Verifying Liveness for Asynchronous Programs *

Pierre Ganty
CS Department, University of California, Los Angeles, CA, USA
pganty@cs.ucla.edu

Rupak Majumdar
CS Department, University of California, Los Angeles, CA, USA
rupak@cs.ucla.edu

Andrey Rybalchenko
Max Planck Institute for Software System, Germany
rybal@mpi-sws.mpg.de

Abstract

Asynchronous or "event-driven" programming is a popular technique to efficiently and flexibly manage concurrent interactions. In these programs, the programmer can post *tasks* that get stored in a task buffer and get executed atomically by a non-preemptive scheduler at a future point. We give a decision procedure for the *fair termination* property of asynchronous programs. The fair termination problem asks, given an asynchronous program and a fairness condition on its executions, does the program always terminate on fair executions? The fairness assumptions rule out certain undesired bad behaviors, such as where the scheduler ignores a set of posted tasks forever, or where a non-deterministic branch is always chosen in one direction. Since every liveness property reduces to a fair termination property, our decision procedure extends to liveness properties of asynchronous programs.

Our decision procedure for the fair termination of asynchronous programs assumes all variables are finite-state. Even though variables are finite-state, asynchronous programs can have an unbounded stack from recursive calls made by tasks, as well as an unbounded task buffer of pending tasks. We show a reduction from the fair termination problem for asynchronous programs to fair termination problems on Petri Nets, and our main technical result is a reduction of the latter problem to Presburger satisfiability. Our decidability result is in contrast to multi-threaded recursive programs, for which liveness properties are undecidable.

While we focus on fair termination, we show our reduction to Petri Nets can be used to prove related properties such as *fair non-starvation* (every posted task is eventually executed) and safety properties such as *boundedness* (find a bound on the maximum number of posted tasks that can be in the task buffer at any point).

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification.
General Terms: Languages, Verification, Reliability.
Keywords: asynchronous (event-driven) programming, liveness, fair termination, Petri Nets.

* This research was sponsored in part by the NSF grants CCF-0546170, CCF-0702743, CNS-0720881, and by Microsoft Research through the European Fellowship Programme.

POPL'09, January 18–24, 2009, Savannah, Georgia, USA.

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 *post* asynchronous procedure calls which are stored in a task buffer for later execution, instead of being executed right away. 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 posted calls from the task buffer and executes them atomically to completion. Execution of the posted calls can lead to further calls being posted. The interleaving of different picks-and-executes (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 [23, 13], embedded systems and sensor networks [8], 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 asynchronously posted calls (including saving and passing appropriate state between calls), 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.

In this paper, we focus on verifying *liveness properties* of asynchronous programs. Informally, liveness properties specify that "something good eventually happens," and can be used to specify progress properties such as every posted call eventually gets dispatched. Specifically, we develop algorithms to check that an asynchronous program terminates under certain fairness constraints on the scheduler and external events. We call this the *fair termination* problem. The fairness conditions on the scheduler rule out certain undesired paths, in which for example the scheduler postpones forever some posted task. It is known that general liveness properties can be reduced to checking fair termination [35].

In the following, we restrict attention to asynchronous programs in which the data ranges over a finite domain of values. The finiteness assumption on the data is necessary, since fair termination is already undecidable for 2-counter machines [22]. However, we do not restrict the depth of the stack or the size of the task buffer which could both be unbounded.

For sequential programs with *synchronous* calls, there has been a lot of work in automatic techniques to proving fair termination [20, 15, 30, 21]. It seems natural that one can reduce reasoning about asynchronous programs to reasoning about synchronous programs by explicitly modeling the task buffer and the scheduling. For example, we can add a counter representing the number of

pending instances for each procedure, increment the appropriate counter at each *post*, and model the scheduler as a *dispatch loop* which picks a non-zero counter, decrements it, and executes the corresponding procedure. While the reduction is sound, applying sequential fair termination checking on this sequential program is not guaranteed to be complete (i.e., terminate with the correct answer on all inputs), since checking termination for programs with integer counters is undecidable in general. The difficulty arises because the number of pending calls in the task buffer (a call is *pending* if it has been posted but not yet picked), and hence the counters, can grow unboundedly large.

Our main result is that fair termination is *decidable* for asynchronous programs. This is unlike the case for multi-threaded programs communicating through shared variables, for which the problem is undecidable [27].

Our decidability proof uses constructions on Petri Nets [28] (an infinite state concurrency model with many decidable properties), via some language-theoretic reductions. As noted in [2, 11, 32], we use 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 calls can only increase, and the number of pending calls decreases precisely when the program stack is empty. Accordingly, our proof of decidability combines two technical constructions. First, we use Parikh's theorem [24] from language theory to precisely summarize the effect of a synchronous call on the state of the counters. Second, with this summarization, we can construct a Petri Net that captures the effect of the execution of a posted call in the system.

Note that by using Parikh's theorem (which preserves finite executions only) we implicitly assume that each executed procedure eventually terminates. We check this property using a decision procedure for synchronous program termination. For this check, the task buffer can be abstracted away (as no dispatches occur from within a call) and we can use techniques for liveness checking for finite-state pushdown systems [1, 36].

With the above check, it remains to show that the Petri Net is fairly terminating. Using known constructions on Petri Nets [33], we reduce this question to proving the absence of certain finite paths in the coverability graph of the Petri Net (for now, think a coverability graph as a finite automaton that captures the runs of the Petri Net). Finally, we reduce the existence of such finite runs to the satisfiability of a formula in the Presburger arithmetic which can be effectively constructed from the coverability graph of the Petri Net. Since each step of the reduction is effective, and satisfiability of Presburger arithmetic is decidable [25], we get our decision procedure.

Our reduction of problems on asynchronous programs to problems on Petri Nets enables the use of an extensive algorithmic repertoire built for Petri Nets. Indeed, we show two alternate proofs for fair termination which follow from the decidability of more general model checking questions on Petri Nets [10, 37].

Moreover, algorithmic analysis on Petri Nets can be used to provide decision procedures for related questions on asynchronous programs. We mention two applications. First, the *fair non-starvation* question asks, given an asynchronous program and a fairness condition on executions, whether every pending call is eventually dispatched (i.e., no posted call waits forever). Fair non-starvation is practically relevant to ensure that an asynchronous program (such as a server) is responsive. We show a decision procedure for this problem by adding new constraints to the Presburger formula for the fair termination problem.

Second, we show a decision procedure for *boundedness*, a safety property that computes the maximum possible size of the task buffer at any point in any execution. For the boundedness property we again use a known result on Petri Nets which allows to compute the maximum possible size D 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 the construction of a coverability graph [12, 33, 34].

Related Work. Event driven programming is a popular way to write high-performance systems, and the style is supported in most programming environments either as libraries (such as libasync [17], libevent [18], or libeel [4]), or as language features [6, 7, 14, 16]. However, it is widely recognized that while the style optimizes for low overhead and efficient execution, programs written in this style may be hard to read and debug.

While we focus on analyzing asynchronous programs written in C and using a library for asynchronous calls, there have been several recent attempts at language-level support for writing asynchronous programs to enable better automatic reasoning [3, 6, 14, 16]. Many of these language extensions "compile down" to our basic model, and hence, our decidability results apply. It will be interesting to see how language level support can be combined with our algorithms to prove deeper properties of systems.

Our work is inspired by recent results that show the decidability of safety properties of asynchronous programs [32, 11, 2], and in particular, of precise meet-over-all-paths dataflow analysis [11].

Our correctness arguments through Petri Nets use powerful algorithmic tools developed for Petri Nets [12, 33, 34]. The reduction from the existence of a fair non-terminating run in a Petri Net to the existence of a certain finite run in its coverability graph uses techniques similar to [33]. The reduction of this latter problem as the satisfiability problem of a Presburger formula is inspired by the encoding found in [31].

2. Asynchronous Programs and Properties

We motivate our problem on a simple imperative language. In our examples, we use C-like syntax with an additional construct `post` $f(e)$ which asynchronously posts a procedure call f with arguments e.

Figure 1(a) 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 of posted calls and the number of calls that have returned (i.e., that have been completed). We abstract the server by the procedure `rpccall` which increments the number of calls that have returned. The procedure `wrpc` first checks how many posted calls have returned. If the number of returned calls is `n` or more, it terminates. If fewer than `n` calls have returned, it posts new calls if required. A new call is posted if (1) the number of calls already posted is fewer than `n`, and (2) the number of pending calls (equal to `sent` $-$ `recv`) is lower than the window size `w`. If these conditions are satisfied, an asynchronous call to `rpccall` is posted, and the variable `sent` is incremented. In either case, `wrpc` reposts a call to itself (this is done by an asynchronous recursive call), either to post more calls or to wait for pending calls to return.

```
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();
```

(a) Windowed RPC implementation

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

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

(b) A fairly terminating asynchronous program

Figure 1. Examples of asynchronous programs.

As mentioned in [14], 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.

2.1 Programming Model

We represent programs using a generalization of control flow graphs, that include special edges corresponding to asynchronous procedure calls. Let P be a finite set of *procedure names*. An *asynchronous control flow graph* (ACFG) G_p for a procedure $p \in P$ is a pair (V_p, E_p) where V_p is the set of *control nodes* of the procedure p, including a unique *start node* v_p^s and a unique *exit node* v_p^e, and E_p is a set of directed *intraprocedural edges* between the control nodes V_p. The edges in E_p are partitioned into edges $E^{(o)}$, $E^{(s)}$, and $E^{(a)}$, corresponding to one of the following:

- *an operation edge* corresponding to a basic block of assignments or an assume predicate derived from a branch condition ($E^{(o)}$);
- *a synchronous call edge* to a procedure $q \in P$ ($E^{(s)}$); or
- *an asynchronous call edge* to a procedure $q \in P$ ($E^{(a)}$).

For each call edge, synchronous or asynchronous, from v to v' we call the source node v the *call-site* node, and the target node v' the *return-site* node.

A *program* $G^{\bowtie}$ comprises a set of pairwise disjoint ACFGs G_p for each procedure in $p \in P$ (we also say *handler*). The control locations of $G^{\bowtie}$ are given by $V^{\bowtie} = \bigcup_{p \in P} V_p$: the union of the control locations of the individual procedures. The edges of $G^{\bowtie}$ are given by $E^{\bowtie} = \bigcup_{p \in P} E_p \cup E^{(i)}$, the union of the (intraprocedural) edges of the individual procedures and a special set $E^{(i)}$ of *interprocedural edges* defined as follows. For each $(v, v') \in E^{(s)}$ that calls procedure q, that is for each synchronous call edge from call-site v to procedure q returning to return-site v', we have:

- an interprocedural *call-to-start* edge from the call-site v to the start node v_q^s of q (i.e., $(v, v_q^s) \in E^{(i)}$); and
- an interprocedural *exit-to-return* edge from the exit node v_q^e of q to the return-site v' (i.e., $(v_q^e, v') \in E^{(i)}$).

As in [29], the call-to-start edges (or exit-to-return edges) allow us to model parameter passing and context restoration in our framework. An *asynchronous program* $A = (P, G^{\bowtie}, \texttt{main})$ consists of a set of procedure names P, a program $G^{\bowtie}$, and an initial procedure $\texttt{main} \in P$ which we assume has no parameters and is never called by any procedure (either synchronously or asynchronously).

Semantics. We consider *abstract* semantics of asynchronous programs relative to finite *dataflow domains* (D, M, d_0), where D is a finite set of dataflow facts, $M\colon S \to D \to D$ is a dataflow transfer function and $S = \bigcup_{p \in P} \{E_p^{(o)}\} \cup E^{(i)} \cup \{(v_{main}^e, v_q^s) \mid q \in P\} \cup \{(v_q^e, v_{main}^e) \mid q \in P\}$, and $d_0 \in D$ the initial dataflow state.

The abstract semantics of an asynchronous program $A = (P, G^{\bowtie}, \texttt{main})$ relative to the dataflow domain (D, M, d_0) is given by a transition system where each state $((v, d), w, m)$ consists in: the *abstract state* $(v, d) \in V^{\bowtie} \times D$, the *program stack* $w \in (V^{\bowtie})^*$, and a multiset m over P called the *multiset of pending calls*. The initial state is $((v_{\texttt{main}}^s, d_0), v_{\texttt{main}}^e, \emptyset)$ in which the abstract state is $(v_{\texttt{main}}^s, d_0)$, the stack content is given by the word $v_{\texttt{main}}^e$ and the multiset is empty. There is a transition from a state $((v, d), s, m)$ to the state $((v', M(v, v')(d)), s, m)$ if there is an edge $(v, v') \in E^{(o)}$ (*internal operation*). There is a transition from $((v, d), s, m)$ to $((v_q^s, M(v, v_q^s)(d)), s \cdot v', m)$ if there is a synchronous call edge (v, v') calling q in $E^{(s)}$ (*procedure call*). There is a transition from $((v_q^e, d), s \cdot v', m)$ to $((v', M(v_q^e, v')(d)), s, m)$ if there is a synchronous call edge (v, v') calling q in $E^{(s)}$ (*procedure return*). There is a transition from $((v, d), s, m)$ to $((v', d), s, m \uplus \{q\})$ if there is an asynchronous call edge $(v, v') \in E^{(a)}$ to a procedure q (*asynchronous post*).[1] There is a transition from $((v_{\texttt{main}}^e, d), \epsilon, m)$ to $((v_q^s, M(v_{\texttt{main}}^e, v_q^s)(d)), v_{\texttt{main}}^e, m \setminus \{q\})$ if $q \in m$ (*asynchronous call dispatch*). Finally, there is a transition from $((v_q^e, d), s \cdot v_{\texttt{main}}^e, m)$ to $((v_{\texttt{main}}^e, M(v_q^e, v_{\texttt{main}}^e)(d)), s, m)$ (*asynchronous return*). A *run* of an asynchronous program relative to the dataflow domain is a path in the associated transition system, starting with the initial state. By abuse of notation, we write A to denote the abstract semantics over a fixed dataflow domain.

We now give some intuition about the control location $v_{\texttt{main}}^e$ which plays a special role in the above semantics. If the current state is such that the control location is $v_{\texttt{main}}^e$ and the stack content is empty (i.e., $((v_{\texttt{main}}^e, d), \epsilon, m)$ for some multiset m and dataflow fact d), then a procedure call from the multiset of pending calls, if

[1] The symbol $\uplus$ denotes the union between multisets.

any, is dispatched. Otherwise, if the multiset of empty, the program terminates. Thus $v^e_{\texttt{main}}$ models a special "dispatch loop."

For simplicity of exposition, we assume that procedures are parameterless and there are no local variables. For a fixed finite dataflow domain, parameters can be modeled by copying and renaming the functions, one copy for each parameter value, and local variables can be encoded into the nodes of the control flow graph.

2.2 Progress Guarantees

Consider the desirable property that the windowed RPC fairly terminates, which implies that, at some point in time, every posted call has returned and the multiset of pending calls is empty. Informally, this property is true because the procedure `wrpc` posts `rpccall` at most `n` times, and posts itself only as long as `recv` is less than `n`. Each run of `rpccall` increments `recv`, so that after `n` dispatches of `rpccall`, the value of `recv` reaches `n`, and from this point, all calls to `wrpc` do not post any new call. Thus, eventually, the multiset of pending calls becomes empty.

Notice that we need the assumption that the scheduler *fairly* dispatches posted calls: a posted call to q is followed by a dispatch to q. Without that assumption the program does not terminate: consider the infinite trace where the scheduler always picks `wrpc` in preference to `rpccall`.

Fair Termination. We now define the fair termination property on the model. An asynchronous program $(P, G^{\bowtie}, \texttt{main})$ *fairly terminates* if (i) every execution of a procedure that is called (synchronous or asynchronous) eventually returns; and (ii) there is no infinite run that is fair. An infinite run is said to be *fair* if for every state $((v, d), w, m)$ along this run, if the multiset m contains a call to procedure q then eventually a call to q is dispatched. 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). However, each request to the server should fairly terminate as defined above. Many event libraries (e.g., libeel [4]) allow identifying individual requests using mechanisms such as group-ids or session-ids.

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 infinite fair execution no handler is starved. A starving handler corresponds to a pending call which is never dispatched, and hence which waits forever to be executed. Consider a handler `h` that posts two calls to itself. An infinite fair execution dispatches a call to `h` each time a dispatch takes place, even though a particular call to `h` may never be run.

Proving fair termination and fair non-starvation for asynchronous programs is difficult for several reasons. First, as the windowed RPC example suggests, reasoning about termination may require reasoning about the dataflow facts (e.g., the fact that `recv` eventually reaches `n` in the example). Second, at each point, there can be an unbounded number of pending calls. This is illustrated by the program in Fig. 1(b), which terminates on each fair execution, but in which the multiset contains unboundedly many pending calls (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. Asynchronous Automata

We now formally introduce *asynchronous automata*, an automaton model for asynchronous programs. We give the semantics of asynchronous programs using asynchronous automata, and develop our theoretical results on asynchronous automata.

Prerequisites. Let Σ be an alphabet. We denote by Σ^* (respectively, Σ^ω) the set of finite (respectively, countably infinite) sequences over Σ. We write Σ_ϵ for the language $\Sigma \cup \{\epsilon\}$. Given $w \in \Sigma^*$, we use $|w|$ to denote its length.

A *counter map* is a mapping from Σ to $\mathbb{N}$. For a counter map $\mathbf{c}$, we write $\mathbf{c}[a \leftarrow i]$ for the counter map that maps $a \in \Sigma$ to $i \in \mathbb{N}$ and maps every $b \in \Sigma \setminus \{a\}$ to $\mathbf{c}(b)$. The counter map $\mathbf{0}$ maps every $a \in \Sigma$ to 0.

The *Parikh image* $\mathsf{Parikh}\colon \Sigma^* \to \mathbb{N}^\Sigma$ maps a word $w \in \Sigma^*$ to a counter map $\mathsf{Parikh}(w)$ such that $\mathsf{Parikh}(w)(a)$ is the number of occurrences of a in w. For example, $\mathsf{Parikh}(abbab)(a) = 2$ and $\mathsf{Parikh}(abbab)(b) = 3$. For a language L, we define $\mathsf{Parikh}(L) = \{\mathsf{Parikh}(w) \mid w \in L\}$.

LEMMA 1. **[Parikh's Lemma [24]]** *For any context free language L there is an effectively computable regular language L' such that* $\mathsf{Parikh}(L) = \mathsf{Parikh}(L')$.

3.1 Asynchronous Automata

An *asynchronous automaton* $A = (Q, \Sigma, \delta, q_0, a_0)$ consists of a finite set Q of *global states*, a finite alphabet Σ, a transition relation $\delta \subseteq (Q \times \Sigma) \times (Q \times \Sigma^* \times \Sigma_\epsilon)$, an initial state $q_0 \in Q$, and an initial handler call $a_0 \in \Sigma$.

A *configuration* of an asynchronous automaton is a tuple $(q, w, \mathbf{c}) \in Q \times \Sigma^* \times \mathbb{N}^\Sigma$ where q is a *global state*, w is a word of Σ^* called the *stack content*, and $\mathbf{c}$ is a *counter map of pending calls* (or simply the counter map). The initial configuration is $(q_0, a_0, \mathbf{0})$, where the global state is q_0, the stack content is a_0, and the counter map is $\mathbf{0}$.

We define a transition relation $\rightarrow$ on configurations as the union of a transition relation $\rightarrow_s$ modeling handler steps and a transition relation $\rightarrow_d$ modeling a dispatch, where $\rightarrow_s$ and $\rightarrow_d$ are defined as follows:

Handler Step There is a transition $(q, wa, \mathbf{c}) \rightarrow_s (q', ww', \mathbf{c}')$, where $((q, a), (q', w', a')) \in \delta$ and $\mathbf{c}' = \mathbf{c}[a' \leftarrow \mathbf{c}(a') + 1]$ if $a' \in \Sigma$ and $\mathbf{c}' = \mathbf{c}$ if $a' = \epsilon$.

Handler Dispatch There is a transition $(q, \epsilon, \mathbf{c}) \rightarrow_d (q, a, \mathbf{c}')$, where $\mathbf{c}(a) > 0$ and $\mathbf{c}' = \mathbf{c}[a \leftarrow \mathbf{c}(a) - 1]$.

In the former case, if $a' \neq \epsilon$ then we say that a call to a' has been posted. In the latter case we say that a pending call to procedure a is dispatched, or that a is dispatched or simply that a dispatch occurs.

A *run* of an asynchronous automaton is a (finite or infinite) sequence $(q_0, a_0, \mathbf{0}) \rightarrow (q_1, w_1, \mathbf{c}_1) \rightarrow \cdots$. Let $\rightarrow^*$ (respectively, $\rightarrow_s^*$) denote the reflexive transitive closure of $\rightarrow$ (respectively, $\rightarrow_s$). A configuration $(q, w, \mathbf{c})$ is *reachable* iff $(q_0, a_0, \mathbf{0}) \rightarrow^* (q, w, \mathbf{c})$. Finally, we write $\mathsf{Reach}_\epsilon(A)$ for the set of pairs $\{(q, \mathbf{c}) \mid \exists q \in Q\colon (q_0, a_0, \mathbf{0}) \rightarrow^* (q, \epsilon, \mathbf{c})\}$, that is the set of reachable configurations in which the stack is empty.

Infinite Runs. The infinite runs of an asynchronous automaton fall into two categories:

1. [Runs of the First Form] infinite runs of the form $(q_0, a_0, \mathbf{0}) \rightarrow_s^* (q_1, \epsilon, \mathbf{c}_1) \rightarrow_d (q_1, a_1, \mathbf{c}_1') \rightarrow_s^* (q_2, \epsilon, \mathbf{c}_2) \rightarrow_d \ldots$ in which there are infinitely many $\rightarrow_d$ steps; and
2. [Runs of the Second Form] infinite runs in which there are finitely many $\rightarrow_d$ steps (and so, in which infinitely many handler steps take place consecutively).

```
parse() {
  post read();
}

read() {
  post send();
}

send() {
}
Initially: parse();
```

Figure 2. A simple event processor

3.2 From Programs to Automata

In the following, we prove our theoretical results on asynchronous automata. We now show that the abstract semantics of asynchronous programs relative dataflow domain (D, M, d_0) can be given by translation to an asynchronous automaton.

Given an asynchronous program $A = (P, G^{\bowtie}, \texttt{main})$ with $G^{\bowtie} = (V^{\bowtie}, E^{\bowtie})$ we define an asynchronous automaton $A_{G^{\bowtie}} = (Q, \Sigma, \delta, q_0, a_0)$ as follows:

- $Q = V^{\bowtie} \times D$;
- $\Sigma = P \cup \{v' \mid \exists (v, v') \in E^{(s)}\} \cup \{v^e_{\texttt{main}}\}$;
- $a_0 = v^e_{\texttt{main}}$;
- $q_0 = (v^s_{\texttt{main}}, d_0)$; and
- δ is the smallest set such that
  - let $(v, v') \in E^{(o)}$, for every $u \in \Sigma$ and $d \in D$ we have $(((v, d), u), ((v', M(v, v')(d)), u, \epsilon)) \in \delta$
  - let $(v, v') \in E^{(s)}$ be an synchronous call edge to procedure $q \in P$ for every $u \in \Sigma$ and $d \in D$ we have
    $$(((v, d), u), ((v^s_q, M(v, v^s_q)(d)), u \cdot v', \epsilon)) \in \delta$$
    $$(((v^e_q, d), v'), ((v', M(v^e_q, v')(d)), \epsilon, \epsilon)) \in \delta$$
  - let $(v, v') \in E^{(a)}$ be an asynchronous call edge to procedure $q \in P$, for every $u \in \Sigma$ and $d \in D$ we have
    $$(((v, d), u), ((v', d), u, q)) \in \delta$$
  - for every $q \in P$, $d \in D$
    $$(((v^e_{\texttt{main}}, d), q), ((v^s_q, M(v^e_{\texttt{main}}, v^s_q)(d)), v^e_{\texttt{main}}, \epsilon)) \in \delta$$
    $$(((v^e_q, d), v^e_{\texttt{main}}), ((v^e_{\texttt{main}}, M(v^e_q, v^e_{\texttt{main}})(d)), \epsilon, \epsilon)) \in \delta$$

3.3 Fair Termination

We now translate the fair termination property of an asynchronous program into the asynchronous automata model.

An infinite run is *a-fair* for a handler $a \in \Sigma$ iff for every state $(q, w, \mathbf{c})$ along this run, if $\mathbf{c}(a) > 1$ then eventually a handler dispatch transition of the form $(q_1, \epsilon, \mathbf{c}_1) \rightarrow_d (q_2, a, \mathbf{c}_2)$ where $\mathbf{c}_1(a) > 0$ and $\mathbf{c}_2 = \mathbf{c}_1[a \leftarrow \mathbf{c}(a) - 1]$ must occur. An infinite run is *fair* iff it is a-fair for all handlers $a \in \Sigma$. An asynchronous automaton *terminates* iff it has no infinite runs. An asynchronous automaton *terminates fairly* iff (i) all infinite runs are of the first form (hence there is no infinite run of the second form), and (ii) there is no infinite run of the first form that is fair.

The asynchronous program of Fig. 2, initialized with the `parse` handler, implements a simple request processor: The program is terminating: there is only one run, starting with `parse`, followed by `read`, and then by `send`. On the other hand, the program in Fig. 1(b) is fair terminating, but not terminating. There is an infinite run in which the scheduler always chooses `h1` over `h2`. This run is not `h2` fair, and hence not fair.

3.4 Recursion-free Asynchronous Automata

We now address one source of unboundedness: the potentially unbounded stack. Our first step will transform, using Parikh's Lemma, a given asynchronous automaton A into an "equivalent" *recursion-free* asynchronous automaton A' in which the stack height is always bounded by one. This equivalence ensures that A does have a (fair) infinite run of the first form iff A' does. Moreover, their reachable configurations coincide as long as we restrict our attention to those with an empty stack (Lem. 2).

An asynchronous automaton A is *recursion-free* if no handler step of A pushes on to the stack, that is, if $\delta \subseteq (Q \times \Sigma) \times (Q \times \Sigma_\epsilon \times \Sigma_\epsilon)$. Intuitively, in a recursion-free asynchronous automaton, no handler makes additional synchronous calls, and in particular, there is no recursion.

We now show that given an asynchronous automaton A, there exists a recursion-free asynchronous automaton A' that preserves the existence of (fair) infinite runs and such that $\mathsf{Reach}_\epsilon(A) = \mathsf{Reach}_\epsilon(A')$. Since we use a construction over finite languages, the transformation we present below preserves runs of the first form but eliminates runs of the second form. Proving that an asynchronous automaton has no infinite runs of the second form is a separate problem which is addressed in Sect. 4.4. Below we assume the reader is familiar with the definitions of pushdown automaton and finite state automaton (see [9] for further details).

LEMMA 2. *Let $A = (Q, \Sigma, \delta, q_0, a_0)$ be an asynchronous automaton. There exists an asynchronous automaton $A' = (Q', \Sigma, \delta', q'_0, a'_0)$ such that A' is recursion-free and such that $(q_0, a_0, \mathbf{0}) \rightarrow^*_s (q_1, \epsilon, \mathbf{c}_1) \rightarrow_d (q_1, z_1, \mathbf{c}'_1) \cdots \rightarrow^*_s (q_i, \epsilon, \mathbf{c}_i) \rightarrow_d (q_i, z_i, \mathbf{c}'_i)$ is a run of A iff it is a run of A'. In particular, $\mathsf{Reach}_\epsilon(A) = \mathsf{Reach}_\epsilon(A')$.*

PROOF: Given $A = (Q, \Sigma, \delta, q_0, a_0)$ we define a finite family of pushdown automata (PDAs) as follows. Define $\Theta \subseteq ((Q \times \Sigma \times \Sigma_\epsilon) \times (Q \times \Sigma^*))$ to be such that $((q, a, b), (q', w)) \in \Theta$ iff there exists $b \in \Sigma_\epsilon$ and $((q, a), (q', w, b)) \in \delta$. The family is given by

$$\{\mathcal{P}_j\}_{j \in J} = \{(Q, \Sigma, \Sigma, \Theta, z, q^i, \{q^f\}) \mid q^i, q^f \in Q,\ z \in \Sigma\} \ .$$

Each member of the family is such that the stack and input alphabets are given by Σ; z is the initial stack symbol; q^i and q^f are the initial and final locations. They share the same transition relation Θ. The only accepting configuration for each PDA is (q^f, ϵ).

From the family $\{\mathcal{P}_j\}_{j \in J}$ of PDAs we define the family $\{\mathcal{F}_j\}_{j \in J}$ where $\mathcal{F}_j = (Q_j, \Sigma, \delta_j, q^i_j, F_j)$ is a finite automaton such that

$$\mathsf{Parikh}(L(\mathcal{F}_j)) = \mathsf{Parikh}(L(\mathcal{P}_j)) \ . \tag{1}$$

By Lemma 1, we can effectively construct such a finite automaton for a given PDA.

Now let us define an asynchronous automaton $A' = (Q', \Sigma, \delta', q'_0, a'_0)$ such that

$$Q' = Q \cup \bigcup_{j \in J} \{Q_j \mid \mathcal{F}_j = (Q_j, \Sigma, \delta_j, q^i_j, F_j)\} \ .$$

Moreover, we define δ' to be the smallest set such that for each automaton $\mathcal{F}_j$ and the associated tuple q_1, z, q_2 we have

- $((q_1, z), (q^i_j, z, \epsilon)) \in \delta'$;
- $((q_j, z), (q'_j, z, a)) \in \delta'$ for each $(q_j, a, q'_j) \in \delta_j$;
- $((q^f, z), (q_2, \epsilon, \epsilon)) \in \delta'$ for each $q^f \in F_j$.

Finally we set $q'_0 = q_0$ and $a'_0 = a_0$.

It is easily seen from the above construction that A' is recursion-free. Then our result is proved by induction on the number i of times that a dispatch (viz., $\rightarrow_d$) occurs.
Case $i = 0$**:** $(q_0, a_0, \mathbf{0}) = (q'_0, a'_0, \mathbf{0})$ holds by definition of a'_0 and q'_0.
Case $i + 1$**:** In this case we consider a run of the form:

$$(q_0, a_0, \mathbf{0}) \rightarrow_s^* (q_1, \epsilon, \mathbf{c}_1) \rightarrow_d (q_1, z_1, \mathbf{c}'_1) \ldots (q_i, \epsilon, \mathbf{c}_i) \rightarrow_d (q_i, z_i, \mathbf{c}'_i) \rightarrow_s^* (q, \epsilon, \mathbf{c}) \ . \quad (2)$$

For the only-if direction, consider the run of A given by (2). By induction hypothesis we have a run in A' which successively visits configurations $(q_0, a_0, \mathbf{0}), (q_1, \epsilon, \mathbf{c}_1), \ldots, (q_i, \epsilon, \mathbf{c}_i)$. By the definition of the asynchronous automaton A', we have $(q_i, \epsilon, \mathbf{c}_i) \rightarrow_d (q_i, z_i, \mathbf{c}'_i)$. Now, let $\mathcal{P}_k$ be the PDA defined above for the triple (q_i, z_i, q). (1) shows that the finite automaton $\mathcal{F}_k$ is such that $\mathsf{Parikh}(L(\mathcal{F}_k)) = \mathsf{Parikh}(L(\mathcal{P}_k))$. By definition of A', we obtain that $(q_i, z_i, \mathbf{c}'_i) \rightarrow_s^* (q, \epsilon, \mathbf{c}_x)$. Moreover $\mathbf{c}_x = \mathbf{c}$ because of $\mathsf{Parikh}(L(\mathcal{F}_k)) = \mathsf{Parikh}(L(\mathcal{P}_k))$, hence the only if direction follows. The if direction can be proved using a similar reasoning. □

4. Decision Procedures

We will show that there is a straightforward encoding of recursion-free asynchronous automata as Petri Nets [28]. Hence, we reduce our fair termination problem on recursion-free asynchronous programs to an equivalent one on Petri Nets. Provided the given asynchronous automaton is recursion-free, the encoding into a Petri Net will be carried out in linear time.

4.1 Petri Nets

A *Petri Net* $N = (S, T, F, \mathbf{m}_0)$ consists of a finite set S of *places*, a finite set T of *transitions* disjoint from S, a *weight function* $F\colon (S \times T) \cup (T \times S) \mapsto \{0, 1\}$, and an *initial marking* $\mathbf{m}_0 \in \mathbb{N}^S$. A *marking* is a map from S to $\mathbb{N}$. For a marking $\mathbf{m}$ of N and $p \in S$, we say that, in $\mathbf{m}$, the place p contains $\mathbf{m}(p)$ *tokens*. For markings $\mathbf{m}, \mathbf{m}'$, we write $\mathbf{m} + \mathbf{m}'$ for the marking obtained by point wise addition of place contents. We write $\mathbf{m} \leq \mathbf{m}'$ if for all $p \in S$ we have $\mathbf{m}(p) \leq \mathbf{m}'(p)$ and write $\mathbf{m} < \mathbf{m}'$ if $\mathbf{m} \leq \mathbf{m}'$ and $\mathbf{m} \neq \mathbf{m}'$. The marking $\mathbf{0}$ maps every $a \in \Sigma$ to 0. The *size* of a Petri Net N is $|S| + |T|$ where $|\cdot|$ denotes the cardinality of a set.

The semantics of a Petri Net $N = (S, T, F, \mathbf{m}_0)$ is given as a labeled transition system (LTS) $(C, [\cdot\rangle, c_0)$ where the set C of states is the set of markings $\mathbb{N}^S$, the initial state $c_0 = \mathbf{m}_0$, and the labeled transition relation $[\cdot\rangle \subseteq C \times T \times C$ is defined as follows. A transition $t \in T$ is *enabled at* $\mathbf{m}$, written $\mathbf{m}\,[t\rangle$, if $\mathbf{m}(p) \geq F(p, t)$ for each $p \in S$. A transition t that is enabled at $\mathbf{m}$ can *fire*, yielding a marking $\mathbf{m}'$ such that $\mathbf{m}' = \mathbf{m} + \Delta(t)$ where $\Delta(t)$, the *effect of a transition* t, is a function from T to $\mathbb{N}^S$ given by $\Delta(t)(p) = F(t, p) - F(p, t)$ for every $p \in S$. In this case, we have $(\mathbf{m}, t, \mathbf{m}') \in [\cdot\rangle$, which we write as $\mathbf{m}\,[t\rangle\,\mathbf{m}'$.

Consider a path $\mathbf{m}_1\,[t_1\rangle\,\mathbf{m}_2\,[t_2\rangle \ldots [t_n\rangle\,\mathbf{m}_{n+1}$ in the LTS. Since each transition is deterministic, we can equally speak about the word that labels the path starting at $\mathbf{m}_1$, forgetting about the intermediate markings.

Using the LTS definition we naturally lift the enabledness and firing notion from transition to sequences (or set of sequences) of transitions as follows. Given $L \subseteq T^*$ and $w \in T^* \cup T^\omega$ we define:

$\mathbf{m}_1\,[L\rangle\,\mathbf{m}_2$ iff some $v \in L$ labels a path of LTS from $\mathbf{m}_1$ to $\mathbf{m}_2$

$\mathbf{m}_1\,[w\rangle$ iff w labels a path of LTS which starts in $\mathbf{m}_1$.

From the above definition we find that $\mathbf{m}$ is a reachable state iff $\mathbf{m}_0\,[T^*\rangle\,\mathbf{m}$ and the set of reachable states coincides with $\{\mathbf{m} \mid \mathbf{m}_0\,[T^*\rangle\,\mathbf{m}\}$.

Also we lift the effect Δ of a single transition to sequences (and sets of sequences) in a natural way: $\Delta(\epsilon) = \mathbf{0}$ and $\Delta(vu) = \Delta(v) + \Delta(u)$. So for every $w \in T^*$ such that $\mathbf{m}\,[w\rangle\,\mathbf{m}'$ we have $\mathbf{m}' = \mathbf{m} + \Delta(w)$. The converse, however, does not hold.

4.2 From Asynchronous Automata to Petri Nets

Let $A' = (Q, \Sigma, \delta, q_0, a_0)$ be a recursion-free asynchronous automaton, its associated Petri Net $N_{A'} = (\Gamma \cup \Sigma, T, F, \mathbf{m}_0)$ is defined as follows:

- $\Gamma = Q \times \Sigma_\epsilon$;
- $T = T_s \cup T_d$ (with $T_s \cap T_d = \emptyset$) and F are the smallest sets which satisfy:
  - T_s contains a transition t for each $((q, a), (q', b, c)) \in \delta'$ such that $F((q, a), t) = 1$ $F(t, (q', b)) = 1$ and $F(t, c) = 1$ if $c \in \Sigma$ ($F(t, x) = F(x, t) = 0$ elsewhere);
  - T_d is given by $\bigcup_{a \in \Sigma} T_d^a$ such that for $(q, \epsilon) \in \Gamma$ the set T_d^a contains a transition t such that $F((q, \epsilon), t) = 1$, $F(a, t) = 1$ and $F(t, (q, a)) = 1$ ($F(t, x) = F(x, t) = 0$ elsewhere);
- $\mathbf{m}_0$ is such that $\mathbf{m}_0(p) = 1$ if $p = (q_0, a_0)$ and 0 otherwise.

Notice that T is well defined since A' is recursion-free. Also we conclude from Σ and Q are finite that $\Gamma \cup \Sigma$ is finite and also T is finite as A' is recursion-free. In the sequel we often refer to a place $a \in \Sigma$ as the counter of pending calls to a. As in the case of asynchronous automaton, whenever a transition $t \in T_d^a$ fires, we say that a pending call to handler a is dispatched. The following lemma connects reachable configurations of an asynchronous automaton with the reachable markings of the associated Petri Net, and allows us to use algorithms on Petri Nets to solve decision questions on asynchronous automata.

LEMMA 3. *Let A' be a recursion-free asynchronous automaton and let $N_{A'} = (\Gamma \cup \Sigma, T, F, \mathbf{m}_0)$ be the associated Petri Net. (1) The size of $N_{A'}$ is linear in the size of A'. (2) Given the following relation between configurations of A' and markings of $N_{A'}$:*

$$\mathbf{m}_i(p) = \begin{cases} 1 & \text{if } p = (q, \epsilon) \\ \mathbf{c}_i(p) & \text{if } p \in \Sigma \\ 0 & \text{otherwise} \end{cases} \qquad \mathbf{m}'_i(p) = \begin{cases} 1 & \text{if } p = (q, z_i) \\ \mathbf{c}'_i(p) & \text{if } p \in \Sigma \\ 0 & \text{otherwise} \end{cases}$$

we have $(q_0, a_0, \mathbf{0}) \rightarrow_s^* (q_1, \epsilon, \mathbf{c}_1) \rightarrow_d (q_1, z_1, \mathbf{c}'_1) \ldots (q_i, \epsilon, \mathbf{c}_i) \rightarrow_d (q_i, z_i, \mathbf{c}'_i)$ is a run of A' iff $\mathbf{m}_0\,[T_s^*\rangle\,\mathbf{m}_1\,[T_d\rangle\,\mathbf{m}'_1\,[T_s^*\rangle \ldots \mathbf{m}_i\,[T_d\rangle\,\mathbf{m}'_i$ is a path in the LTS of $N_{A'}$.

We now translate the fair termination property from recursion-free asynchronous automata to Petri Nets as follows. Let A' be a recursion-free asynchronous automaton. As we say above, because the translation from an asynchronous automaton to a recursion-free one does not preserve infinite runs made of handler steps only, that is runs of $\rightarrow_s^\omega$, it is a separate problem to check that all infinite runs are of the first form. Assuming this, what remains to check is that, in A' there is no infinite run of the first form that is fair. Accordingly, on the associated Petri Net $N_{A'} = (\Gamma \cup \Sigma, T_d \cup T_s, F, \mathbf{m}_0)$ we check that there is no path in the LTS starting from $\mathbf{m}_0$ that is of the form $(T_s^* \cdot T_d)^\omega$ (i.e., of the first form) and also fair. We define fairness of an infinite path $\mathbf{m}_1\,[t_1\rangle\,\mathbf{m}_2\,[t_2\rangle \ldots [t_n\rangle\,\mathbf{m}_{n+1} \ldots$ as follows. An infinite path is *a-fair* for a handler $a \in \Sigma$ if for every marking $\mathbf{m}_j$ along this path, if $\mathbf{m}_j(a) > 1$ then a transition $\mathbf{m}_i\,[T_d^a\rangle\,\mathbf{m}_{i+1}$ for some $i > j$ must fire. An infinite path is *fair* if it is a-fair for all handlers $a \in \Sigma$.

4.3 Coverability Graph

Our decision procedure to check fair termination on a Petri Net N is based on the *coverability graph* of N [33]. We start with some definitions.

Let $\mathbb{N}_\omega = \mathbb{N} \cup \{\omega\}$. An *$\omega$-marking* for a finite set S of places is a mapping from S to $\mathbb{N}_\omega$. Intuitively, ω-markings extend markings, where ω represents an arbitrary large natural. The arithmetic on $\mathbb{N}_\omega$ is defined as follows $\omega \pm c = \omega = \omega + \omega$ for each $c \in \mathbb{N}$. Further, $c < \omega$ for each $c \in \mathbb{N}$ and $\omega \leq \omega$. (In the sequel, whenever ω occurs, it should be clear from the context if ω refers to the one used in markings or to the one used in ω-word or ω-language.) For a set X of ω-markings over S, the *downward closure* of X (written $\downarrow X$) is given by the set $\{\mathbf{m} \in \mathbb{N}^S \mid \exists \mathbf{m}' \in X \colon \mathbf{m} \leq \mathbf{m}'\}$.

Define Accel to be the *acceleration* function that takes as input a set M of ω-markings and an ω-marking $\mathbf{m}$ and returns an ω-marking such that for each $p \in S$ we have $\mathsf{Accel}(M, \mathbf{m})(p) = \omega$ if there exists $\mathbf{m}' \in M$ such that $\mathbf{m}' < \mathbf{m}$ and $\mathbf{m}'(p) < \mathbf{m}(p)$, and $\mathsf{Accel}(M, \mathbf{m})(p) = \mathbf{m}(p)$ otherwise.

Given a Petri Net $N = (S, T, F, \mathbf{m}_0)$, a coverability graph $\mathcal{G}(N)$ is a finite labeled transition system $(\mathsf{nodes}, [\![\cdot\rangle\!\rangle, \mathbf{m}_0)$ where the set of states nodes is a finite subset of ω-markings, $[\![\cdot\rangle\!\rangle \subseteq \mathsf{nodes} \times T \times \mathsf{nodes}$ is the labeled transition relation, and $\mathbf{m}_0$ is the initial state. In this section we often refer to an element of the transition relation $[\![\cdot\rangle\!\rangle$ of a coverability graph as a *edge*. Algorithm 1 shows an algorithm from [33, 34] which given a Petri Net N builds a coverability graph $\mathcal{G}(N)$. The algorithm constructs a coverability graph by a worklist-based algorithm. The set worklist contains the set of pairs of ω-markings $\mathbf{m}$ and transitions t enabled at $\mathbf{m}$ that are to be explored. The set log stores the set of ω-marking, transition pairs that have been explored already. The main loop of the algorithm (lines 5–15) iterates over the worklist, choosing a pair $(\mathbf{m}, t)$ and firing t from $\mathbf{m}$ to get $\mathbf{m}'$. If $\mathbf{m}'$ is not already in $\mathcal{G}(N)$, it is accelerated w.r.t. all states that can reach $\mathbf{m}$ (lines 9, 10), and the accelerated node is added to nodes. The transition relation is updated to reflect an edge from $\mathbf{m}$ to $\mathbf{m}'$ through transition t (line 13), and both log and worklist are updated.

Consider a path $\mathbf{m}_1 [\![t_1\rangle\!\rangle \mathbf{m}_2 [\![t_2\rangle\!\rangle \dots [\![t_n\rangle\!\rangle \mathbf{m}_{n+1}$ in $\mathcal{G}(N)$. Since Algorithm 1 consider every pair $(\mathbf{m}, t)$ at most once, we can equally speak about the word that labels the path starting at $\mathbf{m}_1$, forgetting about the intermediate markings. This is as for the LTS of N.

LEMMA 4. *[33] Given a Petri Net $N = (S, T, F, \mathbf{m}_0)$ Algorithm 1 always terminates and returns a coverability graph $\mathcal{G}(N) = (\mathsf{nodes}, [\![\cdot\rangle\!\rangle, \mathbf{m}_0)$ such that the following hold:*

1. $\downarrow \mathsf{nodes} = \downarrow \{\mathbf{m} \mid \mathbf{m}_0 [T^*\rangle \mathbf{m}\}$.
2. *Let v be a finite path of $\mathcal{G}(N)$ with $\Delta(v) \geq \mathbf{0}$. There exists $u \in T^*$ such that $\mathbf{m}_0 [uv\rangle$.*
3. *If $\mathbf{m} [w\rangle$ for some $\mathbf{m} \in \mathsf{nodes}$ and $w \in T^*$ then there exists a unique node $\mathbf{m}' \in \mathsf{nodes}$ such that $\mathbf{m} [\![w\rangle\!\rangle \mathbf{m}'$ holds and for each $p \in S \colon \mathbf{m}'(p) \neq \omega \rightarrow \mathbf{m}'(p) = \mathbf{m}(p) + \Delta(w)(p)$ (hence, $\{p \in S \mid \mathbf{m}(p) = \omega\} \subseteq \{p \in S \mid \mathbf{m}'(p) = \omega\}$).*

In the following, for a Petri Net N, we define $\mathcal{G}(N)$ to be the coverability graph computed by Algorithm 1.

4.4 Proving the Absence of Runs of the Second Form

We now show how to check the absence of runs of the second form, in which infinitely many handler steps take place consecutively. Intuitively, we have to prove that every handler that is called terminates. We start with some observations.

Given an asynchronous automaton $A = (Q, \Sigma, \delta, q_0, a_0)$, an infinite sequence of handler steps is enabled at a reachable configuration iff an infinite sequence of handler steps is enabled at a reachable configuration that follows a handler dispatch or q_0. Every infinite sequence of handler steps remains enabled when prefixed with a finite sequence of handler steps.

Then we observe that, as far as the handler steps are concerned, the values of the counters of pending calls are irrelevant. In fact, counters do not disable/enable handler steps. Hence we see that an infinite sequence of handler steps is enabled at a configuration $(q, w, \mathbf{c})$ iff the same sequence is enabled at every configuration $\{(q, w, \mathbf{c}') \mid \mathbf{c}' \in \mathbb{N}^\Sigma\}$. This shows that we can abstract away counters of pending calls for the check. Accordingly we obtain a model where the unique source of unboundedness is given by the stack. For this setting, the absence of infinite sequences (of handler steps) can be established by known model checking algorithms for pushdown systems [1, 36].

Our procedure to check for the absence of run of the second form uses one of the above algorithm to check for the absence of infinite sequence of handler steps starting from a finite set of configurations given by $\{(q_0, a_0)\} \cup (Q \times \{a \in \Sigma \mid \exists \mathbf{m} \in \mathcal{G}(N_{A'}) \text{ with } \mathbf{m}(a) > 0\})$ (counters are omitted for the above mentioned reason). Recall that in the above definition A' is the recursion-free asynchronous automaton of Lem. 2.

Algorithm 1: Coverability graph construction

Input: A Petri Net $N = (S, T, F, \mathbf{m}_0)$
Output: A coverability graph $\mathcal{G}(N) = (\mathsf{nodes}, [\![\cdot\rangle\!\rangle, \mathbf{m}_0)$

```
1  begin
2      nodes := {m_0}, [[·⟩⟩ := ∅
3      worklist := {(m_0, t) | m_0 [t⟩}
4      log := ∅
5      while worklist ≠ ∅ do
6          choose (m, t) ∈ worklist and remove from worklist
7          let m' = m + Δ(t)
8          if m' ∉ nodes then
9              Let M = {m'' ∈ nodes | m'' [[T*⟩⟩ m}
10             m' := Accel(M, m')
11             nodes := nodes ∪ {m'}
12         end
13         [[·⟩⟩ := [[·⟩⟩ ∪ {(m, t, m')}; log := log ∪ {(m, t)}
14         worklist := worklist ∪ {(m', t) ∉ log | m' [t⟩}
15     end
16     return G(N)
17 end
```

4.5 Termination

In what follows we define the decision procedure to check the fair termination of a Petri Net. We proceed in two steps. First we give a general decision procedure to check that there is no infinite path (fair or unfair) in the LTS of a Petri Net. Then we will extend this decision procedure to check that there is no *fair* infinite path in the LTS of $N_{A'}$ along which some $t \in T_d$ occurs infinitely often.

Our decision procedure for termination relies on the following lemma from [33] which reduces the existence of an infinite path in the LTS of a Petri Net N to the existence of a finite path in $\mathcal{G}(N)$.

LEMMA 5. *[33] Given a Petri Net $N = (S, T, F, \mathbf{m}_0)$, there exists $w \in T^\omega \colon \mathbf{m}_0 [w\rangle$ in the LTS of N iff in $\mathcal{G}(N)$ there exists a path v such that $\Delta(v) \geq \mathbf{0}$.*

We reduce the search for such a path in $\mathcal{G}(N)$ to the satisfiability of a quantifier-free Presburger formula. Our reduction is inspired by a similar encoding in [31]. Recall that existential Presburger formulas ϕ are defined by the following grammar and interpreted over natural numbers:

$$t ::= 0 \mid 1 \mid x \mid t_1 + t_2 \mid t_1 - t_2 \qquad \text{where } x \text{ is a variable}$$
$$\phi ::= t_1 = t_2 \mid t_1 > t_2 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi_1 \ .$$

$$\sum_{n \in \mathsf{nodes}} z_n = 1 \tag{3}$$

$$\sum_{\substack{n' \in \mathsf{nodes}, t \in T \\ n' [\![t]\!\rangle n}} x_{(n',t,n)} - \sum_{\substack{t \in T, n' \in \mathsf{nodes} \\ n [\![t]\!\rangle n'}} x_{(n,t,n')} = 0 \qquad \text{for } n \in \mathsf{nodes} \tag{4}$$

$$z_n > 0 \rightarrow u_n = 0 \qquad \text{for } n \in \mathsf{nodes} \tag{5}$$

$$\left(z_n = 0 \wedge \sum_{\substack{n' \in \mathsf{nodes}, t \in T \\ n' [\![t]\!\rangle n}} x_{(n',t,n)} > 0 \right) \leftrightarrow u_n \geq 1 \qquad \text{for } n \in \mathsf{nodes} \tag{6}$$

$$u_n \geq 1 \rightarrow \bigvee_{\substack{t \in T, n' \in \mathsf{nodes} \\ n [\![t]\!\rangle n' \text{ or } n' [\![t]\!\rangle n}} u_n > u_{n'} \geq 1 - z_{n'} \qquad \text{for } n \in \mathsf{nodes} \tag{7}$$

$$\sum_{\substack{n,n' \in \mathsf{nodes}, t \in T \\ n [\![t]\!\rangle n'}} x_{(n,t,n')} \geq 1 \tag{8}$$

$$\sum_{\substack{n,n' \in \mathsf{nodes}, t \in T \\ n [\![t]\!\rangle n'}} \Delta(t)(p) \times x_{(n,t,n')} \geq 0 \qquad \text{for } p \in S \tag{9}$$

Figure 3. Presburger constraints for termination

In the following we use standard abbreviations like $c \times x$ for $c \in \mathbb{N}$ for the c-times addition $x + \ldots + x$, $t_1 \geq t_2$ for $t_1 > t_2 \vee t_1 = t_2$, and $\phi_1 \rightarrow \phi_2$ for $\neg\phi_1 \vee \phi_2$.

The encoding as a Presburger formula is based on the notion of flows. A multigraph $M^* = (V, W, \odot)$ consists of a set V of nodes, a multiset W of edges, and a node $\odot \in V$. The multigraph M^* is *consistent* if every node of V has as many incoming edges as outgoing edges. The multigraph M^* is said to be *connected* if there exists an undirected path between every pair of distinct nodes of M^*. A *flow* is a consistent and connected multigraph. Consistency and connectedness imply that there exists a cycle in M^* from $\odot$ to itself which traverses each edge of W exactly once [31].

Given a coverability graph $\mathcal{G}(N) = (\mathsf{nodes}, [\![\cdot\rangle\!], \mathbf{m}_0)$ of a Petri Net N and $\odot \in \mathsf{nodes}$, we define flows $M^* = (V, W, \odot)$ of $\mathcal{G}(N)$ in which $V \subseteq \mathsf{nodes}$ and each edge $e \in (\mathsf{nodes} \times T \times \mathsf{nodes})$ of the multiset W belongs to $[\![\cdot\rangle\!]$. Because clearly a flow induces an Eulerian circuit, we know that a flow defines a (not necessarily simple) cycle in $\mathcal{G}(N)$. The converse also holds: each (not necessarily simple) cycle of $\mathcal{G}(N)$ from $\odot$ to itself corresponds to a flow $M^* = (V, W, \odot)$ where V is the set of nodes along the cycle and W is the multiset of all the edges along the cycle. So in what follows, we often equally speak of cycles and flows.

We now explain how to encode the set of flows of $\mathcal{G}(N) = (\mathsf{nodes}, [\![\cdot\rangle\!], \mathbf{m}_0)$ from $\odot$ to itself as a Presburger formula. In particular, the formula is such that the set of flows from $\odot$ to itself coincides with the set of its satisfying valuations.

Given a coverability graph $\mathcal{G}(N) = (\mathsf{nodes}, [\![\cdot\rangle\!], \mathbf{m}_0)$ of a Petri Net $N = (S, T, F, \mathbf{m}_0)$, we define Ψ_N to be the conjunction of the constraints in Figure 3. We now describe each constraint in the formula in more detail.

The formula Ψ_N uses the following set of variables. First, for each $n \in \mathsf{nodes}$, define the variables z_n which tracks from which node n the flow starts and ends. Second, for each edge $(n, t, n') \in [\![\cdot\rangle\!]$, define the weight variable $x_{(n,t,n')}$ which tracks the number of times the edge (n, t, n') occurs in the flow. Third, for each node $n \in \mathsf{nodes}$, the variable u_n gives a *rank* to node n, and ranks are used (as described below) to ensure connectedness.

The constraint (3) encodes the fact that exactly one of the z_n's valuates to 1 (viz., $z_\odot$), the others to 0. This allows the constraint to choose $\odot$ arbitrarily. Consistency of the flow is encoded by (4) which ensure that at each node, the sum of weights on the incoming edges is equal to the sum of weights on outgoing edges. To encode connectedness of the flow we use the rank variables u_n. The nodes of the flow coincide with the nodes with a positive rank (6), except for $\odot$ which is ranked 0 (5). Hence a node n belongs to the flow iff $u_n + z_n \geq 1$. Then we rely on the property that a flow is connected iff each of its nodes but $\odot$ is adjacent to some node that belongs to the flow and has a strictly lower rank. This encoding of this property is given by (7) which can be intuitively interpreted as follows. If node n belongs to the flow and $n \neq \odot$ (i.e., $u_n \geq 1$) then there exists an adjacent node n' (quantification is encoded by the disjunction) that belongs to the flow ($u_{n'} \geq 1$) and the rank of n' is strictly lower than the rank of n ($u_n > u_{n'}$). The constraint $1 - z_{n'}$ allows to deal with the case in which n is adjacent to $n' = \odot$. It follows that in the disjunct of (7) we have $u_n > u_\odot \geq 0$ (since $z_\odot = 1$) which holds for every positive value of u_n since $u_\odot = 0$ by (5).

Conjunct (9) ensures that the global effect of the selected sequence of transitions is of positive weight and (8) prohibits selecting the empty sequence. Note that $\Delta(t)(p)$ is a constant value.

Flows and Valuations. Consider a satisfying valuations of Ψ_N it naturally induces a flow (or equally a cycle). On the other hand consider a flow $M^* = (V, W, \odot)$ of $\mathcal{G}(N)$, M^* naturally translates to a valuation of the variables of Ψ_N as follows. The node $\odot$ gives which of the $\{z_n\}_{n \in \mathsf{nodes}}$ valuates to 1 else valuates to 0 and the multiset W of edges gives the valuation of the variables $\{x_e\}_{e \in [\![\cdot\rangle\!]}$ such that the edge e occurs c times in W iff x_e valuates to c. Since a flow is consistent and connected by definition, we can extend the valuation to the variables $\{u_n\}_{n \in \mathsf{nodes}}$ such that it satisfies the constraints (4), (5), (6), (7). Because of the above translation, we will sometimes say that a flow M^* of $\mathcal{G}(N)$ satisfies some Presburger formula built upon Ψ_N if the valuation corresponding to the flow M^* satisfies the Presburger formula. From these explanations, it can be seen that the following holds.

LEMMA 6. *Given a Petri Net $N = (S, T, F, \mathbf{m}_0)$ we have that Ψ_N is satisfiable iff there exists a cycle labeled by v in $\mathcal{G}(N)$ such that $\Delta(v) \geq \mathbf{0}$.*

We also have the following relation between cycles and paths in the coverability graph.

LEMMA 7. *Given a Petri Net $N = (S, T, F, \mathbf{m}_0)$ and $v \in T^+$ such that $\Delta(v) \geq \mathbf{0}$, there exists a path labeled by v in $\mathcal{G}(N)$ iff there exists a cycle labeled by v in $\mathcal{G}(N)$.*

PROOF: The if direction trivially follows from the fact that every cycle is also a path. For the only if direction, since v labels a path of $\mathcal{G}(N)$ and $\Delta(v) \geq \mathbf{0}$, by Lem. 4 (point 2), there exists $u \in T^*$ such that $\mathbf{m}_0 [u \cdot v\rangle$. Lemma 4 (point 3) shows that there exists a unique path in $\mathcal{G}(N)$ starting at $\mathbf{m}_0 \in \mathsf{nodes}$ and labeled by $u \cdot v$. Let $\mathbf{m}_1, \mathbf{m}_2 \in \mathsf{nodes}$ be such that $\mathbf{m}_0 [\![u]\!\rangle \mathbf{m}_1$ and $\mathbf{m}_1 [\![v]\!\rangle \mathbf{m}_2$. Since $\Delta(v) \geq \mathbf{0}$ we have that $\mathbf{m}_2 \geq \mathbf{m}_1$. We now prove that $\mathbf{m}_1 [v\rangle$.

We conclude from $\mathbf{m}_0 [u \cdot v\rangle$ that there exists a unique marking $\mathbf{m}'_1$ such that $\mathbf{m}_0 [u\rangle \mathbf{m}'_1$, hence $\mathbf{m}'_1 = \mathbf{m}_0 + \Delta(u)$ by definition of $[u\rangle$ and finally that $\mathbf{m}_1 \in \mathsf{nodes}$ is such that $\mathbf{m}_1 \geq \mathbf{m}'_1$ by Lem. 4 (point 3). By monotonicity of Petri Nets and $\mathbf{m}'_1 [v\rangle$ it follows that $\mathbf{m}_1 [v\rangle$.

Lemma 4 (point 3), $\mathbf{m}_1 [v\rangle$ and $\mathbf{m}_1 \in \mathsf{nodes}$ shows that there exists $\mathbf{m}_2 \in \mathsf{nodes}$ such that $\mathbf{m}_1 [\![v]\!\rangle \mathbf{m}_2$ and also that for every $p \in S$ such that $\mathbf{m}_2(p) \neq \omega$ we have $\mathbf{m}_2(p) = \mathbf{m}_1(p) + \Delta(v)(p)$ (hence $\{p \in S \mid \mathbf{m}_1(p) = \omega\} \subseteq \{p \in S \mid \mathbf{m}_2(p) = \omega\}$).

We conclude from $\Delta(v) \geq \mathbf{0}$ that $\mathbf{m}_2 \geq \mathbf{m}_1$, hence that $\mathbf{m}_2 [v\rangle$ by monotonicity of Petri Nets and finally that there exists $\mathbf{m}_3 \in \mathsf{nodes}$ such that $\mathbf{m}_2 [\![v]\!\rangle \mathbf{m}_3$ by Lem. 4 (point 3). By repeatedly applying the above reasoning we define a sequence of nodes of $\mathcal{G}(N)$ such that $\mathbf{m}_1 [\![v]\!\rangle \mathbf{m}_2 [\![v]\!\rangle \ldots \mathbf{m}_i [\![v]\!\rangle \mathbf{m}_{i+1} \ldots$. Moreover using Lem. 4 (point 3) and $\Delta(v) \geq \mathbf{0}$ we find that for every $i > 1$ we have $\mathbf{m}_{i+1} \geq \mathbf{m}_i$. Now using the observation that since $\mathcal{G}(N)$ is finite and thus that the finite set of natural values that appear in the nodes can be bounded by some $b \in \mathbb{N}$ we conclude that there exists j such that $\mathbf{m}_j = \mathbf{m}_{j+1}$ and so we have a cycle from $\mathbf{m}_j$ to $\mathbf{m}_{j+1}$ labeled by v. □

Finally Lem. 6 and Lem. 7 shows that the following corollary.

COROLLARY 1. *Given a Petri Net $N = (S, T, F, \mathbf{m}_0)$ we have: Ψ_N is satisfiable iff there exists $w \in T^\omega \colon \mathbf{m}_0 [w\rangle$ in the LTS of N.*

Since Presburger satisfiability is decidable, and each step in our construction from asynchronous automata to the Presburger formula is effective, we have that the termination problem is decidable. The problem is EXPSPACE-hard by reduction from the EXPSPACE-hardness for termination of simple programs [5, 19, 32].

THEOREM 1. **[Termination]** *The termination problem for asynchronous automata is decidable and EXPSPACE-hard.*

4.6 Fair Termination

Recall that for fair termination, the definition says that in the LTS of the Petri Net $N_{A'} = (\Gamma \cup \Sigma, T_s \cup T_d, F, \mathbf{m}_0)$ associated with the recursion-free asynchronous automaton A', there is no path that is of the form $(T_s^* \cdot T_d)^\omega$ which is also fair.

Let $\sigma \in (T_s^* \cdot T_d)^\omega$ be such that $\mathbf{m}_0 [\sigma\rangle$, we thus have the infinite path $\mathbf{m}_0 [\sigma_1\rangle \mathbf{m}_1 [\sigma_2\rangle \mathbf{m}_2 \ldots$ where $\sigma = \sigma_1 \sigma_2 \ldots$. By definition we have that σ is fair iff

$$\forall a \in \Sigma \, \forall i \geq 0 \colon \mathbf{m}_i(a) \geq 1 \rightarrow \exists j \geq i \colon \mathbf{m}_j [T_d^a\rangle \mathbf{m}_{j+1} \ . \quad (10)$$

Since $\sigma \in (T_s^* \cdot T_d)^\omega$ for every $i \geq 0$ such that $\mathbf{m}_i(a) \geq 1$ we have $\exists \ell \geq i \colon \mathbf{m}_i [T_s^*\rangle \mathbf{m}_\ell [T_d^a\rangle$.

Hence we have that (10) is equivalent to

$$\forall a \in \Sigma \, \forall i \geq 0 \colon \mathbf{m}_i [T_d^a\rangle \rightarrow \exists j \geq i \colon \mathbf{m}_j [T_d^a\rangle \mathbf{m}_{j+1} \ . \quad (11)$$

As for termination our solution to the fair termination problem checks whether some Presburger formula is satisfiable. The formula for fair termination is given by $\Psi_{N_{A'}}$ of Fig. 3 in conjunction with $\Phi_{N_{A'}}$ given by (12) ∧ (13) where:

$$u_n + z_n \geq 1 \rightarrow \sum_{\substack{n_1, n_2 \in \mathsf{nodes} \\ t \in T_d^a \\ n_1 [\![t]\!\rangle n_2}} x_{(n_1, t, n_2)} \geq 1 \quad (12)$$

for $a \in \Sigma$, $n \in \mathsf{nodes}$ such that $n [\![T_d^a]\!\rangle$.

Intuitively, we require that for each node n of $\mathcal{G}(N_{A'})$ with an outgoing dispatch a edge, if n occurs in the flow (given by $u_n + z_n \geq 1$) then make sure a dispatch to a occurs in the flow as well.

$$\sum_{\substack{n, n' \in \mathsf{nodes}, t \in T_d \\ n [\![t]\!\rangle n'}} x_{(n, t, n')} \geq 1 \ . \quad (13)$$

The above constraint requires that at least one dispatch occurs (note that this entails (8)) since $T_d \subseteq T$.

LEMMA 8. *Given a Petri Net $N_{A'} = (\Gamma \cup \Sigma, T_d \cup T_s, F, \mathbf{m}_0)$ let $M^* = (V, W, \odot)$ be the flow associated to a satisfying valuation for $\Psi_{N_{A'}} \wedge \Phi_{N_{A'}}$. Let $\mathbf{m} \in V$ be such that $\mathbf{m}(a) \geq 1$ for $a \in \Sigma$, then there is a $\mathbf{m}_1 [\![t]\!\rangle \mathbf{m}_2$ in $\mathcal{G}(N_{A'})$ such that $t \in T_d^a$ and $\mathbf{m}_1 [\![t]\!\rangle \mathbf{m}_2$ occurs in W.*

PROOF: We distinguish two cases: (i) each node $\mathbf{m}'$ of M^* is such that $\mathbf{m}'(a) \geq 1$ or (ii) some node $\mathbf{m}'$ of M^* is such that $\mathbf{m}'(a) = 0$. In case (i), constraints (13) shows that at least one dispatch transition $t \in T_d$ must occur in M^* which means by definition of t that some place $p \in \Gamma$ of the form $p = (q, \epsilon)$ is such that $\mathbf{m}''(p) \geq 1$ at some node $\mathbf{m}'' \in V$. Hence we find that there is $t_d^a \in T_d^a$ with $\mathbf{m}'' [t_d^a\rangle$ (both $\mathbf{m}''(p)$ and $\mathbf{m}''(a)$ contain tokens) and so that $\mathbf{m}'' [\![t_d^a]\!\rangle \mathbf{m}_x$ for some $\mathbf{m}_x$ holds by Lem. 4 (point 3). Finally $\mathbf{m}'' \in V$ shows that $u_{\mathbf{m}''} \geq 1$ and so the precondition of constraint (12) holds, thus some $t \in T_a^d$ must occur in M^*.

For the case (ii), i.e., some node $\mathbf{m}'$ of M^* is such that $\mathbf{m}'(a) = 0$, hypothesis $\mathbf{m}(a) \geq 1$ for some node $\mathbf{m}$ of M^* shows that $\mathbf{m}$ and $\mathbf{m}'$ are distinct. Furthermore observe that $\mathbf{m}(a) \in \mathbb{N} \setminus \{0\}$ because we cannot get to 0 if we were at ω as shown by Lem. 4 (point 3). Moreover since $\mathbf{m}$ is a node of M^* we have that either (a) $\mathbf{m}$ occurs before $\mathbf{m}'$ in M^* or (b) $\mathbf{m}'$ occurs before $\mathbf{m}$ in M^*. In case (a) some $t \in T_d^a$ must occur in M^* between node $\mathbf{m}$ and $\mathbf{m}'$ which concludes the proof. In case (b) we find that a call to handler $a \in \Sigma$ must be posted between $\mathbf{m}'$ and $\mathbf{m}$. Because the flow starts and ends in $\odot$ some $t \in T_d^a$ must occur in M^* either between $\mathbf{m}$ and $\odot$ or between $\odot$ and $\mathbf{m}'$. □

We next show that the LTS of $N_{A'}$ has an infinite path of the form $(T_s^* \cdot T_d)^\omega$ that is fair iff the Presburger formula $\Psi_{N_{A'}} \wedge \Phi_{N_{A'}}$ is satisfiable.

PROPOSITION 1. *Given a Petri Net $N_{A'} = (\Gamma \cup \Sigma, T_d \cup T_s, F, \mathbf{m}_0)$ we have there exists $\sigma \in (T_s^* \cdot T_d)^\omega$ such that $\mathbf{m}_0 [\sigma\rangle$ and σ satisfies (11) iff there is a flow $M^* = (V, W, \odot)$ in $\mathcal{G}(N_{A'})$ which satisfies $\Psi_{N_{A'}} \wedge \Phi_{N_{A'}}$.*

PROOF: → By hypothesis, there exists $\sigma \in (T_s^* \cdot T_d)^\omega$ such that $\mathbf{m}_0 [\sigma\rangle$ such that σ satisfies (11), that is, $\forall a \in \Sigma \forall i \geq 0 \colon \mathbf{m}_i [T_d^a\rangle \rightarrow \exists j \geq i \colon \mathbf{m}_j [T_d^a\rangle \mathbf{m}_{j+1}$.

Our next step is to split σ into $\tau_0\tau_1\ldots\tau_n\ldots$ where each $\tau_i \in (T_s^* \cdot T_d)^*$. Let $\mathbf{m}_{I_i}$ be such that $\mathbf{m}_0\,[\tau_0\ldots\tau_{i-1}\rangle\,\mathbf{m}_{I_i}$, the split is such that for each $a \in \Sigma$:

$$\nexists \mathbf{m}\colon \mathbf{m}_{I_i}\left[\{w \mid \exists w' \in T^*\colon ww' = \tau_i\}\right\rangle \mathbf{m} \text{ and } \mathbf{m}\,[T_d^a\rangle \text{ or } \tau_i \cap T_d^a \neq \emptyset \ . \quad (14)$$

This intuitively says that along τ_i for each $a \in \Sigma$ we have that no T_d^a is enabled τ_i or some T_d^a occurs. Splitting this way is possible because for each $a \in \Sigma$ we have that either (i) there are no pending call at the beginning of τ_i and none is posted along τ_i or (ii) there are no pending call at the beginning of τ_i but eventually some call to a is posted along τ_i in which case τ_i must contain a T_d^a which exists by (11) or (iii) there are pending calls to a at the beginning of τ_i and so τ_i must contain a T_d^a which exists by (11). Finally, we show that each τ_i is finite by contradiction, suppose there is a τ_i that is infinite: $\tau_i \in (T_s^* \cdot T_d)^\omega$. So we have $\sigma = \tau_0\ldots\tau_i$. Since by hypothesis σ satisfies (11) we find that for each $a \in \Sigma$ either no transition of T_d^a is enabled along τ_i or that whenever some T_d^a is enabled eventually some transition of T_d^a fires, that is $\tau_i \cap T_d^a \neq \emptyset$ for $\tau_i \in (T_s^* \cdot T_d)^*$. Hence a contradiction.

Now using Dickson's Lemma over the infinite sequence $\mathbf{m}_{I_0}, \mathbf{m}_{I_1}, \ldots, \mathbf{m}_{I_n}, \ldots$ of markings given by $\mathbf{m}_0\,[\tau_0\rangle\,\mathbf{m}_{I_0}\,[\tau_1\rangle\,\mathbf{m}_{I_1}\ldots\mathbf{m}_{I_{n-1}}\,[\tau_n\rangle\,\mathbf{m}_{I_n}\ldots$ we find that there exists $i > j$ such that $\mathbf{m}_{I_j} \leq \mathbf{m}_{I_i}$.

Let $u = \tau_0\ldots\tau_j$, and let $v = \tau_{j+1}\ldots\tau_i$. By definition of the τ's we find that $u \cdot v^\omega \in (T_s^* \cdot T_d)^\omega$ and so $v^\omega \in (T_s^* \cdot T_d)^\omega$.

Next we show that $u \cdot v^\omega$ satisfies (11). Since $v^\omega \in (T_s^* \cdot T_d)^\omega$ we find that v repeatedly enables some T_d^a for every $a \in \Sigma$ for which there is a pending call. So, if some transition of T_d^a is enabled somewhere along τ, some transition t of T_d^a must fire in τ because of (14). Remark also that because v is repeated infinitely often, we have that t eventually fires (if not in the same τ, in its next occurrence) which shows that $u \cdot v^\omega$ satisfies (11).

It follows that $\mathbf{m}_0\,[u \cdot v^\omega\rangle$ ($\mathbf{m}_0\,[\sigma\rangle$ and there is $w \in T^\omega$ such that $\sigma = u \cdot v \cdot w$ shows that $\mathbf{m}_0\,[u \cdot v\rangle$. Moreover $\mathbf{m}_0\,[u \cdot v^i\rangle$ for every $i \geq 1$ since $\Delta(v) \geq \mathbf{0}$).

We conclude from $\mathbf{m}_0\,[u \cdot v\rangle$ and Lem. 4 that there is a path in $\mathcal{G}(N_{A'})$ labeled by v, hence that there is a cycle (or equally a flow) in $\mathcal{G}(N_{A'})$ labeled by v because of $\Delta(v) \geq \mathbf{0}$ and Lem. 7.

The proof ends by showing that the cycle labeled by v (or equally a flow) satisfies $\Psi_{N_{A'}} \wedge \Phi_{N_{A'}}$.

- (12) follows because $u \cdot v^\omega$ satisfies (11);
- (13) follows because $u \cdot v^\omega \in (T_s^* \cdot T_d)^\omega$;
- (3), (4), (5), (6) (7) of $\Psi_{N_{A'}}$ follows because v labels a flow;
- (8) of $\Psi_{N_{A'}}$ is entailed by (13) as we said before;
- (9) of $\Psi_{N_{A'}}$ follows by $\Delta(v) \geq \mathbf{0}$.

$\leftarrow$ Let $v \in T^*$ be a word corresponding to the flow $M^* = (V, W, \odot)$. Since $\Delta(v) \geq \mathbf{0}$, Lem. 4 (point 2) shows that there exists $u \in T^*$ such that $\mathbf{m}_0\,[uv\rangle$. Then we show that the following facts hold:

- $\mathbf{m}_0\,[u \cdot v^\omega\rangle$ (which follows by (9) that says: $\Delta(v) \geq \mathbf{0}$);
- $u \cdot v^\omega \in (T_s^* \cdot T_d)^\omega$ (by (13) that says: at least one T_d occurs in v);
- $u \cdot v^\omega$ satisfies (11). In fact suppose by contradiction that $u \cdot v^\omega$ does not satisfies (11) that is: $\exists a \in \Sigma\,\exists i \geq 0\colon \mathbf{m}_i\,[T_d^a\rangle \wedge (\forall j \geq i\colon \mathbf{m}_j\,[T_d^a\rangle\,\mathbf{m}_{j+1}$ does not hold) Since $u \cdot v^\omega \in (T_s^* \cdot T_d)^\omega$ and by definition of T_d we can assume that i is such that $i \geq |u|$. Hence, $\exists i \geq |u|\colon \mathbf{m}_i\,[T_d^a\rangle$ is true shows that $\mathbf{m}_i(a) \geq 1$ and so by Lem. 8 we find that there is an edge $\mathbf{m}'\,[\![t]\!\rangle\,\mathbf{m}''$ of $\mathcal{G}(N_{A'})$ such that $t \in T_d^a$ and t occurs in W, hence a contradiction since no $t \in T_d^a$ is supposed to occur in W by $(\forall j \geq i\colon \mathbf{m}_j\,[T_d^a\rangle\,\mathbf{m}_{j+1}$ does not hold).

Hence we have $\mathbf{m}_0\,[u \cdot v^\omega\rangle$ is a fair infinite path in the LTS. □

To sum up, the overall decision procedure to check fair termination combines the above steps: first, translate an asynchronous automaton A into a recursion-free automaton A'; second, construct the associated Petri Net $N_{A'}$; third, construct a coverability graph $\mathcal{G}(N_{A'})$; and fourth, construct and check if the Presburger formula $\Phi_{N_{A'}} \wedge \Psi_{N_{A'}}$ is satisfiable. The decision procedure also separately checks that every handler reachable in the coverability graph is terminating using the algorithm in Section 4.4. The EXPSPACE-hardness of the problem again follows from the reduction from simple programs.

THEOREM 2. **[Fair Termination]** *The fair termination problem is decidable for asynchronous automata and EXPSPACE-hard.*

4.7 Fair Non-Starvation

Recall that an asynchronous automaton is fair non-starving if (i) every execution of a handler that is called (synchronously or asynchronously) terminates, and (ii) along every infinite fair execution no handler is starved. A starving handler corresponds to a particular pending call that is never dispatched, thus that waits forever to be executed. Formally, an asynchronous automaton is fair non-starving if (i) there is no infinite run of the second form and (ii) for every fair infinite run $(q_0, a_0, \mathbf{0}) \rightarrow_s^* (q_1, \epsilon, \mathbf{c}_1) \rightarrow_d (q_1, a_1, \mathbf{c}_1') \ldots$ of the first form, for every handler $a \in \Sigma$, we have $\mathbf{c}_i(a) = 0$ for infinitely many $i \geq 0$. As before, we check (i) separately using the decision procedures of Sect. 4.4. We focus on the decision procedure for (ii).

The decision procedure for fair non-starvation is similar to the one for fair termination, the two decision procedure differs when it comes to the Presburger formula. The Presburger formula for fair non-starvation is stronger than the formula for fair termination, and obtained by adding constraints to the Presburger formula for fair termination.

Our encoding is based on the observation that if the asynchronous automaton does not satisfy the fair non-starvation there exists an infinite run $(q_0, a_0, \mathbf{0}) \rightarrow_s^* (q_1, \epsilon, \mathbf{c}_1) \rightarrow_d (q_1, a_1, \mathbf{c}_2) \ldots$ that is fair and for some $a \in \Sigma$ along this path some pending call to a is never dispatched (or equally $\exists i \geq 0\,\forall j \geq i\colon \mathbf{c}_j(a) > 0$).

We now define the Presburger formula $\Upsilon_{N_{A'}}$ the variables of which is given by those of $\Phi_{N_{A'}}$ and $\Psi_{N_{A'}}$ together and the set $\{w_{\mathbf{m}}^a \mid \mathbf{m} \in \mathsf{nodes} \wedge a \in \Sigma\}$. Define a constraint for each variable $w_{\mathbf{m}}^a$ such that $w_{\mathbf{m}}^a$ is positive iff in the ω-marking $\mathbf{m}$ the counter of pending calls to a is positive: $w_{\mathbf{m}}^a \geq 1$ if $\mathbf{m}(a) \geq 1$ and $w_{\mathbf{m}}(a) = 0$ otherwise. The formula $\Upsilon_{N_{A'}}$ is defined as the conjunction of the above constraints with:

$$\bigvee_{a \in \Sigma} \bigwedge_{n \in \mathsf{nodes}} \left(u_n + z_n \geq 1 \rightarrow w_n^a \geq 1\right) \ .$$

If $\Phi_{N_{A'}} \wedge \Psi_{N_{A'}} \wedge \Upsilon_{N_{A'}}$ is satisfiable we have that there exists $w \in T^\omega$ such that $\mathbf{m}_0\,[w\rangle$ is a fair path in the LTS of $N_{A'}$ (given by $\Phi_{N_{A'}} \wedge \Psi_{N_{A'}}$) and along w some counter of pending calls never gets null (given by $\Upsilon_{N_{A'}}$). That means, in the asynchronous automaton A', that some pending call might never be dispatched. Hence we have that if $\Phi_{N_{A'}} \wedge \Psi_{N_{A'}} \wedge \Upsilon_{N_{A'}}$ is satisfiable then some infinite path is fair and along this path some pending call is never dispatched.

THEOREM 3. **[Fair non-starvation]** *Fair non-starvation is decidable for asynchronous automata and EXPSPACE-hard.*

5. Alternative Proofs for Fair Termination

Our reduction to Petri Nets enables the use of powerful algorithmic techniques on Petri Nets to be used to analyze asynchronous programs. Indeed, we now show two alternate proofs for the decidability of fair termination by encoding the problem into more expressive logics whose model checking question remains decidable on Petri Nets.

Path Logics [37]. We recall a class of path formulas (from [37]) for which the model checking problem is decidable. Fix a Petri Net $N = (S, T, F, \mathbf{m}_0)$. We define a subset of the logic of [37] that we need to give an alternative decision procedure for fair termination. Let $\mu_1, \mu_2, \ldots$ be a family of *marking variables* ranging over markings and $\sigma_1, \sigma_2, \ldots$ a family of *transition sequence variables* ranging over T^*. A *basic predicate* is either a *marking predicate* of the form $\mu_j \geq \mu_i$ where μ_j and μ_i with $j > i$ are marking variables or markings, or a *transition predicate* of the form $y \otimes \mathsf{Parikh}(\sigma) \geq c$ or $y \otimes \mathsf{Parikh}(\sigma) = c$, where σ is a transition variable, $c \in \mathbb{N}$ is a constant, y is a vector of integers of dimension $|T|$, and $\otimes$ denotes the inner product (i.e., $(a_1, \ldots, a_k) \otimes (b_1, \ldots, b_k) = \sum_{i=1}^{k} a_i \times b_i$). A *predicate* is either a marking predicate, a transition predicate, or of the forms $P_1 \vee P_2$ or $P_1 \wedge P_2$ where P_1 and P_2 are predicates. A *path formula* is a formula of the form

$$\exists \mu_1, \ldots, \mu_m \colon \exists \sigma_1, \ldots, \sigma_m \colon \Big(\mathbf{m}_0 \xrightarrow{\sigma_1} \mu_1 \xrightarrow{\sigma_2} \ldots \mu_{m-1} \xrightarrow{\sigma_m} \mu_m\Big) \wedge \Phi(\mu_1, \ldots, \mu_m, \sigma_1, \ldots, \sigma_m)$$

where Φ is a predicate. The *model checking problem* for such a path formula asks if there exists a path in the LTS of N of the form $\mathbf{m}_0 [\sigma_1\rangle \mathbf{m}_1 [\sigma_2\rangle \ldots \mathbf{m}_{m-1} [\sigma_m\rangle \mathbf{m}_m$ for markings $\mathbf{m}_1, \ldots, \mathbf{m}_m$ and transition sequences $\sigma_1, \ldots, \sigma_m \in T^*$, such that $\Phi(\mathbf{m}_1, \ldots, \mathbf{m}_m, \sigma_1, \ldots, \sigma_m)$ is true. In this case we say N satisfies the path formula.

Let $N_{A'} = (\Gamma \cup \Sigma, T_s \cup T_d, F, \mathbf{m}_0)$ be the Petri Net associated with a recursion-free asynchronous automaton A'. Let $r = |T|$. We define a path formula such that $N_{A'}$ satisfies the path formula iff there is an infinite fair path in the LTS of $N_{A'}$:

$$\exists \mu_1, \mu_2 \colon \exists \sigma_1, \sigma_2 \colon \mathbf{m}_0 \xrightarrow{\sigma_1} \mu_1 \xrightarrow{\sigma_2} \mu_2 \bigwedge 1^r \otimes \mathsf{Parikh}(\sigma_2) \geq 1 \wedge \mu_2 \geq \mu_1 \bigwedge_{a \in \Sigma} \Big(z_a \otimes \mathsf{Parikh}(\sigma_2) = 0 \rightarrow \big((x_a - z_a) \otimes \mathsf{Parikh}(\sigma_1) = 0 \wedge x_a \otimes \mathsf{Parikh}(\sigma_2) = 0\big)\Big) \ .$$

where z_a and x_a are two vectors of integers of dimension r such that

$$z_a(t) = \begin{cases} 1 & \text{if } t \in T_d \wedge F(a,t) = 1 \\ 0 & \text{otherwise} \end{cases} \qquad \text{and}$$

$$x_a(t) = \begin{cases} 1 & \text{if } t \in T_s \wedge F(t,a) = 1 \\ 0 & \text{otherwise} \ . \end{cases}$$

In the above path formula, the first two lines intuitively say that we want to find a path $\mathbf{m}_0 [\sigma_1 \cdot \sigma_2\rangle$ in the LTS of $N_{A'}$ where $\sigma_2 \neq \epsilon$ and $\mu_2 \geq \mu_1$. The monotonicity property of Petri Nets and $\mu_2 \geq \mu_1$ shows that $\mathbf{m}_0 [\sigma_1 \cdot \sigma_2^\omega\rangle$. Intuitively the first two lines require that $\sigma_1 \cdot \sigma_2^\omega \in T^\omega$ is an infinite path of the LTS of $N_{A'}$. Let us now turn to the fairness.

The fairness constraint of the formula requires the path $\mathbf{m}_0 [\sigma_1 \cdot \sigma_2^\omega\rangle$ to be such that for each handler $a \in \Sigma$ if no T_d^a occurs infinitely often ($z_a \otimes \mathsf{Parikh}(\sigma_2) = 0$) then after firing σ_1 and before firing σ_2 there is no pending call to a ($(x_a - z_a) \otimes \mathsf{Parikh}(\sigma_1) = 0$) and no call to a is posted along σ_2 $x_a \otimes \mathsf{Parikh}(\sigma_2) = 0$.

Above we defined an infinite path to be fair if it is a-fair for every $a \in \Sigma$. An infinite path is a-fair iff for every marking $\mathbf{m}_j$ along this path, if $\mathbf{m}_j(a) \geq 1$ then a transition $\mathbf{m}_i [T_d^a\rangle \mathbf{m}_{i+1}$ for some $i \geq j$ must occur. If the infinite paths we consider are of the form $\sigma_1 \cdot \sigma_2^\omega$ (we have shown above this is not restrictive) then a path is a-fair iff there exists $\sigma_1 \in T^*$ and $\sigma_2 \in T^+$ such that some T_d^a occurs in σ_2 or if the following condition holds: there is no pending call to a after firing σ_1 and no call to a is posted along σ_2. This last condition intuitively says that after firing σ_1, it always holds that there is no pending call to a, i.e., $\mathbf{m}_j(a) = 0$ for every marking $\mathbf{m}_j$ in the sequence after firing σ_1.

Temporal Logic Model Checking [10]. We recall a class of formulas $L(GF)$ (from [10]) for which the model checking problem is decidable. Given a Petri Net $N = (S, T, F, \mathbf{m}_0)$, we define the language $L(GF)$ as follows:

- *atomic formulas (predicates)* are $ge(p, c)$ and $fi(t)$ where $p \in S$, $t \in T$, $c \in \mathbb{N}$ with the following interpretation: for any infinite path $\sigma = \mathbf{m}_1 [u_1\rangle \mathbf{m}_2 [u_2\rangle \ldots$ in the LTS of N and for any $n \in \mathbb{N}$,
$$\langle N, \sigma, n\rangle \models ge(p, c) \text{ iff } \mathbf{m}_n(p) \geq c$$
$$\langle N, \sigma, n\rangle \models fi(t) \text{ iff } u_{n+1} = t \ .$$
- *formulas* are either *literals*, i.e., atomic formulas or their negations ($ge(p, c)$, $\neg ge(p, c)$, $fi(t)$, $\neg fi(t)$), or of the form $GF\ f$, $f_1 \wedge f_2$, $f_1 \vee f_2$ where f, f_1, f_2 are formulas. $GF\ f$ (it is always true that f will hold in future) can be formally defined as follows:
$$\langle N, \sigma, n\rangle \models GF\ f \text{ iff } \forall i \geq n\, \exists j \geq i \colon \langle N, \sigma, j\rangle \models f \ .$$
The rest of the interpretation is natural.

Given $N_{A'} = (\Gamma \cup \Sigma, T_s \cup T_d, F, \mathbf{m}_0)$ the Petri Net associated with a recursion-free asynchronous automaton A' we now define a formula such that there is an infinite path in the LTS of $N_{A'}$ satisfying the formula iff there is an infinite path in the LTS of $N_{A'}$ that is fair:

$$\bigwedge_{a \in \Sigma} GF\Big(\neg ge(a, 1) \vee \bigvee_{t \in T_d^a} fi(t)\Big) \ .$$

The formula intuitively says that along every path in the LTS of N_A, for every handler a, if there is a pending call to a then there is an eventual dispatch to a.

While these produce alternate proofs of decidability, we believe that our presentation of the problem is most amenable to an implementation (the model checking algorithm from [37] unrolls the transition relation a doubly exponential number of steps, and the model checking algorithm of [10] reduces to Petri Net reachability, and both have a computability-theoretic rather than practical value).

6. Boundedness

The reduction of Lem. 3 shows that to every recursion-free asynchronous automaton, there is a Petri Net with the "same" set of reachable states. This allows algorithmic techniques from the Petri Net literature to be applied to asynchronous automata. We show how these techniques can be used to prove *boundedness* properties.

An asynchronous automaton is *bounded* if there exists $N \in \mathbb{N}$ such that for every reachable configuration $(q, w, \mathbf{c})$, for every $a \in \Sigma$ we have $\mathbf{c}(a) \leq N$. The *boundedness problem* takes as input an asynchronous automaton, and asks if it is bounded. (Note that boundedness is a safety property.)

We can decide the boundedness problem for recursion-free asynchronous automaton by deciding the boundedness problem on the associated Petri Net. This problem is defined as follows. Given a Petri Net $N = (S, T, F, \mathbf{m}_0)$, we say N is bounded if there exists $n \in \mathbb{N}$ such that for every reachable marking $\mathbf{m}$, for every place $p \in S$ we have $\mathbf{m}(p) \leq n$. The *boundedness problem* takes as input a Petri Net, and asks if it is bounded.

Let A' be a recursion-free asynchronous automaton and $N_{A'} = (\Gamma \cup \Sigma, T, F, \mathbf{m}_0)$ be the associated Petri Net, Lem. 3 shows that A' is bounded iff $N_{A'}$ is bounded. Furthermore, the Petri Net $N_{A'}$ is bounded iff for each $\mathbf{m} \in \mathsf{nodes}$ of any coverability graph $\mathcal{G}(N_{A'})$ we have $\mathbf{m} \in \mathbb{N}^P$. The reason why a coverability graph is precise enough to check for boundedness is given by Lem. 4 (point 1).

Since the size of the Petri Net is linear in the size of the recursion-free asynchronous automaton, which can be exponential in the size of the original asynchronous automaton, and the search for unbounded executions in the coverability graph can be stopped after a number of steps that is doubly exponential in the size of the Petri Net [26], we get a triply exponential algorithm. The hardness of the algorithm again follows from a reduction from simple programs to asynchronous automata.

THEOREM 4. **[Boundedness]** *The boundedness problem is decidable for asynchronous automata. It is EXPSPACE-hard.*

We note that boundedness is neither a necessary nor a sufficient condition for fair termination. On the one hand, the asynchronous program with one handler `h` which posts itself is bounded but not terminating. On the other hand, the asynchronous program from Fig. 1(b) is fair terminating but unbounded. For each n, there is a fair terminating run which dispatches the handler `h1` n times before dispatching `h2`.

References

[1] O. Burkart and B. Steffen. Pushdown processes: Parallel composition and model checking. In *CONCUR '94*, volume 836 of *LNCS*, pages 98–113. Springer, 1994.

[2] R. Chadha and M. Viswanathan. Decidability results for well-structured transition systems with auxiliary storage. In *CONCUR '07*, volume 4703 of *LNCS*, pages 136–150. Springer, 2007.

[3] P. Chandrasekaran, C.L. Conway, J.M. Joy, and S.K. Rajamani. Programming asynchronous layers with CLARITY. In *ESEC/SIGSOFT FSE*, pages 65–74. ACM, 2007.

[4] R. Cunningham and E. Kohler. Making events less slippery with Eel. In *HotOS-X*, 2005.

[5] J. Esparza and M. Nielsen. Decidability issues for Petri nets - a survey. *Journal of Informatik Processing and Cybernetics*, 30(3):143–160, 1994.

[6] J. Fischer, R. Majumdar, and T. Millstein. Tasks: Language support for event-driven programming. In *PEPM '07*. ACM, 2007.

[7] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI '03*, pages 1–11. ACM, 2003.

[8] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *ASPLOS'00*, pages 93–104. ACM, 2000.

[9] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Addison Wesley, 2000.

[10] P. Jančar. Decidability of a temporal logic problem for petri nets. *Theoretical Computer Science*, 74(1):71–93, 1990.

[11] R. Jhala and R. Majumdar. Interprocedural analysis of asynchronous programs. In *POPL '07*, pages 339–350. ACM, 2007.

[12] R.M. Karp and R.E. Miller. Parallel program schemata. *Journal of Comput. Syst. Sci.*, 3(2):147–195, 1969.

[13] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M.F. Kaashoek. The Click modular router. *ACM TOCS*, 18(3):263–297, 2000.

[14] M. Krohn, E. Kohler, and M.F. Kaashoek. Events can make sense. In *USENIX Annual Technical Conference*, 2007.

[15] D.J. Lehmann, A. Pnueli, and J. Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In *ICALP '81*, volume 115 of *LNCS*, pages 264–277. Springer, 1981.

[16] P. Li and S. Zdancewic. Combining events and threads for scalable network services. In *PLDI: Programming Languages Design and Implementation*, pages 189–199. ACM, 2007.

[17] Libasync. http://pdos.csail.mit.edu/6.824-2004/async/.

[18] Libevent. http://www.monkey.org/∼provos/libevent/.

[19] R. Lipton. The reachability problem is exponential-space hard. Technical Report 62, Department of Computer Science, Yale University, 1976.

[20] Z. Manna and A. Pnueli. Temporal verification of reactive systems: Progress. Draft, 1996.

[21] P. Manolios and D. Vroon. Termination analysis with calling context graphs. In *CAV '06*, volume 4144 of *LNCS*, pages 401–414. Springer, 2006.

[22] M. Minsky. *Finite and Infinite Machines*. Prentice-Hall, 1967.

[23] V.S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *Proc. USENIX Tech. Conf.*, pages 199–212. Usenix, 1999.

[24] R.J. Parikh. On context-free languages. *Journal of the ACM*, 13(4):570–581, 1966.

[25] M. Presburger. Über die vollständigkeit einer gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervortritt. In *Comptes Rendus du I congrès de Mathématiciens des Pays Slaves*, pages 92–101. 1929.

[26] C. Rackoff. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science*, 6(2):223–231, 1978.

[27] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM TOPLAS*, 22(2):416–430, 2000.

[28] W. Reisig. *Petri nets: An introduction*. Springer, 1986.

[29] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95*, pages 49–61. ACM, 1995.

[30] A. Rybalchenko. *Temporal Verification with Transition Invariants*. PhD thesis, Universität des Saarlandes, 2004.

[31] H. Seidl, A. Muscholl, T. Schwentick, and P. Habermehl. Counting in trees for free. In *ICALP'04*, volume 3142 of *LNCS*, pages 1136–1149. Springer, 2004.

[32] K. Sen and M. Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. In *CAV '06*, volume 4144 of *LNCS*, pages 300–314. Springer, 2006.

[33] R. Valk and M. Jantzen. The residue of vector sets with applications to decidability problems in Petri nets. *Acta Informatica*, 21:643–674, 1985.

[34] R. Valk and G. Vidal-Naquet. Petri nets and regular languages. *Journal of Computer and System Sciences*, 23(3):299–325, 1981.

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

[36] I. Walukiewicz. Pushdown Processes: Games and Model-Checking. *Information and Computation*, 164(2):234–263, 2001.

[37] H.-C. Yen. A unified approach for deciding the existence of certain Petri net paths. *Information and Computation*, 96(1):119–137, 1992.