Analysis of Asynchronous Programs with Event-Based Synchronization*

Michael Emmi[1], Pierre Ganty[1], Rupak Majumdar[2], and Fernando Rosa-Velardo[3]

[1] IMDEA Software Institute, Madrid, Spain
{michael.emmi,pierre.ganty}@imdea.org
[2] MPI-SWS, Kaiserslautern, Germany
rupak@mpi-sws.org
[3] Universidad Complutense de Madrid, Spain
fernandorosa@sip.ucm.es

Abstract. Asynchronous event-driven programming has become a central model for building responsive and efficient software systems, from low-level kernel modules, device drivers, and embedded systems, to consumer application on platforms such as .Net, Android, iOS, as well as in the web browser. Being fundamentally concurrent, such systems are vulnerable to subtle and elusive programming errors which, in principle, could be systematically discovered with automated techniques such as model checking. However, current development of such automated techniques are based on formal models which make great simplifications in the name of analysis decidability: they ignore event-based synchronization, and they assume concurrent tasks execute serially. These simplifications can ultimately lead to false positives, in reporting errors which are infeasible considering event-based synchronization, as well as false negatives, overlooking errors which arise due to interaction between concurrent tasks.

In this work, we propose a formal model of asynchronous event-driven programs which goes a long way in bridging the semantic gap between programs and existing models, in particular by allowing the dynamic creation of concurrent tasks, events, task buffers, and threads, and capturing precisely the interaction between these quantities. We demonstrate that (1) the analogous program analysis problems based on our new model remain decidable, and (2) that our new model is strictly more expressive than the existing Petri net based models. Our proof relies on a class of high-level Petri nets called *Data Nets*, whose tokens carry names taken from an infinite and linearly ordered domain. This result represents a significant expansion to the decidability frontier for concurrent program analyses.

1 Introduction

The asynchronous event-driven programming model has emerged as a common approach to building responsive and efficient software. Rather than assigning each computing *task* to a dedicated *thread* which becomes blocked as the task polls for some condition, the system maintains lightweight sets of *events* on which tasks are pending, *buffers* of tasks whose events have been triggered, and worker threads to execute

* This work is supported in part by the N-GREENS Software project (Ref. S2013/ICE-2731), STRONGSOFT (TIN2012-39391-C04-04), and AMAROUT-II (EU-FP7-COFUND-291803).

J. Vitek (Ed.): ESOP 2015, LNCS 9032, pp. 535–559, 2015.
DOI: 10.1007/978-3-662-46669-8_22

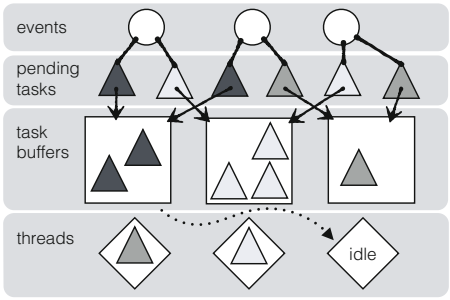

Fig. 1. In asynchronous event-driven programs, pending tasks (drawn as triangles) are moved to their designated task buffers (drawn as boxes) once their designated events (drawn as circles) are triggered. Threads (drawn as diamonds) execute buffered tasks to completion, such that no two tasks from the same buffer (drawn with the same color) execute in parallel.

$$
\begin{aligned}
s ::= {} & s;\ s \mid \textbf{skip} \\
& \mid \textbf{assume}\ e \mid \textbf{assert}\ e \\
& \mid x := e \\
& \mid \textbf{if}\ e\ \textbf{then}\ s\ \textbf{else}\ s \\
& \mid \textbf{while}\ e\ \textbf{do}\ s \\
& \mid \textbf{call}\ x := p\ e \mid \textbf{return}\ e \\
& \mid x := \textbf{(rep) task}\ p\ e\ Y\ z \\
& \mid y := \textbf{new event} \\
& \mid y := \textbf{event} \\
& \mid z := \textbf{new buffer} \\
& \mid z := \textbf{buffer} \\
& \mid \textbf{cancel}\ x \mid \textbf{wait}\ Y \mid \textbf{sync}\ y
\end{aligned}
$$

Fig. 2. The grammar of program statements. Here $x, y, z \in \mathsf{Vars} \cup \{\bot\}$ range over program variables or $\bot$, $Y \subseteq \mathsf{Vars}$ over sets of program variables, e over expressions, and p over procedure names. We assume $\bot$ does not appear on the left-hand side of assignment.

buffered tasks; Figure 1 illustrates the architecture. Besides the possibility of using events for synchronization between tasks, the task buffers themselves provide another means of orchestration: the system can allow tasks from distinct buffers to execute in parallel while ensuring that tasks from the same buffer execute serially. Delegating the management of events, tasks, buffers, and threads to the system generally increases efficiency.

High-performance asynchronous event-driven programs require subtle management of concurrent tasks [1], increasing the possibility of anomalous behavior due to unforeseen task schedules. One important research direction is the development of static program analyses for these programs. Indeed, such analyses have been developed, based on the formal models of *multi-set pushdown systems* (MPDS) [2] and Petri nets [3]. Led by decidability considerations, MPDSs (and Petri nets) model systems with, effectively, just one thread and task buffer, and without events. However, multiple threads, buffers, and events are fundamental to the use of many, if not most, systems; consider the most basic `libevent` API [4] function

```
event_new(buff,fd,flags,proc,arg),
```

which returns a new task destined for buffer `buff` pending an event on the file descriptor `fd`, or the most basic `libdispatch` API [5] functions

```
dispatch_source_create(ty,fd,flags,buff),
```

which returns a new event whose attached tasks are destined to buffer `buff`, and

```
dispatch_source_set_handler(evt,proc),
```

which attaches to event `evt` a task to execute procedure `proc`. Accordingly, MPDS-based analyses may report false positives due to abstraction of event-based synchronization, as well as false negatives due to the restriction to executions in which all tasks are executed serially, by one single thread.

We propose a formal model of event-driven asynchronous programs which captures event-based synchronization and task-buffer partitioning precisely, and yet retains a basis for decidable program analyses. Besides task creation, our model allows event creation and task-buffer creation as primitive operations. Each newly-created task has a set of events on which it is *pending*, and a task buffer in which it resides once activated by the triggering of any such event. An *active* task can be dispatched to run on any idle thread until it completes its execution, possibly blocking its thread by *waiting* for some other set of events. Alternatively, in the spirit of asynchronous event-driven programming, tasks can also create more tasks to continue their work once certain events are triggered, rather than blocking their execution threads. Our model also permits tasks to trigger events and to cancel previously-created tasks. This allows our model to capture many intricate aspects of existing languages and libraries supporting asynchronous programming. We show that these features make our model *strictly* more expressive[1] than usual models for asynchronous programs [2,3].

Our main result is that decidability of safety verification is retained despite such heightened modeling precision. While verification expectedly becomes undecidable once tasks can execute concurrently and call recursive procedures [6], or when task buffers are FIFO-ordered [7], even if data is abstracted into finite domains, we demonstrate that with only non-recursive procedures and unordered buffers, the decidability boundary can be stretched unexpectedly far to include unbounded dynamic creation of tasks *and* events.

We prove our result by a reduction to the coverability (or control-state reachability) problem of *Data nets*, an extension of *Petri nets* [8]. In standard Petri nets each place may contain an arbitrary number of *identical* (black) tokens. These tokens may be consumed from the so-called preconditions of transitions and created in postconditions. In the paradigm of *colored Petri nets*, tokens are distinguishable (colored), and their color is relevant for the semantics of the net. *Data Nets* [9] are a class of colored Petri nets in which tokens carry names taken from an infinite, linearly-ordered, and dense domain. In particular, only the relative order among these names is relevant for the semantics of nets. This reduction essentially works by modeling program tasks as tokens in a Data Net, whose event identifiers are encoded by names, and whose task identifiers, buffer identifiers, and local variable valuations are all encoded by the places of the Data Net. Even if Data Nets can represent unboundedly many names in a configuration, they cannot represent unboundedly many *sets* of names. Therefore, the challenge in the simulation is to represent unboundedly-many tasks that are pending on multiple events, even if we cannot directly represent such sets. We use the linear order to specify the order in which events can take place in the future.

The main contributions and outline of this work are:

§2 A formal model of event-driven asynchronous programs which captures event-based synchronization and task-buffer partitioning.

[1] With respect to the control-state reachability problem.

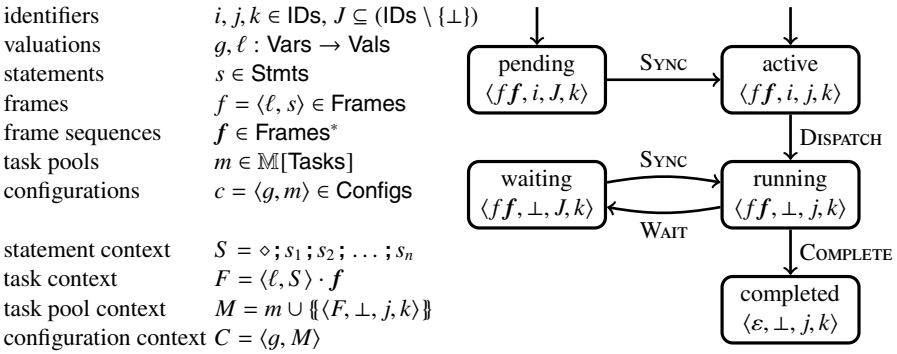

Fig. 3. Syntactic conventions and the task-state transition diagram. Note that $i \neq \bot$ and $J \neq \emptyset$.

§4 The decidability of the control-state reachability problem for this formal model.
§5 A hardness result for the control-state reachability problem.

2 Asynchronous Programs with Event-Based Synchronization

We fix the sets Procs of procedure names, Vars of variables, Vals of values, and $\mathsf{IDs} \subseteq \mathsf{Vals}$ of resource identifiers, such that $\bot \in \mathsf{IDs}$ and $\mathbf{true}, \mathbf{false} \in \mathsf{Vals}$. A *procedure* $p \in \mathsf{Procs}$ is a sequence of parameter and local variable declarations, along with a statement s_p representing the body of p. A *program* P is a sequence of global variable and procedure declarations; we write Stmts to denote the finite set of statements included in the grammar of Figure 2, restricted to those which appear[2] in P. While the syntax of program expressions is mostly unimportant for our concurrency-centric considerations, we do suppose that program expressions are statically typed with distinct types for task, buffer, and event identifiers, and the type of $\bot$ is polymorphic, e.g., as `null` in Java. We also include the nullary nondeterministic choice operator $\star$, which can evaluate to any value of $(\mathsf{Vals} \setminus \mathsf{IDs})$, in order to model programs in which data values have been abstracted [10].

Intuitively, the $y :=$ **new event** and $z :=$ **new buffer** statements store a fresh identifier in variable y, resp., z, which is later used to refer to an event or buffer. The $x :=$ (**rep**) **task** $p\ e\ Y\ z$ statement stores in x the identifier of a new (repeating) task to execute procedure p with argument e, which is to be placed in the buffer identified by z once (resp., each time) any of the events identified by Y are triggered; the **cancel** x statement discards any task(s) identified by x. The **sync** y statement triggers the event identified by y. The **wait** Y statement suspends its executing thread until any of the events identified by Y are triggered. We suppose that local variables do not store identifiers; otherwise, as Appendix B demonstrates, the program analysis problems we consider in this work would become undecidable.

A *(procedure) frame* $f = \langle \ell, s \rangle \in \mathsf{Frames}$ is a (finite) valuation $\ell : \mathsf{Vars} \to \mathsf{Vals}$ to the procedure local variables, along with a statement $s \in \mathsf{Stmts}$ describing the entire

[2] See Appendix A for the precise meaning of *appear* in P.

body of a procedure that remains to be executed, initially set to s_p. A *task* $\langle \boldsymbol{f}, i, j, k\rangle$ or $\langle \boldsymbol{f}, i, J, k\rangle$ is a procedure frame sequence $\boldsymbol{f} \in \mathsf{Frames}^*$, along with a repeating or non-repeating task identifier $i \in \mathsf{IDs}$, an event identifier $j \in \mathsf{IDs}$ or non-empty set of event identifiers $J \subseteq (\mathsf{IDs} \setminus \{\bot\})$, and a buffer identifier $k \in \mathsf{IDs}$. The set of tasks is denoted Tasks. A task $\langle \_, i, \_, \_\rangle$ is *repeating* when i is a repeating task identifier. A task $\langle \boldsymbol{f}, \_, \_, \_\rangle$ is *completed* when $\boldsymbol{f} = \varepsilon$.[3] A non-completed task $\langle \_, i, J, \_\rangle$ is *pending* when $i \neq \bot$, and *waiting* when $i = \bot$. A non-completed task $\langle \_, i, j, \_\rangle$ is *active* when $i \neq \bot$, and *running* when $i = \bot$. Note that we only write capital J for waiting and pending tasks, and lowercase $j \in \mathsf{IDs}$ for active and running tasks. A *task pool* is a finite-support[4] multiset $m \in \mathbb{M}[\mathsf{Tasks}]$. A *configuration* $c = \langle g, m\rangle \in \mathsf{Configs}$ is a valuation $g : \mathsf{Vars} \to \mathsf{Vals}$ to the global variables, along with a task pool m. Figure 3 summarizes our syntactic conventions, and the transitions between the various task states.

The (nondeterministic) evaluation $e(g, \ell) \subseteq \mathsf{Vals}$ is the set of values to which the program expression e can evaluate, given the variable valuations g and ℓ. While we are mostly agnostic to the meaning of program expressions, we assume that $\star(g, \ell) = (\mathsf{Vals} \setminus \mathsf{IDs})$, and that evaluation of each IDs-typed expression e is deterministic, i.e., $e(g, \ell) = \{i\}$ for some $i \in \mathsf{IDs}$; accordingly, we treat $e(g, \ell)$ as an element of IDs rather than as a subset of IDs.

To reduce clutter in the operational program semantics, we introduce a notion of context. A *statement context* S is a term derived from the grammar $S ::= \diamond \mid S\,;s$, where $s \in \mathsf{Stmts}$. We write $S[s]$ for the statement obtained by substituting a statement s for the unique occurrence of $\diamond$ in S. Intuitively, a context filled with s, e.g., $S[s]$, indicates that s is the next statement to execute in the statement sequence $S[s]$. Similarly, a *task context* $F = \langle \ell, S\rangle \cdot \boldsymbol{f}$ is a frame sequence in which the first frame's statement is replaced with a statement context, and we write $F[s]$ to denote the frame sequence $\langle \ell, S[s]\rangle \cdot \boldsymbol{f}$. A *task pool context* $M = m \cup \{\!\{\langle F, \bot, j, k\rangle\}\!\}$ is a task pool in which the frame sequence of one *running* task is replaced by a context, and we write $M[s]$ to denote the task pool $m \cup \{\!\{\langle F[s], \bot, j, k\rangle\}\!\}$. A *configuration context* $C = \langle g, M\rangle$ is a configuration in which the task pool is replaced by a task pool context, and we write $C[s]$ to denote the configuration $\langle g, M[s]\rangle$. We overload expression evaluation to contexts, writing $e(g, F)$, $e(g, M)$, or $e(C)$ for the evaluation $e(g, \ell)$ using the global valuation g and the local valuation ℓ of the selected task's first frame.

Figure 4 defines the program transition relation $\to$ as a set of operational steps on configurations. The New rule simply stores a freshly-allocated identifier, which can be subsequently used to identify events or buffers. The Dispatch rule makes some active task running, so long the number of running and waiting tasks does not exceed the number $T \in (\mathbb{N} \cup \{\omega\})$ of threads,[5] nor does the number of running tasks from any given buffer k exceed the buffer's concurrency limit $T_k \in (\mathbb{N} \cup \{\omega\})$. While our semantics is indifferent, the most common use case sets $T_k = 1$. The Complete rule turns a running task with no statements but **skip** to execute into a completed task. The Task rule adds a newly-created task to the task pool, while the Cancel rule removes any task identified

[3] We write "_" to denote irrelevant entities, and "ε" for the empty sequence.

[4] A multiset m has *finite support* when $m(x) > 0$ for only finitely many $x \in \mathrm{dom}(m)$.

[5] While we consider for simplicity a fixed number T of threads, we claim our theoretical results continue to hold if the number of threads can be changed dynamically, e.g., by the program.

$$\textsc{New}\quad \frac{i \text{ is fresh} \qquad g_2 = g_1(y \mapsto i)}{\langle g_1, M[y := \textbf{new } \_]\rangle \to \langle g_2, M[\textbf{skip}]\rangle}$$

$$\textsc{Dispatch}\quad \frac{i \neq \bot \qquad |\{\!\{\langle f\, \boldsymbol{f}, i', \_, \_\rangle \in m : i' = \bot\}\!\}| < T \qquad |\{\!\{\langle f\, \boldsymbol{f}, i', \_, k'\rangle \in m : i' = \bot \text{ and } k' = k\}\!\}| < T_k}{\langle g, m \cup \{\!\{\langle f, i, j, k\rangle\}\!\}\rangle \to \langle g, m \cup \{\!\{\langle f, \bot, j, k\rangle\}\!\}\rangle}$$

$$\textsc{Complete}\quad \frac{f = \langle \_, \textbf{skip}\rangle}{\langle g, m \cup \{\!\{\langle f, \bot, j, k\rangle\}\!\}\rangle \to \langle g, m \cup \{\!\{\langle \varepsilon, \bot, j, k\rangle\}\!\}\rangle}$$

$$\textsc{Task}\quad \frac{i \text{ is a fresh (repeating) task ID} \qquad g_2 = g_1(x \mapsto i) \qquad \ell \in e(g_1, M) \qquad m = \{\!\{\langle\langle \ell, s_p\rangle, i, Y(g_1, M), z(g_1, M)\rangle\}\!\}}{\langle g_1, M[x := (\textbf{rep})\ \textbf{task}\ p\ e\ Y\ z]\rangle \to \langle g_2, M[\textbf{skip}] \cup m\rangle}$$

$$\textsc{Cancel}\quad \frac{g_2 = g_1(x \mapsto \bot) \qquad m = \{\!\{\langle \_, i, \_, \_\rangle \in M : i = x(g_1, M) \neq \bot\}\!\}}{\langle g_1, M[\textbf{cancel } x]\rangle \to \langle g_2, M[\textbf{skip}] \setminus m\rangle}$$

$$\textsc{Wait}\quad \frac{m_1 = \{\!\{\langle F[\textbf{wait } Y], \bot, \_, k\rangle\}\!\} \qquad m_2 = \{\!\{\langle F[\textbf{skip}], \bot, Y(g, M), k\rangle\}\!\}}{\langle g, m \cup m_1\rangle \to \langle g, m \cup m_2\rangle}$$

$$\textsc{Sync}\quad \frac{j = y(g, M_1) \qquad m \in \{\!\{\langle \_, \_, J, \_\rangle \in M_1 : j \in J\}\!\} \qquad m' = \{\!\{\langle \_, i, \_, \_\rangle \in m : i \text{ is not a repeating task ID}\}\!\} \qquad M_2 = (M_1 \setminus m') \cup \{\!\{\langle \boldsymbol{f}, i, j, k\rangle : \langle \boldsymbol{f}, i, \_, k\rangle \in m\}\!\}}{\langle g, M_1[\textbf{sync } y]\rangle \to \langle g, M_2[\textbf{skip}]\rangle}$$

$$\textsc{Current-Event}\quad \frac{m_1 = \{\!\{F[y := \textbf{event}], \bot, j, k\}\!\} \qquad m_2 = \{\!\{\langle F[\textbf{skip}], \bot, j, k\rangle\}\!\} \qquad g_2 = g_1(y \mapsto j)}{\langle g_1, m \cup m_1\rangle \to \langle g_2, m \cup m_2\rangle}$$

$$\textsc{Current-Buffer}\quad \frac{m_1 = \{\!\{\langle F[z := \textbf{buffer}], \bot, j, k\rangle\}\!\} \qquad m_2 = \{\!\{\langle F[\textbf{skip}], \bot, j, k\rangle\}\!\} \qquad g_2 = g_1(z \mapsto k)}{\langle g_1, m \cup m_1\rangle \to \langle g_2, m \cup m_2\rangle}$$

Fig. 4. Rules defining the program transition relation $\to$. The variables $i, j, k \in \mathsf{IDs}$ range over task, event, and buffer identifiers respectively, and $J \subseteq (\mathsf{IDs} \setminus \{\bot\})$ over non-empty event-identifier sets. We write $Y(g, M)$ in the Task and Wait rules for the set $\{y(g, M) \in \mathsf{IDs} : y \in Y\} \setminus \{\bot\}$ of identifiers referred to by Y, interpreting $Y(g, M)$ as $\bot$ when $Y(g, M) = \emptyset$. We denote irrelevant values with "_" and write $\{\!\{\cdot\}\!\}$, $\cup$, and $\setminus$ to denote the multiset constructor, union, and difference operators.

by x. The Wait rule makes waiting one running task. The Sync rule makes active or running all pending or waiting tasks whose event sets contain the event identified by y. The Current-Buffer and Current-Event rules allow to recover and store in a variable the buffer to which the task belongs and the event that activated the task, respectively. The transition rules for the usual sequential program statements are standard, and are included in Appendix A; those rules modify the state of exactly one *running* task at a time.

The *initial configuration* $c_0 = \langle g_0, m_0\rangle$ of a program P is the valuation g_0 mapping each global variable to $\bot$, along with the task pool m_0 containing a single running task $\langle\langle \ell_0, s_{\texttt{main}}\rangle, \bot, \bot, \bot\rangle$ such that ℓ_0 maps each variable of the `main` procedure to $\bot$. An *execution of P to c_j* is a configuration sequence $c_0 c_1 \ldots c_j$ such that $c_i \to c_{i+1}$ for $0 \leq i < j$. The *control-state reachability problem* asks, given a procedure p of a program P, whether p can be executed in some execution of P, i.e. whether there is a reachable configuration $\langle g, m\rangle$ with $\langle\langle \_, s_p\rangle\, \boldsymbol{f}, \bot, \_, \_\rangle \in m$ for some $\boldsymbol{f} \in \mathsf{Frames}^*$. Typical safety

```
    var x: int
    var b1, b2: buffer id
    var t1, t2: task id

    proc main
s1 :  x := 1000;
s2 :  b1 := new buffer;
s3 :  b2 := new buffer;
s4 :  t1 := task p(1000) {} b1;
s5 :  t2 := task p(1) {} b2
    end

    proc p(var y: int)
      var z: int
s6 :  z := x - y;
s7 :  x := z
    end
```

Fig. 5. A program which executes two tasks from different buffers to subtract a value from global variable x

```
    var x: int
    var e: event id
    var t1, t2: task id

    proc main
s1 :  e := new event;
s2 :  t1 := task p1() {} ⊥;
s3 :  t2 := rep task p2() {e} ⊥
    end

    proc p1()
s4 :  x := 1;
s5 :  sync e
    end

    proc p2()
s6 :  assert x > 0
    end
```

Fig. 6. A program using synchronization

verification questions (e.g., program assertions, mutual exclusion properties) can be reduced to this problem.

Example 1. Consider an execution from the initial configuration

$$\left\langle \begin{pmatrix} \mathtt{x} \mapsto \bot \\ \mathtt{b1} \mapsto \bot, \mathtt{b2} \mapsto \bot \\ \mathtt{t1} \mapsto \bot, \mathtt{t2} \mapsto \bot \end{pmatrix}, \{\!\{ \langle \langle \emptyset, s_1 ; s_2 ; s_3 ; s_4 ; s_5 \rangle, \bot, \bot, \bot \rangle \}\!\} \right\rangle$$

of the program listed in Figure 5. By applying the Assign and New rules to the first three program statements s_1–s_3, and advancing past reduced **skip** statements via the Skip rule, we arrive to the configuration

$$\left\langle \begin{pmatrix} \mathtt{x} \mapsto 1000 \\ \mathtt{b1} \mapsto \mathtt{b}_1, \mathtt{b2} \mapsto \mathtt{b}_2 \\ \mathtt{t1} \mapsto \bot, \mathtt{t2} \mapsto \bot \end{pmatrix}, \{\!\{ \langle \langle \emptyset, s_4 ; s_5 \rangle, \bot, \bot, \bot \rangle \}\!\} \right\rangle$$

in which buffer identifiers $\mathtt{b}_1$ and $\mathtt{b}_2$ have been created and stored in variables b1 and b2. Now applying the Task rule to statement s_4 creates a fresh task identified by $\mathtt{t}_1$ to execute p(1000) from buffer $\mathtt{b}_1$

$$\left\langle \begin{pmatrix} \mathtt{x} \mapsto 1000 \\ \mathtt{b1} \mapsto \mathtt{b}_1, \mathtt{b2} \mapsto \mathtt{b}_2 \\ \mathtt{t1} \mapsto \mathtt{t}_1, \mathtt{t2} \mapsto \bot \end{pmatrix}, \left\{\!\!\left\{ \begin{array}{l} \langle \langle \emptyset, s_5 \rangle, \bot, \bot, \bot \rangle \\ \langle \langle [\mathtt{y} \mapsto 1000, \mathtt{z} \mapsto \bot], s_6 ; s_7 \rangle, \mathtt{t}_1, \bot, \mathtt{b}_1 \rangle \end{array} \right\}\!\!\right\} \right\rangle$$

and subsequently applying the Task rule to statement s_5 creates a fresh task identified by t_2 to execute $\mathsf{p}(1)$ from buffer b_2

$$\left\langle \begin{pmatrix} \mathsf{x} \mapsto 1000 \\ \mathsf{b1} \mapsto \mathsf{b}_1, \mathsf{b2} \mapsto \mathsf{b}_2 \\ \mathsf{t1} \mapsto \mathsf{t}_1, \mathsf{t2} \mapsto \mathsf{t}_2 \end{pmatrix}, \left\{\!\!\left\{ \begin{array}{l} \langle \varepsilon, \bot, \bot, \bot \rangle \\ \langle\langle [\mathsf{y} \mapsto 1000, \mathsf{z} \mapsto \bot], s_6 \,;\, s_7 \rangle, \mathsf{t}_1, \bot, \mathsf{b}_1 \rangle \\ \langle\langle [\mathsf{y} \mapsto 1, \mathsf{z} \mapsto \bot], s_6 \,;\, s_7 \rangle, \mathsf{t}_2, \bot, \mathsf{b}_2 \rangle \end{array} \right\}\!\!\right\} \right\rangle$$

resulting in the completion of the initially-running task, by subsequently applying the Complete rule. Supposing that Task t_2 executes statement s_6 before t_1 executes, via the Dispatch and Assign rules, we arrive at the configuration

$$\left\langle \begin{pmatrix} \mathsf{x} \mapsto 1000 \\ \mathsf{b1} \mapsto \mathsf{b}_1, \mathsf{b2} \mapsto \mathsf{b}_2 \\ \mathsf{t1} \mapsto \mathsf{t}_1, \mathsf{t2} \mapsto \mathsf{t}_2 \end{pmatrix}, \left\{\!\!\left\{ \begin{array}{l} \langle \varepsilon, \bot, \bot, \bot \rangle \\ \langle\langle [\mathsf{y} \mapsto 1000, \mathsf{z} \mapsto \bot], s_6 \,;\, s_7 \rangle, \mathsf{t}_1, \bot, \mathsf{b}_1 \rangle \\ \langle\langle [\mathsf{y} \mapsto 1, \mathsf{z} \mapsto 999], s_7 \rangle, \bot, \bot, \mathsf{b}_2 \rangle \end{array} \right\}\!\!\right\} \right\rangle.$$

Since Task t_1 executes from a different buffer than t_2, it may execute to completion before t_2 makes another move, by applying the Dispatch, Assign, and Complete rules, arriving at the configuration

$$\left\langle \begin{pmatrix} \mathsf{x} \mapsto 0 \\ \mathsf{b1} \mapsto \mathsf{b}_1, \mathsf{b2} \mapsto \mathsf{b}_2 \\ \mathsf{t1} \mapsto \mathsf{t}_1, \mathsf{t2} \mapsto \mathsf{t}_2 \end{pmatrix}, \left\{\!\!\left\{ \begin{array}{l} \langle \varepsilon, \bot, \bot, \bot \rangle \\ \langle \varepsilon, \bot, \bot, \mathsf{b}_1 \rangle \\ \langle\langle [\mathsf{y} \mapsto 1, \mathsf{z} \mapsto 999], s_7 \rangle, \bot, \bot, \mathsf{b}_2 \rangle \end{array} \right\}\!\!\right\} \right\rangle$$

in which t_1 has subtracted `1000` from x, yet t_2 has yet to complete its subtraction of 1 from `1000`. Allowing t_1 to execute to completion, via the Assign and Complete rules, we arrive at the configuration

$$\left\langle \begin{pmatrix} \mathsf{x} \mapsto 999 \\ \mathsf{b1} \mapsto \mathsf{b}_1, \mathsf{b2} \mapsto \mathsf{b}_2 \\ \mathsf{t1} \mapsto \mathsf{t}_1, \mathsf{t2} \mapsto \mathsf{t}_2 \end{pmatrix}, \left\{\!\!\left\{ \begin{array}{l} \langle \varepsilon, \bot, \bot, \bot \rangle \\ \langle \varepsilon, \bot, \bot, \mathsf{b}_1 \rangle \\ \langle \varepsilon, \bot, \bot, \mathsf{b}_2 \rangle \end{array} \right\}\!\!\right\} \right\rangle$$

in which both subtraction tasks have completed, yet only the second's effects are accounted for in the global state. Note that this global state would not be admitted were both tasks to execute serially.

Example 2. Consider an execution from the initial configuration

$$\left\langle \begin{pmatrix} \mathsf{x} \mapsto \bot, \mathsf{e} \mapsto \bot \\ \mathsf{t1} \mapsto \bot, \mathsf{t2} \mapsto \bot \end{pmatrix}, \{\!\!\{ \langle\langle \emptyset, s_1 \,;\, s_2 \,;\, s_3 \rangle, \bot, \bot, \bot \rangle \}\!\!\} \right\rangle$$

of the program listed in Figure 6. By applying the New and Task rules for the first three program statements s_1–s_3 we arrive to the configuration

$$\left\langle \begin{pmatrix} \mathsf{x} \mapsto \bot, \mathsf{e} \mapsto \mathsf{e}_1 \\ \mathsf{t1} \mapsto \mathsf{t}_1, \mathsf{t2} \mapsto \mathsf{t}_2 \end{pmatrix}, \left\{\!\!\left\{ \begin{array}{l} \langle \varepsilon, \bot, \bot, \bot \rangle \\ \langle\langle \emptyset, s_4 \,;\, s_5 \rangle, \mathsf{t}_1, \bot, \bot \rangle \\ \langle\langle \emptyset, s_6 \rangle, \mathsf{t}_2, \{\mathsf{e}_1\}, \bot \rangle \end{array} \right\}\!\!\right\} \right\rangle$$

in which Task t_1 is active and t_2 is pending. Note that neither would have been able to execute before the initial task had completed, since all three tasks are executed from

the same buffer, identified by $\perp$. Now applying Dispatch, which is the only enabled transition, and executing t_1 to completion, we arrive to the configuration

$$\left\langle \begin{pmatrix} \mathtt{x} \mapsto 1, \mathtt{e} \mapsto \mathsf{e}_1 \\ \mathtt{t1} \mapsto \mathsf{t}_1, \mathtt{t2} \mapsto \mathsf{t}_2 \end{pmatrix}, \left\{\!\!\left\{ \begin{array}{l} \langle \varepsilon, \perp, \perp, \perp \rangle \\ \langle \varepsilon, \perp, \perp, \perp \rangle \\ \langle \langle \emptyset, s_6 \rangle, \mathsf{t}_2, \{\mathsf{e}_1\}, \perp \rangle \\ \langle \langle \emptyset, s_6 \rangle, \mathsf{t}_2, \mathsf{e}_1, \perp \rangle \end{array} \right\}\!\!\right\} \right\rangle$$

in which event e_1 has been triggered, and variable $\mathtt{x}$ set to 1. Since t_2 is a repeating task, a pending copy of it remains in the task pool in addition to the newly-activated copy. Now the Dispatch rule can apply to the active copy, which can be executed to completion, succeeding the assertion of statement s_6, resulting in the configuration

$$\left\langle \begin{pmatrix} \mathtt{x} \mapsto 1, \mathtt{e} \mapsto \mathsf{e}_1 \\ \mathtt{t1} \mapsto \mathsf{t}_1, \mathtt{t2} \mapsto \mathsf{t}_2 \end{pmatrix}, \left\{\!\!\left\{ \begin{array}{l} \langle \varepsilon, \perp, \perp, \perp \rangle \\ \langle \varepsilon, \perp, \perp, \perp \rangle \\ \langle \langle \emptyset, s_6 \rangle, \mathsf{t}_2, \{\mathsf{e}_1\}, \perp \rangle \\ \langle \varepsilon, \perp, \mathsf{e}_1, \perp \rangle \end{array} \right\}\!\!\right\} \right\rangle.$$

Had we abstracted the event-based synchronization from our formal model, it would have been possible to violate the assertion of statement s_6, which is not possible in the program as it is written.

While our general model of asynchronous programs gives semantics to arbitrary programs with infinite data domains, our decidability arguments in the following sections rely on the following key assumptions/restrictions:

1. The set ($\mathsf{Vals} \setminus \mathsf{IDs}$) of non-identifier values is finite.
2. Procedures are not recursive: two frames of the same procedure cannot appear on the same procedure stack.
3. Identifiers are not stored in local variables (previously mentioned).
4. Buffer identifiers and repeating task identifiers (other than $\perp$) are stored in global variables from the time they are created.

Assumption 1 is a standard assumption in model checking [11] and data-flow analysis [12], which is obtained, e.g., by predicate abstraction [10]; allowing general-purpose infinite data domains quickly leads to undecidability. Assumption 2 avoids a well-established undecidable class, in which concurrent threads with unbounded procedure stacks (only 2 threads are required) can synchronize (e.g., through global variables) [6]. Assumption 4 together with 1, essentially limits the number of task buffers and repeating tasks (prohibiting unbounded dynamic creation), since their identifiers are always stored among a finite number of global variables. While Appendix B shows Assumption 3 cannot be relaxed, we do not know whether Assumption 4 can be weakened. We believe that it is reasonable to assume that programs keep references to their repeating tasks, in order to eventually cancel them, and do not create unboundedly-many task buffers. Note that these assumptions do not preclude an unbounded number of dynamically-created events correlating tasks in the task pool.

Though we model repeating tasks explicitly in order to describe their semantics precisely, and to stipulate Assumption 4, they can be simulated with regular tasks. Essentially, each **sync** action triggering events on which a repeating task is waiting is

augmented to create active copies of triggered repeating tasks. The required bookkeeping is possible since the number of repeating task identifiers is bounded via Assumption 4 by the number of global variables: for each repeating task i, additional global variables are added to store the identifiers of i's buffer, and events on which i waits.

3 Data Nets

In this section we present basic facts about *Petri Data Nets (PDN)* [9], which extend the classical model of Petri nets [8] with identity-carrying tokens. Despite the fact that PDNs are strictly-more expressive than Petri nets, their coverability problem remains decidable [9]. In Section 4, we show a reduction from the control-state reachability problem of asynchronous programs to the coverability problem on PDNs.

We denote the null tuple $(0, \ldots, 0) \in \mathbb{N}^k$ (for any k) as $\mathbf{0}$, and for $x = (a_1, \ldots, a_k)$ we write $x(i) = a_i$. We denote as $(\mathbb{N}^k)^*$ set of finite words over $\mathbb{N}^k$. For a word $w = x_1 \cdots x_n$ we write $|w| = n$ and $w(i) = x_i$. Formally, a *Petri Data Net* is a Petri net where each token carries an *identity* from a linearly ordered and dense domain $\mathbb{D}$.

Petri Data Nets. A k-dimensional *Petri Data Net* (*PDN*) is a tuple $N = (\mathsf{P}, \mathsf{T}, F, H)$, where:

- $\mathsf{P} = \{p_1, \ldots, p_k\}$ is a finite set of *places*,
- T is a finite set of *transitions*, disjoint from P,
- for every $t \in \mathsf{T}$, F_t and H_t are finite sequences in $(\mathbb{N}^k)^*$ with $|F_t| = |H_t| = n$ (for some n specific to t), and we say t has arity n.

Every transition t is endowed with two sequences F_t and H_t of the same length of (possibly null) vectors; $F_t(i)$ specifies the tokens carrying the i-th identity that are consumed and analogously, $H_t(i)$ specifies the tokens of that identity that are produced, when transition t is taken.

Markings. A marking m of a *PDN* can be seen as a mapping m that maps every place p to a multiset of identities. This will be the intuition that will guide our graphical notations. However, in the formal exposition, we use a different representation of markings, guided by the two following observations:

1. A marking m only has finitely many tokens, carrying some identities $d_1 < \cdots < d_n$. For each i, we can gather all the tokens carrying the name d_i in m, thus obtaining assuming k places a non-null vector $v_i \in \mathbb{N}^k$ (the j-th component of v_i standing for the number of tokens in the j-th place carrying the name d_i). Therefore, m can be written as $(d_1, v_1) \cdots (d_n, v_n)$.
2. The concrete identities d_i are irrelevant, and only their relative *order* is useful with respect to the semantics of *PDN*. Thus, m can be safely abstracted as the sequence $v_1 \cdots v_n$ in $(\mathbb{N}^{|\mathsf{P}|} \setminus \mathbf{0})^*$.

Formally, a *marking* of a k-dimensional *PDN* is a word in $(\mathbb{N}^k \setminus \mathbf{0})^*$. We say a marking $m = x_1 \cdots x_n$ *marks* p_i if $x_j(i) > 0$ for some $j \in \{1, \ldots, n\}$.

Prior to formally defining the transition relation, we start with some intuition. Consider a marking $m \in (\mathbb{N}^k \setminus \mathbf{0})^*$. In order to fire a transition t with arity n, the net nondeterministically selects n identities, consumes some tokens with these identities as specified

Fig. 7. Firing of a *PDN* transition (assuming $a < c < b$)

by F_t, and produces new tokens with the identities specified by H_t. In order to deal with identities that are not present in m, or identities that are removed due to the firing of t, we introduce/remove null vectors where needed. We say $m' \in (\mathbb{N}^k)^*$ is a $\mathbf{0}$-*extension* of a marking m (or m is the $\mathbf{0}$-*contraction* of m') if m is obtained by removing every tuple $\mathbf{0}$ from m'.

Transition Relation of *PDN*. Let m, m' be two markings and $t \in \mathsf{T}$ with arity n. We say t can be *fired* in m, reaching m' if:

1. there exists a $\mathbf{0}$-extension $u_0 x_1 u_1 \cdots u_{n-1} x_n u_n$ of m with $u_i \in (\mathbb{N}^k)^*$ for $i \in \{0, \ldots, n\}$ and $x_i \in \mathbb{N}^k$ for $i \in \{1, \ldots, n\}$,
2. $x_i \geq F_t(i)$ for $i \in \{1, \ldots, n\}$,
3. and taking $y_i = (x_i - F_t(i)) + H_t(i)$, m' is the $\mathbf{0}$-contraction of $u_0 y_1 u_1 \cdots u_{n-1} y_n u_n$.

We write $m \rightarrow m'$ if m' can be reached from m by firing some transition $t \in \mathsf{T}$, and denote $\rightarrow^*$ the reflexive and transitive closure of $\rightarrow$. We assume an initial marking m_0, and say a marking m is reachable if $m_0 \rightarrow^* m$.

We rely on the often-used graphical depiction for *PDN* and use pictures of Petri nets where arcs connected to a transition t are labelled with bags of variables that must be instantiated by ordered identities. The number of these variables is exactly the arity of t and the ordering of the corresponding identities is carried by the transition.

Using these graphical conventions, Figure 7 depicts a *PDN* with a single transition t given by $F_t = (1,0,0)(0,0,0)(0,1,0)$ and $H_t = (0,0,0)(0,0,1)(0,0,0)$ (where places are ordered by their index). The marking shown in the left of the figure is given by the word $m = (1,0,0)(0,1,0)$, where the first tuple represents the identity a (that appears only in p_1) and the second tuple represents the identity b (that appears only in p_2). Since the transition has arity 3, we need to have three tuples in our marking, for which we can add a $\mathbf{0}$-tuple, thus obtaining the $\mathbf{0}$-expansion $m'' = (1,0,0)(0,0,0)(0,1,0)$. Notice that $m''(i) \geq F_t(i)$ for $i = 1,2,3$. After subtracting F_t and adding H_t we obtain $m''' = (0,0,0)(0,0,1)(0,0,0)$, which is $\mathbf{0}$-contracted to the marking $m' = (0,0,1)$ shown in the right of Figure 7.

For brevity and readability, we allow variables that are not totally ordered, which stands for a choice among all possible linearizations. Also, we will allow the labelling of some arcs by an expression of the form $\min\{x_1, \ldots, x_n\}$, which is replaced in each linearization by the minimum variable. For instance, we can simulate a transition t in which two unrelated variables x and y appear, by a non-deterministic choice between three transitions, the first one assuming $x < y$ (and replacing $\min\{x, y\}$ by x), the second one assuming $y < x$ (replacing $\min\{x, y\}$ by y) and the last one assuming $x = y$ (with y and $\min\{x, y\}$ replaced by x). Analogously, a transition with variables x and y so that $x \leq y$, can be simulated by two transitions, assuming $x < y$ and $x = y$, respectively.

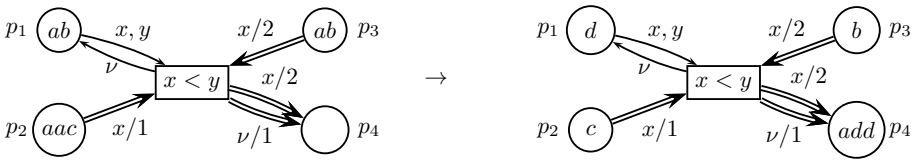

Fig. 8. Firing of a *PDN* transition t with extra features (with $a < b$, and c and d unrestricted)

Additionally, we shall use the following features in Section 4. Abdulla *et al.* [13] prove that each of these features, not present in the basic *PDN* model, can be simulated by the basic model.

Fresh Name Creation. In the *PDN* definition a new name *may* be created whenever some $F_t(i)$ is the null tuple, but freshness is not guaranteed. Abdulla *et al.* [13] give a construction that guarantees some name is created fresh. In pictures we will represent fresh name creation by labeling some postarc of a transition t by a special variable ν, that can only be instantiated to names not appearing in the marking that enables t.

Transfers and Renamings. In our simulation we will need to transfer every token carrying a given name from one place to another. We represent transfers in our pictures by having double arcs labelled by some variable. If a transition has several transfers, we will distinguish them in pictures by numbering them. Moreover, if the variables in the prearc and the postarc of a transfer do not coincide, then the names in the precondition are renamed accordingly in the postcondition.

Figure 8 shows an example that illustrates all the extra features we will need in our simulation. In the firing of t the variables x and y are instantiated to a and b, respectively (we are assuming $a < b$). After the firing of the transition (i) a and b are removed from p_1, (ii) a fresh name d is put in p_1, (iii) every a-token is transfered from p_3 to p_4 (transfer arc labelled by $x/2$), and (iv) every a-token in p_2 is transfered to p_4, and renamed to the same fresh name d (transfer labelled by $x/1$ and $\nu/1$). Notice that the transition specifies a partial order $x < y$ over the set of variables $\{x, y, \nu\}$, so that the position of the fresh name in the order is left unspecified.

We define the *control-state reachability problem*, that given a *PDN* N, an initial marking m_0, and a subset S of places of N, asks whether some marking reachable from m_0 in N marks some place $p \in S$. This problem is proved to be decidable in [9].

In Sect. 5, we show a reduction from a subclass of Data Nets in which names are unordered, and in which transfers are disallowed. This class is referred to as *ν-PN* in the literature [14]. The control-state reachability problem for *ν-PN* is known to be Ackermann-hard [14].

4 Simulation

We now demonstrate a reduction from the control-state reachability problem for asynchronous programs to the control-state reachability problem of Petri Data Nets. Our reduction models program tasks as tokens in a *PDN*, whose event identifiers are encoded

by identities, and whose task identifiers, buffer identifiers, and local variable valuations are all encoded by the places of the *PDN*. We demonstrate that the runs of the *PDN* N_P constructed from a program P simulate the executions of P, transition for transition, such that a given procedure p of P can be executed if and only if one of N_P's reachable markings marks a place which can only be marked when some task executing p is dispatched. For presentational simplicity, we give our simulation result assuming the only program variables have buffer- or event-identifier type, respectively denoted Vars_b and Vars_e. This assumption implies that

- together with Assumption 3, there are no local variables, hence the valuation component from Frames always equals $\emptyset$;
- because there are no local variables, there are no local assignments. Thus, the statement **call** $x := p\ e$ is simply replaced by **call** p and $x :=$ **task** $p\ e\ Y\ z$ is replaced by $x :=$ **task** $p\ Y\ z$;
- because there are no variables storing task identifiers $x :=$ **task** $p\ \ Y\ z$ is further simplified to **task** $p\ Y\ z$; further, there are no **cancel** statements.

We start by describing the encoding of the global valuations for the variables Vars_b and Vars_e. Notice that, because of Assumption 4, the set of buffers identifiers is limited to $|\mathsf{Vars}_\mathrm{b}| + 1$ distinct values. Define IDs_B to be the set buffer identifiers given by $\{\bot\} \cup \{1, \ldots, |\mathsf{Vars}_\mathrm{b}|\}$.

The *PDN* N_P simulating the program P has a set of places given by

$$\begin{aligned}
&\{p_\mathrm{b}(z,k) \mid z \in \mathsf{Vars}_\mathrm{b}, k \in \mathsf{IDs}_B\}\\
\cup&\{p_\mathrm{e}(y) \mid y \in \mathsf{Vars}_\mathrm{e}\}\\
\cup&\{p_\mathrm{t}(f,q,k) \mid f \in \mathsf{Frames}^{\leq F},\ q \in \{\mathsf{A},\mathsf{P},\mathsf{W},\mathsf{R}\},\ k \in \mathsf{IDs}_B\}\\
\cup&\{p_\mathrm{tlim}\} \cup \{p_\mathrm{blim}(k) \mid k \in \mathsf{IDs}_B\} \cup \{p_\mathrm{a}\}
\end{aligned}$$

Intuitively, a token in $p_\mathrm{b}(z,k)$ means that variable z stores the buffer identifier k. A token in $p_\mathrm{e}(y)$ with identity j means that variable y stores the event identifier j. This difference of encoding for event typed variable and buffer typed variable stems from the fact that the number of buffer identifiers is bounded across all executions of P (Assumption 4), whereas the number of event identifiers is not. For each variable $z \in \mathsf{Vars}_\mathrm{b}$, the simulation will enforce that if $p_\mathrm{b}(z,k)$ and $p_\mathrm{b}(z,k')$ are marked then $k = k'$. Intuitively, this is because a buffer typed variable stores exactly one buffer identifier. Also no variables $p_\mathrm{e}(y)$ contains more than 1 token. The places $p_\mathrm{t}(f,q,k)$ are meant to encode the task buffer. In concordance with Assumption 1, we assume a finite set $\mathsf{Frames}^{\leq F} \stackrel{\text{def}}{=} \{\boldsymbol{f} \in \mathsf{Frames}^* \colon\ |\boldsymbol{f}| \leq F\}$ of bounded frame sequences for some $F \in \mathbb{N}$, and we assume the procedure frame sequence of each task belongs to $\mathsf{Frames}^{\leq F}$. Recall that (*i*) it follows from the previous developments that a *frame* f consists of an empty valuation and a statement $s \in \mathsf{Stmts}$ describing the entire body of a procedure p that remains to be executed; and that (*ii*) the set Stmts is finite as explained in §2.

A token in $p_{\mathrm{t}}(f, q, k)$ with identity j encodes a task given by

$$\begin{cases} \langle f, \bot, j, k\rangle & \text{if } q = \mathsf{R}(unning) \\ \langle f, \bot, \{j\}, k\rangle & \text{if } q = \mathsf{W}(aiting) \\ \langle f, \_, j, k\rangle & \text{if } q = \mathsf{A}(ctive) \\ \langle f, \_, \{j\}, k\rangle & \text{if } q = \mathsf{P}(ending) \end{cases}$$

Observe that, since no variable can store a task identifier, they become irrelevant. This is why in the third and fourth case, the component identifying a task is left unspecified (but different from $\bot$).

We remark that the set of places of N_P can be effectively built, since all the finite sets appearing in the definition are not only finite, but they can be statically obtained. This is clearly the case for $\mathsf{Vars}_{\mathrm{b}}$, $\mathsf{Vars}_{\mathrm{e}}$ and IDs_B, which can be obtained by a simple inspection of the program P, but also for $\mathsf{Frames}^{\leq F}$ because of Assumptions 1 to 3.

The initial configuration of P defines the initial marking of N_P, in particular all the variables are initialized to $\bot$.

The encoding of pending tasks, using places $p_{\mathrm{t}}(f, q, k)$, disallows more than one event in their event set. The single event case (i.e., the case in which tasks are pending on a single event) is a special case of the general case, in which tasks are pending on multiple events. Next, we will see how we overcome what seems to be a loss of generality. Since this is the most delicate point in our simulation of programs using *PDN*, let us start by explaining some intuitions (formal developments will follow).

In the simulation using *PDN*, the event set of a task is limited to single events because tokens in *PDN* carry only a single identity.[6] So our goal is to simulate a program where tasks are pending on multiple events using, instead, tasks that are pending on a single event.

We first observe that although tasks are pending on multiple events, only one of them activates the task. At the task creation time, it is not possible to know which event from the set Y will activate the task. However, at creation time, one could guess the event which will activate the task. The non-determinism which is inherent in the model covers all possible such guesses of the event that will eventually activate the task.

However, this is not correct because guesses have to be consistent with the order in which events are triggered along the computation. To see this, consider the following scenario: a task t is created as pending on events $\{e_1, e_2\}$, and every computation triggers first e_2 and then e_1. In this case, the guess which associates to t the single event e_1 wrongly yields a computation with no counterpart in the original program, that in which e_1 and not e_2 activates t.

This example shows that these arbitrary guesses yield a loss of precision by introducing new behaviors. When creating a task, the event that has to be chosen is determined by the ordering in which events are triggered in the future of the computation.

Instead of guessing at task creation time which event will activate the task, we will guess at *event creation time* when this event will be triggered. Formally, this order in the

[6] Similar models in which multiple values can be carried by tokens are undecidable [15], and only become decidable when extra semantic restrictions are added [16], which are too restrictive for us.

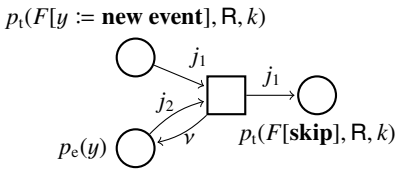

Fig. 9. Widget simulating event creation

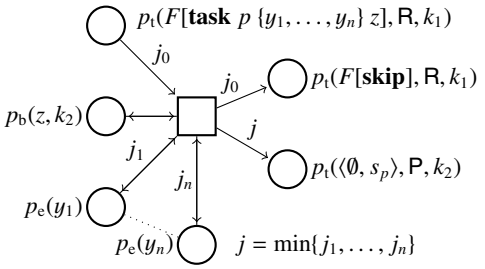

Fig. 10. Widget simulating task creation

triggering of events is given by the linear order on identities in the *PDN*. When creating a task pending on multiple events, the event chosen to activate the task is thus given by the minimal one with respect to that linear order.

Back to the example, since first e_2 and then e_1 are triggered, we have that $e_2 < e_1$ in the *PDN*. Hence, when the task t is created, only $e_2 = \min\{e_1, e_2\}$ can be chosen as the single event associated with t.

Summing up, event identities will be ordered in the simulation according to the (guessed) order in which they are triggered. Then, the simulation needs to guarantee that this order is correct, i.e., that the current computation is consistent with that order. For that purpose, we use a place p_a that holds a token whose identity is the next event that can be triggered, thus separating past events from future events. Also, this token will be used to guarantee that past events (those below the identity in p_a in the linear order) are no longer used. In our example, if the computation guesses $e_1 < e_2$, so that e_1 should first be triggered, and a sync over e_2 is attempted, then the simulation blocks.

Finally, p_{tlim} is a budget place for the number of threads. Its content corresponds to the number of threads available to execute a task. It initially contains $T - 1$ tokens (the -1 accounts for the thread running `main`). The simulation of Dispatch will remove a token from p_{tlim} while the simulation of Complete will add a token into it. Optionally, the budget can vary along the execution. Moreover, for each $k \in \{\bot\} \cup \{1, \ldots, |\mathsf{Vars_b}|\}$ $p_{\text{blim}}(k)$ is a budget place (one per buffer). As above $p_{\text{blim}}(k)$ accounts for the remaining concurrency limit for buffer k. Initially, it is set to the value T_j by putting T_j tokens at buffer creation time, and it is modified as for p_{tlim}.

Let us next see the simulation of each type of instructions. At the end of the section, we will discuss how to encode our programs when we relax the assumptions on variables so that we allow variables to store finite data and task identifiers.

New Event. An instruction $y := \textbf{new event}$ is simply simulated by replacing the name in $p_e(y)$ by a fresh name. The newly created name must be inserted at an arbitrary position in the linear order of identities of the *PDN*, so that the variable ν must be left unrestricted.

The widget of Figure 9 simulates event creation. For every place $p_t(f, \mathsf{R}, k)$ where $f = F[y := \textbf{new event}]$, there is a transition tr such that:

- tr moves a token from $p_t(f, \mathsf{R}, k)$ to $p_t(f', \mathsf{R}, k)$ where $f' = F[\textbf{skip}]$. It does so preserving the identity, j_1, carried by the token.
- tr replaces the token in $p_e(y)$ with one whose identity is fresh.

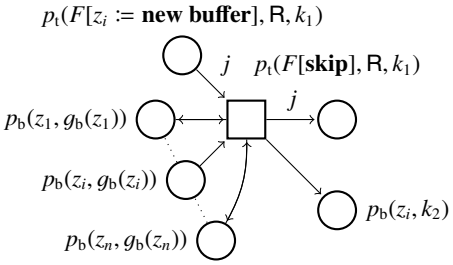

Fig. 11. Widget simulating task buffer creation

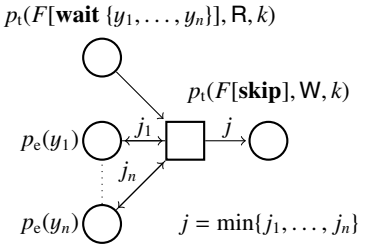

Fig. 12. Widget simulating waiting

New Task. We first treat the case in which the set Y is not empty (the other case is much simpler). The instruction **task** p $\{y_1, \ldots, y_n\}$ z selects the minimum event in places $p_e(y_1), \ldots, p_e(y_n)$ and puts this name in the place representing the initial state of p (in the buffer given by z), so that the new task is pending on the first event to be triggered.

The widget of Figure 10 simulates task creation. For every place $p_t(f, \mathsf{R}, k_1)$ where $f = F[\textbf{task } p \; Y \; z]$, $Y = \{y_1, \ldots, y_n\}$, for every $k_2 \in \mathsf{IDs}_B$, there is a transition tr such that:

- tr moves a token from $p_t(f, \mathsf{R}, k_1)$ to $p_t(f', \mathsf{R}, k_1)$ where $f' = F[\textbf{skip}]$. It does so preserving the identity, j_0, carried by the token.
- tr reads n identities stored in the places $p_e(y_1)$ to $p_e(y_n)$.
- tr tests whether the place $p_b(z, k_2)$ is marked.
- tr adds a token with identity j into $p_t(\langle \emptyset, s_p \rangle, \mathsf{P}, k_2)$ where j is given by the minimum among the identities stored in the places $\{p_e(y_1), \ldots, p_e(y_n)\}$.

For the case $Y = \emptyset$, it is enough to test the place $p_b(z, k_2)$, move the token with identity j_0 and add a token with constant identity $\perp$ to $p_t(\langle \emptyset, s_p \rangle, \mathsf{A}, k_2)$ instead (the task is already active).

New buffer. For the simulation of the creation of buffers, we mostly have to deal with the places of the form $p_b(z, k)$ that contain the valuation of buffer variables.

The widget of Figure 11 simulates task buffer creation. For every place $p_t(f, \mathsf{R}, k)$ where $f = F[z_i := \textbf{new buffer}]$, buffer identifier $k_2 \neq \perp$ and valuation g_b of the buffer typed variables whose range excludes k_2 there is a transition tr such that:

- tr moves a token from $p_t(f, \mathsf{R}, k_1)$ to $p_t(f', \mathsf{R}, k_1)$ where $f' = F[\textbf{skip}]$. It does so preserving the identity, j, carried by the token.
- tr tests that no buffer variables stores k_2. It does so by checking whether $p_b(z, g_b(z))$ is marked for every buffer typed variable z.
- tr moves the token from $p_b(z_i, g_b(z_i))$ to $p_b(z_i, k_2)$.

Sync. The simulation of a **sync** y is perhaps the most involved. On the one hand, we need to guarantee that the triggered event is legal according to the guessed linear order of events. For that purpose, the identity j_2 in $p_e(y)$ must coincide with that in p_a. Also, all the tokens carrying j_2 in each place of the form $p_t(f, P/W, k)$ must be transferred to

$p_1, \ldots, p_n$ an enumeration of $\{p_t(f, P, k), p_t(f, W, k), f \in \mathsf{Frames}, k \in \mathsf{IDs}_B\}$
and $\hat{p} = p_t(f, A/R, k)$ where $p = p_t(f, P/W, k)$

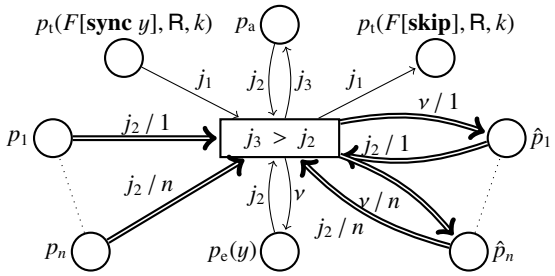

Fig. 13. Widget simulating synchronization

the corresponding place $p_t(f, A/R, k)$, because every task that is pending (waiting) on that name becomes active (running). Then an identity greater than j_2, say j_3, is chosen so as to correspond to the next event to be triggered, by replacing, in p_a, j_2 by j_3. Finally, in order to be able to repeat a **sync** over j_2 we should move j_2 in the linear order to a later position. This is not possible (we cannot change the linear order), but we can replace every j_2 token by a fresh one, which is created at an arbitrary position.

The widget of Figure 13 simulates synchronization. Formally, for every place $p_t(f, \mathsf{R}, k)$ where $f = F[\textbf{sync } y]$, there is a transition tr such that:

- tr moves a token from $p_t(f, \mathsf{R}, k)$ to $p_t(f', \mathsf{R}, k)$ where $f' = F[\textbf{skip}]$. It does so preserving the identity, j_1, carried by the token.
- tr tests that the identity of the token in p_a coincides with that of $p_e(y)$.
- tr transfers, for each place $p_t(f_1, \mathsf{P}, k_1)$ all the tokens whose identity coincide with j_2, the identity of the token in place $p_e(y)$, into $p_t(f_1, \mathsf{A}, k_1)$.
- tr transfers similarly from places $p_t(f_1, \mathsf{W}, k_1)$ into $p_t(f_1, \mathsf{R}, k_1)$.
- tr replace the identity j_2 of the token from p_a with an identity $j_3 > j_2$.
- tr replaces the identity j_2 of the token from $p_e(y)$ with a new, fresh name, ν.
- tr also renames all the tokens in the *PDN* carrying the old name j_2 with this new fresh name ν.

Let us remark that when choosing the next event to be triggered (by instantiating j_3 to a name greater than the instance of j_2) we may choose an event which is different from the one which immediately follows the instance of j_2 in the linear order.[7] For example, if $e_1 < e_2 < e_3$ is the current order of events and p_a contains e_1, the firing of tr may replace e_1 by e_3 in p_a. This means that in that execution e_2 can no longer be triggered, which only implies a loss of behaviour that preserves control-state reachability.

Wait. The simulation of this instruction is similar to the creation of tasks. The widget of Figure 12 simulates waiting. For every place $p_t(f, \mathsf{R}, k)$ where $f = F[\textbf{wait } Y]$, $Y = \{y_1, \ldots, y_n\}$, there is a transition tr such that:

[7] If we allow that operation in PDN the model becomes Turing-complete.

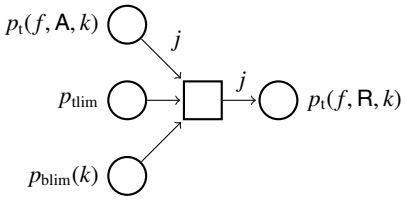

Fig. 14. Widget simulating task dispatch

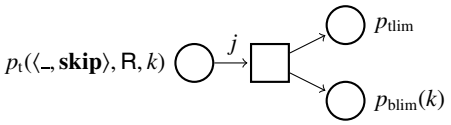

Fig. 15. Widget simulating task completion

- *tr* removes a token with some identity from $p_t(f, \mathsf{R}, k)$.
- *tr* reads n identities stored in the places $p_e(y_1)$ to $p_e(y_n)$.
- *tr* adds a token with identity j to $p_t(f', \mathsf{W}, k)$ where $f' = F[\textbf{skip}]$ and j is given by the minimum among the identities stored in the places $\{p_e(y_1), \ldots, p_e(y_n)\}$.

Dispatch. The widget of Figure 14 simulates task dispatch. For every place $p_t(f, \mathsf{A}, k)$, there is a transition *tr* that: moves a token from $p_t(f, \mathsf{A}, k)$ to $p_t(f, \mathsf{R}, k)$ while preserving its identity j, removes one token from p_{tlim}, the budget place for threads, and also from $p_{\text{blim}}(k)$, the budget place for concurrency limit of buffer k.

Complete. The widget of Figure 15 simulates task completion. For every place $p_t(f, \mathsf{R}, k)$ where $f = \langle \emptyset, \textbf{skip} \rangle$, there is a transition *tr* that removes a token form $p_t(f, \mathsf{R}, k)$, and adds one token into p_{tlim} and one into $p_{\text{blim}}(k)$.

Current-Buffer. For every place $p_t(f, \mathsf{R}, k)$ where $f = F[z := \textbf{buffer}]$ and $k_1 \in \mathsf{IDs}_B$, there is a transition *tr* that: moves a token from $p_t(f, \mathsf{R}, k)$ to $p_t(f', \mathsf{R}, k)$ where $f' = F[\textbf{skip}]$. It does so preserving the identity carried by the token. Transition *tr* also removes a token from $p_b(z, k_1)$ and adds one into $p_b(z, k)$.

Current-Event. For every place $p_t(f, \mathsf{R}, k)$ where $f = F[y := \textbf{event}]$, there is a transition *tr* that moves a token from $p_t(f, \mathsf{R}, k)$ to $p_t(f', \mathsf{R}, k)$ where $f' = F[\textbf{skip}]$. It does so preserving the identity j carried by the token. It also replaces the identity of the token from $p_e(y)$ with j.

At this point, it should be easy for the reader to define the widgets for the remaining statements. They present no particular difficulty.

Variables Over Finite Data Domain. At the beginning of this section we made the simplifying assumptions that there were no other variables than buffer or event typed variables. Let us now explain, how we simulate variables ranging over a finite set Vals of values. They are simulated similarly to the buffer typed variables, namely there is a place in the *PDN* for each pair variable-value. The presence of a token in such a place means the variable currently holds the value. The rest is tedious but follows easily.

Variables Over Task Identifiers. Variables storing task identifiers are trickier to simulate. This is because, although we have a fixed number of them, they store values from the unbounded domain of task identifers. This situation is similar to that of variables storing event identifers with an important difference: there is no comparable mecanism like the $x := \textbf{event}$ statement that allows to recover an event identifer stored in no variables. It follows that, for task identifers, once no variable stores it, its precise value

does not matter anymore. Rather than the actual identifiers, our encoding stores symbolic identifiers. If we call Vars_t the set of task typed variables then define IDs_T to be the set $\{\bot, \top\} \cup \{id_1, id_2, \ldots, id_{|\mathsf{Vars}_t|}\}$ of symbolic identifiers. The encoding keeps track of the symbolic identifers currently in use. Since this information is finite and can be statically obtained, it is easily encoded in the *PDN*. The execution of a $x :=$ **task** $p\ e\ Y\ z$ statement requires a symbolic identifer, not currently in use, to be stored in x. Two situations can occur: either there is a symbolic identifer among $\{id_1, id_2, \ldots, id_{|\mathsf{Vars}_t|}\}$ not in use; or not. However, in the latter case, the symbolic identifier, call it id_j, already stored in x can be re-used as long as all the pending or active tasks with identifier id_j are updated with the symbolic identifier $\top$. Intuitively, a task t with symbolic identifier $\top$ means no program variables storing a task identifier refers to t, hence t cannot be canceled. Furthermore, the set $\{p_t(f, q, k) \mid f \in \mathsf{Frames}^{\leq F}, q \in \{\mathsf{A}, \mathsf{P}, \mathsf{W}, \mathsf{R}\}, k \in \mathsf{IDs}_B\}$ of places encoding the task pool is extended with the information about symbolic identifiers. Hence each place $p_t(f, q, k)$ is replaced by the set $\{p_t(f, i, q, k) \mid i \in \mathsf{IDs}_T\}$. This enables the precise simulation of the **cancel** x statement.

The preceding encoding into *PDN* and the decidability of coverability for *PDN* implies the following main result.

Theorem 3. *The control-state reachability problem is decidable for asynchronous event-driven programs satisfying Assumptions 1 through 4.*

5 Hardness of Control State Reachability

We now show that the control state reachability problem for asynchronous programs is Ackermann-hard by showing that asynchronous programs can simulate *v-PN*, for which the control state reachability problem is Ackermann-hard [14]. Notice that the usual models of asynchronous programs [2,3] are equivalent to Petri nets and therefore have an EXPSPACE-complete coverability problem. Our model is strictly more expressive than these models.

Theorem 4. *The control state reachability problem for asynchronous programs satisfying Assumptions 1 through 4 is Ackermann-hard, even if no tasks are pending on multiple events.*

We simulate a *v-PN* as follows. For each place p, we define a different procedure `place-p` (see Figure 17), so that a task executing `place-p` simulates a token in place p. Furthermore, the procedure `main` schedules the firing of transitions. This scheduler uses boolean global variables `remove-p` and `done-p` for each place p, in order to synchronize with the rest of the tasks. Furthermore, we consider two global event variables, `current` and `aux`, for name matching (whenever a variable labels more than one prearc). Figure 18 shows the general scheme of the scheduler, with an infinite loop that nondeterministically selects the transition that is fired next. Whenever the scheduler decides to fire a transition that is not enabled, the system blocks (notice that this preserves control-state reachability). Instead of showing how it orchestrates the firing of an arbitrary transition, Figure 18 shows the simulation of the transition in Fig. 16 that (*i*) removes two tokens carrying the same name from p_1 and p_2, (*ii*) puts a token with that same name

in p_3, and (*iii*) puts a fresh token in p_4. For that purpose it first signals that a token from p_1 should be removed (by setting `remove-p1 := true`) and blocks until some task executing `place-p1` is done (instruction `assume done-p1`). Notice that this task is done only after putting its own event name in the global variable `aux`, and then it is completed. Then the scheduler does the same with p_2, but it also checks that both events coincide. Then it creates a new task executing `place-p3`, activated by that same event, and finally it creates a fresh event that is used to create and activate the task executing `place-p4`. The general case follows these ideas, but is more tedious.

We are assuming in the reduction that the number of threads and the concurrency limit of the only buffer $\perp$ is 2 ($T = T_\perp$=2), since the task executing `main` must be executed in parallel at each time with at most *one* task of the form `place-p`. The simulation of a transition completes and creates the tasks that represent the tokens involved in the firing of a transition. Therefore, every task of the form `place-p` can be active (not running), except one that has to be completed, which must be dispatched, only to be immediately completed. Notice that if a task that cannot be completed (one with its `remove-p` set to `false`) is dispatched, then the system blocks with that task blocked on `assume remove-p` and `main` blocked on `assume done-p`.

This simulation preserves control-state reachability, that is, a place p in the *ν-PN* can be marked iff some reachable configuration contains a pending task executing `place-p`.

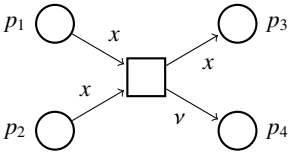

Fig. 16. Simple *ν-PN* transition

```
var remove-p1, done-p1: bool
    ...
var current, aux: event id
proc place-p()
   assume remove-p;
   remove-p := false;
   aux := event;
   done-p := true
end
```

Fig. 17. Global variables and procedure modelling a place

```
proc main
  while true do
    if ⋆ then //transition t1
      remove-p1 := true;
      assume done-p1;
      done-p1 := false;
      current := aux;
      remove-p2 := true;
      assume done-p2 && current = aux;
      done-p2 := false;
      _ := task place-p3() {current} ⊥;
      sync current;
      current := new event;
      _ := task place-p4() {current} ⊥;
      sync current
    else if ⋆ then //transition t2
      ...
end
```

Fig. 18. Scheduler

6 Related Work

Existing decidable models of asynchronous event-driven programs are based on *multi-set pushdown systems* (MPDS) [2], a model with unbounded task creation and recursion,

yet without events, without multiple task buffers, without multiple threads, and without task cancellation. Sen and Viswanathan showed that control-state reachability in MPDS is decidable [2], and Ganty and Majumdar showed that control-state reachability is equivalent to Petri net coverability [3]. Chadha and Viswanathan, Cai and Ogawa generalize these results to show that coverability for well-structured transition systems with one unbounded procedure stack remains decidable, so long as decreasing transitions only occur from a bounded set of configurations [17,18]. Others have shown decidable extensions to MPDS, e.g., with task cancellation [3], or task priorities [19]. While the *Actor Communicating Systems* (ACS) of D'Osualdo et al. do capture multiple task buffers and threads [20] (albeit without recursion, which is modeled in MPDS), they do not model events, and their task buffers are addressed only imprecisely: rather than sending a task to a particular task buffer, tasks are sent to an arbitrary member of a set of equivalent task buffers. Consequently, control-state reachability in ACS is also equivalent to Petri net coverability, and thus to control-state reachability in MPDS. Similarly, while Geeraerts et al. consider multithreaded asynchronous programs with *FIFO-ordered* task buffers, they show decidability for the case of "concurrent queues," in which arbitrarily-many tasks from the same buffer can run concurrently [21]. Effectively, the order in which tasks are added to buffers becomes irrelevant, and their control-state reachability problem is, again, equivalent to Petri net coverability. In contrast, Theorem 4 rules out the possibility of a polynomial time reduction to Petri nets.

Departing from MPDS, Babic and Rakamaric consider an expressive decidable class of asynchronous systems which leverages the decidable properties of visibly pushdown languages [22], equipping visibly pushdown processes with ordered, visibly pushdown, task/message buffers [23]. Kochems and Ong consider a relaxation of MPDS which allows concurrent *and* recursive tasks at the expense of a stack-shape restriction [24]. Bouajjani and Emmi consider decidable subclasses of a hierarchical generalization of MPDS, without communication between concurrent threads [25]. While Atig et al. and Emmi et al. consider MPDSs with multiple buffers and threads [26,27], their algorithms are only under-approximate, analyzing only up to a context-bound [28]. None of these works model events, nor propose sound and complete analysis algorithms in the presence of dynamic creation of threads, events, and buffers.

References

1. Adya, A., Howell, J., Theimer, M., Bolosky, W.J., Douceur, J.R.: Cooperative task management without manual stack management. In: USENIX ATC, pp. 289–302. USENIX (2002)
2. Sen, K., Viswanathan, M.: Model checking multithreaded programs with asynchronous atomic methods. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 300–314. Springer, Heidelberg (2006)
3. Ganty, P., Majumdar, R.: Algorithmic verification of asynchronous programs. ACM Trans. Program. Lang. Syst. 34(1), 6 (2012)
4. Mathewson, N., Provos, N.: libevent: an event notification library, `http://libevent.org`
5. The GCD team: libdispatch, `https://libdispatch.macosforge.org`
6. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM Trans. Program. Lang. Syst. 22(2), 416–430 (2000)
7. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM 30(2), 323–342 (1983)

8. Reisig, W.: Place/transition systems. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 254, pp. 117–141. Springer, Heidelberg (1987)
9. Lazic, R., Newcomb, T., Ouaknine, J., Roscoe, A.W., Worrell, J.: Nets with tokens which carry data. Fundam. Inform. 88(3), 251–274 (2008)
10. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
11. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)
12. Reps, T.W., Horwitz, S., Sagiv, S.: Precise interprocedural dataflow analysis via graph reachability. In: POPL 1995: Proc. 22th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 49–61. ACM (1995)
13. Abdulla, P.A., Delzanno, G., Van Begin, L.: A classification of the expressive power of well-structured transition systems. Inf. Comput. 209(3), 248–279 (2011)
14. Rosa-Velardo, F., de Frutos-Escrig, D.: Decidability and complexity of Petri nets with unordered data. Theor. Comput. Sci. 412(34), 4439–4451 (2011)
15. Rosa-Velardo, F., de Frutos-Escrig, D.: Decidability problems in Petri nets with names and replication. Fundam. Inform. 105(3), 291–317 (2010)
16. Meyer, R.: On boundedness in depth in the pi-calculus. In: Fifth IFIP International Conference On Theoretical Computer Science - TCS 2008, IFIP 20th World Computer Congress, TC 1, Foundations of Computer Science, Milano, Italy, September 7-10, 2008. IFIP, vol. 273, pp. 477–489. Springer, Heidelberg (2008)
17. Chadha, R., Viswanathan, M.: Decidability results for well-structured transition systems with auxiliary storage. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 136–150. Springer, Heidelberg (2007)
18. Cai, X., Ogawa, M.: Well-structured pushdown systems. In: D'Argenio, P.R., Melgratti, H. (eds.) CONCUR 2013 – Concurrency Theory. LNCS, vol. 8052, pp. 121–136. Springer, Heidelberg (2013)
19. Atig, M.F., Bouajjani, A., Touili, T.: Analyzing asynchronous programs with preemption. In: FSTTCS 2008: Proc. IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science. LIPIcs, vol. 2, pp. 37–48. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2008)
20. D'Osualdo, E., Kochems, J., Ong, C.-H.L.: Automatic verification of erlang-style concurrency. In: Logozzo, F., Fähndrich, M. (eds.) Static Analysis. LNCS, vol. 7935, pp. 454–476. Springer, Heidelberg (2013)
21. Geeraerts, G., Heußner, A., Raskin, J.-F.: Queue-dispatch asynchronous systems. CoRR abs/1201.4871 (2012)
22. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: STOC 2004: Proc. 36th Annual ACM Symposium on Theory of Computing, pp. 202–211. ACM (2004)
23. Babić, D., Rakamarić, Z.: Asynchronously communicating visibly pushdown systems. In: Beyer, D., Boreale, M. (eds.) FORTE 2013 and FMOODS 2013. LNCS, vol. 7892, pp. 225–241. Springer, Heidelberg (2013)
24. Kochems, J., Ong, C.-H.L.: Safety verification of asynchronous pushdown systems with shaped stacks. In: D'Argenio, P.R., Melgratti, H. (eds.) CONCUR 2013 – Concurrency Theory. LNCS, vol. 8052, pp. 288–302. Springer, Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-40184-8_21
25. Bouajjani, A., Emmi, M.: Analysis of recursively parallel programs. In: POPL 2012: Proc. 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 203–214. ACM (2012)
26. Atig, M.F., Bouajjani, A., Qadeer, S.: Context-bounded analysis for concurrent programs with dynamic creation of threads. Logical Methods in Computer Science 7(4) (2011)

27. Emmi, M., Lal, A., Qadeer, S.: Asynchronous programs with prioritized task-buffers. In: SIGSOFT FSE 2012: Proc. 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering, p. 48. ACM (2012)
28. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)

A Sequential Program Semantics

The finite set Stmts of statements appearing in a program P, with finite sets Procs of procedures and Vals of values, is defined formally by the inference rules of Figure 19. These rules are defined with respect to the notion of *context* of Section 2, which we define here to avoid circularity: a *context* S is a term derived from the grammar $S ::= \diamond \mid S\,;s$. We write $S[s]$ for the object obtained by substituting s for the unique occurrence of $\diamond$ in S. Intuitively, a context filled with s, e.g., $S[s]$, indicates that s is the first object in a sequence separated by ";".

The transition rules for the sequential program statements complementing those of Figure 4 in Section 2 are listed in Figure 20.

B Storing Unboundedly-Many Identifiers

The ability to store an unbounded number of identifiers, e.g., using the local variables of unboundedly-many running tasks, makes coverability undecidable. In essence, those stored identifiers allow point-to-point communication between arbitrarily many running tasks. While we give an undecidability proof for when *event* identifier storage is unbounded, very similar proofs are carried out using unbounded *buffer* or *task* identifier storage as well. Furthermore, while we assume unlimited threads and buffer concurrency ($T = T_k = \omega$) in what follows, a similar construction is possible with only one thread and one running task per buffer ($T = T_k = 1$), by continuously creating new tasks. Note that we assume an alternate program semantics, in which the CURRENT-EVENT rule of Figure 4 allows local-variable storage.

$$\frac{p \in \mathsf{Procs}}{s_p \in \mathsf{Stmts}} \qquad \frac{S[\_] \in \mathsf{Stmts}}{S[\mathbf{skip}] \in \mathsf{Stmts}} \qquad \frac{S[\mathbf{skip}\,;s] \in \mathsf{Stmts}}{S[s] \in \mathsf{Stmts}} \qquad \frac{S[\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2] \in \mathsf{Stmts}}{S[s_1] \in \mathsf{Stmts}}$$

$$\frac{S[\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2] \in \mathsf{Stmts}}{S[s_2] \in \mathsf{Stmts}} \qquad \frac{S[\mathbf{while}\ e\ \mathbf{do}\ s] \in \mathsf{Stmts}}{S[s\,;\mathbf{while}\ e\ \mathbf{do}\ s] \in \mathsf{Stmts}}$$

$$\frac{v \in \mathsf{Vals} \qquad S[\mathbf{call}\ x := \_] \in \mathsf{Stmts}}{S[x := v] \in \mathsf{Stmts}}$$

Fig. 19. Rules defining the finite set of statements appearing in a program with procedures Procs.

$$\textsc{Skip}\ \frac{}{C[\mathbf{skip}\,;s] \to C[s]} \qquad \textsc{Assume}\ \frac{\mathbf{true} \in e(C)}{C[\mathbf{assume}\ e] \to C[\mathbf{skip}]}$$

$$\textsc{If-Then}\ \frac{\mathbf{true} \in e(C)}{C[\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2] \to C[s_1]} \qquad \textsc{If-Else}\ \frac{\mathbf{false} \in e(C)}{C[\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2] \to C[s_2]}$$

$$\textsc{Loop-Do}\ \frac{\mathbf{true} \in e(C)}{C[\mathbf{while}\ e\ \mathbf{do}\ s] \to C[s\,;\mathbf{while}\ e\ \mathbf{do}\ s]} \qquad \textsc{Loop-End}\ \frac{\mathbf{false} \in e(C)}{C[\mathbf{while}\ e\ \mathbf{do}\ s] \to C[\mathbf{skip}]}$$

$$\textsc{Assign-Global}\ \frac{x \in \mathrm{dom}(g) \qquad v \in e(g, M)}{\langle g, M[x := e]\rangle \to \langle g(x \mapsto v), M[\mathbf{skip}]\rangle} \qquad \textsc{Assign-Local}\ \frac{f_1 = \langle \ell, S\,[x := e]\rangle \qquad x \in \mathrm{dom}(\ell) \qquad v \in e(g, \ell) \qquad f_2 = \langle \ell(x \mapsto v), S\,[\mathbf{skip}]\rangle}{\langle g, m \cup \{\!\{\langle f_1 \boldsymbol{f}, \bot, j, k\rangle\}\!\}\rangle \to \langle g, m \cup \{\!\{\langle f_2 \boldsymbol{f}, \bot, j, k\rangle\}\!\}\rangle}$$

$$\textsc{Call}\ \frac{\boldsymbol{f}_1 = F[\mathbf{call}\ x := p\ e] \qquad \ell \in e(g, F) \qquad f_2 = \langle \ell, s_p\rangle}{\langle g, m \cup \{\!\{\langle \boldsymbol{f}_1, \bot, j, k\rangle\}\!\}\rangle \to \langle g, m \cup \{\!\{\langle f_2 \boldsymbol{f}_1, \bot, j, k\rangle\}\!\}\rangle}$$

$$\textsc{Return}\ \frac{f_1 = \langle \ell_1, S_1[\mathbf{return}\ e]\rangle \qquad v \in e(g, \ell_1) \qquad \boldsymbol{f}_2 = F[\mathbf{call}\ x := \_] \qquad \boldsymbol{f}'_2 = F[x := v]}{\langle g, m \cup \{\!\{\langle f_1 \boldsymbol{f}_2, \bot, j, k\rangle\}\!\}\rangle \to \langle g, m \cup \{\!\{\langle \boldsymbol{f}'_2, \bot, j, k\rangle\}\!\}\rangle}$$

Fig. 20. The semantics for sequential program statements. In the CALL rule, we suppose that the valuation ℓ is obtained by assigning the call arguments of e to the parameters of procedure p.

Theorem 5. *The coverability problem for event-driven asynchronous programs (with unbounded identifier storage) is undecidable.*

Proof. By reduction from the language emptiness problem for Turing machines: given a Turing machine $\mathcal{M}$, we construct the program $P_{\mathcal{M}}$ of Figure 22 which simulates $P_{\mathcal{M}}$ according to Figure 21: for each tape cell we have one running task which executes the procedure `cell`; we assume no limit on the maximum number of threads, $T = \omega$, nor on the running tasks per buffer, $T_k = \omega$. The initial condition dictates that one active task executing `cell` begins with `state` set to the initial state $q_0 \in Q$ of $P_{\mathcal{M}}$. By answering whether the procedure `reached` (whose code is irrelevant and therefore not given) can be executed in some execution of $P_{\mathcal{M}}$, we thus answer whether q_f is reachable in $\mathcal{M}$. □

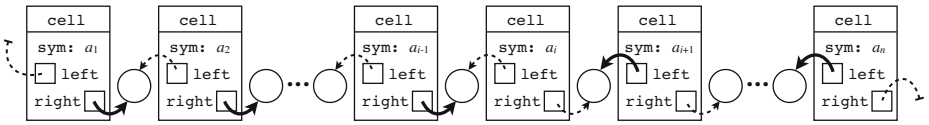

Fig. 21. Encoding a Turing machine tape. Several waiting `cell` tasks, drawn as rectangles, maintain two event variables, `left` and `right`. Arrows depict references to event identifiers, which are drawn as circles; bold arrows originate from tasks waiting on a given event, while dashed arrows denote an otherwise-stored event identifier. The only non-waiting `cell` task is the one pointed to by the tape head; cells to the left (resp., right) wait for their `right` (resp., `left`) event to be signaled.

```
1  proc cell ()
2      var symbol: Σ
3      var left, right: IDs
4
5      // initialize this cell,
6      // and its left neighbor
7      symbol := ⋆;
8      right := event;
9      if ⋆ then
10         left := new event;
11     _ := task cell() {left} ⊥;
12         sync left;
13     wait {left};
14     else left := ⊥
15
16     while true do
17         // test reachability
18         if state = qf then
19             call _ := reached();
20         // choose a transition
21         ...
22         // make the transition
23         ...
24     return
```

```
25 // TM-state stored in
26 // a global variable
27 var state: Q
28
29 // code to choose
30 // an enabled transition
31 let q1,q2: Q, a,b: Σ, d: {L,R} in
32 assume TX(q1,a,d,q2,b);
33 assume state = q1;
34 assume symbol = a;
35
36 // code to make the
37 // chosen transition
38 state := q2;
39 symbol := b;
40 if d = L then
41     sync left;
42     wait {left};
43 else
44     sync right;
45     wait {right};
```

Fig. 22. The program $P_{\mathcal{M}}$ simulating a Turing machine $\mathcal{M} = \langle Q, \Sigma, q_0, q_f, \delta \rangle$ with states Q and alphabet Σ. The predicate $\texttt{TX}(q_1, a, d, q_2, b)$ holds for $\langle q_1, a, d, q_2, b \rangle \in \delta$. The assignment on Line 8 (described in Appendix A) assigns the event identifier on which the current task was activated. Note that here we assume a few trivial syntactic extensions, e.g., **let .. in**, for clarity; they are easily encoded into the rigid syntax of Section 2.