happens.” For example, a safety property can state that a web server does not crash while handling a request, and a liveness property can state that (under suitable fairness constraints) every request to a server is eventually served.

For our results, we focus on *finite-data* asynchronous programs in which data variables range over a finite domain of values. Our main results show that the safety verification for finite-data asynchronous programs is EXPSPACE-complete, and the liveness verification problem is decidable and polynomial-time equivalent to Petri net reachability. The finiteness assumption on the data is necessary for decidability results, since all verification questions are already undecidable for 2-counter machines [Minsky 1967]. However, since the depth of the stack or the size of the task buffer could both be unbounded, even with finitely many data values, asynchronous programs define transition systems with possibly *infinitely many states*.

Specifically, we develop algorithms to check that an asynchronous program (1) reaches a particular data value (*global state reachability*, to which safety questions can be reduced) and (2) terminates under certain fairness constraints on the scheduler and external events (fair termination, to which liveness questions can be reduced [Vardi 1991]). For fair termination, the fairness conditions on the scheduler rule out certain undesired paths, in which for example the scheduler postpones some pending handler forever.

For sequential programs with *synchronous* calls, both safety and liveness verification problems have been studied extensively, and decidability results are well known [Sharir and Pnueli 1981; Burkart and Steffen 1994; Reps et al. 1995; Bouajjani et al. 1997; Walukiewicz 2001]. One simple attempt is to reduce reasoning about asynchronous programs to reasoning about synchronous programs by explicitly modeling the task buffer and the scheduling. A way to model an asynchronous program as a sequential one, is to add a counter representing the number of pending instances for each handler, increment the appropriate counter each time a handler is posted, and model the scheduler as a *dispatch loop* which picks a non-zero counter, decrements it, and executes the corresponding handler code. While the reduction is sound, the resulting system is infinite state, as the counters modeling the pending handler instances can be unbounded, and it is not immediate that existing safety and liveness checkers will be complete in this case (indeed, checking safety and liveness for recursive counter programs is undecidable in general).

Instead, our decidability proofs rely on a connection between asynchronous programs and Petri nets [Reisig 1986], an infinite state concurrency model with many decidable properties. In particular, we show an encoding of asynchronous programs into Petri nets and vice versa. This enables the reduction of decision problems on asynchronous programs to problems on Petri nets. As noted in [Chadha and Viswanathan 2007; Jhala and Majumdar 2007; Sen and Viswanathan 2006], the connection to Petri nets uses the fact that the two sources of unboundedness —unbounded program stack from recursive synchronous calls and unbounded counters from pending asynchronous calls— can be decoupled: while a (possibly recursive) procedure is executing, the number of pending handler instances can only increase, and the number of pending handler instances decreases precisely when the program stack is empty. Accordingly, our proof of decidability proceeds as follows.

First, we note that the change to the state of the task buffer before and after the dispatch of a handler depends only on the number of times each handler is posted. Therefore the ordering in which handler have been posted can be simply ignored. Thus, while the execution of the handler (in general) defines a context-free language over the alphabet of handlers, what is important from the analysis perspective is the Parikh image [Parikh 1966] of this language. (Recall that the Parikh image of a word counts the number of occurrences of each letter in the word, and the Parikh image of a language is the set of Parikh images of each of its words.) We show that the effect of each handler can be encoded by a Petri net which is linear in the size of the grammar representation of the handler. Our Petri net construction builts upon [Esparza 1997] but extends it so as to satisfy one additional property of crucial importance for correctness. Given the Petri net encoding of individual handlers, we can then construct a Petri net that strings together the handlers according to the semantics of asynchronous programs. This Petri net is linear in the size of the asynchronous program and captures in a precise sense the computations of the asynchronous system. Moreover, given a Petri net, we can conversely construct an asynchronous program polynomial in the size of the Petri net that captures in a precise sense the behaviors of the net, a result that is useful to prove lower bounds on asynchronous programs.

Safety verification then reduces to checking coverability of the Petri net for which we can use known decidability results [Karp and Miller 1969; Rackoff 1978]. Together, this gives a tight EXPSPACE-complete decision procedure for safety verification of asynchronous programs. (The lower bound follows from known EXPSPACE-hardness of Petri net coverability [Lipton 1976] and an encoding of an arbitrary Petri net as an asynchronous program that is linear in the size of the Petri net.) Previous decidability proofs for safety verification [Sen and Viswanathan 2006; Jhala and Majumdar 2007] used backward reachability of well-structured transition systems [Abdulla et al. 1996] to argue decidability, and did not yield any upper bound on the complexity of the problem.

An alternate route to safety verification [Sen and Viswanathan 2006] explicitly invokes Parikh's theorem [Parikh 1966] to construct, for each handler, a regular language which has the same Parikh image. Coupled with our construction of Petri nets, this gives another algorithm for safety verification. Unfortunately, this construction does not give a tight complexity bound. It is known that the automaton representation of a regular set with the same Parikh image as a context-free grammar can be at least exponential in the size of the grammar. Thus, the Petri net obtained using the methods of [Sen and Viswanathan 2006] can be exponential in the size of the original asynchronous program. This only gives a 2EXPSPACE upper bound on safety verification (using the EXPSPACE upper bound for Petri net coverability [Rackoff 1978]).

For fair termination, we proceed in two steps. An asynchronous program can fail to terminate in two ways. First, a particular handler execution can loop forever. Second, each dispatch can terminate, but there can be infinite sequence of posted handler and dispatches.

For infinite runs of the first kind, the task buffer can be abstracted away (as no dispatches occur from within a dispatched handler) and we can use a combination of safety verification (checking that a particular handler can ever be dispatched) and techniques for liveness checking for finite-state pushdown systems [Burkart and Steffen 1994; Walukiewicz 2001] (checking that a handler loops forever).

The second case above is more interesting, and we focus on this problem. For infinite runs of the second form, we note that the Petri net constructed from an asynchronous program preserves all infinite behaviors, and we can reduce fair termination of the asynchronous program (assuming each individual dispatched handler terminates) to an analogous property on the Petri net. We show that this property can be encoded in a logic on Petri nets [Yen 1992], which can be reduced to checking certain reachability properties of Petri nets [Atig and Habermehl 2009]. Conversely, we show that the Petri net reachability problem can be reduced in polynomial time to a fair termination question on asynchronous programs. Together, we show that the fair termination problem for asynchronous programs is polynomial-time equivalent to the Petri net reachability problem. Again, this gives an EXPSPACE-hard lower bound on the problem [Lipton 1976]. On the other hand, the best known upper bounds for Petri net reachability take non-primitive recursive space [Kosaraju 1982; Lambert 1992; Mayr 1981; Mayr and Meyer 1981]. (In the absence of fairness, i.e., for the termination problem, we get an EXPSPACE-complete algorithm. Previously, [Chadha and Viswanathan 2009] gave a decision procedure for this problem, but the complexity of their procedure is not apparent.)

The reduction to Petri nets also enables us to provide decision procedures for related verification questions on asynchronous programs. First, we show a decision procedure for *boundedness*, a safety property that asserts there exists some finite N such that the maximum possible size of the task buffer at any point in any execution is at most N. For the boundedness property we again use a known result on Petri nets which allows to decide the existence of an upper bound D on the size of the task buffer at any point in any execution (or return infinity, if the task buffer is unbounded). Since the task buffer is often implemented as a finite buffer, let us say of size d, if $D > d$ holds then there is an execution of the system that leads to an overflow of the buffer, and to a possible crash. Our decision procedure for the boundedness problem uses the above reduction to Petri nets, and checks boundedness of Petri nets using standard algorithms in EXPSPACE. Second, the *fair non-starvation* question asks, given an asynchronous program and a fairness condition on executions, whether every pending handler instance is eventually dispatched (i.e., no pending handler instance waits forever). Fair non-starvation is practically relevant to ensure that an asynchronous program (such as a server) is responsive. We show fair non-starvation is decidable by showing a reduction to Petri nets.

We also study safety and liveness verification for natural extensions to asynchronous programs inspired by features supported in common asynchronous programming languages and libraries. For the model of asynchronous programs where a handler can cancel all pending instances of a handler, we show that safety is decidable, but boundedness and termination are not. If in addition, a handler can test (at most once in every execution) the absence of pending instances for a specific handler, safety becomes undecidable as well. The decidability result uses decidability of coverability of Petri nets extended with reset arcs [Abdulla et al. 1996]. The undecidability results are based on undecidability of boundedness or reachability of Petri nets with reset arcs, or the undecidability of reachability of two-counter machines.

2. INFORMAL EXAMPLES

We start by giving informal examples of asynchronous programs using, for readability, a simple imperative language. We use C-like syntax with an additional construct `post` $f(e)$ which is the syntax for an *asynchronous call* to procedure f with arguments e. Operationally, the execution of `post` $f(e)$ posts handler $f(e)$: an *instance* of handler $f(e)$ is added to the task buffer.

In the initial state of an asynchronous program, the task buffer is specified by the programmer and the program stack is empty. Whenever the program stack is empty, the scheduler dispatches a pending handler instance, if any. The program *stops* when the scheduler has no pending handler instances to dispatch.

In our formal development, we use a more abstract language acceptor based model. Compiling our imperative programs to the formal model (assuming all data variables range over finite types) is straightforward although laborious.

2.1. Safety Properties

Figure 1 shows an abstracted example of a server that runs in a loop (procedure `server`) responding to external events to connect. When a client connects to the server, the server loop allocates a data structure for the connection, reads data asynchronously, sends data back to the client, and disconnects. If there is an error reading data, the connection is disconnected.

The implementation uses asynchronous calls to procedures `read` and `send`. The server allocates data specific to a connection (`alloc_client`), sets the state of the connection to `TO_READ` and posts handler `process_client` to process the connection and posts itself to wait for the next connection.

The handler `process_client` performs data read and data send. It looks at the state of the connection and posts `read` or `send` based on the state. It is an error to execute `process_client` if the connection is in any other state (and the code is expected never to reach the label `E`).

The handler `read` can disconnect a connection based on some error (lines 1,2), or read data. If the data has not been read completely (modeled by the then-branch of the non-deterministic conditional on line 4), the state is kept at `TO_READ`. If the data has been read completely (modeled by the else-branch of the non-deterministic conditional on line 4), the state is changed to `DONE_READ`. In both cases, the procedure `process_client` is called (synchronously) which, in turn, posts `read` or `send`.

The handler `send` closes the connection by calling disconnect. It expects a connection whose state `DONE_READ` denotes data has been read (the assertion on line 1), and the state is marked `CLOSED`.

The example is representative of many server implementations, and demonstrates the difficulty of writing asynchronous programs. The sequential flow of control, in which a connection is accepted, data is read, data is sent to the client, and the connection is closed, gets broken into individual handlers and the control flow is obscured. Moreover, the state space can be unbounded as an arbitrary number of connections can be in flight at the same time.

For correct behavior of the server, the programmer expects the connection is in specific states at various stages of processing. These are demonstrated by the assertions in the code.

In this example, the assertion in `send` holds for all program executions, but the assertion in `process_client` does *not*. The assertion in `send` holds because the condition is checked in `process_client` (line 3) before `send` is posted. However, there can be an arbitrary delay between the check and the execution of `send` for this connection, with any number of other connections executing in the middle.

The assertion in `process_client` can be violated in an execution which `read` terminates a connection on line 2 by calling `disconnect` (which sets the state to `CLOSED`), and subsequently `process_client` is called on line 7. The bug occurs because the author forgot a `return` on line 2 after the `disconnect`.

Our first goal is to get a sound and complete algorithm which can automatically check an asynchronous program for safety properties such as assertions.

```
server() {
1: client *c = alloc_client();
2: if (c != 0) {
3:   c->state = TO_READ;
4:   post process_client(c);
   }
5: post server();
}

process_client(c) {
1: if (c->state == TO_READ) {
2:   post read(c); return;
   }
3: if (c->state == DONE_READ) {
4:   post send(c); return;
   }
E: assert(false);
}

read(c) {
1: if (*) {
2:   disconnect(c);   //ERROR: should return here
3: } else {
4:   if (*) { c->state = TO_READ;   }
5:   else   { c->state = DONE_READ; }
6: }
7: process_client(c);
}

send(c) {
1: assert(c->state == DONE_READ);
2: disconnect(c);  //done processing
}

disconnect(c) { // close connection
1: c->state = CLOSED;
2: return;
}

Initially:  server();
```

Fig. 1. Server example with bug

2.2. Liveness Properties

Figure 2 shows a simplified asynchronous implementation of *windowed RPC*, in which a client makes `n` asynchronous procedure calls in all, of which at most `w` $\leq$ `n` are pending at any one time. (Assume that `n` and `w` are fixed constants.) Windowed RPC is a common systems programming idiom which enables concurrent interaction with a server without overloading it.

The windowed RPC client is implemented in the procedure `wrpc`. Two global counters, `sent` and `recv`, respectively track the number times `rpccall` has been posted and the number pending instances of `rpccall` that have completed. The server is abstracted by

```
global int sent = 0, recv = 0;
global int n, w;
wrpc() {
  if (recv < n) {
    if (sent < n && sent - recv < w) {
      post rpccall();
      sent++;
    }
    post wrpc();
  } else {
    return;
  }
}
rpccall() { recv ++; }
Initially: wrpc();
```

Fig. 2. Windowed RPC implementation

the procedure `rpccall` which increments `recv`. The procedure `wrpc` first checks how many instances of `rpccall` have completed. If the number is `n` or more, it terminates. Otherwise if fewer than `n` instances to `rpccall` have been posted and the number of pending instances (equal to `sent` − `recv`) is lower than the window size `w` then `wrpc` posts `rpccall`. Finally, `wrpc` posts itself (this is done by an asynchronous recursive call), either to further post handlers or to wait for pending instances of `rpccall` to complete.

As mentioned in [Krohn et al. 2007], already in this simple case, asynchronous code with windowed control flow is quite complex as the control decisions are spread across multiple pieces of code.

Consider the desirable property that the windowed RPC fairly terminates, which implies that, at some point in time, every pending instances of `rpccall` completed and the task buffer is empty. Informally, this property is true because `wrpc` posts `rpccall` at most `n` times, and posts itself only as long as `recv` is less than `n`. Each execution of `rpccall` increments `recv`, so that after `n` dispatches of `rpccall`, the value of `recv` reaches `n`, and from this point, each dispatch to `wrpc` does not post new handler. Thus, eventually, the task buffer becomes empty.

Notice that we need the assumption that the scheduler *fairly* dispatches pending handlers: a post to q is followed by a dispatch of q. Without that assumption the program does not terminate: consider the infinite run where the scheduler always picks `wrpc` in preference to `rpccall`.

Fair Termination. An asynchronous program *fairly terminates* if (i) every time a procedure is called (synchronously or asynchronously), it eventually returns; and (ii) there is no infinite run that is fair. An infinite run is said to be *fair* if for every handler q and for every step along the run, a pending instance to q is followed by a dispatch of q. The fairness constraint is expressible as a ω-regular property.

Of course, for most server applications, the asynchronous program implementing the server should *not* terminate (indeed, termination of a server points to a bug).

Fair Non-starvation. A second "progress condition" is fair non-starvation. When an asynchronous program does not terminate, we can still require that (i) every execution of a procedure that is called (synchronous or asynchronous) eventually returns; and (ii) along every fair infinite run no handler is starved. A starving handler corresponds to a particular pending handler instance which is never dispatched, and hence which waits forever to be executed. Consider a handler `h` that posts itself twice. A fair infinite execution dispatches `h` infinitely often, even though a particular pending instance to `h` may never get to run.

```
global bit = 0;
h1() {
  if (bit == 0) {
    post h1();
    post h2();
  }
}

h2() {
  bit = 1;
}
Initially: h1();
```

Fig. 3. A fairly terminating asynchronous program

Our second goal is to provide sound and complete algorithms to check fair termination and fair non-starvation properties of asynchronous programs.

Proving safety and liveness properties for asynchronous programs is difficult for several reasons. First, as the server and the windowed RPC example suggests, reasoning about termination may require reasoning about the dataflow facts (e.g., the fact that the state is checked to be `DONE_READ` before posting `send` in server or that `recv` eventually reaches `n` in RPC). Second, at each point, there can be an unbounded number of pending handler instances. This is illustrated by the program in Fig. 3, which terminates on each fair execution, but in which the task buffer contains unboundedly many pending instances (to `h2`). Third, each handler can potentially be recursive, so the program stack can be unbounded as well.

We remark that if the finite dataflow domain induces a sound abstraction of a concrete asynchronous program in which data variables range over infinite domains, that is, if the finite abstraction has more behaviors, then our analysis is sound: if the analysis with the finite dataflow domains shows the asynchronous program fairly terminates (resp. is fair non-starving) then the original asynchronous program fairly terminates (resp. is fair non-starving).

3. PRELIMINARIES

3.1. Basics

An *alphabet* is a finite non-empty set of *symbols*. For an alphabet Σ, we write Σ^* for the set of finite sequences of symbols (also called *words*) over Σ. A set $L \subseteq \Sigma^*$ of words defines a *language*. The length of a word $w \in \Sigma^*$, denoted $|w|$, is defined as expected. An infinite word ω alphabet Σ is an infinite sequence of symbols. For a finite non-empty word $w \in \Sigma^* \setminus \{\varepsilon\}$, we write w^ω for the infinite word given by the infinite repetition of w, that is, $w \cdot w \cdot w \cdots$. The projection of word w onto some alphabet Σ', written $\mathit{Proj}_{\Sigma'}(w)$, is the word obtained by erasing from w each symbol which does not belong to Σ'. For a language L, define $\mathit{Proj}_{\Sigma'}(L) = \{\mathit{Proj}_{\Sigma'}(w) \mid w \in L\}$.

A *multiset* $\mathbf{m}\colon \Sigma \to \mathbb{N}$ 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.

We sometimes write $\mathbf{m} = [\![q_1, q_1, q_3]\!]$ for the multiset $\mathbf{m} \in \mathbb{M}[\{q_1, q_2, q_3, q_4\}]$ such that $\mathbf{m}(q_1) = 2$, $\mathbf{m}(q_2) = \mathbf{m}(q_4) = 0$, and $\mathbf{m}(q_3) = 1$. The empty multiset $[\![]\!]$ is denoted $\varnothing$. The size of a multiset $\mathbf{m}$, denoted $|\mathbf{m}|$, is given by $\sum_{\gamma \in \Sigma} \mathbf{m}(\gamma)$. Note that this definition applies to sets as well.

Given two multisets $\mathbf{m}, \mathbf{m}' \in \mathbb{M}[\Sigma]$ we define $\mathbf{m} \oplus \mathbf{m}' \in \mathbb{M}[\Sigma]$ to be multiset such that $\forall a \in \Sigma \colon (\mathbf{m} \oplus \mathbf{m}')(a) = \mathbf{m}(a) + \mathbf{m}'(a)$, we also define the natural order $\preceq$ on $\mathbb{M}[\Sigma]$ as follows: $\mathbf{m} \preceq \mathbf{m}'$ iff there exists $\mathbf{m}^{\Delta} \in \mathbb{M}[\Sigma]$ such that $\mathbf{m} \oplus \mathbf{m}^{\Delta} = \mathbf{m}'$.

Given $\mathbf{m}$, we define $\downarrow\mathbf{m}$ and $\uparrow\mathbf{m}$ to be the *downward closure* and *upward closure of* $\mathbf{m}$, defined by $\{\mathbf{m}' \in \mathbb{M}[\Sigma] \mid \mathbf{m}' \preceq \mathbf{m}\}$ and $\{\mathbf{m}' \in \mathbb{M}[\Sigma] \mid \mathbf{m} \preceq \mathbf{m}'\}$, respectively. The downward and upward closure are naturally extended to sets of multisets.

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. We define the projection of $\mathbf{m}' \in \mathbb{M}[\Sigma']$ onto $\Sigma \subseteq \Sigma'$ as the multiset $\mathbf{m} \in \mathbb{M}[\Sigma]$ such that $\forall \sigma \in \Sigma \colon \mathbf{m}(\sigma) = \mathbf{m}'(\sigma)$. We write this as follows $\mathit{Proj}_{\Sigma}(\mathbf{m}')$.

The *Parikh image* $\mathsf{Parikh} \colon \Sigma^* \to \mathbb{M}[\Sigma]$ maps a word $w \in \Sigma^*$ to a multiset $\mathsf{Parikh}(w)$ such that $\mathsf{Parikh}(w)(a)$ is the number of occurrences of a in w. For example, $\mathsf{Parikh}(abbab)(a) = 2$, $\mathsf{Parikh}(abbab)(b) = 3$ and $\mathsf{Parikh}(\varepsilon) = \varnothing$. For a language L, we define $\mathsf{Parikh}(L) = \{\mathsf{Parikh}(w) \mid w \in L\}$. Given an alphabet Σ', define $\mathsf{Parikh}_{\Sigma'}$ to be the function $\mathsf{Parikh} \circ \mathit{Proj}_{\Sigma'}$ where $\circ$ denotes the function composition.

3.2. Formal Languages

A *context-free grammar* (CFG for short) G is a tuple $(\mathcal{X}, \Sigma, \mathcal{P})$ where $\mathcal{X}$ is a finite set of variables (non-terminal letters), Σ is an alphabet of terminal letters and $\mathcal{P} \subseteq \mathcal{X} \times (\Sigma \cup \mathcal{X})^*$ a finite set of productions (the production (X, w) may also be noted $X \to w$). Given two strings $u, v \in (\Sigma \cup \mathcal{X})^*$ we define the relation $u \underset{G}{\Rightarrow} v$, if there exists a production $(X, w) \in \mathcal{P}$ and some words $y, z \in (\Sigma \cup \mathcal{X})^*$ such that $u = yXz$ and $v = ywz$. We use $\underset{G}{\Rightarrow}^*$ for the reflexive transitive closure of $\underset{G}{\Rightarrow}$. A word $w \in \Sigma^*$ is *recognized* (we also say *accepted*) from the state $X \in \mathcal{X}$ if $X \underset{G}{\Rightarrow}^* w$. We sometimes simply write $\Rightarrow$ instead of $\underset{G}{\Rightarrow}$ if G is clear from the context.

An *initialized context-free grammar* G is given by a tuple $(\mathcal{X}, \Sigma, \mathcal{P}, X_0)$ where $(\mathcal{X}, \Sigma, \mathcal{P})$ is a CFG and $X_0 \in \mathcal{X}$ is the *initial variable*. When the initial variable is clear from the context, we simply say context-free grammar.

We define the language of an initialized CFG G, denoted $L(G)$, as $\{w \in \Sigma^* \mid X_0 \Rightarrow^* w\}$. A language L is *context-free* (written CFL) if there exists an initialized CFG G such that $L = L(G)$.

A *regular grammar* R is a context-free grammar such that each production is in $\mathcal{X} \times \big((\Sigma \cdot \mathcal{X}) \cup \{\varepsilon\}\big)$. It is known that a language L is *regular* iff $L = L(R)$ for some initialized regular grammar R.

We usually use the letters G and R to denote grammars and regular grammars, respectively. Given a CFG $G = (\mathcal{X}, \Sigma, \mathcal{P})$ its *size*, denoted $\|G\|$, is given by $|\mathcal{X}| + |\Sigma| + \sum \{|Xw| \mid (X, w) \in \mathcal{P}\}$.

We will use the following result from language theory in our proofs.

LEMMA 3.1. *(Parikh's Lemma [Parikh 1966]) For any context free language L there is an effectively computable regular language L' such that* $\mathsf{Parikh}(L) = \mathsf{Parikh}(L')$.

Any two languages L and L' such that $\mathsf{Parikh}(L) = \mathsf{Parikh}(L')$ are said to be *Parikh-equivalent*.

Throughout the paper, we make the following assumption without loss of generality.

ASSUMPTION 1. $\mathcal{P} \subseteq \big(\mathcal{X} \times (\mathcal{X}^2 \cup \Sigma \cup \{\varepsilon\})\big)$ *for every* CFG $G = (\mathcal{X}, \Sigma, \mathcal{P})$.

It has been shown, see for instance in [Lange and Leiß 2010], that every CFG can be transformed, in polynomial time, into an equivalent grammar of the above form.

4. FORMAL MODEL

As noted in the informal example, our formal model consists of three ingredients: a global store of data values, a set of potentially recursive handlers, and a task buffer that maintains a multiset of pending handler instances. We formalize the representation using asynchronous programs.

4.1. Asynchronous Programs

An asynchronous program $\mathfrak{P} = (D, \Sigma, \Sigma_i, G, R, d_0, \mathbf{m}_0)$ consists of a finite set of *global states* D, an alphabet Σ of *handler names*, an alphabet Σ_i of *internal actions* disjoint from Σ, a CFG $G = (\mathcal{X}, \Sigma \cup \Sigma_i, \mathcal{P})$, a regular grammar $R = (D, \Sigma \cup \Sigma_i, \delta)$, a multiset $\mathbf{m}_0 \in \mathbb{M}[\Sigma]$ of initial pending handler instances, and an initial state $d_0 \in D$. We assume that for each $\sigma \in \Sigma$, there is a non-terminal $X_\sigma \in \mathcal{X}$ of G.

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 an asynchronous program 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 $\sigma \in \Sigma$

$$(d, \mathbf{m} \oplus [\![\sigma]\!]) \xrightarrow{\sigma} (d', \mathbf{m} \oplus \mathbf{m}')$$
$$\text{iff}$$
$$\exists w \in (\Sigma \cup \Sigma_i)^* \colon d \underset{R}{\Rightarrow}^* w \cdot d' \wedge X_\sigma \underset{G}{\Rightarrow}^* w \wedge \mathbf{m}' = \mathsf{Parikh}_\Sigma(w) \quad .$$

Intuitively, we model the (potentially recursive) code of a handler using a context-free grammar. The code of a handler does two things: first, it can change the global state (through R), and second, it can add new pending handler instances (through derivation of a word in Σ^*). Together, the transition relation $\rightarrow$ states that there is a transition from configuration $(d, \mathbf{m} \oplus [\![\sigma]\!])$ to $(d', \mathbf{m} \oplus \mathbf{m}')$ if there is an execution of handler σ that changes the global state from d to d' and adds to the task buffer the handler instances given by $\mathbf{m}'$. Note that the multiset $\mathbf{m}$ (the current content of the task buffer minus the pending handler instance σ) is unchanged while σ executes, and that the order in which the handler instances are added to the task buffer is immaterial (hence, in our definition, we take the Parikh image of w).

Finally, we conclude from the definition of their semantics that asynchronous programs satisfy the following form of *monotonicity*. Let us first define the ordering $\sqsubseteq \subseteq (D \times \mathbb{M}[\Sigma]) \times (D \times \mathbb{M}[\Sigma])$ such that $c \sqsubseteq c'$ iff $c.d = c'.d \wedge c.\mathbf{m} \preceq c'.\mathbf{m}$. Also we have:

$$\forall \sigma \in \Sigma \, \forall c_1 \, \forall c_2 \, \forall c_3 \, \exists c_4 \colon c_1 \xrightarrow{\sigma} c_2 \wedge c_1 \sqsubseteq c_3 \text{ implies } c_3 \xrightarrow{\sigma} c_4 \wedge c_2 \sqsubseteq c_4 \quad .$$

Therefore, as already pointed in [Sen and Viswanathan 2006; Chadha and Viswanathan 2009], the transitions system $\big((D \times \mathbb{M}[\Sigma], \sqsubseteq), \rightarrow, c_0\big)$ defined by asynchronous programs are *well-structured transition systems* as given in [Abdulla et al. 1996; Finkel and Schnoebelen 2001].

A *run* of an asynchronous program is a finite or infinite sequence

$$c_0 \xrightarrow{\sigma_0} c_1 \cdots c_k \xrightarrow{\sigma_k} c_{k+1} \cdots$$

of configurations c_i starting from the initial configuration c_0. A configuration c is *reachable* if there is a finite run $c_0 \xrightarrow{\sigma_0} \cdots \xrightarrow{\sigma_{k-1}} c_k$ with $c_k = c$.

A handler $\sigma \in \Sigma$ is *pending* at a configuration c if $c.\mathbf{m}(\sigma) > 0$. The handler σ is said to be *dispatched* in the transition $c \xrightarrow{\sigma} c'$.

An infinite run $c_0 \xrightarrow{\sigma_0} \cdots c_k \xrightarrow{\sigma_k} \cdots$ is *fair* if for every $\sigma \in \Sigma$, if σ is dispatched only finitely many times along the run, then σ is not pending at c_j for infinitely many j's. Intuitively, an infinite run is unfair if at some point some handler is pending and is never dispatched.

For complexity considerations, we encode an asynchronous program as follows. The grammar G and R are encoded as given in Sect. 3.2. The initial multiset is encoded as a list of pairs $(\sigma, \mathbf{m}_0(\sigma))$, and using a binary representation for $\mathbf{m}_0(\sigma)$. The *size* of an asynchronous program A encoded as above is denoted $\|A\|$.

4.2. From Program Flow Graphs to Asynchronous Programs

We briefly describe how program flow graphs can be represented formally as asynchronous programs.

We represent programs using control flow graphs [Aho et al. 1986], one for each procedure. The set of procedure names is denoted Σ. The *control flow graph* for a procedure $\sigma \in \Sigma$ consists of a labeled, directed graph (V_σ, E_σ), together with a unique entry node $v^e_\sigma \in V_\sigma$, a unique exit node $v^x_\sigma \in V_\sigma$, and an edge labeling which labels each edge with either a statement (such as assignments or conditionals) taken from a set stmts, or a *synchronous* procedure call (that gets executed immediately) or an *asynchronous* procedure call (that gets added to the task buffer). The nodes of the control flow graph correspond to control points in the procedure, the entry and exit nodes represent the point where execution begins and ends, respectively. Moreover, control flow graphs are well-formed: every node of V_σ is reachable from v^e_σ and co-reachable from v^x_σ. We allow arbitrary recursion.

Let D be a fixed finite set of dataflow values. We assume that there is an abstract transfer function $M\colon D \times (\Sigma \cup \mathsf{stmts}) \to D$ which maps dataflow values and statements to a dataflow value, and captures the abstract semantics of the program.

Let us now define an asynchronous program $\mathfrak{P} = (D, \Sigma, \mathsf{stmts}, G, R, d_0, \mathbf{m}_0)$. The reasoning underlying the definition of $\mathfrak{P}$ is to map the control flow graphs to G and the abstract transfer function to R.

We define the CFG $G = (\mathcal{X}, \Sigma \cup \mathsf{stmts}, \mathcal{P})$ where the set of nonterminals $\mathcal{X}$ is the set of all nodes in all control flow graphs.

The set of productions $\mathcal{P}$ is defined as the smallest set such that:

— $(X \to \sigma \cdot Y) \in \mathcal{P}$ if the edge (X, Y) in the control flow graph is labeled with an asynchronous call to procedure $\sigma \in \Sigma$;
— $(X \to st \cdot Y) \in \mathcal{P}$ if the edge (X, Y) is labeled with a statement $st \in \mathsf{stmts}$;
— $(X \to v^e_\sigma \cdot Y) \in \mathcal{P}$ if the edge (X, Y) is labeled with a synchronous call to procedure $\sigma \in \Sigma$;
— $(v^x_\sigma \to \varepsilon) \in \mathcal{P}$ for each procedure $\sigma \in \Sigma$.

Assumption 1 does not hold on G. However it can be enforced easily (in this case in linear time) by replacing productions of the form $X \to \gamma \cdot Y$ ($\gamma \in (\Sigma \cup \mathsf{stmts})$) by $X \to G \cdot Y$ and $G \to \gamma$) where G is a fresh variable.

We define the regular grammar $R = (D, \Sigma \cup \mathsf{stmts}, \delta)$ where $\delta = \{d \to st \cdot d' \mid d, d' \in D \wedge st \in \Sigma \cup \mathsf{stmts} \wedge M(d, st) = d'\}$.

Let $\sigma_0 \in \Sigma$ be the main procedure. Intuitively, a leftmost derivation in the grammar G starting from $v^e_{\sigma_0}$ corresponds to an interprocedurally valid path in the program. The derived word is the sequence of asynchronous calls to procedures of Σ and statements of stmts made along that path. The global state is given by executing the program along the path with the abstract semantics specified by M on the domain D starting from an initial dataflow value $d_\imath$. Therefore, $\mathfrak{P}$ is such that $\mathbf{m}_0 = [\![\sigma_0]\!]$ and $d_0 = d_\imath$.

Remark 4.1. Observe that by modelling handlers using language acceptors we are abstracting away the non terminating executions within a handler.

4.3. A Technical Construction

Given an asynchronous program $\mathfrak{P} = (D, \Sigma, \Sigma_i, G, R, d_0, \mathbf{m}_0)$, we define a "product grammar" G^R which synchronizes derivations in G and R. The CFG G^R simplifies some subsequent constructions on asynchronous programs.

Definition 4.2. Given a CFG $G = (\mathcal{X}, \Sigma \cup \Sigma_i, \mathcal{P})$ and a regular grammar $R = (D, \Sigma \cup \Sigma_i, \delta)$, define the CFG $G^R = (\mathcal{X}^R, \Sigma, \mathcal{P}^R)$ where $\mathcal{X}^R = \{[dXd'] \mid d, d' \in D, X \in \mathcal{X}\}$, and $\mathcal{P}^R$ is the least set such that each of the following holds:

— if $(X \to \varepsilon) \in \mathcal{P}$ and $d \in D$ then $([dXd] \to \varepsilon) \in \mathcal{P}^R$.
— if $(X \to a) \in \mathcal{P}$ and $(d \to a \cdot d') \in \delta$ then $([dXd'] \to \mathit{Proj}_\Sigma(a)) \in \mathcal{P}^R$.
— if $[d_0Ad_1], [d_1Bd_2] \in \mathcal{X}^R$ and $(X \to AB) \in \mathcal{P}$ then $([d_0Xd_2] \to [d_0Ad_1][d_1Bd_2]) \in \mathcal{P}^R$.

LEMMA 4.3. *Let G, R and G^R as in Def. 4.2. For every $d, d' \in D$, $X \in \mathcal{X}$, $w_1 \in \Sigma^*$ and $w \in (\Sigma \cup \Sigma_i)^*$ we have:*

$$[dXd'] \underset{G^R}{\Rightarrow}^* w_1 \textit{ implies } \exists w_2 \in (\Sigma \cup \Sigma_i)^* \colon \mathit{Proj}_\Sigma(w_2) = w_1 \wedge d \underset{R}{\Rightarrow}^* w_2 \cdot d' \wedge X \underset{G}{\Rightarrow}^* w_2 \quad (1)$$

$$d \underset{R}{\Rightarrow}^* w \cdot d' \wedge X \underset{G}{\Rightarrow}^* w \textit{ implies } [dXd'] \underset{G^R}{\Rightarrow}^* \mathit{Proj}_\Sigma(w) \ . \quad (2)$$

Moreover, G^R can be computed in time polynomial in the size of G and R.

PROOF. See Sect. A for a proof of (1) and (2). Given def. 4.2, it is routine to check that the time complexity bound holds. ∎

Lem. 4.5 below makes clear the purpose of this section: it gives an equivalent but simpler definition for the semantics of an asynchronous program.

Definition 4.4. Let $\mathfrak{P} = (D, \Sigma, \Sigma_i, G, R, d_0, \mathbf{m}_0)$ be an asynchronous program. We define a *context* to be an element of $D \times \Sigma \times D$. We also introduce the abbreviation $\mathfrak{C} = D \times \Sigma \times D$ for the set of all contexts. Let $c = (d_i, \sigma, d_f) \in \mathfrak{C}$, define G^c to be an initialized CFG which is given by G^R with the initial symbol $[d_iX_\sigma d_f]$, that is $G^c = (\mathcal{X}^R, \Sigma, \mathcal{P}^R, [d_iX_\sigma d_f])$.

LEMMA 4.5. *Let $c = (d_1, \sigma, d_2) \in \mathfrak{C}$ and $\mathbf{m} \in \mathbb{M}[\Sigma]$, we have:*

$$(d_1, [\![\sigma]\!]) \xrightarrow{\sigma} (d_2, \mathbf{m}) \quad \textit{iff} \quad \mathbf{m} \in \mathsf{Parikh}(L(G^c)) \ .$$

PROOF. The definition of $\to$ shows that

$$\begin{array}{lr}
(d_1, [\![\sigma]\!]) \xrightarrow{\sigma} (d_2, \mathbf{m}) & \\
\text{iff } \exists w \in (\Sigma \cup \Sigma_i)^* \colon d_1 \underset{R}{\Rightarrow}^* w \cdot d_2 \wedge X_\sigma \underset{G}{\Rightarrow}^* w \wedge \mathbf{m} = \mathsf{Parikh}_\Sigma(w) & \text{def. of } \to \\
\text{iff } \exists w \in (\Sigma \cup \Sigma_i)^* \colon [d_1X_\sigma d_2] \underset{G^R}{\Rightarrow}^* \mathit{Proj}_\Sigma(w) \wedge \mathbf{m} = \mathsf{Parikh}_\Sigma(w) & \text{Lem. 4.3} \\
\text{iff } \exists w \in (\Sigma \cup \Sigma_i)^* \colon [d_1X_\sigma d_2] \underset{G^R}{\Rightarrow}^* \mathit{Proj}_\Sigma(w) \wedge \mathbf{m} = \mathsf{Parikh} \circ \mathit{Proj}_\Sigma(w) & \text{def. of } \mathsf{Parikh}_\Sigma \\
\text{iff } \exists w' \in \Sigma^* \colon [d_1X_\sigma d_2] \underset{G^R}{\Rightarrow}^* w' \wedge \mathbf{m} = \mathsf{Parikh}(w') & \text{elim. } \mathit{Proj}_\Sigma \\
\text{iff } \mathbf{m} \in \mathsf{Parikh}(L(G^c)) & \text{def. of } G^c, \mathsf{Parikh}
\end{array}$$

∎

Observe that this equivalent semantics completely ignores the ordering in which handlers are posted. Using the above constructions, we have eliminated the need to explicitly carry around the internal actions Σ_i. Consequently, in what follows, we shall omit the internal actions from our description of asynchronous programs.

4.4. Properties of Asynchronous Programs

In this paper, we study the following decision problems for asynchronous programs. The first set of problems relate to properties of finite runs.

Definition 4.6.

— **Safety (Global state reachability)**:
Instance: An asynchronous program $\mathfrak{P}$ and a global state $d_f \in D$
Question: Is there a reachable configuration c such that $c.d = d_f$?
If so d_f is said to be *reachable* (in $\mathfrak{P}$); otherwise *unreachable*.
— **Boundedness (of the task buffer)**:
Instance: An asynchronous program $\mathfrak{P}$
Question: 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 *unbounded*.
— **Configuration reachability**:
Instance: An asynchronous program $\mathfrak{P}$ and a configuration c
Question: Is c reachable?

The next set of problems relate to properties of infinite runs.

Definition 4.7. All the following problems have a common input given by an asynchronous program $\mathfrak{P}$

— **Non Termination**: Is there an infinite run?
— **Fair Non Termination**: Is there a fair infinite run?
— **Fair Starvation**: Is there a fair infinite run $c_0, c_1, \ldots, c_i, \ldots$, a handler $\sigma \in \Sigma$ and some index $J \geq 0$ such that for each $j \geq J$ we have (i) $c_j.\mathbf{m}(\sigma) \geq 1$, and (ii) if $c_j \xrightarrow{\sigma} c_{j+1}$ then $c_j.\mathbf{m}(\sigma) \geq 2$?

We provide some intuition on the fair starvation property. A run could be fair, but a specific pending handler instance may never get chosen in the run. We say that the handler instance is *starved* in the run. Of course, the desired property for a program is the complement: that no handler is starved on any run (i.e., that every infinite fair run does not starve any handler).

5. PETRI NET SEMANTICS

In this section we show how asynchronous programs can be modelled by Petri nets. We review a reduction from asynchronous programs to Petri nets and sharpen the reduction to get optimal complexity bounds.

5.1. Petri nets

A *Petri net* (PN for short) $N = (S, T, F = \langle I, O \rangle)$ consists of a finite non-empty set S of *places*, a finite set T of *transitions* disjoint from S, and a pair $F = \langle I, O \rangle$ of functions $I \colon T \to \mathbb{M}[S]$ and $O \colon T \to \mathbb{M}[S]$.

To define the semantics of a PN we introduce the definition of *marking*. Given a PN $N = (S, T, F)$, a marking $\mathbf{m} \in \mathbb{M}[S]$ is a multiset which maps each $p \in S$ to a non-negative integer. For a marking $\mathbf{m}$, we say that $\mathbf{m}(p)$ gives the number of *tokens* contained in place p.

A transition $t \in T$ is *enabled at* marking $\mathbf{m}$, written $\mathbf{m}\,[t\rangle$, if $I(t) \preceq \mathbf{m}$. A transition t that is enabled at $\mathbf{m}$ can *fire*, yielding a marking $\mathbf{m}'$ such that $\mathbf{m}' \oplus I(t) = \mathbf{m} \oplus O(t)$. We write this fact as follows: $\mathbf{m}\,[t\rangle\,\mathbf{m}'$.

We extend enabledness and firing inductively to finite sequences of transitions as follows. Let $w \in T^*$. If $w = \varepsilon$ we define $\mathbf{m}\,[w\rangle\,\mathbf{m}'$ iff $\mathbf{m}' = \mathbf{m}$; else if $w = u \cdot v$ we have $\mathbf{m}\,[w\rangle\,\mathbf{m}'$ iff there exists $\mathbf{m}_1$ such that $\mathbf{m}\,[u\rangle\,\mathbf{m}_1$ and $\mathbf{m}_1\,[v\rangle\,\mathbf{m}'$.

Let $w_\infty = t_0, t_1, \ldots$ be an infinite sequence of transitions. We write $\mathbf{m}\,[w_\infty\rangle$ iff there exist markings $\mathbf{m}_0, \mathbf{m}_1, \ldots$ such that $\mathbf{m}_0 = \mathbf{m}$ and $\mathbf{m}_i\,[t_i\rangle\,\mathbf{m}_{i+1}$.

An *initialized* PN is given by a pair $(N, \mathbf{m}_\imath)$ where $N = (S, T, F)$ is a Petri net and $\mathbf{m}_\imath \in \mathbb{M}[S]$ is called the *initial marking* of N.

A marking $\mathbf{m}$ is reachable from $\mathbf{m}_0$ iff there exists $w \in T^*$ such that $\mathbf{m}_0\,[w\rangle\,\mathbf{m}$. The *set of reachable states from* $\mathbf{m}_0$, written $[\mathbf{m}_0\rangle$, is thus $\{\mathbf{m} \mid \exists w \in T^* \colon \mathbf{m}_0\,[w\rangle\,\mathbf{m}\}$. When the starting marking is omitted, it is assumed to be $\mathbf{m}_\imath$.

We now define the size of the encoding of a PN and of their markings. First, let us recall the encoding of a multiset $\mathbf{m} \in \mathbb{M}[S]$. It is encoded as a list of pairs $(p, \mathbf{m}(p))$ symbol/value for each symbol $p \in S$. The size of the encoding, noted $\|\mathbf{m}\|$, is given by the number of bits needed to write down the list of pairs, where we assume $\mathbf{m}(p)$ is encoded in binary. The encoding of a PN N is given by a list of lists. Each transition $t \in T$ is encoded by two lists corresponding to $I(t)$ and $O(t)$. The size of N, written $\|N\|$, is thus defined as $\sum_{t\in T}\|I(t)\| + \sum_{t\in T}\|O(t)\|$.

We now define the boundedness, the reachability and the coverability problem for Petri nets. Let $(N, \mathbf{m}_\imath)$ be a initialized PN. The *boundedness problem* asks if $[\mathbf{m}_\imath\rangle$ is finite set. Let $\mathbf{m} \in \mathbb{M}[S]$, the *reachability problem* (resp. *coverability problem*) asks if $\mathbf{m} \in [\mathbf{m}_\imath\rangle$ (resp. $\uparrow\mathbf{m} \cap [\mathbf{m}_\imath\rangle \neq \emptyset$) and if so $\mathbf{m}$ is said to be *reachable* (resp. *coverable*). In each of the above problem, the *size* of an instance is given by the $\|N\| + \|\mathbf{m}_\imath\|$ plus $\|\mathbf{m}\|$, if any.

A marking $\mathbf{m}$ is Boolean if for each place $p \in S$, we have $\mathbf{m}(p) \in \{0, 1\}$. An initialized Petri net is *Boolean* if $\mathbf{m}_\imath$ is Boolean and for each $t \in T$, both $I(t)$ and $O(t)$ are Boolean. The following technical lemma shows that for any initalized Petri net, one can compute in polynomial time a Boolean initialized Petri net that is equivalent w.r.t. the boundedness problem (i.e., the original Petri net is bounded iff the Boolean Petri net is). Similarly, for an initialized Petri net and a marking, one can compute a Boolean initialized Petri net and a Boolean marking that is equivalent w.r.t. the coverability and reachability problems.

LEMMA 5.1. *(1) Let $(N, \mathbf{m}_\imath)$ be an initialized* PN. *There exists a Boolean initialized* PN *$(N', \mathbf{m}'_\imath)$ computable in polynomial time in the size of $(N, \mathbf{m}_\imath)$ such that $(N, \mathbf{m}_\imath)$ is bounded iff $(N', \mathbf{m}'_\imath)$ is bounded.*

(2) Let $(N, \mathbf{m}_\imath, \mathbf{m}_f)$ be an instance of the reachability (respectively, coverability) problem. There exists a Boolean initialized Petri net $(N', \mathbf{m}'_\imath)$ and a Boolean marking $\mathbf{m}'_f$ computable in polynomial time such that $\mathbf{m}_f$ is reachable (respectively, coverable) in $(N, \mathbf{m}_\imath)$ iff $\mathbf{m}'_f$ is reachable (respectively, coverable) in $(N', \mathbf{m}'_\imath)$.

Lem. 5.1 which proof is in the appendix shows that lower bounds for Petri nets already hold for Boolean Petri nets. This will be useful in the next sections to get lower bounds on asynchronous programs.

The following results are known from the PN literature.

THEOREM 5.2.

(1) *[Rackoff 1978] The boundedness and coverability problems for* PN *are* EXPSPACE-*complete.*
(2) *[Kosaraju 1982; Lipton 1976] The reachability problem for* PN *is decidable and* EXPSPACE-*hard.*

While the best known lower bound for Petri net reachability is EXPSPACE-hard, the best known upper bounds take non-primitive recursive space [Kosaraju 1982; Mayr and Meyer 1981; Mayr 1981; Lambert 1992]. Moreover, Lem. 5.1 shows that the lower bounds hold already for Boolean Petri nets.

5.2. Petri net semantics of asynchronous programs

We now show how to model an asynchronous program $\mathfrak{P} = (D, \Sigma, G, R, d_0, \mathbf{m}_0)$ as an initialized PN $(N_{\mathfrak{P}}, \mathbf{m}_\imath)$, parameterized by a family of *widgets* $\mathcal{N}^\clubsuit = \{N_c^\clubsuit \mid c \in D \times \Sigma \times D\}$. Each widget $N^\clubsuit_{(d,a,d')}$ is a Petri net, intuitively capturing the effect of executing a handler a taking the system from global state d to global state d'.

Fix an asynchronous program $\mathfrak{P} = (D, \Sigma, G, R, d_0, \mathbf{m}_0)$. Let $\mathcal{N}^\clubsuit = \{N_c^\clubsuit \mid c \in \mathfrak{C}\}$ be a family of Petri nets, called widgets, one for each context in $\mathfrak{C}$. We say that the family $\mathcal{N}^\clubsuit$ is *adequate* if the following conditions hold. For each $c = (d_1, a, d_2) \in \mathfrak{C}$, the widget $N_c^\clubsuit = (S_c^\clubsuit, T_c^\clubsuit, F_c^\clubsuit)$ is a PN with a distinguished *entry* place $(\mathit{begin}, c) \in S_c^\clubsuit$ and a distinct *exit* place $(\mathit{end}, c) \in S_c^\clubsuit$. Moreover for every $\mathbf{m} \in \mathbb{M}[\Sigma]$ we have:

$$\exists w \in (T_c^\clubsuit)^* \colon [\![(\mathit{begin}, c)]\!] \, [w\rangle \, ([\![(\mathit{end}, c)]\!] \oplus \mathbf{m}) \text{ iff } (d_1, [\![a]\!]) \xrightarrow{a} (d_2, \mathbf{m}) \ . \tag{3}$$

Construction 1 below shows how an adequate family of widgets is “stitched together” to give a Petri net model for an asynchronous program.

CONSTRUCTION 1. *Let $\mathfrak{P} = (D, \Sigma, G, R, d_0, \mathbf{m}_0)$ be an asynchronous program and $\mathcal{N}^\clubsuit$ an adequate family of widgets for $\mathfrak{P}$. Define $(N_{\mathfrak{P}}(\mathcal{N}^\clubsuit), \mathbf{m}_\imath)$ to be an initialized* PN *where (1) $N_{\mathfrak{P}}(\mathcal{N}^\clubsuit) = (S_{\mathfrak{P}}, T_{\mathfrak{P}}, F_{\mathfrak{P}})$ is given as follows:*

— *the set $S_{\mathfrak{P}}$ of places is given by $D \cup \Sigma \cup \bigcup_{c \in \mathfrak{C}} S_c^\clubsuit$;*
— *the set $T_{\mathfrak{P}}$ of transitions is given by $\bigcup_{c \in \mathfrak{C}} \left(\{t_c^<\} \cup T_c^\clubsuit \cup \{t_c^>\}\right)$;*
— *$F_{\mathfrak{P}}$ is such that for each $c = (d_1, a, d_2) \in \mathfrak{C}$ we have*

$$\begin{aligned} F_{\mathfrak{P}}(t_c^<) &= \langle [\![d_1, a]\!], [\![(\mathit{begin}, c)]\!] \rangle \\ F_{\mathfrak{P}}(t) &= F_c^\clubsuit(t) & t \in T_c^\clubsuit \\ F_{\mathfrak{P}}(t_c^>) &= \langle [\![(\mathit{end}, c)]\!], [\![d_2]\!] \rangle \end{aligned}$$

and (2) $\mathbf{m}_\imath = [\![d_0]\!] \oplus \mathbf{m}_0$.

In what follows we use the notation $N_{\mathfrak{P}}$ to denote $N_{\mathfrak{P}}(\mathcal{N}^\clubsuit)$, which is parameterized by an adequate family $\mathcal{N}^\clubsuit$.

We show two constructions of adequate families. First, we recall a simple definition of an adequate family of widgets, inspired by a similar construction in [Sen and Viswanathan 2006], that leads to a Petri net $N_{\mathfrak{P}}$ which is exponential in the size of $\mathfrak{P}$. Next, we give a new construction of an adequate family of widgets that leads to a Petri net $N_{\mathfrak{P}}$ of size *polynomial* in $\mathfrak{P}$. As we shall see later our definition allows to infer the existence of optimal EXPSPACE algorithms for checking safety and boundedness properties.

First construction of an adequate family. Let us now define the widgets $\mathcal{N}^\star = \{N_c^\star\}_{c \in \mathfrak{C}}$ using ideas from [Sen and Viswanathan 2006]. The central idea is to rely on the effective construction of Lem. 3.1 which, given an initialized CFG G, returns an initialized regular grammar A such that the languages $L(G)$ and $L(A)$ are Parikh-equivalent.

Definition 5.3. Let $c = (d_1, a, d_2) \in \mathfrak{C}$. Let $A^c = (Q^c, \Sigma, \delta^c, q_0)$ be a regular grammar such that $\mathsf{Parikh}(L(G^c)) = \mathsf{Parikh}(L(A^c))$. Define the Petri net $N_c^\star = (S_c^\star, T_c^\star, F_c^\star)$ given as follows:

— the set $S_c^\star$ of places is given by $\{(\mathit{begin}, c), (\mathit{end}, c)\} \cup Q^c \cup \Sigma$;
— $T_c^\star = \delta^c \cup \{t_i\}$;

— the sets $F_c^\star$ are such that

$$\begin{aligned}F_c^\star(t_i) =& \langle [\![(begin, c)]\!], [\![q_0]\!]\rangle \\ F_c^\star(q \rightarrow \sigma \cdot q') =& \langle [\![q]\!], [\![q', \sigma]\!]\rangle \\ F_c^\star(q \rightarrow \varepsilon) =& \langle [\![q]\!], [\![(end, c)]\!]\rangle\end{aligned}$$

Finally, define $\mathcal{N}^\star = \{N_c^\star\}_{c\in\mathfrak{C}}$.

An invariant of $N_c^\star$ is that every reachable marking from $[\![(begin, c)]\!]$ is such that the tokens in places Q^c never exceed 1.

Lem. 5.4 shows that $\mathcal{N}^\star$ is an adequate family.

LEMMA 5.4. *Let $c = (d_1, a, d_2) \in \mathfrak{C}$, and $\mathbf{m} \in \mathbb{M}[\Sigma]$ all the following statements are equivalent:*

(1) $(d_1, [\![a]\!]) \xrightarrow{a} (d_2, \mathbf{m})$;
(2) $\mathbf{m} \in \mathsf{Parikh}(L(G^c))$;
(3) $\mathbf{m} \in \mathsf{Parikh}(L(A^c))$;
(4) $\exists w \in (T_c^\star)^* \colon [\![(begin, c)]\!]\, [w\rangle_{N_c^\star} \big(\mathbf{m} \oplus [\![(end, c)]\!]\big)$.

PROOF. (1) and (2) are equivalent by Lem. 4.5. (2) and (3) are equivalent by assumption on A^c. Finally, (3) and (4) are equivalent by Def. 5.3. ∎

Note that for some $c \in \mathfrak{C}$ the set $S_c^\star$ of places in $N_c^\star$ may be exponentially larger than the set $\mathcal{X}^c$ of variables of G^c. As an example consider the following CFG $G = (\{A_n, \ldots, A_0\}, \{a\}, \mathcal{P}, A_n)$ for some $n \geq 0$ where $\mathcal{P} = \{A_k \rightarrow A_{k-1}A_{k-1} \colon 1 \leq k \leq n\} \cup \{A_0 \rightarrow a\}$. Clearly $L(G) = \{a^{2^n}\}$ and therefore there is no regular grammar with less than 2^n variables which accepts the same language.

Second construction of an adequate family. We now define a new family $\mathcal{N} = \{N_c\}_{c\in\mathfrak{C}}$ of widgets which improves on $\mathcal{N}^\star$ by providing more compact widgets, in particular, widgets polynomial in the size of G. Given a context $c = (d_1, a, d_2) \in \mathfrak{C}$ and the associated initialized CFG $G^c = (\mathcal{X}^c, \Sigma, \mathcal{P}^c, [d_1 X_a d_2])$, the widget $N_c = (S_c, T_c, F_c)$ will be such that $|S_c| = \mathcal{O}(|\mathcal{X}^c|)$ and $|T_c| = \mathcal{O}(|\mathcal{P}^c|)$.

Our construction combines two ingredients.

The first ingredient is the following construction of [Esparza 1997] which, given an initialized CFG $G = (\mathcal{X}, \Sigma, \mathcal{P}, S)$, returns an initialized PN $(N_G, \mathbf{m}_\imath)$ where (1) $N_G = (S_G, T_G, F_G)$ is given by

— $S_G = \mathcal{X} \cup \Sigma$ and $T_G = \mathcal{P}$;
— $F_G(X \rightarrow \alpha) = \langle [\![X]\!], \mathsf{Parikh}(\alpha)\rangle$;

and (2) $\mathbf{m}_\imath = [\![S]\!]$. Let $\mathfrak{S}$ be the set of transition sequences that are enabled in $\mathbf{m}_\imath$. We conclude from [Esparza 1997] that there is a total surjective function f from the set of derivations of G onto $\mathfrak{S}$ such that for every $\alpha \in (\mathcal{X}\cup\Sigma)^*$ if $S \underset{G}{\Rightarrow}^* \alpha$ then $\mathbf{m}_\imath\, [f(S \Rightarrow^* \alpha)\rangle\, \mathsf{Parikh}(\alpha)$.

Unfortunately, the above construction cannot be used directly to build an adequate family because of the following problem. Recall that for each $c \in \mathfrak{C}$ the widget $N_c = (S_c, T_c, F_c)$ has an exit place $(end, c) \in S_c$ and condition (3) must hold. Using Lem. 4.5 we obtain that (3) is equivalent to:

$$\exists w \in (T_c)^* \colon [\![(begin, c)]\!]\, [w\rangle\, ([\![(end, c)]\!] \oplus \mathbf{m}) \text{ iff } \mathbf{m} \in \mathsf{Parikh}(L(G^c)) \enspace .$$

This means that widget N_c should put a token in (end, c) only after some $\mathbf{m} \in \mathsf{Parikh}(L(G^c))$ has been generated, that is, it should check that the derivation $S \underset{G}{\Rightarrow}^* \alpha$ it is simulating cannot be further extended, i.e., $\alpha \in \Sigma^*$. This is equivalent to checking that each place which corresponds to a variable in $\mathcal{X}^c$ is empty. However the definition of PN transitions do not allow for such a test for 0. Therefore we need an additional ingredient in the widget in order to ensure that N_c puts a token in (end, c) only after some $\mathbf{m} \in \mathsf{Parikh}(L(G^c))$ has been generated.

The second ingredient in our construction is the observation from [Esparza et al. 2010; Esparza et al. 2011] that as long as we are interested in the Parikh image of a context-free language, it suffices to only consider derivations of *bounded index*. Let us first introduce a few notions on derivations of CFG. Let $G = (\mathcal{X}, \Sigma, \mathcal{P}, S)$ be an initialized CFG. Given a word $w \in (\Sigma \cup \mathcal{X})^*$, we denote by $\#_{\mathcal{X}}(w)$ the number of symbols of w that belongs to $\mathcal{X}$. Formally, $\#_{\mathcal{X}}(w) = |\mathsf{Parikh}_{\mathcal{X}}(w)|$. A derivation $S = \alpha_0 \Rightarrow \cdots \Rightarrow \alpha_m$ of G has index k if $\#_{\mathcal{X}}(\alpha_i) \leq k$ for each $i \in \{0, \ldots, m\}$. The set of words of Σ^* derivable through derivations of index k is denoted by $L^{(k)}(G)$.

LEMMA 5.5. *(from [Esparza et al. 2011]) Let $G = (\mathcal{X}, \Sigma, \mathcal{P}, S)$ be an initialized* CFG*, and let $k = |\mathcal{X}|$, we have:* $\mathsf{Parikh}(L(G)) = \mathsf{Parikh}(L^{(k+1)}(G))$.

Our next widget construction is directly based on this result. In the following, our widget definition only differs from the construction of [Esparza 1997] by our use of an incidental budget place \$.

In the construction of $(N_G, \mathbf{m}_\imath)$ above, define $\mathfrak{s}$ to be the subset of $\mathfrak{S}$ such that every marking reachable through a sequence in $\mathfrak{s}$ has no more than k tokens in the places $\mathcal{X}$. It is routine to check that $\{f^{-1}(w) \mid w \in \mathfrak{s}\}$ corresponds the set of derivations of index k.

Let us define N_G^k which adds an extra place \$ to N_G in order to allow exactly the sequences of transitions of $\mathfrak{s}$.

Definition 5.6. Let $G = (\mathcal{X}, \Sigma, \mathcal{P}, S)$ be an initialized CFG and let $k > 0$, we define $(N_G^k, \mathbf{m}_\imath)$ to be an initialized PN where (1) $N_G^k = (S_G, T_G, F_G)$ is given by

— $S_G = \mathcal{X} \cup \Sigma \cup \{\$\}$;
— $T_G = \mathcal{P}$; and
— F_G is such that

$$F_G(X \to Z \cdot Y) = \langle [\![X, \$]\!], [\![Z, Y]\!] \rangle \qquad \text{and} \qquad F_G(X \to \sigma) = \langle [\![X]\!], \mathsf{Parikh}(\sigma) \oplus [\![\$]\!] \rangle$$

and (2) $\mathbf{m}_\imath = [\![S]\!] \oplus [\![\$^{k-1}]\!]$.

The set of enabled transition sequences of N_G^k coincides with the set of derivation of index k. In fact, every reachable marking has exactly k tokens in places $\mathcal{X} \cup \{\$\}$. Therefore no reachable marking puts more than k tokens in places $\mathcal{X}$ which coincides with the condition imposed on derivations of index k.

LEMMA 5.7. *Let $G = (\mathcal{X}, \Sigma, \mathcal{P}, S)$ be an initialized* CFG*, let $k > 0$, and let $(N_G^k = (S_G, T_G, F_G), \mathbf{m}_\imath)$. For every $\mathbf{m} \in \mathbb{M}[S_G]$*

$$(\mathbf{m} \oplus [\![\$^k]\!]) \in [\mathbf{m}_\imath\rangle_{N_G^k} \textit{ iff } \mathbf{m} \in \mathsf{Parikh}(L^{(k)}(G)) \ .$$

PROOF. We prove that for every $\alpha_1, \alpha_2 \in (\mathcal{X} \cup \Sigma)^*$ where both $\#_{\mathcal{X}}(\alpha_1) \leq k$ and $\#_{\mathcal{X}}(\alpha_2) \leq k$, we have $\alpha_1 \Rightarrow \alpha_2$ iff there exists $t \in T_G$ such that $\mathsf{Parikh}(\alpha_1) \oplus [\![\$^{k-\#_{\mathcal{X}}(\alpha_1)}]\!] \, [t\rangle_{N_G^k} \, \mathsf{Parikh}(\alpha_2) \oplus [\![\$^{k-\#_{\mathcal{X}}(\alpha_2)}]\!]$. This holds by definition of F_G. Also observe that $\mathbf{m}_\imath = \mathsf{Parikh}(S) \oplus [\![\$^{k-\#_{\mathcal{X}}(S)}]\!]$.

Now note that the right hand side is equivalent to saying that there exist $\alpha_1, \ldots, \alpha_{n+1} \in (\mathcal{X} \cup \Sigma)^*$ where each α_i is such that $\#_{\mathcal{X}}(\alpha_i) \leq k$, $S = \alpha_1$, $\alpha_{n+1} \in \Sigma^*$, $\alpha_1 \Rightarrow \alpha_2 \cdots \alpha_n \Rightarrow \alpha_{n+1}$ and $\mathbf{m} = \mathsf{Parikh}(\alpha_{n+1})$, and use induction on n. ∎

Let us now turn to our widget definition which directly relies on the above results.

Definition 5.8. Let $c = (d_1, a, d_2) \in \mathfrak{C}$, and let $G^c = (\mathcal{X}^c, \Sigma, \mathcal{P}^c, [d_1 X_a d_2])$ its associated initialized CFG. Define $k = |\mathcal{X}^c|$ and $N_c = (S_c, T_c, F_c)$ such that:

— the set S_c of places is given by $\{(\mathit{begin}, c), (\mathit{end}, c)\} \cup \mathcal{X}^c \cup \{(\$, c)\} \cup \Sigma$;
— $T_c = \{t_i, t_e\} \cup \mathcal{P}^c$; and
— the set F_c is such that

$$\begin{aligned} F_c(t_i) =& \langle [\![(\mathit{begin}, c)]\!], [\![[d_1 X_a d_2]]\!] \oplus [\![(\$, c)^k]\!]\rangle \\ F_c(X \to Z \cdot Y) =& \langle [\![X, (\$, c)]\!], [\![Z, Y]\!]\rangle \\ F_c(X \to \sigma) =& \langle [\![X]\!], \mathsf{Parikh}(\sigma) \oplus [\![(\$, c)]\!]\rangle \\ F_c(t_e) =& \langle [\![(\$, c)^{k+1}]\!], [\![(\mathit{end}, c)]\!]\rangle \end{aligned}$$

Define $\mathcal{N} = \{N_c\}_{c \in \mathfrak{C}}$.

The following lemma shows that the family constructed above is adequate.

LEMMA 5.9. *Let $c = (d_1, a, d_2) \in \mathfrak{C}$, and $\mathbf{m} \in \mathbb{M}[\Sigma]$ we have:*

$$(d_1, [\![a]\!]) \xrightarrow{a} (d_2, \mathbf{m}) \quad \text{iff} \quad \exists w \in (T_c)^* \colon [\![(\mathit{begin}, c)]\!] \, [w\rangle_{N_c} \left([\![(\mathit{end}, c)]\!] \oplus \mathbf{m}\right) \ .$$

Moreover, N_c is computable in time polynomial in the size of G^c.

PROOF. Lem. 4.5 shows that the left hand side of the equivalence can be replaced by $\mathbf{m} \in \mathsf{Parikh}(L(G^c))$. Moreover, Lem. 5.5 shows that $L^{(k+1)}(G^c)$ and $L(G^c)$ are Parikh-equivalent, hence that the left hand side of the equivalence can be replaced by $\mathbf{m} \in \mathsf{Parikh}(L^{(k+1)}(G^c))$. Finally, we conclude from Lem. 5.7 and Def. 5.8 that $\mathbf{m} \in \mathsf{Parikh}(L^{(k+1)}(G^c))$ iff $\exists w \in (T_c)^* \colon [\![(\mathit{begin}, c)]\!] \, [w\rangle_{N_c} \left([\![(\mathit{end}, c)]\!] \oplus \mathbf{m}\right)$ and we are done. Fianlly, given def. 5.8, it is routine to check that the polynomial time upper bound holds. ∎

6. MODEL CHECKING

6.1. Safety and Boundedness

In this section, we provide algorithms for checking safety (global state reachability), boundedness, and configuration reachability for asynchronous programs by reduction to equivalent problems on PN. Conversely, we show that any PN can be simulated by an asynchronous program with no recursion.

LEMMA 6.1. *Let $\mathfrak{P}$ be an asynchronous program and let $(N_{\mathfrak{P}}, \mathbf{m}_\imath)$ be an initialized* PN *as given in Constr. 1. We have*

— *$(N_{\mathfrak{P}}, \mathbf{m}_\imath)$ is bounded iff $\mathfrak{P}$ is bounded.*
— *$(d, \mathbf{m})$ is reachable in $\mathfrak{P}$ iff $[\![d]\!] \oplus \mathbf{m}$ is reachable in $N_{\mathfrak{P}}$ from $\mathbf{m}_\imath$.*

Moreover, $(N_{\mathfrak{P}}(\mathcal{N}), \mathbf{m}_\imath)$ can be computed in polynomial time from $\mathfrak{P}$.

PROOF. The results for boundedness and reachability essentially follows from requirement (3) in the definition of adequacy. For the polynomial time algorithm we first show that polynomial time is sufficient to compute $\mathcal{N}$ given $\mathfrak{P}$. In fact, the polynomial time upper bounds follows from the fact that $\mathcal{N} = \{N_c\}_{c \in \mathfrak{C}}$ contains polynomially many widgets, that each N_c is computable in polynomial time given G^c (Lem. 6.1) and that each G^c is computable in polynomial time given $\mathfrak{P}$ (basically G, R) and c (Lem. 4.3). Then given $\mathcal{N}$, it is routine to check from Constr. 1 that $(N_{\mathfrak{P}}(\mathcal{N}), \mathbf{m}_\imath)$ can be computed in polynomial time from $\mathfrak{P}$. ∎

Let us now consider the boundedness, the safety and the configuration reachability problems for asynchronous programs. Lem. 6.1 shows that for the boundedness, the safety and the configuration reachability problem for asynchronous programs there is an equivalent instance of, respectively, the boundedness, the coverability and the reachability problem for PN. Moreover each of the reduction can be carried out in polynomial time. In [Rackoff 1978] Rackoff gives EXPSPACE algorithms to solve the coverability and boundedness problem for PN, therefore we obtain an exponential space upper bound for the safety and boundedness problems for asynchronous programs. For the reachability problem, the best known upper bounds take non-primitive recursive space [Esparza and Nielsen 1994].

```
global st = $(\varepsilon, \varepsilon)$;

runPN () {
    if st $\in (T \cup \{\varepsilon\}) \times \{\varepsilon\}$ {
        pick $t \in T$ non det.;
        st = $(t, \hat{I}(t))$;
    }
    post runPN();
}

Initially: $\mathbf{m}_\imath \oplus [\![\texttt{runPN}]\!]$
```

```
$p()$ {
    if st == $(t, p \cdot w)$ {
        st = $(t, w)$;
        if $w$ == $\varepsilon$ {
            for each $p \in S$ do {
                if $O(t)(p) > 0$ {
                    post $p()$;
                }
            }
        }
    } else {
        post $p()$;
    }
}
```

Fig. 4. Let $(N = (S, T, F = \langle I, O\rangle), \mathbf{m}_\imath)$ be an initialized Boolean PN. We assume that N is such that $\forall t \in T\colon |I(t)| > 0$. The encoding of N is given by an asynchronous program with $|S| + 1$ handlers.

We now give the reverse reductions in order to derive lower complexity bounds. In fact, we show how to reduce instances of the boundedness, the coverability and the reachability problem for Boolean PN into equivalent instances of, respectively, the boundedness, the safety and the configuration reachability problem for asynchronous programs. Each of those reduction is carried out in polynomial time in the size of the given instance. From known EXPSPACE lower bounds for Petri nets, and the construction in Lem. 5.1, we get EXPSPACE lower bounds for the boundedness, the safety, and the configuration reachability problems for asynchronous programs.

Fix a Boolean initialized PN $(N, \mathbf{m}_\imath)$. The encoding of a PN as an asynchronous program given in Fig. 4 is the main ingredient of our reductions.

For readability, we describe the asynchronous program in pseudocode syntax. It is easy to convert the pseudocode to a formal asynchronous program.

Let us fix an (arbitrary) linear ordering $<$ on the places in S. For each $t \in T$, let $\hat{I}(t)$ be the sequence obtained by ordering the set $I(t)$ according to the ordering $<$ on S, and let $\mathit{suffix}(\hat{I}(t))$ be the set of suffixes of $\hat{I}(t)$. Clearly, for any $t \in T$, there are at most $|S| + 1$ elements in $\mathit{suffix}(\hat{I}(t))$.

The intuition behind the construction of Fig. 4 is the following. The asynchronous program has $|S| + 1$ procedures, one procedure p for each place $p \in S$, and a procedure `runPN` that simulates the transitions of PN. The content of the task buffer (roughly) corresponds to the marking of the Petri net.

The procedure `runPN` initiates the simulation of PN by selecting nondeterministically a transition to be fired. A global variable `st` keeps track of the transition t selected and also the preconditions that have yet to be checked in order for t to be enabled. The possible values for `st` are $(\varepsilon, \varepsilon)$ (which holds initially), and (t, w) for transition $t \in T$ and $w \in \mathit{suffix}(\hat{I}(t))$ (encoding the fact that the current transition being simulated is t, and we need to reduce the number of pending instances of each $p \in w$ by one in order to fire t). Thus, the maximum number of possible values to `st` is $|T| \cdot (|S| + 1) + 1$.

The code for `runPN` works as follows. If `st` $\in (T \cup \{\varepsilon\}) \times \{\varepsilon\}$, it nondeterministically selects an arbitrary transition t of the PN (not necessarily an enabled transition) to be fired, sets `st` to $(t, \hat{I}(t))$, and reposts itself. If `st` $\notin (T \cup \{\varepsilon\}) \times \{\varepsilon\}$, it simply reposts itself.

We now describe how a transition is fired based on the global state `st`. When `runPN` sets `st` to $(t, \hat{I}(t))$, it means that we must consume a token from each place in $I(t)$ in order to fire t. Then the intuition is the following. Each time a handler p is dispatched it will check if it is the first element in the precondition, i.e., if `st` $= (t, p \cdot w)$ for some w. If p is not the first element in the precondition, it simply reposts itself, so that the number of pending instances to each $p' \in S$ before and after the dispatch of p are equal. However, if `st` $= (t, p \cdot w)$, there are two possibilities. If $w \neq \epsilon$, then handler p updates `st` to (t, w), but does not repost itself. This ensures that after the execution of p, the number of pending instances to p is one fewer than before the execution of p (and thus, we make progress in firing the transition t by consuming a token from its precondition). If $w = \epsilon$, then additionally, handler p posts p' for each $p' \in O(t)$. This ensures that the execution of the transition t is complete, and moreover, each place in $O(t)$ now has a pending handler instance corresponding to the firing of t.

The initial task buffer is the multiset $\mathbf{m}_\imath \oplus [\![\texttt{runPN}]\!]$ and the initial value of `st` is $(\varepsilon, \varepsilon)$.

The following invariant is preserved by the program of Fig. 4, whenever `st` $= (tr, \varepsilon)$ for $tr \in T \cup \{\varepsilon\}$ we have that the multiset $\mathbf{m}$ given by the number of pending instances to procedure p for each $p \in S$ is such that $\mathbf{m}_\imath\, [w \cdot tr\rangle\, \mathbf{m}$ for $w \in T^*$.

Let us prove the invariant. Initially, we have `st` $= (\varepsilon, \varepsilon)$ and the task buffer is precisely $\mathbf{m}_\imath$, so the invariant holds because we have $\mathbf{m}_\imath\, [\varepsilon\rangle\, \mathbf{m}_\imath$.

By induction hypothesis, assume the invariant holds at some configuration of the program in which `st` $\in (T \cup \{\varepsilon\}) \times \{\varepsilon\}$. We show the invariant holds the next time `st` $\in (T \cup \{\varepsilon\}) \times \{\varepsilon\}$.

Whenever `st` is of the form (tr, ε), each dispatch to p for $p \in S$ simply reposts itself. When procedure `runPN` is dispatched, it picks a transition t to be fired. Hence `st` is updated $(t, \hat{I}(t))$. Suppose $\mathbf{m}\, [t\rangle$. Then, for each $p \in I(t)$, the program configuration has a pending instance of p. A sequence of dispatches corresponding to $\hat{I}(t)$ will reduce `st` to (t, p) for some $p \in S$, and at this point, the dispatch of p will post as many calls as $O(t)$. The configuration reached after this dispatch of p sets `st` $= (t, \varepsilon)$ and the configuration of the program corresponds to a marking $\mathbf{m}' \oplus I(t) = \mathbf{m} \oplus O(t)$.

Now suppose t is not enabled in $\mathbf{m}$. Then in the simulation, `st` will get to some value $(t, p \cdot w)$ such that there is no pending instance to p. In this case, the state `st` will never be set to some value in $(T \cup \{\varepsilon\}) \times \{\varepsilon\}$, and hence the invariant holds vacuously.

We conclude by establishing the EXPSPACE lower bounds for boundedness, safety and configuration reachability.

Boundedness. Consider the reduction given at Fig. 4 which given an initialized Boolean PN $(N, \mathbf{m}_\imath)$ builds an asynchronous program $\mathfrak{P}$. We deduce from above that $\mathfrak{P}$ is bounded iff $(N, \mathbf{m}_\imath)$ is bounded. Moreover, it is routine to check that $\mathfrak{P}$ can be computed in time polynomial in the size of $(N, \mathbf{m}_\imath)$.

Safety. Consider an instance of the coverability problem for Boolean PN. Because of the result of Lem. 5.1 we can assume this instance has the following form: a PN $N^\flat = (S \cup \{p_i, p_c\}, T \cup \{t_i, t_c\}, F^\flat)$, an initial marking $[\![p_i]\!]$ and a marking to cover $[\![p_c]\!]$. Moreover the only way to create a token in place p_c is by firing transition t_c. Observe that $\uparrow [\![p_c]\!] \cap [[\![p_i]\!]\rangle_{N^\flat}$, namely p_c is coverable, iff there exists $\mathbf{m}$ such that $\mathbf{m} \in [[\![p_i]\!]\rangle_{N^\flat}$ and $\mathbf{m}\, [t_c\rangle$.

Then using the polynomial time construction given at Fig. 4 we obtain a asynchronous program $\mathfrak{P}$ which satisfies the property that the global state `st` $= (t_c, \varepsilon)$ is reachable in $\mathfrak{P}$ iff $\uparrow [\![p_c]\!] \cap [[\![p_i]\!]\rangle_{N^\flat}$.

Configuration reachability. Consider an instance of the reachability problem for Boolean PN. Because of the result of Lem. 5.1 we can assume this instance has the following form: a PN $N^\flat = (S \cup \{p_i, p_r\}, T \cup \{t_i, t_r\}, F^\flat)$ an initial marking $[\![p_i, p_r]\!]$ and $\varnothing$ a marking to reach. Moreover, every transition sequence which reaches $\varnothing$ ends with the firing of t_r. Therefore using the polynomial time construction given at Fig. 4 we obtain a asynchronous program $\mathfrak{P}$ which satisfies the property that the configuration c such that $c.d$ is given by $\mathtt{st} = (t_r, \varepsilon)$ and $c.\mathbf{m} = \varnothing$ is reachable in $\mathfrak{P}$ iff $\varnothing \in [[\![p_i, p_r]\!]\rangle_{N^\flat}$.

Hence we finally obtain the following results.

THEOREM 6.2.

(1) *The global state reachability and boundedness problems for asynchronous programs are* EXPSPACE-*complete.*
(2) *The configuration reachability problem for asynchronous programs is polynomial-time equivalent to the* PN *reachability problem. The configuration reachability problem is* EXPSPACE-*hard.*

6.2. Termination

Since we now study properties of infinite runs of Petri nets modelling asynchronous programs, there is a subset of transitions which becomes of particular interest. This subset allows to distinguish the runs where some widget enters a non terminating execution from those runs where each time a widget runs, it eventually terminates. Since our definition of asynchronous program does not allow for non-terminating runs of a handler (see Rmk. 4.1) we need a way to discriminate non-terminating runs in the corresponding PN widget.

Definition 6.3. Let $T_{\mathfrak{P}}^{d(a)} = \{t_c^{>} \in T_{\mathfrak{P}} \mid c \in \mathfrak{C} \cap (D \times \{a\} \times D)\}$ for some $a \in \Sigma$ and let $T_{\mathfrak{P}}^{d} = \bigcup_{a \in \Sigma} T_{\mathfrak{P}}^{d(a)}$.

Definition 6.4. Let $\mathfrak{P}$ be an asynchronous program, and let $(N_{\mathfrak{P}}, \mathbf{m}_\imath)$ be an initialized PN as given in Constr. 1. Let $\rho = \mathbf{m}_0\,[t_0\rangle\,\mathbf{m}_1\,[t_1\rangle \ldots \mathbf{m}_n\,[t_n\rangle \ldots$ be an infinite run of $N_{\mathfrak{P}}$ where $\mathbf{m}_0 = \mathbf{m}_\imath$.

— ρ is an infinite $\mathfrak{P}$-run iff $t_i \in T_{\mathfrak{P}}^d$ for infinitely many i's;

— ρ is a fair infinite run iff $\begin{cases} \rho \text{ is an infinite } \mathfrak{P}\text{-run, and} \\ \text{for all } a \in \Sigma,\ \text{if } t_i \in T_{\mathfrak{P}}^{d(a)} \text{ for finitely many } i\text{'s} \\ \text{then } \mathbf{m}_j(a) = 0 \text{ for infinitely many } j\text{'s} \end{cases}$;

— ρ fairly starves $b (\in \Sigma)$ iff $\begin{cases} \rho \text{ is a fair infinite run, and} \\ \text{there is a } J \geq 0 \text{ such that for each } j \geq J \\ \mathbf{m}_j(b) \geq 1 \wedge (t_j \in T_{\mathfrak{P}}^{d(b)} \rightarrow \mathbf{m}_j(b) \geq 2) \end{cases}$.

LEMMA 6.5. *Let $\mathfrak{P}$ be an asynchronous program and let $(N_{\mathfrak{P}}, \mathbf{m}_\imath)$ be an initialized* PN *as given in Constr. 1.*

— *$\mathfrak{P}$ has an infinite run iff $(N_{\mathfrak{P}}, \mathbf{m}_\imath)$ has an infinite $\mathfrak{P}$-run;*
— *$\mathfrak{P}$ has a fair infinite run iff $(N_{\mathfrak{P}}, \mathbf{m}_\imath)$ has a fair infinite run;*
— *$\mathfrak{P}$ fairly starves a iff $(N_{\mathfrak{P}}, \mathbf{m}_\imath)$ fairly starves a.*

Moreover, $(N_{\mathfrak{P}}(\mathcal{N}), \mathbf{m}_\imath)$ can be computed in polynomial time from $\mathfrak{P}$.

PROOF. It suffices to observe that the Def. 6.4 and Def. 4.7 are equivalent using (3). The polynomial time construction was proven in Lem. 6.1. ∎

We now give an EXPSPACE-complete decision procedure for termination.

Remark 6.6. In what follows we assume a fixed linear order on the set of transitions T (resp. places S) which allow us to identify a multiset with a vector of $\mathbb{N}^T$ (resp. $\mathbb{N}^S$).

We recall a class of path formulas for which the model checking problem is decidable. This class was originally defined in [Yen 1992], but the model checking procedure in that paper had an error which was subsequently fixed in [Atig and Habermehl 2009]. For simplicity, our definition below captures only a subset of the path formulas defined in [Yen 1992], but this subset is sufficient to specify termination.

Fix a PN $N = (S, T, F, \mathbf{m}_\imath)$. Let $\mu_1, \mu_2, \ldots$ be a family of *marking variables* ranging over $\mathbb{N}^S$ and $\sigma_1, \sigma_2, \ldots$ a family of *transition variables* ranging over T^*.

Terms are defined recursively as follows:

— every $\mathbf{c} \in \mathbb{N}^S$ is a term;
— for all $j > i$, and marking variables μ_j and μ_i, we have $\mu_j - \mu_i$ is a term.
— $\mathcal{T}_1 + \mathcal{T}_2$ and $\mathcal{T}_1 - \mathcal{T}_2$ are terms if $\mathcal{T}_1$ and $\mathcal{T}_2$ are terms. (Consequently, every mapping $\mathbf{c} \in \mathbb{Z}^S$ is also a term)

Atomic predicates are of two types: marking predicates and transition predicates.

Marking predicates. There are two types of marking predicates. The first type consists in the forms $\mathcal{T}_1(p_1) = \mathcal{T}_2(p_2)$, $\mathcal{T}_1(p_1) < \mathcal{T}_2(p_2)$, and $\mathcal{T}_1(p_1) > \mathcal{T}_2(p_2)$, where $\mathcal{T}_1$ and $\mathcal{T}_2$ are terms and $p_1, p_2 \in S$ are two places of N. The second type consists in the forms $\mu(p) \geq z$ and $\mu(p) > z$, where μ is a marking variable, $p \in S$, and $z \in \mathbb{Z}$.

Transition predicates. Define the *inner product* $\otimes : \mathbb{Z}^T \times \mathbb{Z}^T \to \mathbb{Z}^T$ as $\mathbf{c}_1 \otimes \mathbf{c}_2 = \sum_{t \in T} \mathbf{c}_1(t) \cdot \mathbf{c}_2(t)$. for $\mathbf{c}_1, \mathbf{c}_2 \in \mathbb{Z}^T$. A transition predicate is either of the form $\mathsf{Parikh}(\sigma_1)(t) \leq c$, where $c \in \mathbb{N}$ and $t \in T$, or of the forms $\mathbf{y} \otimes \mathsf{Parikh}(\sigma_i) \geq c$ and $\mathbf{y} \otimes \mathsf{Parikh}(\sigma_i) \leq c$, where $i > 1$, $c \in \mathbb{N}$, $\mathbf{y} \in \mathbb{Z}^T$, and $\otimes$ denotes the inner product as defined above.

A *predicate* is a finite positive boolean combination of atomic predicates. A *path formula* Λ is a formula of the form:

$$\exists \mu_1, \ldots, \mu_m \exists \sigma_1, \ldots, \sigma_m \colon \big(\mathbf{m}_\imath \, [\sigma_1\rangle \, \mu_1 \, [\sigma_2\rangle \ldots [\sigma_m\rangle \, \mu_m\big) \wedge \Phi(\mu_1, \ldots, \mu_m, \sigma_1, \ldots, \sigma_m)$$

where Φ is a predicate. A *path formula* Λ is increasing if Φ implies $\mu_1 \leq \mu_m$ (where $\mu_i \leq \mu_j$ for $i < j$ is an abbreviation for $\bigwedge_{p \in S} (\mu_j - \mu_i)(p) > (-1^S)(p)$) and contains no transition predicate. The size of a path formula is the number of symbols in the description of the formula, where constants are encoded in binary.

The *satisfiability problem* for a path formula Λ asks if there exists a run of N of the form $\mathbf{m}_\imath \, [w_1\rangle \, \mathbf{m}_1 \, [w_2\rangle \ldots \mathbf{m}_{m-1} \, [w_m\rangle \, \mathbf{m}_m$ for markings $\mathbf{m}_1, \ldots, \mathbf{m}_m$ and transition sequences $w_1, \ldots, w_m \in T^*$, such that $\Phi(\mathbf{m}_1, \ldots, \mathbf{m}_m, w_1, \ldots, w_m)$ is true. If Λ is satisfiable, we write $N \models \Lambda$.

THEOREM 6.7. *(from [Atig and Habermehl 2009])*

— *The satisfiability problem for a path formula is reducible in polynomial time to the reachability problem for Petri nets. Hence, the satisfiability problem is* EXPSPACE*-hard.*
— *The satisfiability problem for an increasing path formula is* EXPSPACE*-complete.*

We know define our reduction of the termination problem to the satisfiability problem for an increasing path formula.

Remark 6.8. Without loss of generality, we assume that in $\mathfrak{P}$, the set $\mathbf{m}_0$ of initial pending handler instances is given by the singleton $[\![a_0]\!]$ for some $a_0 \in \Sigma$ and a_0 is never posted.

Lemma 6.9. *Let $\mathfrak{P}$ be an asynchronous program and let $(N_{\mathfrak{P}}, \mathbf{m}_\imath)$ be an initialized* PN *as given in Constr. 1. Let Λ_t be the path formula given by*

$$\exists \mu_1, \mu_2 \colon \exists \sigma_1, \sigma_2 \colon \big(\mathbf{m}_\imath \, [\sigma_1\rangle \, \mu_1 \, [\sigma_2\rangle \, \mu_2\big) \wedge \mu_1 \leq \mu_2 \wedge T^d_{\mathfrak{P}} \otimes \mathsf{Parikh}(\sigma_2) \geq 1 \ .$$

We have $(N_{\mathfrak{P}}, \mathbf{m}_\imath) \models \Lambda_t$ *iff* $(N_{\mathfrak{P}}, \mathbf{m}_\imath)$ *has an infinite $\mathfrak{P}$-run.*

Proof. Let us first give a few facts about Λ_t:

— *Fact 0:* Λ_t is polynomial in the size of $(N_{\mathfrak{P}}, \mathbf{m}_\imath)$.
— *Fact 1:* $T^d_{\mathfrak{P}} \otimes \mathsf{Parikh}(\sigma_2) \geq 1$ implies that $\sigma_2 \in T^*_{\mathfrak{P}} \cdot T^d_{\mathfrak{P}} \cdot T^*_{\mathfrak{P}}$ because it requires that some transition of $T^d_{\mathfrak{P}}$ is fired along σ_2;
— *Fact 2:* $\mu_1 \leq \mu_2$ implies the sequence of transition given by σ_2 can be fired over and over.

Let us now turn to the proof.

Only if: Let $\mathbf{m}_1$, $\mathbf{m}_2$, w_1 and w_2 be a valuation of μ_1, μ_2, σ_1 and σ_2 respectively such that Λ_t is satisfied. Fact 1 shows that $w_2 \neq \varepsilon$ and $\mathsf{Parikh}(w_2)(t) > 0$ for some $t \in T^d_{\mathfrak{P}}$. Then Fact 2 shows that $\mathbf{m}_\imath \, [w_1\rangle \, \mathbf{m}_1 \, [w_2^\omega\rangle$ is an infinite $\mathfrak{P}$-run of $N_{\mathfrak{P}}$ and we are done.

If: Let ρ be an infinite $\mathfrak{P}$-run of $(N_{\mathfrak{P}}, \mathbf{m}_\imath)$. By definition of infinite $\mathfrak{P}$-run, ρ can be written as $\mathbf{m}_0 \, [w_0\rangle \, \mathbf{m}_1 \ldots \mathbf{m}_n \, [w_n\rangle \ldots$ where $\mathbf{m}_0 = \mathbf{m}_\imath$ and for each $k \geq 0$, we have $w_k \in T^*_{\mathfrak{P}} \cdot T^d_{\mathfrak{P}}$. By Dickson's Lemma [Dickson 1913], there exists two indices $i < j$ in the above infinite run such that $\mathbf{m}_i \preceq \mathbf{m}_j$. Let $\sigma_1 = w_0 \ldots w_{i-1}$, $\sigma_2 = w_i \ldots w_j$, $\mu_1 = \mathbf{m}_i$ and $\mu_2 = \mathbf{m}_{j+1}$. Clearly $\mu_1 \leq \mu_2$. Also we have that $\sigma_2 \neq \varepsilon$ because some transition of $T^d_{\mathfrak{P}}$ is in each w_k, and hence $T^d_{\mathfrak{P}} \otimes \mathsf{Parikh}(\sigma_2) \geq 1$. Thus, every conjunction of Λ_t is satisfied. ∎

Proposition 6.10. *Given an asynchronous program $\mathfrak{P}$, determining the existence of an infinite run is* EXPSPACE*-complete.*

Proof. As expected our decision procedure relies on reductions to equivalent PN problems. We start by observing that the PN $N_{\mathfrak{P}}$ can be computed in time polynomial in the size of $\mathfrak{P}$. Lem. 6.5 shows that $\mathfrak{P}$ has an infinite run iff $(N_{\mathfrak{P}}, \mathbf{m}_\imath)$ has an infinite $\mathfrak{P}$-run. Next, Lem. 6.9 shows that determining whether $(N_{\mathfrak{P}}, \mathbf{m}_\imath)$ has an infinite $\mathfrak{P}$-run is equivalent to determining the satisfiability of $(N_{\mathfrak{P}}, \mathbf{m}_\imath) \models \Lambda_t$ where Λ_t can be computed in time polynomial in the size of $N_{\mathfrak{P}}$. The formula Λ_t is not an increasing path formula because it contains a transition predicate ($T^d_{\mathfrak{P}} \otimes \mathsf{Parikh}(\sigma_2) \geq 1$). However the problem instance $(N_{\mathfrak{P}}, \mathbf{m}_\imath, \Lambda_t)$ can easily be turned into an equivalent instance $(N'_{\mathfrak{P}}, \mathbf{m}'_\imath, \Lambda'_t)$ that is computable in polynomial time and such that Λ'_t is a increasing path formula. This is accomplished by adding a place p_w to which a token is added each time some transitions of $T^d_{\mathfrak{P}}$ is fired. Then it suffices to replace $T^d_{\mathfrak{P}} \otimes \mathsf{Parikh}(\sigma_2) \geq 1$ by $(\mu_2 - \mu_1)(p_w) > 0^S(p_w)$. It is routine to check that Λ'_t is a increasing path formula.

Finally, the result of Thm. 6.7 together with the fact that Λ'_t is an increasing path formula shows that the satisfiability of $(N'_{\mathfrak{P}}, \mathbf{m}'_\imath) \models \Lambda'_t$ can be determined in space exponential in the size of the input. Therefore we conclude that determining the existence of an infinite run in a given $\mathfrak{P}$ has an EXPSPACE upper bound. The EXPSPACE lower bound follows by reduction from the termination of simple programs [Lipton 1976]. Indeed, the construction of [Lipton 1976] (see also [Esparza 1998]) shows how a deterministic 2^{2^n}-bounded counter machine of size $O(n)$ can be simulated by a Petri net of size $O(n^2)$ such that the counter machine has an infinite computation iff the Petri net has an infinite execution and this construction is easily adapted to use asynchronous programs. ∎

6.3. Fair Termination

We now turn to fair termination.

LEMMA 6.11. *Let $\mathfrak{P}$ be an asynchronous program and let $(N_{\mathfrak{P}}, \mathbf{m}_\imath)$ be an initialized* PN *as given in Constr. 1. Let Λ_{ft} be the path formula given by*

$$\exists \mu_1, \mu_2, \mu_3 \colon \exists \sigma_1, \sigma_2, \sigma_3 \colon \mathbf{m}_\imath \, [\sigma_1\rangle \, \mu_1 \, [\sigma_2\rangle \, \mu_2 \, [\sigma_3\rangle \, \mu_3$$
$$T_{\mathfrak{P}} \otimes \mathsf{Parikh}(\sigma_1) \leq 0 \wedge \mu_2 \leq \mu_3 \wedge T^d_{\mathfrak{P}} \otimes \mathsf{Parikh}(\sigma_3) \geq 1$$
$$\bigwedge\nolimits_{a \in \Sigma} \Big(\mathbf{c}_a \otimes \mathsf{Parikh}(\sigma_3) = 0 \rightarrow \big((\mathbf{p}_a - \mathbf{c}_a) \otimes \mathsf{Parikh}(\sigma_2) = 0 \wedge \mathbf{p}_a \otimes \mathsf{Parikh}(\sigma_3) = 0 \big) \Big)$$

where $\mathbf{c}_a, \mathbf{p}_a \in \mathbb{M}[T_{\mathfrak{P}}]$ are s.t. $\mathbf{c}_a(t) = I(t)(a)$ and $\mathbf{p}_a(t) = O(t)(a)$ for every $t \in T_{\mathfrak{P}}$. We have

$$(N_{\mathfrak{P}}, \mathbf{m}_\imath) \models \Lambda_{ft} \quad \textit{iff} \quad (N_{\mathfrak{P}}, \mathbf{m}_\imath) \textit{ has a fair infinite run.}$$

PROOF. As for termination (see Lem. 6.9) we start with a few facts about Λ_{ft}:

(1) For the sake of clarity we used an implication in Λ_{ft}. However the equivalences $A \rightarrow B \equiv \neg A \vee B$ and $\mathbf{c}_a \otimes \mathsf{Parikh}(\sigma_3) \neq 0 \equiv \mathbf{c}_a \otimes \mathsf{Parikh}(\sigma_3) > 0$ shows that the above predicate is indeed a positive boolean combination of atomic predicates, hence Λ_{ft} is indeed a path formula.
(2) Λ_{ft} is polynomial in the size of the PN.
(3) $T_{\mathfrak{P}} \otimes \mathsf{Parikh}(\sigma_1) \leq 0$ ensures that $\sigma_1 = \varepsilon$, hence that $\mu_1 = \mathbf{m}_\imath$. The reason for this is to be able to use the more expressive transition predicate starting right from the initial marking.
(4) $\mu_2 \leq \mu_3$ implies the sequence of transition given by σ_3 can be fired over and over (by monotonicity).
(5) $T^d_{\mathfrak{P}} \otimes \mathsf{Parikh}(\sigma_3) \geq 1$ ensures that $\sigma_3 \in T^*_{\mathfrak{P}} \cdot T^d_{\mathfrak{P}} \cdot T^*_{\mathfrak{P}}$ as for termination.
(6) The last conjunction ensures that each $a \in \Sigma$ is treated fairly. Intuitively, it says that if σ_3 does not dispatch $a \in \Sigma$ (given by $\mathbf{c}_a \otimes \mathsf{Parikh}(\sigma_3) = 0$) then it must hold that (i) a has been posted as many times as it has been dispatched along σ_2 (given by $(\mathbf{p}_a - \mathbf{c}_a) \otimes \mathsf{Parikh}(\sigma_2) = 0$), and (ii) σ_3 is not posting any call to a (given by $\mathbf{p}_a \otimes \mathsf{Parikh}(\sigma_3) = 0$). Together, this means that there is no pending call to a along the execution.

We now turn to the proof.

Only if: Let $\mathbf{m}_{\mu_2}$, $\mathbf{m}_{\mu_3}$, w_2 and w_3 be a valuation of μ_2, μ_3, σ_2 and σ_3 respectively such that Λ_{ft} is satisfied. Note that by Fact (3) we know that since Λ_{ft} holds we have $\sigma_1 = \varepsilon$. Hence we find that $\mathbf{m}_\imath \, [w_2\rangle \, \mathbf{m}_{\mu_2} \, [w_3\rangle \, \mathbf{m}_{\mu_3}$ where $w_3 \in T^*_{\mathfrak{P}} \cdot T^d_{\mathfrak{P}} \cdot T^*_{\mathfrak{P}}$ by Fact (5). Then Fact (4) shows that the run ρ given by $\mathbf{m}_\imath \, [w_2\rangle \, \mathbf{m}_{\mu_2} \, [w_3^\omega\rangle$ is an infinite $\mathfrak{P}$-run of $(N_{\mathfrak{P}}, \mathbf{m}_\imath)$.

Let us now show ρ is also a fair infinite run. We first rewrite ρ as $\mathbf{m}_0 \, [t_0\rangle \, \mathbf{m}_1 \, [t_1\rangle \ldots [t_{i-1}\rangle \, \mathbf{m}_i \, [t_i\rangle \ldots$ where $\mathbf{m}_0 = \mathbf{m}_\imath$, $w_2 = t_0 \ldots t_{i-1}$ and $w_3^\omega = t_i t_{i+1} \ldots$ So we have that $\mathbf{m}_i = \mathbf{m}_{\mu_2}$.

Our final step is to show that ρ matches a fair infinite run in $(N_{\mathfrak{P}}, \mathbf{m}_\imath)$. By hypothesis, Λ_{ft} holds, so each implication holds. Fix $a \in \Sigma$. We examine what the satisfaction of the implication entails.

(a) Assume that the left hand side of the implication does not hold. This means that w_3 fires some $t \in T^{d(a)}_{\mathfrak{P}}$, that is, some $T^{d(a)}_{\mathfrak{P}}$ occurs infinitely often along w_3^ω, and the run is fair w.r.t. a.

(b) If the left hand side of the implication holds, it means that no $T^{d(a)}_{\mathfrak{P}}$ is fired along w_3, hence $t_i \in T^{d(a)}_{\mathfrak{P}}$ holds for finitely many i's in ρ. Because the implication is satisfied, Fact (6) shows that, along w_2, a is posted as many times as it is dispatched.

We conclude from Remark 6.8 and $\mathbf{m}_\imath(a) = 0$, that $\mathbf{m}_i(a) = \mathbf{m}_{\mu_2}(a) = 0$, hence that for every position $j \geq i$ we have $\mathbf{m}_j(a) = 0$, namely $\mathbf{m}_j(a) = 0$ holds for infinitely many j's.

We conclude from the above cases that for every $a \in \Sigma$, we have that if $t_i \in T_{\mathfrak{P}}^{d(a)}$ for finitely many i's then $\mathbf{m}_j(a) = 0$ for infinitely many j's, namely ρ is a fair infinite run and we are done.

If: Let $\rho = \mathbf{m}_0 [t_0\rangle \mathbf{m}_1 [t_1\rangle \ldots [t_{i-1}\rangle \mathbf{m}_i [t_i\rangle \ldots$ where $\mathbf{m}_0 = \mathbf{m}_\imath$ be a infinite fair run of $(N_{\mathfrak{P}}, \mathbf{m}_\imath)$. By definition we find that ρ is an infinite $\mathfrak{P}$-run and that for all $a \in \Sigma$, if $t_i \in T_{\mathfrak{P}}^{d(a)}$ for finitely many i's then $\mathbf{m}_j(a) = 0$ for infinitely many j's. Define S to be the set $\{a \in \Sigma \mid t_i \in T_{\mathfrak{P}}^{d(a)}$ for finitely many i's$\}$. Let m denote a positive integer such that for all $n \geq m$ we have $t_n \in T_{\mathfrak{P}} \setminus \bigcup_{a \in S} T_{\mathfrak{P}}^{d(a)}$. Observe that, because the run is fair, for every $a \in S$ and for all $n \geq m$, we have $\mathbf{m}_n(a) = 0$.

Let us now rewrite ρ as $\mathbf{m}_0 [t_0\rangle \mathbf{m}_1 \ldots \mathbf{m}_m [t_m\rangle \mathbf{m}_{i_0} [w_{i_0}\rangle \mathbf{m}_{i_1} [w_{i_1}\rangle \ldots$ such that $\mathbf{m}_0 = \mathbf{m}_\imath$ and for all $a \in \Sigma \setminus S$ some $T_{\mathfrak{P}}^{d(a)}$ occurs in w_{i_j} for all $j \geq 0$.

Now using Dickson's Lemma [Dickson 1913] over the infinite sequence $\mathbf{m}_{i_0}, \mathbf{m}_{i_1}, \ldots, \mathbf{m}_{i_n}, \ldots$ of markings defined above we find that there exists $\ell > k$ such that $\mathbf{m}_{i_k} \preceq \mathbf{m}_{i_\ell}$.

Define $\sigma_1 = \varepsilon$, $\sigma_2 = t_0 \ldots t_m w_{i_0} \ldots w_{i_{k-1}}$, $\sigma_3 = w_{i_k} \ldots w_{i_{\ell-1}}$, $\mu_1 = \mathbf{m}_\imath$, $\mu_2 = \mathbf{m}_{i_k}$ and $\mu_3 = \mathbf{m}_{i_\ell}$. Clearly $\mu_2 \leq \mu_3$ and $T_{\mathfrak{P}} \otimes \mathsf{Parikh}(\sigma_1) \leq 0$. Also, some transition of $T_{\mathfrak{P}}^d$ occurs in σ_3 by definition of w_{i_j}, hence we find that $T_{\mathfrak{P}}^d \otimes \mathsf{Parikh}(\sigma_3) \geq 1$.

Let $a \in \Sigma$. The implication $\mathbf{c}_a \otimes \mathsf{Parikh}(\sigma_3) = 0 \rightarrow \big((\mathbf{p}_a - \mathbf{c}_a) \otimes \mathsf{Parikh}(\sigma_2) = 0 \wedge \mathbf{p}_a \otimes \mathsf{Parikh}(\sigma_3) = 0\big)$ is divided into two cases.

First, if $a \in S$ then we find that no $T_{\mathfrak{P}}^{d(a)}$ occurs after t_m. In particular no $T_{\mathfrak{P}}^{d(a)}$ occurs in σ_3 and the left hand side of the implication holds. We now show that so does the right hand side. We showed above that $\mathbf{m}_n(a) = 0$ for every $n \geq m$. By Rmk. 6.8, initially $\mathbf{m}_\imath = [\![a_0]\!]$ and a_0 never reappears in the task buffer. So, we find that $(\mathbf{p}_a - \mathbf{c}_a) \otimes \mathsf{Parikh}(\sigma_2) = 0$ holds. Also $\mathbf{p}_a \otimes \mathsf{Parikh}(\sigma_3) = 0$ holds because $\mathbf{m}_n(a) = 0$ for each $n \geq m$ and no $T_{\mathfrak{P}}^{d(a)}$ occurs in σ_3, hence no post of a can occur in σ_3.

Second, if $a \in \Sigma \setminus S$ then we find that some $T_{\mathfrak{P}}^{d(a)}$ occurs along σ_3 by definition of the w_{i_j}'s. Therefore the implication evaluates to true because its left hand side evaluates to false.

This concludes the proof since every conjunction of Λ_{ft} is satisfied. ∎

Remark 6.12. Λ_{ft} is not an increasing path formula because we cannot conclude it implies $\mu_1 \leq \mu_3$. Since $\sigma_1 = \varepsilon$, for $\mu_1 \leq \mu_3$ to hold we must have $\mathbf{m}_\imath \leq \mu_3$. Because of Rmk. 6.8 it is clearly the case that $\mathbf{m}_\imath \not\leq \mu_3$ since $\mathbf{m}_\imath = [\![a_0]\!]$ and a_0 is first dispatched and never posted eventually.

We now show a lower bound on the fair termination problem. Given an initialized Boolean PN $(N = (S, T, F), \mathbf{m}_0)$ and a place $p \in P$, we reduce the problem of checking if there exists a reachable marking with no token in place p (which is recursively equivalent to the reachability problem of a marking [Hack 1976]) iff an asynchronous program constructed from the PN has a fair infinite run. For the sake of clarity, let us index $S = \{p_1, \ldots, p_{|S|}\}$ and assume that p_1 plays the role of place p in the above definition.

Fig. 5 shows an outline of the reduction from the reachability problem for PN to the fair termination problem for asynchronous programs. The reduction is similar to the simulation shown in Fig. 4. In particular, we again define a global state `st`, a procedure `runPN` to fire transitions, and $|S|$ procedures, one for each $p_i \in S$.

The program has three global variables, two booleans `terminate` and `p_1_is_null` and the variable `st` which ranges over a finite subset of $(T \cup \{\varepsilon\}) \times S^*$. The program has $|S| + 3$ procedures: one procedure for each $p_i \in S$, `main`, `guess` and `runPN`. The role of `main` is to initialize the global variables, and to post `runPN` and `guess`. As before, the role of `runPN` is to simulate the transitions of the PN. The role of `guess` is related to checking whether there exists some marking $\mathbf{m} \in [\mathbf{m}_0\rangle$ such that $\mathbf{m}(p_1) = 0$, and is explained below.

The program of Fig. 5 preserves the same invariant as the program of Fig. 4 and is as follows. Whenever the program state is such that `st` coincides with (t, ε) for some $t \in T \cup \{\epsilon\}$ we have that the multiset $\mathbf{m}$ given by the pending instances to handler $p \in S$ is such that $\mathbf{m} \in [\mathbf{m}_0\rangle$ and there exists $w \in T^*$ such that $\mathbf{m}_0\,[w \cdot t\rangle\,\mathbf{m}$.

We now explain the role played by procedure `guess` and the variables `p_1_is_null` and `terminate`. After the dispatch of `main`, `guess` is pending. As long as `guess` does not run the program behaves exactly like the program of Fig. 4. That is, `runPN` selects a transition which, if enabled, fires. Once the firing is complete `runPN` selects a transition, and so on. Now consider the dispatch of `guess` which must eventually occur by fairness. It sets `p_1_is_null` to true. This prevents `runPN` to repost itself, hence to select a transition to fire. So the dispatch of `guess` stops the simulation. Now we will see that if the program has an infinite run then the dispatch of `guess` has to occur in a configuration where (i) `st` $\in (T \cup \{\varepsilon\}) \times \{\varepsilon\}$ and (ii) the marking $\mathbf{m}$ corresponding to the current configuration is such that $\mathbf{m}(p_1) = 0$. For (i), we see that if the precondition of `st` does not equal ε then `terminate` is set to true in `guess`, hence every dispatch that follows does not post, and the program eventually terminates. For (ii), suppose that `guess` runs and that in the current configuration there is a pending instance to p_1. By fairness we find that eventually p_1 has to be dispatched. Since `guess` has set `p_1_is_null` to true we have that the dispatch of p_1 sets `terminate` to true and the program will eventually terminate following the same reasoning as above. So if the program has a fair infinite run then it cannot have any pending instance of handler p_1 after the dispatch of `guess`. The rest of the infinite run looks like this. After the dispatch of `guess` we have that `runPN` is dispatched at most once. Every dispatch of a p_i for $i \in \{2, \ldots, |S|\}$ will simply repost itself since `st` has an empty precondition and the value of `terminate` is false. This way we have a run ρ with infinitely many dispatches and no effect: ρ leaves the program in the exact same configuration that corresponds to a marking $\mathbf{m} \in [\mathbf{m}_0\rangle$ such that $\mathbf{m}(p_1) = 0$. Notice that if current configuration of the program corresponds to the marking $\mathbf{m} = \varnothing$ we have that $\mathbf{m}(p_1) = 0$ but the program terminates. We can avoid this undesirable situation by adding one more place p^g to the PN such that it is marked initially and no transition is connected to p^g.

Let us now turn to the other direction. Suppose there exists $w \in T^*$ such that $\mathbf{m}_\imath\,[w\rangle\,\mathbf{m}$ with $\mathbf{m}(p_1) = 0$. The infinite fair run of the asynchronous program has the following form. The invariant shows that the program can simulate the firing of w and ends up in a configuration with no pending instance to handler p_1 and such that the precondition of `st` is ε. Then `guess` is dispatched followed by a fair infinite sequence of dispatch for p_i where $i \in \{2, \ldots, |S|\}$. Because of `st` the dispatch of p_i has no effect but reposting p_i. So we have a fair infinite run.

This shows that the fair termination problem is polynomial-time equivalent to the Petri net reachability problem.

The reduction also suggests that finding an increasing path formula for fair termination will be non-trivial, since it would imply that Petri net reachability is in EXPSPACE.

PROPOSITION 6.13. *Given an asynchronous program $\mathfrak{P}$, determining the existence of a fair infinite run is polynomial-time equivalent to the reachability problem for* PN. *Hence, it is* EXPSPACE-*hard and can be solved in non-primitive recursive space.*

PROOF. As in Prop. 6.10 our decision procedure relies on reductions to equivalent PN problems. Define $N_{\mathfrak{P}}$ to be the PN given by $N_{\mathfrak{P}}(\mathcal{N})$. Lem. 6.5 shows that $\mathfrak{P}$ has a fair infinite run iff so does $(N_{\mathfrak{P}}, \mathbf{m}_\imath)$. Next, Lem. 6.11 shows that determining whether $(N_{\mathfrak{P}}, \mathbf{m}_\imath)$ has a fair infinite run is equivalent to determining the satisfiability of $(N_{\mathfrak{P}}, \mathbf{m}_\imath) \models \Lambda_{ft}$ where Λ_{ft} is computable in time polynomial in the size of $N_{\mathfrak{P}}$. Finally, Th. 6.7 shows that the

```
*global* st, p_1_is_null, terminate;

main() {
   st= $(\varepsilon,\varepsilon)$;
   p_1_is_null=false;
   terminate=false;

   post runPN();
   post guess();
}

guess() {
   p_1_is_null=true;
   if (st $\notin (T\cup\{\varepsilon\})\times\{\varepsilon\}$){
      terminate=true;
   }
}

runPN() {
   if p_1_is_null==false {
      if (st $\in (T\cup\{\varepsilon\})\times\{\varepsilon\}$) {
         pick $t'\in T$ non det.;
         st=$(t',\hat{I}(t'))$;
      }
      post runPN();
   }
}

Initially: $\mathbf{m}_0 \oplus [\![\texttt{main}]\!]$
```

```
$p_1$() {
   if p_1_is_null==true {
      terminate=true;
   } else {
      if st==$(t, p_1 \cdot w')$ {
         st=$(t, w')$;
         if $w'$==$\varepsilon$ {
            for each $j \in \{1,...,|S|\}$ do {
               if $O(t)(p_j) > 0$ {
                  post $p_j$();
               }
            }
         }
      } else {
         if terminate == false {
            post $p_1$();
         }
      }
   }
}

$p_i$() { // for $i \in \{2,...,|S|\}$
   if st==$(t, p_i \cdot w')$ {
      st=$(t, w')$;
      if $w'$==$\varepsilon$ {
         for each $j \in \{1,...,|S|\}$ do {
            if $O(t)(p_j) > 0$ {
               post $p_j$();
            }
         }
      }
   } else {
      if terminate == false {
         post $p_i$();
      }
   }
}
```

Fig. 5. Let $(N = (S,T,F), \mathbf{m}_0)$ be an initialized Boolean PN such that $p_1 \in S$ and $\forall t \in T$: $|I(t)| > 0$. $\exists \mathbf{m} \in [\mathbf{m}_0\rangle : \mathbf{m}(p_1) = 0$ iff the asynchronous program has a fair infinite execution.

satisfiability of $(N_{\mathfrak{P}}, \mathbf{m}_\iota) \models \Lambda_{ft}$ is reducible to a reachability problem for PN. The best known upper bounds for the reachability problem in PN take non-primitive recursive space. Therefore, we conclude that determining the existence of a fair infinite run in a given $\mathfrak{P}$ can be solved in non-primitive recursive space.

The lower bound is a consequence of (1) the reduction from the reachability problem for PN to the fair termination problem for asynchronous program given at Fig. 5 and (2) the EXPSPACE lower bound for the reachability problem for PN. ∎

6.4. Fair starvation

Recall that the fair starvation property states that there is no pending handler instance that is starved (i.e. never leaves the task buffer) along any fair infinite run.

In order to solve the fair starvation problem, we first define Constr. 2 which modifies Constr. 1 by introducing constructs specific to the starvation problem. In what follows, we assume that the assumption of Rmk. 6.8 holds.

We first give some intuition. A particular pending instance of handler a starves if there exists a fair infinite execution such that from some point in time — call it † — there exists an instance of handler a in the task buffer and it never leaves it. Because the run is fair and there exists at least one instance of handler a in the task buffer, we find that a is going to be dispatched infinitely often. In this case, a particular instance of handler a never leaves the task buffer iff each time a dispatch to a occurs the task buffer contains two or more instances of a.

In order to capture infinite fair runs of an asynchronous program that starves a specific handler a, we modify the Petri net construction as follows. The PN has two parts: the first part simulates the asynchronous program as before, and the second part which also simulated the asynchronous program ensures that an instance of handler a never leaves the task buffer. In order to ensure that condition, the Petri net simply requires that any dispatch of a requires at least *two* pending instances of a rather than just one (as in normal simulation), and the dispatch transition consumes one instance of a and puts back the second instance. The Petri net non-deterministically transitions from the first part of the simulation to the second. The transition point serves as a guess of time point † from which the task buffer always contains at least pending instance of handler a. We now formalize the intuition.

Construction 2 (Petri net for fair starvation). *Let $\mathfrak{P} = (D, \Sigma, G, R, d_0, \mathbf{m}_0)$ be an asynchronous program. Let $\mathcal{N}^\spadesuit = \{N_c^\spadesuit\}_{c\in\mathfrak{C}}$ and $N_c^\spadesuit = (S_c^\spadesuit, T_c^\spadesuit, F_c^\spadesuit)$ be an adequate family of* widgets.

Let $a \in \Sigma$. Define $\mathfrak{C}^a$ to be the set $\mathfrak{C} \cap (D \times \{a\} \times D)$ and $(N^a_{\mathfrak{P}}(\mathcal{N}^\spadesuit), \mathbf{m}'_\iota)$ to be an initialized PN *where (1) $N^a_{\mathfrak{P}}(\mathcal{N}^\spadesuit) = (S_{\mathfrak{P}}, T_{\mathfrak{P}}, F_{\mathfrak{P}})$ is given as follows:*

— $S_{\mathfrak{P}} = D \cup \Sigma \cup \bigcup_{c\in\mathfrak{C}} S_c^\spadesuit \cup \{p_f, p_\infty\}$
— $T_{\mathfrak{P}} = \{t^{f/\infty}\} \cup \bigcup_{c\in\mathfrak{C}} T_c^\spadesuit \cup \{t_c^<\}_{c\in\mathfrak{C}\setminus\mathfrak{C}^a} \cup \{t_c^{<f}, t_c^{<\infty}\}_{c\in\mathfrak{C}^a} \cup \{t_c^>\}_{c\in\mathfrak{C}}$
— $F_{\mathfrak{P}}$ *is given by*

$$\begin{aligned}
F_{\mathfrak{P}}(t^{f/\infty}) &= \langle [\![p_f]\!], [\![p_\infty]\!]\rangle && \\
F_{\mathfrak{P}}(t_c^<) &= \langle [\![d_1, b]\!], [\![(begin, c)]\!]\rangle && c = (d_1, b, d_2) \in \mathfrak{C}\setminus\mathfrak{C}^a \\
F_{\mathfrak{P}}(t_c^{<f}) &= \langle [\![d_1, a, p_f]\!], [\![(begin, c), p_f]\!]\rangle && c = (d_1, a, d_2) \in \mathfrak{C}^a \\
F_{\mathfrak{P}}(t_c^{<\infty}) &= \langle [\![d_1, a, a, p_\infty]\!], [\![(begin, c), a, p_\infty]\!]\rangle && c = (d_1, a, d_2) \in \mathfrak{C}^a \\
F_{\mathfrak{P}}(t) &= F_c^\spadesuit(t) && t \in T_c^\spadesuit \\
F_{\mathfrak{P}}(t_c^>) &= \langle [\![(end, c)]\!], [\![d_2]\!]\rangle && c = (d_1, b, d_2) \in \mathfrak{C}
\end{aligned}$$

and (2) $\mathbf{m}'_\iota = [\![d_0, p_f]\!] \oplus \mathbf{m}_0$.

In an execution of the PN, the occurence of transition $t^{f/\infty}$ corresponds to the Petri net's transition from the first mode of simulation to the second, i.e., the guess of the point † in time from which an instance of a never leaves the task buffer.

In what follows we use the notation $N^a_{\mathfrak{P}}$ to denote an adequate family $N^a_{\mathfrak{P}}(\mathcal{N}^\spadesuit)$.

Lemma 6.14. *Let $\mathfrak{P}$ be an asynchronous program and let $\mathcal{N} = \{N_c\}_{c\in\mathfrak{C}}$ be an adequate family. Define $(N_{\mathfrak{P}}, \mathbf{m}_\iota)$ to be the initialized* PN $(N_{\mathfrak{P}}(\mathcal{N}), \mathbf{m}_\iota)$ *as in Constr. 1 and given*

$a \in \Sigma$ define $(N^a_{\mathfrak{P}}, \mathbf{m}'_\imath)$ to be the initialized PN *$(N^a_{\mathfrak{P}}(\mathcal{N}), \mathbf{m}'_\imath)$ as in Constr. 2 Let the path formula Λ^a_{fs} given by*

$$\exists \mu_1, \mu_2, \mu_3 \exists \sigma_1, \sigma_2, \sigma_3 \colon \mathbf{m}_\imath \left[\sigma_1\right\rangle \mu_1 \left[\sigma_2\right\rangle \mu_2 \left[\sigma_3\right\rangle \mu_3$$

$$T_{\mathfrak{P}} \otimes \mathsf{Parikh}(\sigma_1) \leq 0 \wedge \mu_2 \leq \mu_3 \wedge T^{d(a)}_{\mathfrak{P}} \otimes \mathsf{Parikh}(\sigma_3) \geq 1 \wedge [\![t^{f/\infty}]\!] \otimes \mathsf{Parikh}(\sigma_2) > 0$$

$$\bigwedge\nolimits_{b \in \Sigma} \Big(\mathbf{c}_b \otimes \mathsf{Parikh}(\sigma_3) = 0 \rightarrow \big((\mathbf{p}_b - \mathbf{c}_b) \otimes \mathsf{Parikh}(\sigma_2) = 0 \wedge \mathbf{p}_b \otimes \mathsf{Parikh}(\sigma_3) = 0 \big) \Big)$$

where $\mathbf{c}_b(t) = I(t)(b)$ and $\mathbf{p}_b(t) = O(t)(b)$ for every $t \in T_{\mathfrak{P}}$.
We have

$$(N^a_{\mathfrak{P}}, \mathbf{m}'_\imath) \models \Lambda^a_{fs} \quad \textit{iff} \quad (N_{\mathfrak{P}}, \mathbf{m}_\imath) \textit{ fairly starves } a$$

PROOF. **If**: $(N_{\mathfrak{P}}, \mathbf{m}_\imath)$ fairly starves a implies the existence of a fair infinite run $\rho = \mathbf{m}_0 \left[t_0\right\rangle \mathbf{m}_1 \left[t_1\right\rangle \ldots$ and an index $J \geq 0$ such that for each $j \geq J$ we have $\mathbf{m}_j(a) \geq 1 \wedge (t_j \in T^{d(a)}_{\mathfrak{P}} \rightarrow \mathbf{m}_j(a) \geq 2)$.

To show ρ yields the existence of a run ρ' in $(N^a_{\mathfrak{P}}, \mathbf{m}'_\imath)$ which satisfies Λ^a_{fs}, we first define a set of positions in ρ as we did in Lem. 6.11 for fair termination. Let $b \in \Sigma$, we define m_b such that if every transition in $T^{d(b)}_{\mathfrak{P}}$ occur finitely often then m_b is greater than the last such occurrence; else (some $t \in T^{d(b)}_{\mathfrak{P}}$ occur infinitely often) $m_b = 0$. Define m to be the maximum over $\{J\} \cup \{m_b \mid b \in \Sigma\}$.

Let us now rewrite ρ as the following infinite run

$$\mathbf{m}_0 \left[t_0\right\rangle \mathbf{m}_1 \ldots \mathbf{m}_m \left[t_m\right\rangle \mathbf{m}_{i_0} \left[w_{i_0}\right\rangle \mathbf{m}_{i_1} \left[w_{i_1}\right\rangle \ldots \tag{4}$$

such that for every $b \in \Sigma$ if some $t \in T^{d(b)}_{\mathfrak{P}}$ occurs infinitely often then that t occurs in each w_{i_j} for $j \geq 0$.

Our next step is to associate to ρ a counterpart ρ' in $(N^a_{\mathfrak{P}}, \mathbf{m}'_\imath)$. The run ρ from Eqn. 4 is associated with the trace ρ' given by

$$\mathbf{m}_0 \oplus [\![p_f]\!] \left[t'_0\right\rangle \ldots \mathbf{m}_m \oplus [\![p_f]\!] \left[t'_m\right\rangle \mathbf{m}_{i_0} \oplus [\![p_f]\!] \left[t^{f/\infty}\right\rangle \mathbf{m}_{i_0} \oplus [\![p_\infty]\!] \left[w'_{i_0}\right\rangle \mathbf{m}_{i_1} \oplus [\![p_\infty]\!] \ldots$$

where $\mathbf{m}_\imath = \mathbf{m}_0$, $\mathbf{m}'_\imath = \mathbf{m}_0 \oplus [\![p_f]\!]$. ρ' is such that before the occurrence of $t^{f/\infty}$, if $t_i = t^<_c$ where $c \in \mathfrak{C}^a$ then $t'_i = t^{<f}_c$; else ($c \in \mathfrak{C} \setminus \mathfrak{C}^a$) $t'_i = t_i$. Moreover after the occurrence of $t^{f/\infty}$, if $t_i = t^<_c$ where $c \in \mathfrak{C}^a$ then $t'_i = t^{<\infty}_c$; else $t'_i = t_i$.

Since $m \geq J$ and ρ fairly starves a, we deduce that for every $j \geq m$ we have $\mathbf{m}_j(a) \geq 1$ and $t_j \in T^{d(a)}_{\mathfrak{P}} \rightarrow \mathbf{m}_j(a) \geq 2$. This implies that the transitions of the form $t^{<\infty}_c$ which occur after $t^{f/\infty}$ only, hence after m, are enabled because their counterpart t_c in $N_{\mathfrak{P}}$ is enabled in ρ. Hence we conclude that ρ' is a run of $(N^a_{\mathfrak{P}}, \mathbf{m}'_\imath)$,

Now using Dickson's Lemma [Dickson 1913] over the infinite sequence $\mathbf{m}_{i_0}, \mathbf{m}_{i_1}, \ldots, \mathbf{m}_{i_n}, \ldots$ of markings defined above we find that there exists $\ell > k$ such that $\mathbf{m}_{i_k} \preceq \mathbf{m}_{i_\ell}$.

Finally, let $\sigma_1 = \varepsilon$, $\sigma_2 = t'_0 \ldots t'_m t^{f/\infty} w'_{i_0} \ldots w'_{i_{k-1}}$, $\sigma_3 = w'_{i_k} \ldots w'_{i_{\ell-1}}$, $\mu_1 = \mathbf{m}_\imath$, $\mu_2 = \mathbf{m}_{i_k}$ and $\mu_3 = \mathbf{m}_{i_\ell}$. Clearly $\mu_2 \leq \mu_3$, $T_{\mathfrak{P}} \otimes \mathsf{Parikh}(\sigma_1) \leq 0$ and $[\![t^{f/\infty}]\!] \otimes \mathsf{Parikh}(\sigma_2) > 0$. We conclude from $\mathbf{m}_\ell(a) \geq 1$ for all $\ell \geq m$ and because ρ is fair that some $T^{d(a)}_{\mathfrak{P}}$ must occur infinitely often, hence that it occurs in w'_{i_j} for all $j \geq 0$, and finally that $T^{d(a)}_{\mathfrak{P}} \otimes \mathsf{Parikh}(\sigma_3) \geq 1$ by definition of σ_3. Finally let $b \in \Sigma$, the implication $\mathbf{c}_b \otimes \mathsf{Parikh}(\sigma_3) = 0 \rightarrow \big((\mathbf{p}_b - \mathbf{c}_b) \otimes \mathsf{Parikh}(\sigma_2) = 0 \wedge \mathbf{p}_b \otimes \mathsf{Parikh}(\sigma_3) = 0\big)$ holds using arguments similar to the proof of Lem. 6.11. This concludes this part of the proof since every conjunction of Λ^a_{fs} is satisfied.

Only if: The arguments used here are close to the ones of Lem. 6.11. Let $\mathbf{m}_{\mu_2}$, $\mathbf{m}_{\mu_3}$, w_2 and w_3 be a valuation of μ_2, μ_3, σ_2 and σ_3 respectively such that Λ^a_{fs} is satisfied. $T_{\mathfrak{P}} \otimes \mathsf{Parikh}(\sigma_1) \leq 0$ shows that $\sigma_1 = \varepsilon$. Hence we find that $\mathbf{m}_\imath\,[w_2\rangle\,\mathbf{m}_{\mu_2}\,[w_3\rangle\,\mathbf{m}_{\mu_3}$ where $w_3 \in (T_{\mathfrak{P}})^* \cdot T^{d(a)}_{\mathfrak{P}} \cdot (T_{\mathfrak{P}})^*$ because $T^{d(a)}_{\mathfrak{P}} \oplus \mathsf{Parikh}(\sigma_3) \geq 1$ holds. Then $\mu_2 \leq \mu_3$ shows that the run ρ given by $\mathbf{m}_\imath\,[w_2\rangle\,\mathbf{m}_{\mu_2}\,[w_3^\omega\rangle$ is an infinite run of $(N^a_{\mathfrak{P}}, \mathbf{m}'_\imath)$. $[\![t^{f/\infty}]\!] \oplus \mathsf{Parikh}(\sigma_2) > 0$ where $F_{\mathfrak{P}}(t^{f/\infty}) = \langle [\![p_f]\!], [\![p_\infty]\!]\rangle$ shows that the token initially in p_f moves to p_∞ while w_2 executes.

Our next step is to show that ρ matches a run ρ' in $(N_{\mathfrak{P}}, \mathbf{m}_\imath)$ which fairly starves a. By hypothesis, Λ^a_{fs} holds and so does each implication. Let $b \in \Sigma$, we examine the satisfiability of the implication.

(a) Assume that the left hand side does not hold which means that w_3 fires some $t \in T^{d(b)}_{\mathfrak{P}}$, that is some $T^{d(b)}_{\mathfrak{P}}$ occurs infinitely often along w_3^ω.

(b) If the left hand side of the implication holds we find that no $T^{d(b)}_{\mathfrak{P}}$ is fired along w_3, hence $t_i \in T^{d(b)}_{\mathfrak{P}}$ holds for finitely many i's in ρ. Observe that $b \neq a$ because we showed some $T^{d(a)}_{\mathfrak{P}}$ fires infinitely often in ρ. Because the implication is satisfied, along w_2, b is posted as many times as it is dispatched.

Hence, using similar arguments as those of Lem. 6.11 that we will not repeat here, we find that ρ' is a fair infinite run.

Also since $[\![t^{f/\infty}]\!] \otimes \mathsf{Parikh}(\sigma_2) > 0$ holds, we find that $t^{f/\infty}$ occurs in w_2. This together with the fact that some transition of $T^{d(a)}_{\mathfrak{P}}$ fires infinitely often in w_3^ω implies that each time a token is removed from a (through some $t_c^{<\infty}$ for some c) at least one token remains, hence $\mathbf{m}_i(a) \geq 2$ before a token is removed from a, hence ρ fairly starves a.

Our last step shows that ρ has a counterpart ρ' in $(N_{\mathfrak{P}}, \mathbf{m}_\imath)$ and ρ' is fairly starving a. Let us define ρ' by abstracting away from ρ the places $\{p_f, p_\infty\}$ and the occurrence of $t^{f/\infty}$. Clearly ρ' is an infinite run of $(N_{\mathfrak{P}}, \mathbf{m}_\imath)$ fairly starving a. ∎

Proposition 6.15. *Given an asynchronous program $\mathfrak{P}$, determining the existence of a run that fairly starves some $a \in \Sigma$ is polynomial-time equivalent to* PN *reachability. Fair starvation for asynchronous programs is* EXPSPACE*-hard and can be solved in non-primitive recursive space.*

Proof. As in Prop. 6.13 our decision procedure relies on reductions to equivalent PN problems. Fix $N^a_{\mathfrak{P}}$ to be the PN given by $N^a_{\mathfrak{P}}(\mathcal{N})$. Lem. 6.5 shows that $\mathfrak{P}$ has a run that fairly starves some $a \in \Sigma$ iff so does $(N_{\mathfrak{P}}, \mathbf{m}_\imath)$. Next, Lem. 6.14 shows that determining whether $(N_{\mathfrak{P}}, \mathbf{m}_\imath)$ has a run that fairly starves a given $a \in \Sigma$ is equivalent to determining the satisfiability of $(N^a_{\mathfrak{P}}, \mathbf{m}'_\imath) \models \Lambda^a_{fs}$ where $N^a_{\mathfrak{P}}$ and $\mathbf{m}'_\imath$ are given as in Constr. 2. The reduction from the problem of determining if $\mathfrak{P}$ fairly starves to the problem of checking whether $(N^a_{\mathfrak{P}}, \mathbf{m}'_\imath) \models \Lambda^a_{fs}$ holds can be carried out in polynomial time.

Finally, Th. 6.7 shows that the satisfiability of $(N^a_{\mathfrak{P}}, \mathbf{m}'_\imath) \models \Lambda^a_{fs}$ is reducible to a reachability problem for PN which can be solved using non-primitive recursive space. Therefore, we conclude that determining the existence of a run that fairly starves a for a given $\mathfrak{P}$ and $a \in \Sigma$ can be solved using non-primitive recursive space.

The lower bound is established similarly to the reduction for fair termination (see the asynchronous program $\mathfrak{P}$ of Fig. 5). Let us recall some intuition. After a finite amount of time, $\mathfrak{P}$ guesses that the current state of the task buffer has no pending instance to p_1. If the guess is wrong, $\mathfrak{P}$ will eventually terminate. If the guess is correct then the program will enter into a fair infinite run ρ. We can massage $\mathfrak{P}$ so that ρ is a fair infinite run starving a given handler $p_\spadesuit$. Initially, the task buffer contains one pending instance to a special handler

$p_\spadesuit$. If `terminate` is false, then $p_\spadesuit$ posts itself twice; otherwise it does not do anything. This guarantess that if $\mathfrak{P}$ incorrectly guesses when p_1 is empty, then the number of pending instance to $p_\spadesuit$ will eventually be 0 and $\mathfrak{P}$ will terminate as above. Otherwise, if $\mathfrak{P}$ correctly guesses when p_1 is empty, the number of pending instances of $p_\spadesuit$ will grow unboundedly, therefore preventing some pending $p_\spadesuit$ to ever complete. The EXPSPACE-hardness follows from the corresponding hardness for Petri net reachability. ∎

7. EXTENSIONS: ASYNCHRONOUS PROGRAMS WITH CANCELLATION

The basic model for asynchronous programming considered so far allows *posting* a handler, but not doing any other changes to the task buffer. In practice, APIs or languages for asynchronous programming provide additional capabilities, such as *canceling* one or all pending instances of a given handler, and checking if there are pending instances of a handler. For example, the `node.js` library for Javascript allows canceling all posted handlers of a certain kind. A model with cancellation can also be used to abstractly model asynchronous programs with timeouts associated with handlers, i.e., where a handler should not be called after a specific amount of time has passed since the post.

We now discuss extensions of asynchronous programs that model cancellation of handlers.

7.1. Formal model

We now give a model for asynchronous programming in which the programmer can perform asynchronous calls as before, but in addition can *cancel* pending instances of a given handler. Informally, the command `cancel` $f()$ immediately removes every pending handler instances for f from the task buffer.

To model this extension, we define an extension of asynchronous programs called *asynchronous programs with cancel*. The first step is to associate to every handler f an additional symbol $\bar{f}$, which intuitively represents a cancellation of handler $f \in \Sigma$.

Let Σ be the set of handler names, we denote by $\overline{\Sigma}$ a distinct copy of Σ such that for each $\sigma \in \Sigma$ we have $\bar{\sigma} \in \overline{\Sigma}$. So in the settings with cancel, an asynchronous program defines an extended alphabet $\Gamma = \Sigma_i \cup \Sigma \cup \overline{\Sigma}$ which respectively model the statements, the posting and cancellation of handler instances. We thus have that an asynchronous program with cancel $\mathfrak{P} = (D, \Sigma \cup \overline{\Sigma}, \Sigma_i, G, R, d_0, \mathbf{m}_0)$ consists of a finite set of global states D, an alphabet $\Sigma \cup \overline{\Sigma}$ of for handler calls and cancels, a CFG $G = (\mathcal{X}, \Gamma, \mathcal{P})$, a regular grammar $R = (D, \Gamma, \delta)$, a multiset $\mathbf{m}_0$ of initial pending handler instances, and an initial state $d_0 \in D$.

As with asynchronous programs without cancel, we model the (potentially recursive) code of a handler using a context-free grammar. The code of a handler does two things: first, it can change the global state (through R), and second, it can add and remove pending handler instances from the task buffer (through derivation of a word in $(\Sigma \cup \overline{\Sigma})^*$). In fact, a symbol $\sigma \in \Sigma$ is interpreted as a post of handler σ and a symbol $\bar{\sigma} \in \overline{\Sigma}$ is interpreted as the removal of all pending instances to handler σ.

The set of configurations of $\mathfrak{P}$ is given by $D \times \mathbb{M}[\Sigma]$. Observe it does not differ from asynchronous programs without cancel. The transition relation $\rightarrow \subseteq (D \times \mathbb{M}[\Sigma]) \times (D \times \mathbb{M}[\Sigma])$ is defined as follows: let $\mathbf{m}, \mathbf{m}' \in \mathbb{M}[\Sigma]$, $d, d' \in D$ and $\sigma \in \Sigma$

$$(d, \mathbf{m} \oplus [\![\sigma]\!]) \xrightarrow{\sigma} (d', \mathbf{m}')$$
$$\text{iff}$$
$$\exists w \in \Gamma^* \colon d \underset{R}{\Rightarrow}^* w \cdot d' \wedge X_\sigma \underset{G}{\Rightarrow}^* w \wedge \forall b \in \Sigma \colon \Psi_1(b) \vee \Psi_2(b)$$

where $\Psi_1(b)$ is given by

$$\exists w_1 \in \Gamma^* \exists w_2 \in (\Gamma \setminus \{\bar{b}\})^* \colon w = w_1 \cdot \bar{b} \cdot w_2 \wedge \mathbf{m}'(b) = \mathsf{Parikh}(w_2)(b)$$

and $\Psi_2(b)$ is given by

$$w \in (\Gamma \setminus \{\bar{b}\})^* \wedge \mathbf{m}'(b) = \mathbf{m}(b) + \mathsf{Parikh}(w)(b)$$

The transition relation $\rightarrow$ states that there is a transition from configuration $(d, \mathbf{m} \oplus [\![\sigma]\!])$ to $(d', \mathbf{m}')$ if there is an execution of handler σ that changes the global state from d to d' and operates a sequence of posts and cancel which leaves the task buffer in state $\mathbf{m}'$. A cancel immediately removes every pending instance of the handler being canceled. Note that contrary to the case without cancel the order in which the handler instances are added to and removed from the task buffer does matter.

Finally, let us observe that asynchronous programs with cancel $(D, \Sigma \cup \overline{\Sigma}, \Sigma_i, G, R, d_0, \mathbf{m}_0)$ define a well-structured transition systems $((D \times \mathbb{M}[\Sigma], \sqsubseteq), \rightarrow, c_0)$ where $\sqsubseteq$ is the ordering used for asynchronous programs: $\sqsubseteq \subseteq (D \times \mathbb{M}[\Sigma]) \times (D \times \mathbb{M}[\Sigma])$ is given by $c \sqsubseteq c'$ iff $c.d = c'.d \wedge c.\mathbf{m} \preceq c'.\mathbf{m}$.

The *safety*, *boundedness*, *configuration reachability* and *(fair) non termination* problems for asynchronous programs with cancel are defined as for asynchronous programs (without cancel).

7.2. Construction of an equivalent asynchronous program

Similarly to what we have done for Lem. 4.5 we now give a simpler yet equivalent semantics to asynchronous programs with cancel. To compute the task buffer content after the run ρ of a handler h, the following information is needed: (i) the current content of the task buffer, (ii) the set of cancelled handlers along ρ, and (iii) for each handler $b \in \Sigma$ the number of posts to b that are still pending after ρ, that is the number of posts to b that have not been subsequently neutralized by a cancel to b.

Intuitively, our construction uses the following steps.

First, using the construction of Def. 4.2, we eliminate the need to carry around internal actions Σ_i and the regular grammar R. We get a CFG G^R as a result of this step, and for each context $c = (d, a, d')$, we get the initialized CFG G^c using Def. 4.4. Remember that in G^R and G^c, the alphabet is $\Sigma \cup \overline{\Sigma}$, that is, both posts and cancels are visible.

Now, consider a run of G^c. For each handler a, we want to remember how many posts to a were issued after the *last* call (if any) to cancel a, and also to remember if a cancel to a was issued in the handler along the execution. To update the task buffer, for each handler a for which no cancel was issued, we proceed as before and add all the new posts of a to the buffer. For each handler a for which a cancel was called, we first remove all pending instances of a from the task buffer, and then add all instances of a posted after the last issuance of a cancel. We now give a formal construction that takes any grammar G and computes a new grammar from which we can get these two pieces of information.

Let $G = (\mathcal{X}, \Sigma \cup \overline{\Sigma}, \mathcal{P})$ be a CFG. Define the *reverse* $r(G) = (\mathcal{X}, \Sigma \cup \overline{\Sigma}, \overline{\mathcal{P}})$ as the CFG where $\overline{\mathcal{P}}$ is the least set containing the production $X \rightarrow a$ for each $X \rightarrow a$ in $\mathcal{P}$ and the production $X \rightarrow BA$ for each production $X \rightarrow AB$ in $\mathcal{P}$. It is easy to see that for each $X \in \mathcal{X}$ and each $w \in (\Sigma \cup \overline{\Sigma})^*$, we have $X \underset{G}{\Rightarrow}^* w$ iff $X \underset{r(G)}{\Rightarrow}^* w^r$, where w^r is the reverse of w.

Define the regular grammar $\mathcal{C} = (\mathcal{Y}, \Sigma \cup \overline{\Sigma}, \mathcal{P}_{\mathcal{Y}})$, where $\mathcal{Y} = \{Y_S \mid S \subseteq \Sigma\}$, and $\mathcal{P}_{\mathcal{Y}}$ consists of production rules $Y_S \rightarrow \bar{c} Y_{S \cup \{c\}}$ for each $S \subseteq \Sigma$, and $Y_S \rightarrow c Y_S$ for each $S \subseteq \Sigma$. Intuitively, the regular grammar tracks the set of handlers for which a cancel has been seen. Formally, $Y_\emptyset \underset{\mathcal{C}}{\Rightarrow}^* w Y_S$ implies that for each $\bar{b} \in \overline{\Sigma}$, we have $\mathsf{Parikh}(w)(\bar{b}) > 0$ iff $b \in S$.

Now, we construct a grammar $r(G) \times \mathcal{C} = (\mathcal{Z}, \Sigma, \mathcal{P}_{\mathcal{Z}})$, where $\mathcal{Z} = \{[Y_{S_1} X Y_{S_2}] \mid Y_{S_1}, Y_{S_2} \in \mathcal{Y}, X \in \mathcal{X}\}$, and $\mathcal{P}_{\mathcal{Z}}$ is the least set of rules such that

— if $(X \rightarrow \varepsilon) \in \mathcal{P}$ then $[Y_S X Y_S] \rightarrow \varepsilon$ for all $S \subseteq \Sigma$;
— if $(X \rightarrow c) \in \mathcal{P}$, $c \in \Sigma \cup \overline{\Sigma}$ and $(Y_S \rightarrow c Y_{S'}) \in \mathcal{P}_{\mathcal{Y}}$, then $([Y_S X Y_{S'}] \rightarrow \mathit{Proj}_{\Sigma \setminus S}(c)) \in \mathcal{P}_{\mathcal{Z}}$;

— if $(X \to AB) \in \mathcal{P}$ and then $([Y_{S_0} X Y_{S_2}] \to [Y_{S_0} A Y_{S_1}][Y_{S_1} B Y_{S_2}]) \in \mathcal{P}_{\mathcal{Z}}$ for each $S_0 \subseteq S_1 \subseteq S_2 \subseteq \Sigma$.

Intuitively, a leftmost derivation of the grammar generates derivations of words in $r(G)$ while tracking which symbols from $\overline{\Sigma}$ have been seen. Additionally, it suppresses all symbols in $\overline{\Sigma}$ as well as all symbols $c \in \Sigma$ such that $\bar{c}$ has been seen. Formally, the grammar $r(G) \times \mathcal{C}$ has the following property. The proof is by induction on the derivation of w, similar to Lem. 4.3.

LEMMA 7.1. *For $w \in \Sigma^*$ and $S \subseteq \Sigma$, we have $[Y_\emptyset X Y_S] \underset{r(G)\times\mathcal{C}}{\Rightarrow}^* w$ iff there exists $w' \in (\Sigma \cup \overline{\Sigma})^*$ such that $X \underset{G}{\Rightarrow}^* w'$ and for each $b \in \Sigma$, we have (1) either $w' \in (\Sigma \cup \overline{\Sigma} \setminus \{\bar{b}\})^*$ and $\mathsf{Parikh}(w)(b) = \mathsf{Parikh}(w')(b)$ and $b \notin S$, or (2) there exists $w_1' \in (\Sigma \cup \overline{\Sigma})^*$, $w_2' \in (\Sigma \cup \overline{\Sigma} \setminus \{\bar{b}\})^*$, $w' = w_1' \bar{b} w_2'$, and $\mathsf{Parikh}(w)(b) = \mathsf{Parikh}(w_2')(b)$ and $b \in S$.*

Lem. 7.1, when instantiated with the grammar G^c, provides the following corollary.

COROLLARY 7.2. *Let $\mathfrak{P}$ be an asynchronous program with cancel, and let $\mathbf{m}, \mathbf{m}' \in \mathbb{M}[\Sigma]$. For $c = (d_1, \sigma, d_2) \in \mathfrak{C}$, let G^c be defined as in Def. 4.4 (with Σ replaced by $\Sigma \cup \overline{\Sigma}$). The following statements are equivalent:*

(1) $(d_1, \mathbf{m} \oplus [\![\sigma]\!]) \xrightarrow{\sigma} (d_2, \mathbf{m}')$
(2) $\exists w \in \Sigma^* \colon [Y_\emptyset [d_1 X_\sigma d_2] Y_S] \underset{r(G^c)\times\mathcal{C}}{\Rightarrow}^* w$ *and for all $b \in \Sigma$, we have*

$$\mathbf{m}'(b) = \begin{cases} \mathbf{m}(b) + \mathsf{Parikh}(w)(b) & \text{if } b \notin S \\ \mathsf{Parikh}(w)(b) & \text{if } b \in S \end{cases}$$

PROOF. We have

$$\begin{aligned}
& (d_1, \mathbf{m} \oplus [\![\sigma]\!]) \xrightarrow{\sigma} (d_1, \mathbf{m}') \\
\text{iff } & \exists w \in (\Sigma \cup \overline{\Sigma})^* \colon [d_1 X_\sigma d_2] \underset{G^c}{\Rightarrow}^* w \wedge \forall b \in \Sigma \colon \Psi_1(b) \vee \Psi_2(b) && \text{def. of } \xrightarrow{\sigma} \text{ and } G^c \\
\text{iff } & \left(\begin{array}{c} \exists w \in \Sigma^* \exists S \subseteq \Sigma \colon [Y_\emptyset [d_1 X_\sigma d_2] Y_S] \underset{r(G^c)\times\mathcal{C}}{\Rightarrow}^* w \\ \text{and} \\ \forall b \in \Sigma \colon \mathbf{m}'(b) = \begin{cases} \mathsf{Parikh}(w)(b) & \text{if } b \in S \text{ and} \\ \mathbf{m}(b) + \mathsf{Parikh}(w)(b) & \text{otherwise} \end{cases} \end{array} \right) && \text{Lem. 7.2}
\end{aligned}$$

∎

7.3. PN with reset arcs

Let us now introduce an extension of the PN model which will serve to model the semantics of asynchronous programs with cancel.

Definition 7.3. A Petri net with reset arcs, PN + R for short, is a tuple $(S, T, F = \langle I, O, Z \rangle, \mathbf{m}_0)$ where S, T and F are defined as for PN except that F is extended with a mapping Z such that $Z(t) \subseteq S$ for each $t \in T$. As for PN, $\mathbf{m}_0 \in \mathbb{M}[S]$ defines the initial marking.

Semantics. Given a tuple $(S, T, F, \mathbf{m}_0)$, and a marking $\mathbf{m} \in \mathbb{M}[S]$, a transition $t \in T$ is *enabled* at $\mathbf{m}$, written $\mathbf{m}\,[t\rangle$, if $I(t) \preceq \mathbf{m}$. We write $\mathbf{m}\,[t\rangle\,\mathbf{m}'$ if transition t is enabled at $\mathbf{m}$ and its *firing* yields to marking $\mathbf{m}'$ defined as follows:

(1) Let $\mathbf{m}_1$ be such that $\mathbf{m}_1 \oplus I(t) = \mathbf{m}$.
(2) Let $\mathbf{m}_2$ be such that $\mathbf{m}_2(p) = \begin{cases} 0 & \text{if } p \in Z(t) \\ \mathbf{m}_1(p) & \text{else.} \end{cases}$

(3) $\mathbf{m}'$ is such that $\mathbf{m}' = \mathbf{m}_2 \oplus O(t)$.

The semantics as well as the boundedness and coverability problems naturally follows from their counterpart for PN. Note that if $Z(t) = \emptyset$ for each $t \in T$, then N reduces to a PN.

THEOREM 7.4. *[Dufourd et al. 1998] The coverability problem for* PN + R *is decidable. The boundedness problem and the reachability problem for* PN + R *are both undecidable.*

7.4. PN + R semantics of asynchronous programs with cancel

Definition 7.5. Let $c = (d_1, a, d_2) \in \mathfrak{C}$, and let $r(G^c) \times \mathcal{C} = (\mathcal{Z}, \Sigma, \mathcal{P}_{\mathcal{Z}})$. Define $k = |\mathcal{Z}|$ and the PN + R $N_c^{\sim} = (S_c^{\sim}, T_c^{\sim}, F_c^{\sim})$ such that:

— $S_c^{\sim} = \{(begin, c), (end, c)\} \cup \mathcal{Z} \cup \{(\$, c)\} \cup \Sigma$;
— the sets $T_c^{\sim}$ and $F_c^{\sim}$ are such that $t \in T_c^{\sim}$ iff one of the following holds

$$\begin{aligned}
F_c^{\sim}(t) &= \langle [\![(begin, c)]\!], [\![[Y_\emptyset[d_1 X_a d_2] Y_{S_1}]\!] \oplus [\![(\$, c)^k]\!], S_1\rangle && \text{for each } S_1 \subseteq \Sigma\\
F_c^{\sim}(t) &= \langle [\![X, (\$, c)]\!], [\![Z, Y]\!], \emptyset\rangle && (X \to Z \cdot Y) \in \mathcal{P}_{\mathcal{Z}}\\
F_c^{\sim}(t) &= \langle [\![X]\!], \mathsf{Parikh}(\sigma) \oplus [\![(\$, c)]\!], \emptyset\rangle && (X \to \sigma) \in \mathcal{P}_{\mathcal{Z}}\\
F_c^{\sim}(t) &= \langle [\![(\$, c)^{k+1}]\!], [\![(end, c)]\!], \emptyset\rangle
\end{aligned}$$

Finally, define $\mathcal{N}^{\sim} = \{N_c^{\sim}\}_{c \in \mathfrak{C}}$.

The following lemma is proved similar to Lem. 5.9.

LEMMA 7.6. *Let $\mathfrak{P}$ be an asynchronous program with cancel and let $d, d' \in D$ and $\mathbf{m}, \mathbf{m}' \in \mathbb{M}[\Sigma]$. Define $c = (d, \sigma, d') \in \mathfrak{C}$, we have:*

$$(d, \mathbf{m}) \xrightarrow{\sigma} (d', \mathbf{m}') \textit{ iff } \exists w \in (T_c^{\sim})^* \colon \big([\![(begin, c)]\!] \oplus \mathbf{m}\big)\, [w\rangle_{N_c^{\sim}} \big([\![(end, c)]\!] \oplus \mathbf{m}'\big)\ .$$

CONSTRUCTION 3. *Let $\mathfrak{P} = (D, \Sigma \cup \overline{\Sigma}, \Sigma_i, G, R, d_0, \mathbf{m}_0)$ be an asynchronous program with cancel. Define $(N_{\mathfrak{P}}, \mathbf{m}_\imath)$ to be an initialized* PN + R *where (1) $N_{\mathfrak{P}} = (S_{\mathfrak{P}}, T_{\mathfrak{P}}, F_{\mathfrak{P}})$ is given as follows:*

— *the set $S_{\mathfrak{P}}$ is given by $D \cup \Sigma \cup \bigcup_{c \in \mathfrak{C}} S_c^{\sim}$*
— *the set $T_{\mathfrak{P}}$ of transitions is given by $\bigcup_{c \in \mathfrak{C}} (\{t_c^{<}\} \cup T_c^{\sim} \cup \{t_c^{>}\})$*
— *$F_{\mathfrak{P}}$ is such that for each $c = (d_1, a, d_2) \in \mathfrak{C}$ we have*

$$\begin{aligned}
F_{\mathfrak{P}}(t_c^{<}) &= \langle [\![d_1, a]\!], [\![(begin, c)]\!], \emptyset\rangle\\
F_{\mathfrak{P}}(T_c^{\sim}) &= F_c^{\sim}(T_c^{\sim})\\
F_{\mathfrak{P}}(t_c^{>}) &= \langle [\![(end, c)]\!], [\![d_2]\!], \emptyset\rangle
\end{aligned}$$

and (2) $\mathbf{m}_\imath = [\![d_0]\!] \oplus \mathbf{m}_0$.

From the previous lemma, it follows that.

LEMMA 7.7. *Let $\mathfrak{P}$ be an asynchronous program with cancel and let $(N_{\mathfrak{P}}, \mathbf{m}_\imath)$ be an initialized* PN *as given in Constr. 3. We have $(d, \mathbf{m})$ is reachable in $\mathfrak{P}$ iff $[\![d]\!] \oplus \mathbf{m}$ is reachable in $N_{\mathfrak{P}}$ from $\mathbf{m}_\imath$.*

7.5. Model checking

We now summarize the status of model checking asynchronous programs with cancel.

THEOREM 7.8.

(*1*) *The safety (global state reachability) problem for asynchronous programs with cancel is decidable.*

```
global st = (ε, ε);

runPN () {
  if st ∈ (T ∪ {ε}) × {ε} {
    pick t ∈ T non det.;
    st = (t, Î(t));
  }
  post runPN();
}

Initially: m_ι ⊕ ⟦runPN⟧
```

```
p′() { // for p′ ∈ S
  if st == (t, p′ · w) {
    st = (t, w);
    if w == ε {
      for each p ∈ S do {
        if p ∈ Z(t) {
          cancel p();
        }
        if O(t)(p) > 0 {
          post p();
        }
      }
    }
  } else {
    post p′();
  }
}
```

Fig. 6. Let $N = (S, T, F = \langle I, O, Z\rangle, \mathbf{m}_0)$ be an initialized $\mathsf{PN+R}$ such that $\forall t \in T\colon |I(t)| > 0$. N is unbounded (that is $[\mathbf{m}_\imath\rangle$ is infinite) iff the asynchronous program is unbounded.

(*2*) *The configuration reachability problem for asynchronous programs with cancel is undecidable.*
(*3*) *The boundedness problem for asynchronous programs with cancel is undecidable.*

Proof. Part (1) of Theorem. 7.8 follows from Thm. 7.4 and Lem. 7.7.
To show configuration reachability and boundedness are undecidable, we use a reduction similar to what we have previously seen at Fig. 4 for PN. We reduce the reachability and boundedness problems for $\mathsf{PN+R}$ to the configuration reachability and boundedness problems for asynchronous programs with cancel, respectively. The reachability and the boundedness problems for $\mathsf{PN+R}$ are both undecidable [Dufourd et al. 1998]. Our reduction from the boundedness of $\mathsf{PN+R}$ is given at Fig. 6. We omit the details, which are similar to the construction for PN. The reduction for configuration reachability is similar. ∎

We now show undecidability results when it comes to determine properties related to infinite runs. Our proofs use undecidability results for counter machines, which we now introduce.

Definition 7.9. A *n-counter machine C* (nCM for short), is a tuple $\langle \{c_i\}_{1\le i\le n}, L, \mathsf{Instr}\rangle$ where:

— each c_i takes its values in $\mathbb{N}$;
— $L = \{l_1, \ldots, l_u\}$ is a finite non-empty set of locations;
— Instr is a function that labels each location $l \in L$ with an instruction that has one of the following forms:
  — l: $c_j := c_j + 1$; `goto` l' where $1 \le j \le n$ and $l' \in L$, this is called an *increment*, and we define $\mathsf{TypeInst}(l) = \langle \mathsf{inc}_j, l'\rangle$;
  — l: $c_j := c_j - 1$; `goto` l' where $1 \le j \le n$ and $l' \in L$, this is called a *decrement*, and we define $\mathsf{TypeInst}(l) = \langle \mathsf{dec}_j, l'\rangle$;
  — l: `if` $c_j = 0$ `then goto` l' `else goto` l'' where $1 \le j \le n$ and $l', l'' \in L$, this is called a *zero-test*, and we define $\mathsf{TypeInst}(l) = \langle \mathsf{zerotest}_j, l', l''\rangle$;

We define 2CM and 3CM as the class of 2-counter and 3-counter machines, respectively.

```
global loc=$l_1$;

main() {
    while(*)
        post $I$();
}

$c_j$() { // for $j \in \{1,2,3\}$
    if TypeInst(loc) == $\langle \mathsf{dec}_j, l' \rangle$ {
        loc=$l'$;
        post $I$();
    } else if
        TypeInst(loc) == $\langle \mathsf{zerotest}_j, l', l'' \rangle$ {
        loc=$l''$;
        post $c_j$();
    } else
        loc=$\perp$;
}

Initially: $[\![$main$]\!]$

$I$() {
    if TypeInst(loc) == $\langle \mathsf{inc}_j, l' \rangle$ {
        loc=$l'$;
        post $c_j$();
    } else if
        TypeInst(loc) == $\langle \mathsf{zerotest}_j, l', l'' \rangle$ {
        loc=$l'$;
        cancel $c_j$();
        post $I$();
    } else
        loc=$\perp$;
}
```

Fig. 7. Let $C' = (\{c_1, c_2, c_3\}, L, \mathsf{Instr})$ be the 3CM defined upon a reachability problem instance for 2CM, the above asynchronous program with cancel has an infinite computation iff C' has an infinite bounded computation. In the above program, whenever loc equals $\perp$ then every conditional fails.

Semantics. The instructions have their usual obvious semantics, in particular, decrement can only be done if the value of the counter is strictly greater than zero.

A *configuration* of an nCM $\langle \{c_1, \ldots, c_n\}, L, \mathsf{Instr} \rangle$ is a tuple $\langle loc, v_1, v_2, \ldots, v_n \rangle$ where $loc \in L$ is the value of the program counter and, $v_1, \ldots, v_n$ are positive integers that gives the values of counters $c_1, \ldots, c_n$, respectively. We adopt the convention that every nCM is such that L contains a special location l_1 called the *initial location*.

A *computation* γ of an nCM is a finite sequence of configurations $\langle loc^1, v_1^1, \ldots, v_n^1 \rangle, \langle loc^2, v_1^2, \ldots, v_n^2 \rangle, \ldots, \langle loc^r, v_1^r, \ldots, v_n^r \rangle$ such that the following conditions hold. (i) "Initialization": $loc^1 = l_1$ and for each $i \in \{1, \ldots, n\}$, we have $v_i^1 = 0$. That is, a computation starts in l_1 and all counters are initialized to 0. (ii) "Consecution": for each $i \in \mathbb{N}$ such that $1 \le i \le |\gamma|$ we have that $\langle loc^{i+1}, v_1^{i+1}, \ldots, v_n^{i+1} \rangle$ is the configuration obtained from $\langle loc^i, v_1^i, \ldots, v_n^i \rangle$ by applying instruction $\mathsf{Instr}(loc_i)$. A configuration c is *reachable* if there exists a finite computation γ whose last configuration c. A location $\ell \in L$ is reachable if there exists a reachable configuration $\langle \ell, v_1, \ldots, v_n \rangle$ for some $v_1, \ldots, v_n \in \mathbb{N}$.

Given an nCM C and $F \subseteq L$, the *reachability problem* asks if some $\ell \in F$ is reachable. If so, we say C *reaches* F.

THEOREM 7.10. *[Minsky 1967] The reachability problem for nCM is undecidable for $n \ge 2$.*

THEOREM 7.11. *Determining if an asynchronous program with cancel has an infinite run is undecidable.*

PROOF. Our proof follows the proof of [Esparza et al. 1999] which reduces the termination of broadcast protocols to the reachability problem for nCM.

We first start with some additional notions on counter machines. A configuration $\langle loc, v_1, v_2, \ldots, v_n \rangle$ of an nCM is *k-bounded* if $\sum_{i=1}^{n} v_i \le k$. A computation γ is k-bounded if all its configurations are k-bounded, and *bounded* if it is k-bounded for some positive integer k.

Consider an instance of the reachability problem of a 2CM given by $C = \langle \{c_1, c_2\}, L, \mathsf{Instr}\rangle$ and $F \subseteq L$. Without loss of generality, we assume that l_1 does not have an "incoming edge" in C. Define C' to be a 3CM that behaves as follows. C' simulates C on counters c_1 and c_2 and increases c_3 by 1 after each step of simulation. If C reaches some location in F, then C' goes back to its initial configuration $\langle l_1, 0, 0, 0\rangle$. We make the following two observations about C':

— C' has an infinite bounded computation iff C reaches F. Because after each step C' increments counter c_3, the only infinite bounded computation of C', if any, corresponds to the infinite iteration of a run of C that reaches F.
— In every infinite bounded computation of C', the initial configuration $\langle l_1, 0, 0, 0\rangle$ occurs infinitely often.

We can simulate $C' = \langle \{c_1, c_2, c_3\}, L, \mathsf{Instr}\rangle$ in a weak sense by using an asynchronous program with cancel $\mathfrak{P}$ given at Fig. 7. The simulation uses procedures c_1, c_2, and c_3 to simulate decrements of counters as well as zero-tests where the "else branch" is taken. It additionally uses a procedure I to simulate increments to variables, as well as the "then branch" for a zero-test. The location $\perp$ is a special "halt" location with no instructions (so the simulation eventually terminates once the location is set to $\perp$).

We call a simulation *faithful* if whenever the then-branch of a zero test is executed, there are no pending instances of handler c_j (and thus the cancel is a no-op). A simulation may not be faithful because the dispatch of handler I amounts to guess that the then-branch is taken, and cancels any pending instances of handler c_j. If there were pending instances of c_j, this guess is wrong, but these instances get removed anyway by the cancel. In that case we say that $\mathfrak{P}$ *cheats*.

We prove that if C reaches F, then by the above observation $\mathfrak{P}$ has an infinite run. If C reaches F, then C' has an infinite bounded computation γ, which iterates infinitely often a computation of C that reaches F. By definition of bounded computation, there exists $b \geq 0$ such that γ is b-bounded. Let ρ be a run of $\mathfrak{P}$ that initially executes "`post` $I()$" b times and then faithfully simulates γ. Since this is a faithful simulation, each time a "`cancel` c_i" (for $i \in \{1, 2\}$) statement is executed, there is no pending instance of handler c_i to remove. Since ρ can simulate every step of γ, it is infinite.

We now prove that if $\mathfrak{P}$ has an infinite run, then C reaches F. Here, we have to take into account possible cheating in the simulation. Let ρ be an infinite run of $\mathfrak{P}$. Notice that in this run, the variable `loc` can never be set to $\perp$ (since any run of $\mathfrak{P}$ where `loc` $= \perp$ eventually terminates. Suppose in this run, the statement "`post`$I()$" was executed b times in `main`. After the execution of `main`, the number of pending handlers is always at most b, and thus the execution encodes a b-bounded run of the counter machine. Moreover, the number of pending handlers only decreases if there is a cheat (that is, some pending handler c_j is canceled). Thus, the infinite execution ρ can have only finitely many cheats. Take a suffix of ρ containing no cheats. It corresponds to a bounded infinite simulation γ of C'. Now recall that every infinite bounded run of C' contains infinitely many initial configurations. So some suffix γ' of γ is an infinite computation of C'. Thus, C reaches F. ∎

It can also be shown that the fair non termination and fair starvation problem for asynchronous program with cancel are also undecidable. Let us sketch the main intuitions here. For the fair non termination problem, it suffices to modify the 3CM C' as follows. In the initial configuration $\langle l_1, 0, 0, 0\rangle$, instead of simulating C, C' first increments and then decrement each counter c_i for $i \in \{1, 2, 3\}$. Then C' simulates C as given above. Observe that this modification preserves the correctness of the above proof. Let us now turn to the asynchronous program with cancel $\mathfrak{P}$ simulating this updated C'. We conclude from the above modification that if $\mathfrak{P}$ simulates the bounded infinite run of C' faithfully then the run is fair because a faithful simulation requires the dispatch of every handler (i.e. $c_1(), c_2(), c_3()$ and $I()$). Therefore the infinite run is fair.

For the fair starvation problem, let k denote the value such that there is a k-bounded infinite computation in C'. We will now show there exists a fair infinite run that starves handler $I()$. In this run, `main` posts at least $k+2$ instances of handler $I()$. This will ensure that after executing the `main` procedure there are at least 2 pending instances of $I()$ along the fair infinite run and we are done.

THEOREM 7.12. *Determining if an asynchronous program with cancel $\mathfrak{P}$ has a fair infinite run or determining if $\mathfrak{P}$ fairly starves some $a \in \Sigma$ is undecidable.*

7.6. Asynchronous Programs with Cancel and Test

Our final results investigate the decidability of natural extensions to asynchronous programs with cancel, where additionally, the program can test for the absence of pending instances to a particular handler p. We model an additional instruction `assertnopending` $p()$ that succeeds if there is no pending instance of p. Here, we show that safety verification becomes undecidable as well. Our proof reduces the coverability problem for an extension of $\mathsf{PN}+\mathsf{R}$ where we additionally allow one transition whose enabling condition is augmented by requiring the absence of token in a given place. We call this transition a transition with *inhibitor arc*.

We first introduce an extension of $\mathsf{PN}+\mathsf{R}$ with one transition with inhibitor arc.

Definition 7.13. A reset net with one inhibitor arc N ($\mathsf{PN}+\mathsf{R}+!$ for short) is a tuple $\langle S, T, F = \langle I, O, Z\rangle, !, \mathbf{m}_0\rangle$ where $\langle S, T, F = \langle I, O, Z\rangle, \mathbf{m}_0\rangle$ is a $\mathsf{PN}+\mathsf{R}$ and $! \in (T \times S)$.

We know define the semantics for $\mathsf{PN}+\mathsf{R}+!$ by extending the one for $\mathsf{PN}+\mathsf{R}$.

Semantics. Given a $\mathsf{PN}+\mathsf{R}+!$ $N = \langle S, T, F, !, \mathbf{m}_0\rangle$, and a marking $\mathbf{m}$ of N, a transition $t \in T$ is *enabled* at $\mathbf{m}$, written $\mathbf{m}\left[t\right\rangle$, if (1) $I(t) \preceq \mathbf{m}$ and (2) $! = (t,p)$ implies $\mathbf{m}(p) = 0$. We write $\mathbf{m}\left[t\right\rangle \mathbf{m}'$ if transition t is enabled at $\mathbf{m}$ and its *firing* yields to marking $\mathbf{m}'$ defined as in Sect. 7.3.

The coverability problem for $\mathsf{PN}+\mathsf{R}+!$ naturally follows from the definition for $\mathsf{PN}+\mathsf{R}$. The following result, due to Laurent Van Begin, shows that coverability is undecidable in this model.

THEOREM 7.14. *The coverability problem for $\mathsf{PN}+\mathsf{R}+!$ is undecidable.*

PROOF. Our proof reduces the reachability problem for 2CM to the coverability problem for $\mathsf{PN}+\mathsf{R}+!$. We consider here a particular case of the reachability problem which asks whether a particular control location, e.g. l_f, with null counter values is reachable (Is $\langle l_f, 0, 0\rangle$ reachable?). This problem is known to be undecidable.

Fix an instance ($C = \langle\{c_1, c_2\}, L, \mathsf{Instr}\rangle, l_f$) of that problem where C is the 2CM and $l_f \in L$ is a control location of C.

We define the $\mathsf{PN}+\mathsf{R}+!$ $N = (S, T, F = \langle I, O, Z\rangle, !, \mathbf{m}_0)$ such that N simulates C in a weak sense we define below.

— $S = L \cup \{c_1, c_2\} \cup \{\mathit{cnt}, \mathit{2cover}\}$
— T and F are such that $t \in T$ iff one of the following holds:
  — $F(t) = \langle [\![l]\!], [\![c_j, l', \mathit{cnt}]\!], \emptyset\rangle$ where $\mathsf{TypeInst}(l) = \langle \mathsf{inc}_j, l'\rangle$;
  — $F(t) = \langle [\![c_j, l, \mathit{cnt}]\!], [\![l']\!], \emptyset\rangle$ where $\mathsf{TypeInst}(l) = \langle \mathsf{dec}_j, l'\rangle$;
  — $F(t) = \langle [\![l]\!], [\![l']\!], \{c_j\}\rangle$ where $\mathsf{TypeInst}(l) = \langle \mathsf{zerotest}_j, l', l''\rangle$;
  — $F(t) = \langle [\![l, c_j]\!], [\![l'', c_j]\!], \emptyset\rangle$ where $\mathsf{TypeInst}(l) = \langle \mathsf{zerotest}_j, l', l''\rangle$;
  — $F(t) = \langle [\![l_f]\!], [\![\mathit{2cover}]\!], \emptyset\rangle$.
— $! = (t, \mathit{cnt})$ such that $F(t) = \langle [\![l_f]\!], [\![\mathit{2cover}]\!], \emptyset\rangle$ namely a token is produced in *2cover* provided l_f contains some token and *cnt* does not;

— $\mathbf{m}_0 = [\![l_1]\!]$.

Define $(N, [\![\mathit{2cover}]\!])$ to be an instance of the coverability problem for PN +R+!. The rest of the proof shows that $[\![\mathit{2cover}]\!]$ is coverable iff C reaches the configuration $\langle l_f, 0, 0\rangle$.

Intuitively, the following property is maintained by N: as long as N simulates faithfully C the place cnt holds as many tokens as the sum of tokens in c_1 and c_2; once N does not faithfully simulate C we have that cnt holds strictly more tokens than c_1 and c_2.

The definition of $\mathbf{m}_0$ shows that initially $\mathbf{m}_0(\mathit{cnt}) = \mathbf{m}_0(c_1) + \mathbf{m}_0(c_2) = 0$, that is cnt holds as many tokens as c_1 and c_2. Moreover the definition of N shows that whenever a transition which resets c_j $j = 1, 2$ is fired and removes at least one token from c_j then cnt holds more tokens than c_1 and c_2. This will reflect that N incorrectly simulated C. In fact, if a transition resets c_j and removes at least one token from it then we find that some zerotest instruction was inaccurately simulated because the "then" branch was taken while the counter tested for 0 contained a token. Therefore a token was removed from c_j. Observe that once a reset transition of N has removed a token from c_1 or c_2 then from this point on cnt holds strictly more than the sum of tokens in c_1 and c_2.

Therefore, given a sequence of transitions $w \in T^*$, such that $\mathbf{m}_0\,[w\rangle\,\mathbf{m}$, we have $\mathbf{m}(\mathit{cnt}) = \mathbf{m}(c_1) + \mathbf{m}(c_2)$ iff each occurrence of a transition t such that $Z(t) = \{c_j\}$ along w removes no token from c_j. We thus interpret w as an accurate simulation of C.

Now suppose $\langle l_f, 0, 0\rangle$ is reachable in C through some computation γ. By accurately simulating γ in N we find that a marking with some tokens in l_f and no tokens elsewhere is reachable, hence that $[\![\mathit{2cover}]\!]$ is coverable. The other direction is proven by contradiction.

Assume that $\langle l_f, 0, 0\rangle$ is not reachable in C but $[\![\mathit{2cover}]\!]$ is coverable in N. Hence there exists $w \in T^*$ such that $\mathbf{m}_0\,[w\rangle\,\mathbf{m}$, $\mathbf{m}(l_f) \geq 1$ and $\mathbf{m}(\mathit{cnt}) = 0$. It follows that $\mathbf{m}(c_1) + \mathbf{m}(c_2) = 0 = \mathbf{m}(\mathit{cnt})$. But we showed above that in this case w is a precise simulation of a computation in C, hence a contradiction.

In fact, whenever N does not faithfully simulate C, every marking $\mathbf{m}$ reachable from this point is such that $\mathbf{m}(c_1) + \mathbf{m}(c_2) < \mathbf{m}(\mathit{cnt})$, hence that $\mathbf{m}(\mathit{cnt}) > 0$ since the minimum value for $\mathbf{m}(c_1) + \mathbf{m}(c_2)$ is 0. This means cnt can never be emptied, hence that the enabling condition expressed by ! can never be satisfied, and finally that $[\![\mathit{2cover}]\!]$ can never be marked. ∎

We finally obtain the following negative result for the safety problem of asynchronous programs with cancel and a test for the absence of pending instances to a particular hander p. Recall that boundedness, configuration reachability, and liveness properties are undecidable already for the more restricted class without testing for the absence of a handler.

LEMMA 7.15. *The safety problem for asynchronous programs with cancel and test for absence of pending instances is undecidable.*

PROOF. We reduce from coverability problem for PN +R+! which has been shown to be undecidable at Thm. 7.14. The reduction is similar to the one given at Fig. 6 only that `runPN` has to be slightly modified in order take the augmented enabling condition of PN +R+! into account. As in Sect. 6.1 we assume w.l.o.g. that instead of asking if some given marking $\mathbf{m}$ is such that $\uparrow\mathbf{m} \in [\mathbf{m}_\imath\rangle_N$ where N is a PN +R+!, we equivalently asks if there exists a marking $\mathbf{m} \in [\mathbf{m}_\imath\rangle_N$ for a PN +R+! N such that $\mathbf{m}$ enables some given transition t_c, namely $\mathbf{m}\,[t_c\rangle$. We thus obtain that there exists $\mathbf{m} \in [\mathbf{m}_\imath\rangle$ such that $\mathbf{m}\,[t_f\rangle$ iff $\mathtt{st} = (t_c, \varepsilon)$ is reachable in $\mathfrak{P}$. The resulting code for `runPN` is given at Fig. 8. ∎

8. CONCLUSION

Asynchronous programming is ubiquitous in computing systems. The results in this paper provide a fairly complete theoretical characterization of the safety and liveness verification problems for this model. Initial implementations for safety verification of asynchronous programs were reported in [Jhala and Majumdar 2007]. One interesting direction will be to

```
$\mathit{global}$ st $= (\varepsilon, \varepsilon)$;

runPN () {
  if st $\in (T \cup \{\varepsilon\}) \times \{\varepsilon\}$ {
    pick $t \in T$ non det.;
    st = $(t, \hat{I}(t))$;
    if $! = (t, p)$ {
      assertnopending $p()$;
    }
  }
  post runPN();
}

Initially: $\mathbf{m}_\imath \oplus [\![\texttt{runPN}]\!]$
```

Fig. 8. Let $N = (S, T, F = \langle I, O, Z \rangle, !, \mathbf{m}_0)$ be an initialized PN +R+! such that $\forall t \in T\colon |I(t)| > 0$. N enables some given t_f iff $\texttt{st} = (t_c, \varepsilon)$ is reachable in $\mathfrak{P}$.

apply tools for coverability analysis of PN to this problem, using the reduction outlined in this paper. For liveness verification, the PN reachability lower bound is somewhat disappointing. It will be interesting to see what heuristic approximations can work well in practice.

Since our initial work [Ganty et al. 2009], there have been several other related results. The problem of whether an asynchronous program is simulated by or simulates a finite state machine is shown to be decidable in [Chadha and Viswanathan 2009]. The authors also show how to solve the control state maintainability problem which asks whether an asynchronous program has an infinite (or terminating) run such that each of its state belongs to a given upward closed set of configurations. Safety verification was shown to be decidable for a model augmenting asynchronous programs with priorities (and letting higher priority handlers interrupt lower priority ones) in [Atig et al. 2008]. Safety verification was shown to be undecidable for a natural extension of asynchronous programs with timing [Ganty and Majumdar 2009]. A model of asynchronous programs in which emptiness of a fixed subset of handlers can be checked has been proposed in the Linux kernel (see http://lwn.net/Articles/314808/). For this model, safety and boundedness are decidable. This follows from recent results in [Abdulla and Mayr 2009] (for safety) and [Finkel and Sangnier 2010] (for boundedness). As far as we known, the decidability of termination is still open. When extended with cancellation of handlers, safety verification becomes undecidable, using Thm. 7.14.

A. APPENDIX: CONSTRUCTION OF THE GRAMMAR G^R

Definition A.1. Given a CFG $G = (\mathcal{X}, \Sigma \cup \Sigma_i, \mathcal{P})$ and a regular grammar $R = (D, \Sigma \cup \Sigma_i, \delta)$, define $G^r = (\mathcal{X}^r, \Sigma \cup \Sigma_i, \mathcal{P}^r)$ where $\mathcal{X}^r = \{[dXd'] \mid X \in \mathcal{X}, d, d' \in D\}$, and $\mathcal{P}^r$ is the least set such that each of the following holds:

(1) if $(X \to \varepsilon) \in \mathcal{P}$ and $d \in D$ then $([dXd] \to \varepsilon) \in \mathcal{P}^r$.
(2) if $(X \to a) \in \mathcal{P}$ and $(d \to a \cdot d') \in \delta$ then $([dXd'] \to a) \in \mathcal{P}^r$.
(3) if $[d_0Ad_1], [d_1Bd_2] \in \mathcal{X}^r$ and $(X \to AB) \in \mathcal{P}$ then $([d_0Xd_2] \to [d_0Ad_1][d_1Bd_2]) \in \mathcal{P}^r$.

LEMMA A.2. *Let $\sigma \in (\Sigma \cup \Sigma_i \cup \{\varepsilon\})$, $d, d' \in D$ and $X \in \mathcal{X}$.*

$$\textit{if} \quad d \underset{R}{\Rightarrow}^* \sigma \cdot d' \wedge X \underset{G}{\Rightarrow}^* \sigma \quad \textit{then} \quad [dXd'] \underset{G^r}{\Rightarrow}^* \sigma \enspace .$$

PROOF. The proof is by induction on the length of the derivation $X \underset{G}{\Rightarrow}^* \sigma$.

$\mathbf{i = 1}$. Then $X \Rightarrow \sigma$. Moreover $d \underset{R}{\Rightarrow}^* \sigma \cdot d'$ shows that either $d \underset{R}{\Rightarrow} \sigma \cdot d'$ or $d = d'$ and $\sigma = \varepsilon$ (i.e. $d \underset{R}{\Rightarrow}^0 \sigma \cdot d'$).

In any case we have that $([dXd'] \to \sigma) \in \mathcal{P}^r$ by definition of G^r, hence we find that $[dXd'] \underset{G^r}{\Rightarrow} \sigma$.

$\mathbf{i > 1}$. We have $X \Rightarrow^i \sigma$. Then we necessarily have $X \Rightarrow ZY \Rightarrow^j w_1 Y \Rightarrow^k w_1 w_2 = \sigma$ where $j + k = i - 1$. Two cases may arise: $w_1 = \sigma$ and $w_2 = \varepsilon$ or $w_1 = \varepsilon$ and $w_2 = \sigma$. Let us prove the case $w_1 = \sigma$ and $w_2 = \varepsilon$. The other one is treated similarly.

We have $Y \Rightarrow^k w_2 (= \varepsilon)$ with $k \leq i - 1$. Moreover for each $d \in D$, we have $d \underset{R}{\Rightarrow}^* w_2 \cdot d$. Next, because $k \leq i - 1$ we can apply the induction hypothesis to conclude that $[dYd] \underset{G^r}{\Rightarrow}^* \varepsilon$ for all $d \in D$.

Also $Z \Rightarrow^j w_1 (= \sigma)$ with $j \leq i - 1$. Moreover $d \underset{R}{\Rightarrow}^* \sigma \cdot d'$ shows by induction that $[dZd'] \underset{G^r}{\Rightarrow}^* \sigma$. Finally, $(X \to ZY) \in \mathcal{P}$ and the definition of G^r shows that $([dXd'] \to [dYd][dZd']) \in \mathcal{P}^r$, hence that $[dXd'] \underset{G^r}{\Rightarrow}^* \sigma$ and we are done. ∎

LEMMA A.3. *Let $X_0 \underset{G}{\Rightarrow}^* w$ where $|w| > 1$. There exist $X, X_1, X_2 \in \mathcal{X}$ and $w_1, w_2 \in (\Sigma \cup \Sigma_i)^* \setminus \{\varepsilon\}$ such that each of the following holds:*

— $X \Rightarrow X_1 X_2 \Rightarrow^* w_1 X_2 \Rightarrow^* w_1 w_2 = w$
— $X_0 \Rightarrow^* X$

PROOF. The proof is by induction of the length of the derivation $X_0 \underset{G}{\Rightarrow}^* w$. Since $|w| > 1$, the smallest derivation for w needs no less than three steps.

$\mathbf{i = 3}$. Then $X_0 \underset{G}{\Rightarrow}^3 w$ is necessarily of the form $X_0 \Rightarrow X_1 X_2 \Rightarrow \sigma_1 X_2 \Rightarrow \sigma_1 \sigma_2 = w$ where $\sigma_1 \neq \varepsilon \neq \sigma_2$. By choosing $X = X_0$ we have $X_0 \Rightarrow^* X$ which concludes the proof of this case.

$\mathbf{i > 3}$. Then $X_0 \underset{G}{\Rightarrow}^i w$ is necessarily of the form $X_0 \Rightarrow X_1 X_2 \Rightarrow^j w_1 X_2 \Rightarrow^k w_1 w_2 = w$ with $j + k = i - 1$.

Three cases may arise:

$w_1 = \varepsilon$ and $w_2 = w$. Therefore we have that $X_1 \Rightarrow^* w_1 = \varepsilon$ and $X_2 \Rightarrow^k w_2 = w$ with $k \leq i - 1$. The induction hypothesis shows that there exists X', X'_1, X'_2 and $w'_1, w'_2 \in (\Sigma \cup \Sigma_i)^* \setminus \{\varepsilon\}$ such that $X' \Rightarrow X'_1 X'_2 \Rightarrow^* w'_1 X'_2 \Rightarrow^* w'_1 w'_2 (= w_2 = w)$ and $X_2 \Rightarrow^* X'$. Finally we find that $X_0 \Rightarrow^* X' \Rightarrow X'_1 X'_2 \Rightarrow^* w'_1 X'_2 \Rightarrow^* w'_1 w'_2 = w$ and we are done.
$w_1 = \varepsilon$ and $w_2 = w$. This case is similar to the previous one.
$w_1 \neq \varepsilon$ and $w_2 \neq \varepsilon$. By choosing $X = X_0$ we have $X_0 \Rightarrow^* X$ which concludes the proof of this case.

∎

LEMMA A.4. *If $X_0 \underset{G}{\Rightarrow}^* X \underset{G}{\Rightarrow}^* w$ and $[dXd'] \underset{G^r}{\Rightarrow}^* w$ then $[dX_0 d'] \underset{G^r}{\Rightarrow}^* w$.*

PROOF. The proof is by induction on the length of the derivation $X_0 \underset{G}{\Rightarrow}^* X$

$\mathbf{i = 0}$. So we have $X_0 = X$ and the result trivially holds.

$\mathbf{i > 0}$. We have $X_0 \Rightarrow^i X \Rightarrow^* w$. It follows that $X_0 \Rightarrow YZ \Rightarrow^{i-1} X \Rightarrow^* w$.

Two cases may arise: $Y \Rightarrow^* \varepsilon$ and $Z \Rightarrow^* X$ or $Y \Rightarrow^* X$ and $Z \Rightarrow^* \varepsilon$. We solve the former, the proof of the latter being similar.

Applying Lem. A.2 to $Y \underset{G}{\Rightarrow}^* \varepsilon$ and $d \underset{R}{\Rightarrow}^* d$ we find that $[dYd] \underset{G^r}{\Rightarrow}^* \varepsilon$. Next since $Z \Rightarrow^k X$ with $k < i-1$ we find by induction hypothesis that $[dZd'] \underset{G^r}{\Rightarrow}^* w$, hence that $[dX_0d'] \underset{G^r}{\Rightarrow}^* w$ since $([dX_0d'] \to [dYd][dZd']) \in \mathcal{P}^r$ and we are done. ∎

Lemma A.5. *Let $w \in (\Sigma \cup \Sigma_i)^*$, $d, d' \in D$ and $X \in \mathcal{X}$.*

$$[dXd'] \underset{G^r}{\Rightarrow}^* w \quad \textit{iff} \quad d \underset{R}{\Rightarrow}^* w \cdot d' \wedge X \underset{G}{\Rightarrow}^* w \ .$$

Proof. The proof for the only if direction is by induction on the length of the derivation of $[dXd'] \Rightarrow^* w$.

$\mathbf{i = 1}$. So we conclude from $[dXd'] \Rightarrow \sigma$ that $([dXd'] \to \sigma) \in \mathcal{P}^r$, hence that $(X \to \sigma) \in \mathcal{P}$ and $(d \to \sigma \cdot d') \in \delta$ or $d = d'$ by definition of G^r, and finally that $X \Rightarrow \sigma$ and $d \Rightarrow \sigma \cdot d'$ and we are done.

$\mathbf{i > 1}$. If the derivation of G^r has i steps with $i > 1$, it must be the case that:

$[dXd'] \Rightarrow [dZd_\ell][d_\ell Yd'] \Rightarrow^j w_1 \cdot [d_\ell Yd'] \Rightarrow^k w_1w_2$ where $w = w_1w_2$ and $j + k = i - 1$. By induction hypothesis, we have $d \Rightarrow^* w_1 \cdot d_\ell$ and $Z \Rightarrow^* w_1$. Also $d_\ell \Rightarrow^* w_2 \cdot d'$ and $Y \Rightarrow^* w_2$. Hence we find that $d \Rightarrow^* w_1w_2 \cdot d'$ and $X \Rightarrow^* w_1w_2$ since $(X \to ZY) \in \mathcal{P}$ and we are done since $w = w_1w_2$.

For the if direction, let $w \in \Sigma^*$ such that $X \underset{G}{\Rightarrow}^* w$ and $d \underset{R}{\Rightarrow}^* w \cdot d'$. Then the proof goes by induction on the length i of w.

$\mathbf{i = 0, 1}$. We have $d \underset{R}{\Rightarrow}^* \sigma \cdot d' \wedge X \underset{G}{\Rightarrow}^* \sigma$ with $\sigma \in (\Sigma \cup \Sigma_i \cup \{\varepsilon\})$. This coincides with the result of Lem. A.2.

$\mathbf{i > 1}$. Lem. A.3 shows that there exist $X', X_1, X_2 \in \mathcal{X}$ and $w_1, w_2 \in (\Sigma \cup \Sigma_i)^* \setminus \{\varepsilon\}$ such that $X \Rightarrow^* X' \Rightarrow X_1X_2 \Rightarrow^* w_1X_2 \Rightarrow w_1w_2 = w$.

Since $d \Rightarrow^* w \cdot d'$ and $w_1w_2 = w$, the definition of R shows that there exists $d_\ell \in D$ such that $d \Rightarrow^* w_1 \cdot d_\ell \Rightarrow^* w_1w_2 \cdot d'$.

Hence we can use that induction hypothesis for w_1 and w_2 which shows that $[dX_1d_\ell] \Rightarrow^* w_1$ and $[d_\ell X_2d'] \Rightarrow^* w_2$. Next, we conclude from $(X' \to X_1X_2) \in \mathcal{P}$ that $([dX'd'] \to [dX_1d_\ell][d_\ell X_2d']) \in \mathcal{P}^r$, hence that $[dX'd'] \Rightarrow^* w_1w_2 = w$.

Finally $X \Rightarrow^* X'$ and the result of Lem. A.4 shows that $[dXd'] \Rightarrow^* w$. ∎

Definition A.6. Given $G^r = (\mathcal{X}^r, \Sigma \cup \Sigma_i, \mathcal{P}^r)$ as given in Def. 4.2. Define $G^R = (\mathcal{X}^R, \Sigma, \mathcal{P}^R)$ where $\mathcal{X}^R = \mathcal{X}^r$; and $\mathcal{P}^R$ is the smallest set such that if $(X \to \alpha) \in \mathcal{P}^r$ then $(X \to \mathit{Proj}_{\Sigma \cup \mathcal{X}^R}(\alpha) \in \mathcal{P}^R)$.

It is routine to check that Def. A.6 is equivalent to Def. 4.2 p. 12. Finally, we conclude from Lem. A.5 and Def. A.6 that for every $d, d' \in D$ and $X \in \mathcal{X}$ we have: (i) let $w_1 \in \Sigma^*$ such that $[dXd'] \underset{G^R}{\Rightarrow}^* w_1$ then there exists $w_2 \in (\Sigma \cup \Sigma_i)^*$ such that $d \underset{R}{\Rightarrow}^* w_2 \cdot d'$, $X \underset{G}{\Rightarrow}^* w_2$, and $\mathit{Proj}_\Sigma(w_2) = w_1$; (ii) let $w \in (\Sigma \cup \Sigma_i)^*$ such that $d \underset{R}{\Rightarrow}^* w \cdot d'$, $X \underset{G}{\Rightarrow}^* w$ then $[dXd'] \underset{G^R}{\Rightarrow}^* \mathit{Proj}_\Sigma(w)$. Hence Lem. 4.3 holds.

A.1. Reduction from Petri Nets to Boolean Petri Nets

Lemma A.7. *(1) Let $(N, \mathbf{m}_\imath)$ be an initialized* PN*. There exists a Boolean initialized* PN *$(N', \mathbf{m}'_\imath)$ computable in polynomial time in the size of $(N, \mathbf{m}_\imath)$ such that $(N, \mathbf{m}_\imath)$ is bounded iff $(N', \mathbf{m}'_\imath)$ is bounded.*

(2) Let $(N, \mathbf{m}_\imath, \mathbf{m}_f)$ be an instance of the reachability (respectively, coverability) problem. There exists a Boolean initialized Petri net $(N', \mathbf{m}'_\imath)$ and a Boolean marking $\mathbf{m}'_f$ computable in polynomial time such that $\mathbf{m}_f$ is reachable (respectively, coverable) in $(N, \mathbf{m}_\imath)$ iff $\mathbf{m}'_f$ is reachable (respectively, coverable) in $(N', \mathbf{m}'_\imath)$.

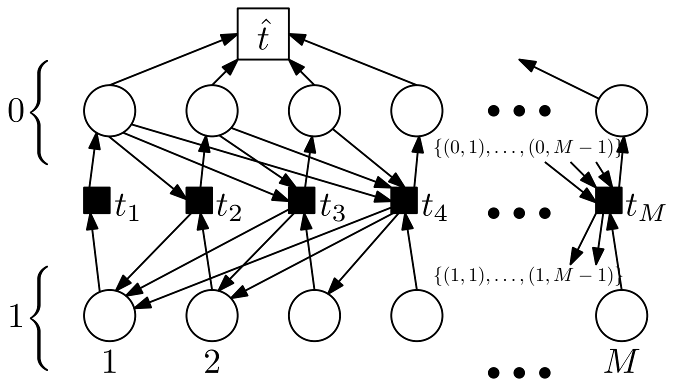

Fig. 9. A Petri net widget, left to right is from the least significant bit to the most significant bit.

Proof. We prove the result in two steps. First, we transform the instances so that the initial marking and (in case of coverability and reachability) the target markings are Boolean. Second, we transform the instances so that $I(t)$ and $O(t)$ are Boolean for each transition t.

Consider a boundedness problem instance $(N = (S, T, F), \mathbf{m}_\imath)$. In the first step, we define an equivalent instance $(N^\flat, \mathbf{m}_\imath^\flat)$ where the marking $\mathbf{m}_\imath^\flat$ is Boolean (but transitions in $N^\flat$ need not be Boolean). We perform the transformation by adding a new place p_i and a new transition t_i that consumes a token from p_i and puts $\mathbf{m}_\imath$ tokens in the other places. Initially, $\mathbf{m}_\imath^\flat$ has one token in p_i and zero tokens in all other places. Formally, $N^\flat = (S \cup \{p_i\}, T \cup \{t_i\}, F^\flat = \langle I^\flat, O^\flat\rangle)$, where $I^\flat(t) = I(t)$ and $O^\flat(t) = O(t)$ for all $t \in T$ and $I^\flat(t_i) = [\![p_i]\!]$ and $O^\flat(t_i) = \mathbf{m}_\imath$.

Consider now a coverability problem instance $(N, \mathbf{m}_\imath, \mathbf{m})$. To replace $\mathbf{m}_\imath$ and $\mathbf{m}$ by Boolean markings, intuitively, we add two new places p_i and p_c to N. As in the case of boundedness, there is a single transition out of p_i that consumes one token and produces $\mathbf{m}_\imath$. Additionally, there is one transition that consumes $\mathbf{m}$ and produces a single token in p_c. Formally, define $N^\flat = (S \cup \{p_i, p_c\}, T \cup \{t_i, t_c\}, F^\flat)$ with $F^\flat(T) = F(T)$, $F^\flat(t_i) = \langle [\![p_i]\!], \mathbf{m}_\imath\rangle$ and $F^\flat(t_c) = \langle \mathbf{m}, [\![p_c]\!]\rangle$. The initial and target marking are respectively given by $[\![p_i]\!]$ and $[\![p_c]\!]$ each of which is Boolean.

Let us turn to a reachability problem instance $(N, \mathbf{m}_\imath, \mathbf{m})$. The initial marking is made Boolean using the same trick: add a new place p_i and add a transition that consumes one token from p_i and produces $\mathbf{m}_\imath$ tokens. To get rid of $\mathbf{m}$, we use a construction from [Hack 1976] and additionally, we add a new place p_r. Then, we change each transition of N to additionally consume a token from p_r and produce a token back in p_r. Finally, we add a new transition that consumes $\mathbf{m} \oplus [\![p_r]\!]$ tokens and produces no tokens. The initial marking puts one token each at p_i and p_r, and we ask if the marking where every place has zero tokens is reachable. Formally, define $N^\flat = (S \cup \{p_i, p_r\}, T \cup \{t_i, t_r\}, F^\flat)$ such that $F^\flat(t) = \langle [\![p_r]\!] \oplus I(t), [\![p_r]\!] \oplus O(t)\rangle$ where $F(t) = \langle I(t), O(t)\rangle$, $F^\flat(t_i) = \langle [\![p_i]\!], \mathbf{m}_\imath\rangle$ and $F^\flat(t_r) = \langle \mathbf{m} \oplus [\![p_r]\!], \varnothing\rangle$. The initial and target marking are respectively given by $[\![p_i, p_r]\!]$ and $\varnothing$ the empty marking each of those marking being a set.

We now move to the second step of the construction. Given a PN $N = (S, T, F)$, we show how to compute in polynomial time a PN $N' = (S', T', F')$ such that for every transition $t \in T'$ the multisets $I(t)$ and $O(t)$ are Boolean. The construction is independent of the decision problem (boundedness, coverability, or reachability).

Assume that S is given by $\{s_1, \ldots, s_n\}$ and T is given by $\{t_1, \ldots, t_k\}$.

We convert N to a Boolean Petri net in five steps. First, we define the PN $N_1 = (S_1, T_1, F_1)$. The set of places $S_1 = S$. For each $t \in T$, we define the transitions $t^I_1, t^I_2, \ldots, t^I_n$, $t^O_1, t^O_2, \ldots, t^O_n$ in T_1 such that:

— $F_1(t^I_i) = \langle \mathit{Proj}_{\{s_i\}}(I(t)), \varnothing \rangle$ and $F_1(t^O_i) = \langle \varnothing, \mathit{Proj}_{\{s_i\}}(O(t)) \rangle$ for $i \in \{1, \ldots, n\}$.

Intuitively, to each pair (s_i, t) ($i \in \{1, \ldots, n\}$, $t \in T$) we associate two transitions t^I_i and t^O_i of T_1 which we will use to simulate the effect of t on s_i.

Second, we define the PN N_2 which is given by the synchronized product of N_1 with the following regular language over alphabet T_1:

$$L \overset{def}{=} (w_1 + \cdots + w_k)^*$$

where each $w_i = t^I_{i1} t^O_{i2} \ldots t^I_{in} t^O_{in}$ is a finite word that simulates the firing of transition $t_i \in T$ for $i \in \{1, \ldots, k\}$. Clearly, since each w_i corresponds to the firing of transition $t_i \in T$ we find that N_2 simulates N (i.e., $\mathbf{m}\,[t\rangle$ does not hold in T iff $\mathbf{m}\,\left[t^I_i\right\rangle$ does not hold from some $i \in \{1, \ldots, n\}$; and $\mathbf{m}\,[t\rangle\,\mathbf{m}'$ iff $\mathbf{m}\,[w\rangle\,\mathbf{m}'$).

Observe that N_2 is still not a Boolean PN. In the third step, we replace each transition t^O_i (resp. t^I_i) which produce (resp. consume) $\mathit{Proj}_{\{s_i\}}(O(t))$ (resp. $\mathit{Proj}_{\{s_i\}}(I(t))$) tokens to (resp. from) place s_i by a Boolean PN $N_{t^O_i}$ (resp. $N_{t^I_i}$). We do this by defining the following class of widgets.

Let us consider a transition t^O_i which produces m tokens into s_i, and let $M = \lceil \log_2 m \rceil$. We will substitute t^O_i with a Boolean PN $N_{t^O_i}$. We call such a PN a widget. A generic description of a widget is given in Fig. 9.

Intuitively, the widget behaves like a binary decrementer. To begin with, we shall put a $(0, 1)$-marking on the widget, where for each "column" labeled $1, \ldots, M$, we put a single token in either the 0th row or the 1st row. Each $(0, 1)$-marking coincides with the binary representation of a number in the range $[0, 2^M - 1]$, obatined by $\sum_{i=1}^{M} \delta_i 2^i$, where $\delta_i = 1$ if the $(0, 1)$-marking places a token in the 1st row of column i and $\delta_i = 0$ if the $(0, 1)$-marking places a token in the 0th row of column i. Conversely, every number in the range $[0, 2^M - 1]$ corresponds to exactly one $(0, 1)$-marking of the widget. Let f be the function which takes as input a number in the range $[0, 2^M - 1]$ and returns the corresponding $(0, 1)$-marking.

One can check that the widget defines a Boolean PN. Moreover, from every $(0, 1)$-marking there exists exactly one enabled transition in the widget. Hence the widget behaves as follows: starting from marking $f(m)$ there exists a unique maximal sequence of enabled transitions which consists of m transitions in $\{t_1, \ldots, t_n\}$ followed by $\hat{t}$ enabled at the marking which represents 0 in binary (i.e., the $(0, 1)$-marking that puts a single token each in the 0th row of each column). Next, we add transition $\check{t}$ whose role is to initialize the widget with marking $f(m)$. Therefore we have $F_{t^O_i}(\check{t}) = \langle \varnothing, f(m) \rangle$. Finally let us add an arc from every transition of the widget except $\hat{t}$ and $\check{t}$ into place s_i.

From the above construction, we observe that the firing of any sequence in the language $\check{t} \cdot (\{t_1, \ldots, t_n\})^* \cdot \hat{t}$ has the effect of producing exactly m tokens in place s_i.

Using a similar reasoning one can define a widget for t^I_i.

In the fourth step, let us define N_3 as the PN which is given by the union of all the widgets (therefore S is contained in the places of N_3). Given $i \in \{1, \ldots, n\}$, let us denote by $T_{t^I_i}$ and $T_{t^O_i}$ the set of transitions of the widget corresponding to t^I_i and t^O_i, respectively. Also we have transitions $\check{t}^I_i, \hat{t}^I_i, \check{t}^O_i, \hat{t}^O_i$. Observe that N_3 is a Boolean PN.

Finally, to conclude the construction of the Boolean PN N', we define N' as the synchronized product of N_3 with the language $\tau(L)$ where τ is a substitution which maps t^I_i onto the language $(\check{t}^I_i \cdot (T_{t^I_i})^* \cdot \hat{t}^I_i)$ and t^O_i onto the language $(\check{t}^O_i \cdot (T_{t^O_i})^* \cdot \hat{t}^O_i)$.

It is routine to check that the obtained PN is Boolean and it can be computed in polynomial time in the size of N. ∎

Received April 2011; revised xxx XXXX; accepted xxx XXXX

ACKNOWLEDGMENTS

We thank Mohamed-Faouzi Atig, Andrey Rybalchenko, Bishesh Adhikari and Laurent Van Begin for useful discussions, and the anonymous referees for many useful comments. We thank Laurent Van Begin for the proof of Th. 7.14.

REFERENCES

ABDULLA, P. A., CERANS, K., JONSSON, B., AND TSAY, Y.-K. 1996. General decidability theorems for infinite-state systems. In *LICS '96: Proc. 11th Annual IEEE Symp. on Logic in Computer Science*. IEEE Computer Society, 313–321.

ABDULLA, P. A. AND MAYR, R. 2009. Minimal cost reachability/coverability in priced timed petri nets. In *FOSSACS '09: Proc. 12th Int. Conf. Foundations of Software Science and Computational Structures*. LNCS Series, vol. 5504. Springer, 348–363.

AHO, A., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.

ATIG, M. F., BOUAJJANI, A., AND TOUILI, T. 2008. Analyzing asynchronous programs with preemption. In *FSTTCS '08: Proc. 28th Int. Conf. on Fondation of Software Technology and Theoretical Computer Science*. Leibniz International Proceedings in Informatics (LIPIcs) Series, vol. 2. Leibniz-Zentrum fuer Informatik, 37–48.

ATIG, M. F. AND HABERMEHL, P. 2009. On Yen's path logic for Petri nets. In *RP '09: Proc. 3rd Workshop on Reachability Problems*. LNCS Series, vol. 5797. Springer, 51–63.

BOUAJJANI, A., ESPARZA, J., AND MALER, O. 1997. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR '97: Proc. 8th Int. Conf. on Concurrency Theory*. LNCS Series, vol. 1243. Springer, 135–150.

BURKART, O. AND STEFFEN, B. 1994. Pushdown processes: Parallel composition and model checking. In *CONCUR '94: Proc. 5th Int. Conf. on Concurrency Theory*. LNCS Series, vol. 836. Springer, 98–113.

CHADHA, R. AND VISWANATHAN, M. 2007. Decidability results for well-structured transition systems with auxiliary storage. In *CONCUR '07: Proc. 18th Int. Conf. on Concurrency Theory*. LNCS Series, vol. 4703. Springer, 136–150.

CHADHA, R. AND VISWANATHAN, M. 2009. Deciding branching time properties for asynchronous programs. *Theor. Comput. Sci. 410,* 42, 4169–4179.

DICKSON, L. E. 1913. Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. *Amer. J. Math. 35*, 413–422.

DUFOURD, C., FINKEL, A., AND SCHNOEBELEN, P. 1998. Reset nets between decidability and undecidability. In *ICALP '98: Proc. of 25th Int. Colloquium on Automata, Languages and Programming*. LNCS Series, vol. 1443. Springer, 103–115.

ESPARZA, J. 1997. Petri nets, commutative context-free grammars, and basic parallel processes. *Fundamenta Informaticae 31*, 13–26.

ESPARZA, J. 1998. Decidability and complexity of petri net problems – an introduction. In *Lectures on Petri Nets I: Basic Models*. LNCS Series, vol. 1491. Springer, 374–428.

ESPARZA, J., FINKEL, A., AND MAYR, R. 1999. On the verification of broadcast protocols. In *LICS '99: Proc. 14th Annual IEEE Symp. on Logic in Computer Science*. IEEE Computer Society, 352–359.

ESPARZA, J., GANTY, P., KIEFER, S., AND LUTTENBERGER, M. 2011. Parikh's theorem: A simple and direct automaton construction. *Information Processing Letters 111*, 614–619.

ESPARZA, J., KIEFER, S., AND LUTTENBERGER, M. 2010. Newtonian program analysis. *Journal of the ACM 57,* 6, 33:1–33:47.

ESPARZA, J. AND NIELSEN, M. 1994. Decibility issues for Petri nets - a survey. *Journal of Informatik Processing and Cybernetics 30,* 3, 143–160.

FINKEL, A. AND SANGNIER, A. 2010. Mixing coverability and reachability to analyze vass with one zero-test. In *SOFSEM '10: Proc. 36th Conf. on Current Trends in Theory and Practice of Computer Science*. LNCS Series, vol. 5901. Springer, 394–406.

FINKEL, A. AND SCHNOEBELEN, P. 2001. Well-structured transition systems everywhere! *Theoretical Computer Science 256,* 1-2, 63–92.

GANTY, P. AND MAJUMDAR, R. 2009. Analyzing real-time event-driven programs. In *FORMATS '09: Proc. 7th Int. Conf. on Formal Modelling and Analysis of Timed Systems*. LNCS Series, vol. 5813. Springer, 164–178.

GANTY, P., MAJUMDAR, R., AND RYBALCHENKO, A. 2009. Verifying liveness for asynchronous programs. In *POPL '09: Proc. 36th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages*. ACM Press, 102–113.

HACK, M. H. T. 1976. Decidability questions for Petri nets. Tech. Rep. 161, MIT. June.

HILL, J. L., SZEWCZYK, R., WOO, A., HOLLAR, S., CULLER, D. E., AND PISTER, K. S. J. 2000. System architecture directions for networked sensors. In *ASPLOS '00 Proc. 9th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*. ACM, 93–104.

JHALA, R. AND MAJUMDAR, R. 2007. Interprocedural analysis of asynchronous programs. In *POPL '07: Proc. 34th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages*. ACM Press, 339–350.

KARP, R. M. AND MILLER, R. E. 1969. Parallel program schemata. *Journal of Comput. Syst. Sci. 3,* 2, 147–195.

KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. 2000. The Click modular router. *ACM TOCS 18,* 3, 263–297.

KOSARAJU, S. R. 1982. Decidability of reachability in vector addition systems (preliminary version). In *STOC '82: Proc. of 14th ACM symp. on Theory of Computing*. ACM, 267–281.

KROHN, M., KOHLER, E., AND KAASHOEK, M. 2007. Events can make sense. In *USENIX Annual Technical Conference*. USENIX Association.

LAMBERT, J. L. 1992. A structure to decide reachability in petri nets. *Theor. Comput. Sci. 99,* 1, 79–104.

LANGE, M. AND LEISS, H. 2008-2010. To CNF or not to CNF? An efficient yet presentable version of the CYK algorithm. *Informatica Didactica 8*, 1–21.

LIPTON, R. 1976. The reachability problem is exponential-space hard. Tech. Rep. 62, Department of Computer Science, Yale University. Jan.

MAYR, E. W. 1981. An algorithm for the general petri net reachability problem. In *STOC '81: Proc. of 13th ACM symp. on Theory of computing*. ACM, 238–246.

MAYR, E. W. AND MEYER, A. R. 1981. The complexity of the finite containment problem for petri nets. *Journal of the ACM 28,* 3, 561–576.

MINSKY, M. 1967. *Finite and Infinite Machines*. Englewood Cliffs, N.J., Prentice-Hall.

PAI, V., DRUSCHEL, P., AND ZWAENEPOEL, W. 1999. Flash: An efficient and portable web server. In *Proc. USENIX Tech. Conf.* Usenix, 199–212.

PARIKH, R. J. 1966. On context-free languages. *Journal of the ACM 13,* 4, 570–581.

RACKOFF, C. 1978. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science 6,* 2, 223–231.

REISIG, W. 1986. *Petri Nets. An introduction*. Springer.

REPS, T., HORWITZ, S., AND SAGIV, M. 1995. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95: Proc. 22nd ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages*. ACM, 49–61.

SEN, K. AND VISWANATHAN, M. 2006. Model checking multithreaded programs with asynchronous atomic methods. In *CAV '06: Proc. 18th Int. Conf. on Computer Aided Verification*. LNCS Series, vol. 4144. Springer, 300–314.

SHARIR, M. AND PNUELI, A. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Inc., Chapter 7, 189–233.

VARDI, M. Y. 1991. Verification of concurrent programs — the automata-theoretic approach. *Annals of Pure and Applied Logic 51*, 79–98.

WALUKIEWICZ, I. 2001. Pushdown Processes: Games and Model-Checking. *Information and Computation 164,* 2, 234–263.

YEN, H.-C. 1992. A unified approach for deciding the existence of certain petri net paths. *Information and Computation 96,* 1, 119–137.