On the Verification of Concurrent, Asynchronous Programs with Waiting Queues

GILLES GEERAERTS, Université Libre de Bruxelles, Belgium
ALEXANDER HEUẞNER, Otto-Friedrich-Universität Bamberg, Germany
JEAN-FRANÇOIS RASKIN, Université Libre de Bruxelles, Belgium

Recently, new libraries, such as Grand Central Dispatch (GCD), have been proposed to directly harness the power of multi-core platforms and to make the development of concurrent software more accessible to software engineers. When using such a library, the programmer writes so-called *blocks*, which are chunks of code, and dispatches them, using *synchronous* or *asynchronous* calls, to several types of waiting queues. A scheduler is then responsible for dispatching those blocks among the available cores. Blocks can synchronize via a global memory. In this paper, we propose Queue-Dispatch Asynchronous Systems as a mathematical model that faithfully formalizes the synchronization mechanisms and the behavior of the scheduler in those systems. We study in detail their relationships to classical formalisms such as pushdown systems, Petri nets, FIFO systems, and counter systems. Our main technical contributions are precise worst-case complexity results for the Parikh coverability problem and the termination problem for several subclasses of our model. We also consider an extension of QDAS with a fork-join mechanism. Adding fork-join to any of the subclasses that we have identified leads to undecidability of the coverability problem. This motivates the study of over-approximations. Finally, we consider handmade abstractions as a practical way of verifying programs that cannot be faithfully modelled by decidable subclasses of QDAS.

Categories and Subject Descriptors: D.2.4 [**Software**]: Software/Program Verification; F.2 [**Analysis of algorithms and problem complexity**]

General Terms: algorithms, theory, verification

Additional Key Words and Phrases: asynchronous programming, waiting queues

ACM Reference Format:
G. Geeraerts, A. Heußner, J.-F. Raskin. 2013. On the Verification of Concurrent, Asynchronous Programs with Waiting Queues. *ACM Trans. Embedd. Comput. Syst.* 0, 0, Article 0 (2013), 25 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

The computing power delivered by computers has followed an exponential growing rate during the last decades. One of the main reasons was the steady increase of the CPU clock rates. This growth, however, has come to an end a few years ago, because further increasing the clock rate would incur major engineering challenges related to power dissipation. In order to overcome this and meet the continuous need for more computing power, multi-core CPU's have been introduced and are now ubiquitous. However, in order to harness the power of multiple cores, software applications need to be fundamentally modified and the programmers now have to write programs with parallelism in mind. But writing parallel programs is a notoriously difficult and error prone task. Also, writing *efficient* and *portable* parallel code for multi-core platforms is difficult, as the number of available cores will vary greatly from one platform to another, and might also depend on the current load, the energy management policy, and so forth.

In order to alleviate the task of the programmer, several high level programming interfaces have been proposed, and are now available on several operating systems. A popular example is *Grand Central Dispatch*, GCD for short, a technology that is present in Mac OS X (since 10.6), iOS (since version 4), and FreeBSD—Windows and Linux ports are currently under development. In GCD, the programmer writes so-called *blocks*, which are chunks of code, and sends them to *queues*, together with several dependency constraints between those blocks (for instance, one block cannot

Table I. The Parikh Coverability Problem for submodels of QDAS

		queue types		
		concurrent	serial	both
dispatch	synchr.	EXPTIME-C	PSPACE-C	EXPTIME-C
	asynchr.	EXPSPACE-C	↯	(↯)
	both	↯	(↯)	(↯)

Legend:
↯: "undecidable",
(–): directly derivable

start before the previous one in the queue has finished). The scheduler is then responsible for dispatching those blocks on the available cores, through a thread pool that the scheduler manages (thereby avoiding the explicit and costly creation/destruction of threads by the programmer that is in addition extremely error-prone).

So far, to the best of our knowledge, no formal model has been proposed for systems relying on GCD or similar technologies, making those programs *de facto* out of reach of current verification methods and tools. This is particularly unfortunate as the control structure of such programs is rich and may exhibit complex behaviors. Indeed, the state-space of such programs is infinite even when types of variables are abstracted to finite domains of values. This is not surprising as asynchronous calls and recursive synchronous calls can send an unbounded number of blocks to queues. Also, those programs are, as any parallel program, subject to concurrency bugs that are difficult to detect using testing only.

Contributions: In this article, we introduce *Queue-Dispatch Asynchronous Systems*, QDAS for short, as a formal model for programs written using libraries such as GCD. Our model is composed of *blocks*, that are finite transition systems with finite data-domain variables that can do *asynchronous* (non-blocking) and *synchronous* (blocking) calls to other blocks (possibly recursively). However, a call does not immediately trigger the execution of the callee: the block is inserted into a queue that can be either *concurrent* or *serial*. In concurrent queues, several blocks can be taken from the queue and executed in parallel, while in serial queues, a block can be dequeued only if the previous block in the queue has completed its execution. Queues are maintained with a FIFO policy. To formalize configurations of such systems, our formal semantics relies on *call task graphs*, CTG for short, in which nodes model tasks that are either in queues or executing, and edges model dependencies between tasks and within queues.

We study the decidability border for the *Parikh coverability problem* and the *termination problem* on several subclasses of QDAS. Our results are summarized in Table I. The *Parikh image* of a CTG is an abstraction that counts for each type and state of blocks the number of occurrences in the CTG. The *Parikh coverability* problem asks for the reachability of a CTG that contains at least a given number of blocks of each type that are in a given set of states. Not surprisingly, this problem is undecidable for QDAS, but we identify several subclasses for which the problem is decidable. For those decidable cases, we characterize the exact complexity of the problem.

The main positive decidability results with precise complexity are as follows. First, we show that QDAS with *only* synchronous calls are essentially equivalent to pushdown systems with finite domain data-variables, and we show that the Parikh coverability problem is EXPTIME-C for QDAS with only synchronous calls to concurrent queues (Theorem 4.6). Second, for QDAS with *only* serials queues with *only* synchronous calls, the problem is PSPACE-C (Theorem 4.7). Third, we show that QDAS with *only* asynchronous calls and *only* concurrent queues are essentially equivalent to lossy Petri nets and show that the Parikh coverability problem is EXPSPACE-C for that class (Theorem 4.14). This decidability border is precise as we show that if we permit either (i) asynchronous calls with synchronous queues, or (ii) synchronous and asynchronous calls with concurrent queues, then the Parikh coverability problem becomes undecidable (Theorem 5.2 and Theorem 5.3). Similar constructions allow us to prove similar results for the termination problem.

Then, we extend QDAS with a fork/join mechanism. When performing a `forkjoin` action, n copies of a block are dispatched to a queue (forked) simultaneously and the execution of the caller is resumed only when all the forked blocks have terminated their execution. Unfortunately, adding this mechanism leads to undecidability of the Parikh coverability problem for all the subclasses of QDAS discussed above.

Finally, in section 8, we show how to design handmade abstractions which are precise enough for the analysis of programs that fall outside decidable cases.

This is an extended version of a preliminary conference version [Geeraerts et al. 2013a].

Related Works: The basic model checking result for asynchronous programs is the EXPSPACE-hardness for the control-state reachability problem. The original reduction by Sen and Viswanathan [2006] is based on Parikh's theorem and derives the lower bound from the Petri net reachability problem [Esparza 1998]. Another reduction that does not rely on the Parikh theorem but on interprocedural dataflow analysis can be found in Jhala and Majumdar [2007].

The close relation between asynchronous programs and (extensions of) Petri nets has also been used in [Ganty et al. 2009; Ganty and Majumdar 2012] to obtain the following results: *fair* termination (i.e., do all dispatched calls terminate) is EXPSPACE-c; boundedness (i.e., is the number of pending calls bounded) is decidable in EXPSPACE, fair non-starvation (i.e., is every pending call eventually dispatched under fairness assumptions) is decidable. The authors also consider extensions of asynchronous programs with cancellation (i.e., an additional operation removing all pending instances of a block) and testing whether there is *no* pending instance of a given block. In the first case, they show a reduction to Petri nets with transfer or reset arcs, in the latter case they show a reduction to Petri nets with one inhibitor arc.

Observe that none of those works consider FIFO queues of waiting blocks, a key point of the GCD programming model, while our model of QDAS does.

A series of parallel programming libraries and techniques is formalized in Bouajjani and Emmi [2012] with the help of *recursively parallel programs*. These allow one to model fork/join based parallel computations based on a reduction to recursive vector addition systems with states. With respect to QDAS and asynchronous programming, recursively parallel programs only cover the classical asynchronous models presented above and not the advanced scheduling strategies for different queues that introduce more sophisticated behaviors.

As QDAS relate to both recursive programs and message passing via unbounded, reliable FIFO queues (cf. the constructions of Section 5.1), our work also relates to ongoing research on the decidability frontier for *communicating pushdown automata*. For instance, [La Torre et al. 2008; Heußner et al. 2012] provide positive decidability results for the reachability problem when restricting the underlying communication topology (and also adding a condition on the stack emptiness when accessing the queues). However, the proposed restrictions cannot be lifted to our setting as in QDAS waiting queues are shared between all running tasks. Another recent and related result is that of [Kochems and Ong 2013], which considers communicating pushdown systems with some restrictions, but, again, these results cannot be applied to our setting.

Acknowledgments The authors are deeply indebted to the anonymous reviewers for their many comments that have definitely improved the quality of the paper.

```
1  global int const l,m,n
2  global int[l][m] matrix1, int[m][n] matrix2, int[l][n] res
3  global c_queue workqueue, s_queue semaphore, int count
4  block increase():
5    count = count + 1
6  block one_cell(int i, int j):
7    for k in range(m):
8      res[i][j]+= matrix1[i][k] * matrix2[k][j]
9    dispatch_s(semaphore,increase())
10 def main():
11   // read input matrix1, matrix2
12   count = 0
13   for i in range(l):
14     for j in range(n):
15       dispatch_a(workqueue,one_cell(i,j))
16   wait(count = l*n)
17   // print the result res
```

Fig. 1. GCD (-like) program for parallel matrix multiplication

2. PRELIMINARIES

Grand Central Dispatch (GCD) is a technology developed by Apple [2010; 2011] that is publicly available at [libdispatch] under a free license. GCD is the main inspiration for the formal model of queue-dispatch asynchronous systems. In the following, we often present our examples as pseudo code using a syntax inspired by GCD. In the GCD framework, the programmer has to organize his code into *blocks*. During the execution of a GCD program, one or several *tasks* run in parallel, each executing a given block (initially, only the `main` block is running). Tasks can call (or *dispatch* in the GCD vocabulary) other blocks, either *synchronously* (the call is blocking), or *asynchronously* (the call is not blocking). A *dispatch* consists of inserting the block into a FIFO *queue*. In our examples, we use the keywords $\mathtt{dispatch}_{\mathrm{a}}$ and $\mathtt{dispatch}_{\mathrm{s}}$ to refer to asynchronous and synchronous dispatches respectively. At any time, the scheduler can decide to *dequeue* blocks from the queues and to assign them to tasks for execution. All queues ensure that the blocks are dequeued in FIFO order; however, the actual scheduling policy depends on the type of queue. GCD supports two types of queues: *concurrent queues* allow several tasks from the same queue to run in parallel, whereas *serial queues* guarantee that *at most one* task from this queue is running. In our examples, concurrent (or serial) queues are declared as global variables of type `c_queue` (`s_queue`). In addition, all blocks have access to the same set of *global variables* (in this work, we assume that the variables range over finite domains).

Example 2.1 (*Matrix Multiplication*). Let us consider the pseudo code in Fig. 1 that computes the product of two integer matrices `matrix1` and `matrix2` of constant size (`l`,`m`,`n`) in a matrix `matrix`. The `main` task forks a series of `one_cell` blocks. Each `one_cell` computes the value of a single cell of the result. The parallelism is achieved via the GCD scheduler, thanks to *asynchronous dispatches* on the *concurrent* queue `workqueue`. Asynchronous dispatches are needed to make sure that `main` is not blocked after each dispatch, and a concurrent queue allows all the `one_cell` block to run in parallel. The variable `count` is incremented each time the computation of a cell is finished and acts as a semaphore for the `main` block, to ensure that `matrix` contains the final result. As only reading and writing to a variable are atomic, we need to guarantee mutual exclusive access of two consecutive operations on `count` (line 5 of Figure 1). This is achieved by a dedicated block `increase` that is dispatched to the *serial* queue `semaphore`. As only `increase` blocks can increase `count`, this queue implicitly locks the access to the variable. Moreover, the synchronous dispatch in line 9 guarantees that a block terminates only after it has increased `count`.

```
1  global bool s_state = 0 // external socket (O)pen/in(U)se
2  global s_queue q
3  global bool disp_close //internal flag: did dispatch close
4  block close():
5    ⋆ // ⋆=close socket,i.e., set s_state = 0
6  block treat(msg):
7    if msg=="Hello":
8       // do something
9    elseif msg=="DoSmth":
10      //... implement client-server protocol
11   else:
12      disp_close=True
13      dispatch_s(q,close) // dangerous dispatch: possible source of a deadlock
14 block listen():
15   if ⋆ and !disp_close: // ⋆ = receive msg on socket
16      dispatch_a(q,treat(msg))
17   if !disp_close:
18      dispatch_a(q,listen)
19 def main():
20   while True:
21     if ⋆ and !disp_close: //⋆=switch s_state from O to U
22      disp_close=False
23      ispatch_a(q,listen
```

Fig. 2. GCD (-style) simple asynchronous server (socket API calls replaced by "$\star$", ignoring return values)

Example 2.2 (*Asynchronous Server*). The pseudo code in Fig. 2 presents a simple asynchronous server similar to the web server example in the Mac Developer Library [Apple 2009]. The `main` task waits for incoming connections on a socket and then transfers the concrete handling of the communication protocol to a listener task. The latter is *asynchronously dispatched* to a *serial* queue. In case of an incoming request, the listener asynchronously dispatches the block `treat` that implements the server's communication protocol by analyzing the message and dispatching corresponding response tasks like `close`. If the protocol did not yet dispatch `close`, i.e., the internal flag `disp_close` is false, the listener automatically restarts by *asynchronously dispatching* itself again. The serial queue guarantees the mutually exclusive access to sockets, and ensures that incoming requests are treated sequentially. In Section 8, we prove automatically that this protocol guarantees that everytime a connection is closed, no `listen` associated to it remaisn active. We also show that our techniques allow to detect a *deadlock*. This occurs when a `close` block is *synchronously* dispatched to queue `q` (in line 13) *and* the `treat` block executing this line has been dispatched on the same queue `q`. For simplicity, we restrict the server to handle only one socket (which however can be directly extended to a pool of queue/flag-pairs that each handle a separate connection in parallel), abstract the socket itself by a Boolean variable, and replace calls to the socket API by the non deterministic choice operator "$\star$". We implicitly assume the external socket library to be independent and (already verified to be) correct.

Notations: Given a set S, let $|S|$ denote its cardinality. For an I-indexed family of sets $(S_i)_{i\in I}$, we write elements of $\prod_{i\in I} S_i$ in bold face, i.e., $\mathbf{s} \in \prod_{i\in I} S_i$. The i-component of $\mathbf{s}$ is written $s_i \in S_i$, and we identify $\mathbf{s}$ with the indexed family of elements $(s_i)_{i\in I}$. We use $\uplus$ to denote the disjoint union of sets. An *alphabet* Σ is a finite set of *letters*. We write Σ^* for the set of all *finite words*, over Σ and denote the empty word by ε. The concatenation of two words w, w' is represented by $w \cdot w'$. For a letter $\sigma \in \Sigma$ and a word $w \in \Sigma^*$, let $|w|_\sigma$ be the number of occurrences of σ in w. We use standard complexity classes, e.g., polynomial time (PTIME) or deterministic exponential time (EXPTIME), and mark completeness by appending "-C" (PSPACE-C).

Let $\mathbb{D}$ be a finite **data domain** with an *initial element* $d_0 \in \mathbb{D}$, and let $\mathcal{X}$ be a finite set of variables ranging over $\mathbb{D}$. A *valuation* of the variables in $\mathcal{X}$ is a function $\mathbb{d} : \mathcal{X} \to \mathbb{D}$. An *atom* is an expression of the form $x = d$ or $x \neq d$, where $x \in \mathcal{X}$ and $d \in \mathbb{D}$. A *guard* is a finite conjunction of atoms. An assignment is an expression of the form $x \leftarrow v$, where $x \in \mathcal{X}$ and $v \in \mathbb{D}$. Let $\mathsf{guards}\,(\mathcal{X})$, $\mathsf{assign}\,(\mathcal{X})$ and $\mathsf{vals}\,(\mathcal{X})$ denote respectively the sets of all guards, assignments and valuations over variables from $\mathcal{X}$. Guards, atoms and valuations have their usual semantics: for all valuations $\mathbb{d}$ of $\mathcal{X}$ and all $g \in \mathsf{guards}\,(\mathcal{X})$, we write $\mathbb{d} \models g$ iff $\mathbb{d}$ satisfies g.

Let us now recall classical models of computation that we use in the rest of the paper to establish undecidability and complexity results.

A **FIFO system** is a tuple $F = \langle S_F, s_F^0, M, \Delta_F \rangle$ where S_F is a finite set of *states*, $s_F^0 \in S_F$ is the *initial state*, M is a finite set of *messages* and $\Delta_F \subseteq S_F \times (\{!a, ?a \mid a \in M\} \cup \{\varepsilon\}) \times S_F$ is the transition relation. We assume the usual semantics of FIFO systems: configurations are pairs of the form $(s, w) \in S_F \times M^*$; an $!a$-labeled transition enqueues a at the channel tail while an $?a$-labeled transition reads (dequeues) an a from the channel head. The reachability problem for FIFO systems asks, given a state $s \in S_f$, whether a configuration of the form (s, w) is reachable in F. This problem is known to be undecidable [Brand and Zafiropulo 1983].

A **two counter machine** (2CM for short) is a tuple $\mathcal{P} = \langle S_\mathcal{P}, s_\mathcal{P}^0, \Delta_\mathcal{P} \rangle$ where $S_\mathcal{P}$ is a finite set of states, $s_\mathcal{P}^0 \in S_\mathcal{P}$ is the initial state and $\Delta_\mathcal{P} \subseteq S_\mathcal{P} \times \{\mathtt{incr}(i), \mathtt{decr}(i), \mathtt{is\_zero}(i) \mid i = 1, 2\} \times S_\mathcal{P}$ is the transition relation. We rely on the classical semantics for 2CM. In particular, counters are always non-negative (a decrement of a null counter cannot be performed). The reachability problem for 2CM is known to be undecidable [Minsky 1967].

A **pushdown system with data** is a pushdown system (see [Bouajjani et al. 1997] for details) equipped with a finite set of variables $\mathcal{X}$ over a finite domain $\mathbb{D}$, and whose transitions can test and modify those variables. Formally a PDS with data is a tuple $\mathcal{P} = \langle Y, \mathcal{X}, y^0, \Gamma, \Delta \rangle$, where Y is a finite set of states, $\mathcal{X}$ is a finite set of variables over a finite domain $\mathbb{D}$, $y^0 \in Y$ is the initial state, Γ is the stack alphabet, and $\Delta \subseteq Y \times \Sigma \times Y$ is the transition relation, with $\Sigma = (\{\mathsf{push}, \mathsf{pop}\} \times \Gamma) \cup \{\mathtt{empty?}\} \cup \mathsf{guards}\,(\mathcal{X}) \cup \mathsf{assign}\,(\mathcal{X})$. We rely on the classical semantics of PDS with data. In particular, a configuration of a PDS with data is a pair $(s, w, \mathbb{d})$ where $s \in Y$ is a control state, $w \in \Gamma^*$ is the stack content, and $\mathbb{d}$ is a valuation of the variables. Clearly, each PDS with data $\mathcal{P} = \langle Y, \mathcal{X}, y^0, \Gamma, \Delta \rangle$ can be translated into an equivalent (as far as reachability is concerned) PDS *without* variables $\mathcal{P}' = \langle Y \times \mathbb{D}^{|\mathcal{X}|}, \emptyset, (y^0, \mathbb{d}_0, \dots, \mathbb{d}_0), \Gamma, \Delta' \rangle$, by encoding the valuations of the variables in the discrete states, and adapting the transition relation accordingly. Further, by reducing the emptiness test of the intersection of a context-free language with n regular languages to the reachability problem in such a PDS (as in [J. Esparza 2003]), we obtain:

PROPOSITION 2.3. *The reachability problem is* EXPTIME-C *for* PDS *with data.*

A **Petri net** (PN) is a tuple $N = \langle P, T, m_0 \rangle$ where P is a finite set of places, a *marking* of the places is a function $m : P \to \mathbb{N}$ that associates, to each place $p \in P$ a number $m(p)$ of tokens, T is finite set of transitions, each transition $t \in T$ is a pair (I_t, O_t) where[1] $I_t : P \to \{0, 1\}$ and $O_t : P \to \{0, 1\}$ are respectively the *input* and *output functions* of t, and m_0 is the *initial marking*. Given two markings m_1 and m_2, we let $m_1 \preceq m_2$ iff

[1]Note that we restrict to PN where transitions can consume and produce at most one token in each place. However, places are still unbounded, and this restriction is w.l.o.g. as far as complexity is concerned [Esparza 1998]. When discussion approximations in Section 7, we extend the discussion to nets that can produce/consume an arbitrary number of tokens.

$m_1(p) \leq m_2(p)$ for all $p \in P$. Given a marking m, a transition $t = (I_t, O_t)$ is *enabled* in m iff $m(p) \geq I_t(p)$ for all $p \in P$. When t is enabled in m, one can *fire* the transition t in m, which produces a new marking m' s.t. $m'(p) = m(p) - I_t(p) + O_t(p)$ for all p. This is denoted $m \xrightarrow{t} m'$, or simply $m \rightarrow m'$ when the transition identity is irrelevant. A *execution* of a PN is either a sequence $m_0 m_1 \ldots m_n$ s.t. for all $1 \leq i \leq n$: $m_{i-1} \rightarrow m_i$ or an infinite sequence $m_0 m_1 \ldots m_j \ldots$ s.t. for all $i \geq 1$: $m_{i-1} \rightarrow m_i$. For a PN N, we denote by *Reach*(N) (resp. *Cover*(N)) the *reachability (coverability) set* of N, i.e. the set of all markings m s.t. there exists a run $m_0 m_1 \ldots m_n$ of N with $m = m_n$ ($m \preceq m_n$). The *coverability problem* asks, given a PN N and a marking m, whether $m \in$ *Cover*(N). It is EXPSPACE-C [Esparza 1998]. The termination problem, i.e., whether all executions of the Petri net are finite, is EXPSPACE-C [Lipton 1976; Rackoff 1978].

3. QUEUE-DISPATCH ASYNCHRONOUS SYSTEMS

Syntax: Let us define our formal model for queue-dispatch asynchronous systems. Let $\mathbb{D}$ be a finite data domain containing an *initial value* d_0. A *queue-dispatch asynchronous system* (QDAS) $\mathcal{A}$ is a tuple $\langle CQID, SQID, \Gamma, main, \mathcal{X}, \Sigma, (\mathcal{TS}_\gamma)_{\gamma \in \Gamma} \rangle$ where:

— $CQID$ and $SQID$ are sets of *(c)oncurrent* and *(s)erial queues* respectively;
— Γ is a finite set of *blocks* and $main \in \Gamma$ is the *initial block*. A block $\gamma \in \Gamma$ is a tuple $\mathcal{TS}_\gamma = \langle S_\gamma, s^0_\gamma, f_\gamma, \Sigma, \Delta_\gamma \rangle$ where $\langle S_\gamma, s^0_\gamma, \Sigma, \Delta_\gamma \rangle$ is an LTS with final state $f_\gamma \in S_\gamma$;
— $\mathcal{X}$ is a finite set of variables, taking value in $\mathbb{D}$;
— $\Sigma = \Sigma_d \cup \mathsf{guards}\,(\mathcal{X}) \cup \mathsf{assign}\,(\mathcal{X})$ is the set of *actions*, where Σ_d is the set of possible dispatch actions: $\Sigma_d = \{\mathtt{dispatch_s}, \mathtt{dispatch_a}\} \times (CQID \cup SQID) \times (\Gamma \setminus \{main\})$.

We assume that $SQID, CQID, \Gamma, \mathcal{X}$, and all S_γ for $\gamma \in \Gamma$ are disjoint from each other. Let $S = \biguplus_{\gamma \in \Gamma} S_\gamma$, $F = \biguplus_{\gamma \in \Gamma} \{f_\gamma\}$, $\Delta = \biguplus_{\gamma \in \Gamma} \Delta_\gamma$, and $QID = SQID \uplus CQID \uplus \{\imath\}$ (where $\imath \notin SQID \cup CQID$). We further assume that there is a symbol τ with $\tau \notin \Sigma$.

Remark 3.1. Observe that Examples 2.1 and 2.2 can be encoded as a QDAS. The control flow graph of each GCD block must be expressed as an LTS, and an adequate abstraction must be applied in order to reduce the data domain to a finite one (using, for instance, classical abstract interpretation techniques [Cousot and Cousot 1977]).

Call-task graphs: We formalize the semantics of QDAS using the notion of *call-task graph* (CTG) to describe the system's global configurations.

A configuration of a QDAS (see Fig. 3 for an example) contains a set of running tasks, represented by *task vertices* (round nodes in the figure), and a set of called but unscheduled blocks, represented by *call vertices* (square nodes). Call vertices are held by queues, and the FIFO order of each queue is represented by *queue edges* (solid edges) from each node to his predecessor in the queue (i.e., the only node of the queue without outgoing edge is the head of the queue). Synchronous calls add additional dependencies between nodes as the caller needs to wait for the termination of the callee. This is represented by a *wait edge* (dashed edges) from the caller to the callee. Wait edges are also inserted between the head of a *serial* queue and the running task that has been extracted from this queue (if it exists) to indicate that the task has to terminate before a new block can be dequeued. Note that the tasks that can execute are precisely those whose vertices have no outgoing edge (others are currently blocked). Each node v is labeled by a block $\lambda(v)$, and by the identifier *queue*(v) of the queue that contains it (for call vertices) or that contained it (for task vertices). Task vertices are labeled by their current state *state*(v) (for convenience, we also label call vertices by the initial state of their respective blocks – not shown in the figure).

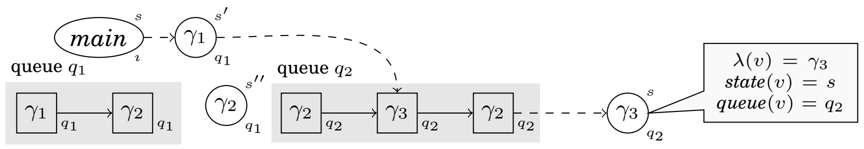


Fig. 3. CTG for a QDAS with a concurrent queue q_1 and a serial queue q_2

Example 3.2. The CTG in Fig. 3 depicts a configuration of a QDAS with two queues. Queue q_2 is serial (note the wait edge from the γ_2 task in the head of the queue, to the running γ_3 task) and contains $\gamma_2\gamma_3\gamma_2$, and q_1 is parallel with content $\gamma_1\gamma_2$. There are 4 active tasks, two of which (`main` and the task running γ_1) are blocked. The task running γ_3 has been dequeued from q_2 and is currently at location s.

Formally, given a QDAS $\mathcal{A} = \langle CQID, SQID, \Gamma, main, \mathcal{X}, \Sigma, (\mathcal{TS}_\gamma)_{\gamma\in\Gamma}\rangle$, a *call-task graph* over $\mathcal{A}$ is a tuple $G_\mathcal{A} = \langle V, E, \lambda, \mathit{queue}, \mathit{state}\rangle$ where: $V = V_C \uplus V_T$ is a finite set of *vertices*, partitioned into a set V_C of *call vertices* and a set V_T of *task vertices*; $E \subseteq V \times V$ is a set of *edges*; $\lambda : V \to \Gamma$ labels each vertex by a block; $\mathit{queue} : V \to QID \uplus \{\imath\}$ associates each vertex to a queue identifier (or $\imath$); and $\mathit{state} : V \to S$ associates each vertex to an LTS state. For each $q \in QID$, let $V_q = \{v \in V_C \,|\, \mathit{queue}(v) = q\}$. We let $E_Q = \biguplus_{q\in QID} E_q$ be the set of *queue edges*, with $E_q = E \cap (V_q \times V_q)$ for all $q \in QID$; and let $E_w = E \setminus E_q$.

A CTG is *empty* iff $V = \emptyset$. A *path* (of length n) in $G_\mathcal{A}$ is a sequence of vertices $v_0, v_1, \ldots, v_n$ s.t. for all $1 \leq i \leq n$: $(v_{i-1}, v_i) \in E$. Such a path is *simple* iff $v_i \neq v_j$ for all $1 \leq i < j \leq n$. The *restriction* of $G_\mathcal{A}$ to $V' \subseteq V$ is the CTG $G'_\mathcal{A} = \langle V', E', \lambda', \mathit{queue}', \mathit{state}'\rangle$, where $E' = E \cap (V' \times V')$, and λ', queue' and state' are respectively the restrictions of λ, queue and state to V'.

In the rest of the paper, we assume that all CTG are *well-formed* as defined by:

(1) For each $v \in V_T$: $\mathit{state}(v) \in S_{\lambda(v)}$ where $S_{\lambda(v)}$ are the states of $\mathcal{TS}_{\lambda(v)}$.
(2) Each *call* vertex has at most one outgoing (queue or wait) edge, at most one incoming *wait* edge, and at most one incoming *queue* edge. Each *task* vertex has at most one outgoing, and at most one incoming *wait* edge.
(3) For each $q \in QID$, the restriction of $G_\mathcal{A}$ to V_q is either empty or contains one and only one simple path of length $|V_q| - 1$. This ensures that queue nodes are arranged in the graph as a list (the queue is thus well-formed).
(4) For each $q \in SQID$, there is at most one task vertex v s.t. $\mathit{queue}(v) = q$. This ensures that serial queues force the serial execution of the block they contain.

For convenience, we also introduce the following notations: Let $G_\mathcal{A}$ be a CTG, and let q be a queue identifier of $\mathcal{A}$. Then, $\mathit{head}(q, G_\mathcal{A})$ and $\mathit{tail}(q, G_\mathcal{A})$ denote respectively the head and the tail of q in the configuration described by $G_\mathcal{A}$. That is, $\mathit{tail}(q, G_\mathcal{A})$ is the call vertex $v \in V_q$ that has no incoming queue edge, or $\perp$, if such a vertex does not exist; and $\mathit{head}(q, G_\mathcal{A})$ is the call vertex $v \in V_q$ that has no outgoing *queue* edge (but possibly an outgoing *wait* edge), or $\perp$, if such a vertex does not exist. Note that, when they exist, these vertices are necessarily unique because of the well-formedness assumptions. Finally, we say that a vertex v is *unblocked* iff it has no outgoing edge, and that it is *final* iff (i) v is an *unblocked task* vertex; and (ii) $\mathit{state}(v) = f_\lambda(v)$ (that is, v represents a task that has reached the final state of its transition system and is not waiting for another task).

Let us now define several operations to manipulate queues and blocks in CTG. We will rely on these operations when defining the formal semantics of QDAS. For the following operations, let $\mathcal{A}$ be a QDAS and $G_\mathcal{A} = \langle V, E, \lambda, \mathit{queue}, \mathit{state}\rangle$ be a CTG for $\mathcal{A}$:

Restriction: for all $v \in V$: $G \setminus v$ is the restriction of G to $V \setminus \{v\}$.

Enqueue operation: for all $\gamma \in \Gamma$ and $q \in QID$, $\mathsf{enqueue}(q,\gamma)(G_{\mathcal{A}})$ is the CTG $\langle V', E', \lambda', \textit{queue}', \textit{state}'\rangle$ where: $V' = V \cup \{v'\}$, v' is a fresh call vertex, $\lambda(v') = \gamma$, $\textit{queue}'(v') = q$, $\textit{state}'(v') = s^0_\gamma$, and for all $v \in V$: $\lambda'(v) = \lambda(v)$, $\textit{queue}'(v) = \textit{queue}(v)$, $\textit{state}'(v) = \textit{state}(v)$. Finally, $E' = E \cup E_1 \cup E_2$, where: (i) $E_1 = \{(v', \textit{tail}(G_{\mathcal{A}}, q))\}$ if $\textit{tail}(G_{\mathcal{A}}, q) \neq \bot$, and $E_1 = \emptyset$ otherwise, and (ii) if $v \in V$ is a *task* node s.t. $\textit{queue}(v) = q \in SQID$ and $\textit{tail}(G_{\mathcal{A}}, q) \neq \bot$, then $E_2 = \{(v', v)\}$, otherwise $E_2 = \emptyset$. Intuitively, this operation inserts a call to γ in the queue q, by creating a new vertex v' and adding an edge to maintain the FIFO ordering, if necessary (set E_1). In the case of a *serial* queue that was empty before the enqueue, a supplementary edge (in set E_2) might be necessary to ensure that v' is blocked by a currently running v which has been extracted from q.

Dequeue operation: for all $q \in QID$, if $\textit{head}(q)$ is different from $\bot$ and *unblocked*, then $\mathsf{dequeue}(q)(G_{\mathcal{A}})$ is the CTG $\langle V'_C \uplus V'_T, E', \lambda, \textit{queue}, \textit{state}\rangle$ where $V'_C = V_C \setminus \{\textit{head}(q)\}$ and $V'_T = V'_T \cup \{\textit{head}(q)\}$. Otherwise, $\textit{head}(q) = \bot$ and $\mathsf{dequeue}(q)(G_{\mathcal{A}})$ is undefined. Intuitively, this operation removes the first (with respect to the FIFO ordering) block from q and turns the corresponding *call* vertex $\textit{head}(q)$ into a *task* vertex, meaning that the block is now running as a task.

Executing one step in a task: for all $\delta = (s, a, s') \in \Delta$, $\mathsf{step}(\delta)(G_{\mathcal{A}})$ is a *set* of CTG defined as follows. $\langle V, E, \lambda, \textit{queue}, \textit{state}'\rangle \in \mathsf{step}(\delta)(G_{\mathcal{A}})$ iff there exists an *unblocked* $v \in V_T$ s.t. $\textit{state}(v) = s$, $\textit{state}'(v) = s'$ and for all $v' \neq v$: $\textit{state}'(v') = \textit{state}(v')$. Note that $\mathsf{step}(\delta)(G_{\mathcal{A}})$ can be empty. Intuitively, each graph in $\mathsf{step}(\delta)(G_{\mathcal{A}})$ corresponds to the firing of an a-labeled transition by a task that is not blocked.

Introducing a wait dependency between two tasks: for all unblocked $v \in V \cup \{\bot\}$, all $v' \in V$: $\mathsf{letwait}(v, v')(G_{\mathcal{A}})$ is either the CTG $G_{\mathcal{A}}$ if $v = \bot$, or the CTG $\langle V, E \cup (v, v'), \lambda, \textit{queue}, \textit{state}\rangle$ if $v \neq \bot$. Intuitively, this operation adds a wait edge between nodes v and v' when $v \neq \bot$, and does not modify the CTG otherwise.

Semantics of QDAS: With all these ingredients, we can now define the formal semantics of QDAS, as infinite transitions systems whose configurations contain a CTG to express the current state of the running tasks and queues of the system. Let $\mathcal{A} = \langle CQID, SQID, \Gamma, main, \mathcal{X}, \Sigma, (\mathcal{TS}_\gamma)_{\gamma\in\Gamma}\rangle$ be a QDAS. A *configuration* is a pair $(G, \mathbb{d})$, where G is a CTG of $\mathcal{A}$ and $\mathbb{d} \in \mathsf{vals}(\mathcal{X})$. The semantics of $\mathcal{A}$ is the labeled transition system $[\![\mathcal{A}]\!] = \langle C, c^0, \widetilde{\Sigma}, \Longrightarrow\rangle$ where: (i) C contains all the pairs $(G, \mathbb{d})$ where $\mathbb{d} \in \mathsf{vals}(\mathcal{X})$, and G is a CTG of $\mathcal{A}$, (ii) $c^0 = (G^0, \mathbb{d}^0)$ with $\mathbb{d}^0(x) = d_0$ for all $x \in \mathcal{X}$, and $G^0 = \langle \{v^0\}, \emptyset, \lambda, \textit{queue}, \textit{state}\rangle$, where v^0 is a *task node*, $\lambda(v^0) = main$, $\textit{state}(v^0) = s^0_{\text{main}}$ and $\textit{queue}(v^0) = \imath$, (iii) $\widetilde{\Sigma} = \Sigma \uplus \{\tau\}$ and (iv) $\big((G, \mathbb{d}), a, (G', \mathbb{d}')\big) \in \Longrightarrow$ iff one of the following cases holds:

Asynchronous dispatch: $a = \mathtt{dispatch}_{\mathtt{a}}(q, \gamma)$, $\mathbb{d}' = \mathbb{d}$, and there are $\delta = (s, a, s') \in \Delta$ and $G'' \in \mathsf{step}(\delta)(G)$ s.t.: $G' = \mathsf{enqueue}(q, \gamma)(G'')$.

Synchronous dispatch: $a = \mathtt{dispatch}_{\mathtt{s}}(q, \gamma)$, $\mathbb{d}' = \mathbb{d}$ and there are $\delta = (s, a, s') \in \Delta$ and $G'' \in \mathsf{step}(\delta(G))$ s.t.: $G' = \mathsf{letwait}(v, v')\big(\mathsf{enqueue}(q, \gamma)(G'')\big)$ where v is the node whose *state* has changed during the step operation, and v' is the fresh node that has been created by the $\mathsf{enqueue}$ operation. That is, a queue vertex v' labeled by γ is added to q and a *wait* edge is added between the node v representing the task that performs the *synchronous* dispatch, and v', as the dispatch is *synchronous*.

Test: $a = g \in \mathsf{guards}(\mathcal{X})$, $\mathbb{d}' = \mathbb{d}$, $\mathbb{d} \models g$, and there is $\delta = (s, a, s') \in \Delta$ s.t. $G' \in \mathsf{step}(\delta)(G)$.

Assignment: $a = x \leftarrow v \in \mathsf{assign}(\mathcal{X})$, $\mathbb{d}'(x) = v$, for all $x' \neq x$: $\mathbb{d}'(x') = \mathbb{d}(x')$ and there is $\delta = (s, a, s') \in \Delta$ s.t. $G' \in \mathsf{step}(\delta)(G)$.

Scheduler action: $a = \tau$, $\mathbb{d}' = \mathbb{d}$ and either

— there is a final vertex v s.t. $G' = G \setminus v$;
— or there is $q \in CQID$ s.t. *head*$(q, G) \neq \bot$ and $G' = \mathsf{dequeue}(q)(G)$. That is, the scheduler schedules a block (represented by v) from a concurrent queue.
— or there is $q \in SQID$ s.t. *head*$(q, G) = v$, v is *unblocked*, as well as $G' = \mathsf{letwait}(\textit{head}(q, G''), v)(G'')$ and $G'' = \mathsf{dequeue}(q)(G)$. That is, the scheduler schedules a block (represented by v) from the serial queue q. As the queue is serial, a *wait* edge is inserted between the next waiting block in q (now represented by *head*(q, G'')) and v.

A *run* ρ of a QDAS is either a finite sequence $c_0a_1c_1a_2 \dots a_nc_n$ or an infinite sequence $c_0a_1c_1a_2 \dots a_jc_j \dots$ where each c_i is a configuration of $\mathcal{A}$, each a_i is an action of $\mathcal{A}$, $c_0 = c^0$ and $(c_i, a_{i+1}, c_{i+1}) \in \Longrightarrow$ for all i, and $c_0 = c^0$. A configuration c is *reachable* in $\mathcal{A}$ iff there exists a finite run $c_0a_1c_1a_2 \dots a_nc_n$ of $\mathcal{A}$ s.t. $c_n = c$. We let *Reach*$(\mathcal{A})$ be the set of reachable configurations of $\mathcal{A}$.

Decision Problems for QDAS: Typical verification questions on concurrent systems in general and QDAS in particular ask whether there exists an execution in which at least a certain number of processes reach a designated state. For instance, a *mutual exclusion* problem on QDAS would ask whether it is possible to reach a configuration in which at least two tasks are executing the same block γ and are in the same control state s. If yes, the mutual exclusion (of control state s) is violated. A concrete example is to check, in Example 2.1, whether there can be at least two blocks of type `increase` running simultaneously.

To formalise such problems, we rely on the notion of a *Parikh image* of a CTG G (for some QDAS $\mathcal{A}$) which is a function $f : S \to \mathbb{N}$, s.t. for all $s \in S$: $f(s) = |\{v \in V \mid \textit{state}(v) = s\}|$. Given two Parikh images $\mathsf{Parikh}(G)$ and $\mathsf{Parikh}(G')$, we let $\mathsf{Parikh}(G) \preceq \mathsf{Parikh}(G')$ iff for all $s \in S$: $\mathsf{Parikh}(G)(s) \leq \mathsf{Parikh}(G')(s)$. Clearly, the mutual exclusion of control state s is violated in $\mathcal{A}$ iff a configuration $(G, \mathbb{d})$ with $\mathsf{Parikh}(G)(s) \geq 2$ is reachable in $\mathcal{A}$, which is an instance of the *Parikh coverability problem* for QDAS. Formally, given a QDAS $\mathcal{A}$ and a Parikh image f, this problem asks whether there exists a $c = (G, \mathbb{d}) \in \textit{Reach}(\mathcal{A})$ such that $f \preceq \mathsf{Parikh}(G)$. When the answer to this question is 'yes', we say that f is *Parikh-coverable* in $\mathcal{A}$.

The other problem we consider is the *(universal) termination problem*: given a QDAS $\mathcal{A}$, it asks whether all executions of $\mathcal{A}$ are finite. Considering Example 2.1 again, this allows to check whether the `main` task terminates, i.e., all dispatched blocks eventually terminate.

The Family of QDAS Submodels: In the following section, we study the Parikh coverability problem from a computational point of view. As expected, this problem is undecidable in general. However, when restricting the types of queues and dispatches that are allowed, it is possible to retain decidability. Formally, we consider the following subclasses of QDAS. A QDAS $\mathcal{A}$ with set of transitions Δ, set of serial queues $SQID$ and set of concurrent queues $CQID$, is *synchronous* iff there exists no $(s, a, s') \in \Delta$ with $a \in \{\texttt{dispatch\_a}\} \times QID \times \Gamma$; it is *asynchronous* iff there exists no $(s, a, s') \in \Delta$ with $a \in \{\texttt{dispatch\_s}\} \times QID \times \Gamma$; it is *concurrent* iff $SQID = \emptyset$ and $CQID \neq \emptyset$; it is *serial* iff $CQID = \emptyset$ and $SQID \neq \emptyset$; it is *queueless* iff $CQID = SQID = \emptyset$.

4. DECIDING THE PARIKH COVERABILITY PROBLEM

4.1. Queueless QDAS

In a queueless QDAS, there is no possible dispatch, so the only task that can execute at all time is the `main` one. Configurations of queueless QDAS can thus be encoded as tuples $(s, \mathbb{d})$, where s is a state of `main`, and $\mathbb{d}$ is a valuation of the variables. Hence queueless QDAS are essentially LTS with variables over a finite data domain, and thus:

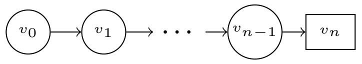
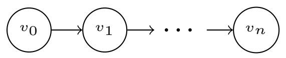
(a)  (b) 

Fig. 4. The two possible forms of reachable CTGs in a synchronous QDAS

PROPOSITION 4.1. *The Parikh coverability is* PSPACE-C *for queueless* QDAS.

4.2. Synchronous QDAS

In synchronous QDAS, no concurrency can occur in the sense there is at most one running task that can execute an action at all times. All the other tasks have necessarily performed a *synchronous* dispatch and are thus blocked. More precisely, in every reachable configuration $(G, \mathbb{d})$ of a synchronous QDAS, G is of one of the forms (a) or (b) depicted in Fig. 4 (with $v_0, \ldots, v_{n-1} \in V_T$ and either $v_n \in V_T$ or $v_n \in V_C$). When the current CTG is of the form (a), the only possible action is that the scheduler starts running v_n's block and we obtain a graph of the form (b). In the case where the CTG is of the form (b), either v_n terminates, which removes v_n from the CTG; or v_n executes an internal action, which does not change the shape of the CTG; or v_n performs a synchronous dispatch, which adds a call vertex as successor of v_n. W.l.o.g., we assume in the following that for synchronous QDAS the combined action of $\mathtt{dispatch_s}$ and scheduling the dispatched block is atomic. Thus, intuitively, all CTGs behave like a *stack*: a task termination corresponds to a pop, while a synchronous call corresponds to a push.

From QDAS to PDS: Let $\mathcal{A} = \langle \emptyset, SQID, \Gamma, main, \mathcal{X}, \Sigma, (\mathcal{TS}_\gamma)_{\gamma\in\Gamma} \rangle$ be a synchronous QDAS with set of locations S, set of transitions Δ, set of final states F. Let G be a CTG of one of the forms given in Fig. 4, and let $w = w_0 w_1 \cdots w_n$ be a word in S^*. Then, G *is encoded by* w, written $G \triangleright w$, iff either G is empty and $w = \varepsilon$, or G is not empty and $w_i = \mathit{state}(v_i)$ for all $0 \le i \le n$. From $\mathcal{A}$, we build a pushdown system with data $\mathcal{P}_\mathcal{A} = \langle Y, \mathcal{X}, y^0, S, \Delta_\mathcal{P} \rangle$ where:

— the set of states is $Y = S \cup \{\tau\}$ and the initial state is $y^0 = s^0_{main}$
— a tuple $(y, a, y') \in \Delta_\mathcal{P}$ iff one of the following holds: (i) either $a \in \mathsf{guards}(\mathcal{X}) \cup \mathsf{assign}(\mathcal{X})$ and $(y, a, y') \in \Delta$ (ii) or $a = \mathtt{push}(s')$, $(s, \mathtt{dispatch_s}(q, \gamma), s') \in \Delta$ and $y' = s^0_\gamma$ (iii) or $a = \mathtt{pop}(s)$, $y \in F$ and $y' = s$ (iv) or $a = \mathtt{empty?}$, $y = f_{main}$, and $y' = \tau$.

Thus, at all times, the current location of $P_\mathcal{A}$ encodes the current location of the (single) running block in $\mathcal{A}$, and the stack content records the sequence of synchronous dispatches, as described above. A guard or assignment in $\mathcal{A}$ is kept as is in $P_\mathcal{A}$. A synchronous dispatch $(s, \mathtt{dispatch_s}(q, \gamma), s')$ in $\mathcal{A}$ is simulated by a push of s' (to record the local state that has to be reached when the callee terminates) and moves the current state of $P_\mathcal{A}$ to the initial state of γ. The termination of a block is simulated by a pop (and we use the $\mathtt{empty?}$ action for the termination of $main$).

PROPOSITION 4.2. *For all synchronous* QDAS $\mathcal{A}$*, we can construct a pushdown system with data* $\mathcal{P}_\mathcal{A}$ *such that:*

(i) for all runs $\rho = (G_0, \mathbb{d}_0)a_1(G_1, \mathbb{d}_1)\ldots a_n(G_n, \mathbb{d}_n)$ *of* $\mathcal{A}$*, there exists a run* $\pi = (s_0, w_0, \mathbb{d}'_0)a_1(s_1, w_1, \mathbb{d}'_1)\ldots a_n(s_n, w_n, \mathbb{d}'_n)$ *of* $\mathcal{P}_\mathcal{A}$ *with* $\mathbb{d}_i = \mathbb{d}'_i$ *and* $G_i \triangleright w_i$ *for all* $0 \le i \le n$ *and, symmetrically*
(ii) for all runs $\pi = (s_0, w_0, \mathbb{d}'_0)a_1(s_1, w_1, \mathbb{d}'_1)\ldots a_n(s_n, w_n, \mathbb{d}'_n)$ *of* $\mathcal{P}_\mathcal{A}$ *there exists a run* $\rho = (G_0, \mathbb{d}_0)a_1(G_1, \mathbb{d}_1)\ldots a_n(G_1, \mathbb{d}_1)$ *of* $\mathcal{A}$ *with* $\mathbb{d}_i = \mathbb{d}'_i$ *and* $G_i \triangleright w_i$ *for all* $0 \le i \le n$

PROOF. We only prove point (i), the other direction follows from similar arguments. In the usual semantics for pushdown systems with data, configurations are of the form $(y, w, d) \in Y \times S^* \times \mathbb{D}^\mathcal{X}$, where y is the current state and w is the stack content. Since

$Y = S \cup \{\tau\}$, by definition, we rewrite each configuration (y, w, d) as a pair $(w \cdot y, d)$, i.e., the first element is a word from $S^* \cdot (S \cup \{\tau\})$ that encodes both the stack content and the current state of $\mathcal{P}_\mathcal{A}$, and $d \in \mathbb{D}^\mathcal{X}$ as before.

Let $\rho = (G_0, \mathbb{d}_0)a_1(G_1, \mathbb{d}_1)a_2 \ldots a_n(G_n, \mathbb{d}_n)$ be a run of $\mathcal{A}$. Let us build by induction a corresponding run $(x_0, \widehat{\mathbb{d}}_0)a_1(x_1, \widehat{\mathbb{d}}_1)a_2 \ldots a_n(x_n, \widehat{\mathbb{d}}_n)$ of $\mathcal{P}_\mathcal{A}$ with $x_i = w_i \cdot y_i \in S^* \cdot (S \cup \{\tau\})$ and $\widehat{\mathbb{d}}_i \in \mathbb{D}^\mathcal{X}$ for $1 \le i \le n$. The base case holds by definition of $\mathcal{P}_\mathcal{A}$. Let us assume that $(x_0, \widehat{\mathbb{d}}_0) \ldots (x_j, \widehat{\mathbb{d}}_j)$ is a run of $\mathcal{P}_\mathcal{A}$ s.t. $\mathbb{d}_i = \mathbb{d}'_i = \widehat{\mathbb{d}}_i$ and $G_i \triangleright w_i$ for all $0 \le i \le j$ and let us show how to extend it. Since ρ is a run of a synchronous QDAS, G_j is a path of vertices $v_0 \ldots v_n$ connected by wait edges. We consider several cases depending on a_{j+1}:

Sync. dispatch: $a_{j+1} = \mathtt{dispatch_s}(\gamma, q)$ In this case, $\mathbb{d}_j = \mathbb{d}_{j+1}$ and G_{j+1} is a path graph $v_0 v_1 \ldots v_n v_{n+1}$ where v_{n+1} is a fresh vertex with $\mathit{state}(v_{n+1}) = v^0_\gamma$. In the run of $\mathcal{P}_\mathcal{A}$, we let $\widehat{\mathbb{d}}_j = \widehat{\mathbb{d}}_{j+1}$ and $x_{j+1} = x_j \cdot s^0_\gamma$ (i.e. $\mathcal{P}_\mathcal{A}$ pushes the current state on the stack, jumps to state s^0_γ).

Test/Assignment: In this case, G_{j+1} equals G_j except for $\mathit{state}_j(v_n) = s$ and $\mathit{state}_{j+1}(v_n) = s'$ and a possible change of $\mathbb{d}_{j+1}$ if a_{j+1} is an assignment. We simulate this action in $\mathcal{P}_\mathcal{A}$ by letting $\widehat{\mathbb{d}}_{j+1} = \mathbb{d}_{j+1}$ and $x_{j+1} = w \cdot s'$ (assuming $x_j = w \cdot s$).

Termination In this case, G_j is a non-empty path ending in v with $\mathit{state}_j(v) \in F$, $G_{j+1} = G_j \setminus v$, and $\mathbb{d}_j = \mathbb{d}_{j+1}$ (note that G_{j+1} could be empty). We let $\widehat{\mathbb{d}}_j = \widehat{\mathbb{d}}_{j+1}$. To define x_{j+1},we consider two further cases: (i) Either $x_j = w_j \cdot y_j$ with $w_j \neq \varepsilon$. In this case, we use a pop action in $\mathcal{P}_\mathcal{A}$, and let $x_{j+1} = w$. (ii) Or $x_j = y_j \in S$ (i.e., the stack is empty). In this case, $x_j = f_{main}$, G_j is a path of length 1 and G_{j+1} is empty. We thus execute the $\mathtt{empty}?$ action in $\mathcal{P}_\mathcal{A}$, and obtain $x_{j+1} = \tau$

Clearly, in all those cases, $G_{j+1} \triangleright w_{j+1}$ and $\mathbb{d}_{j+1} = \widehat{\mathbb{d}}_{j+1}$. Hence the proposition holds. □

The following Lemma shows that we can test a Parikh condition thanks to a finite automaton.

LEMMA 4.3. *Let S be a finite set, and let $f : S \to \mathbb{N}$ be a function. There is a finite automaton $\mathcal{F}_f$ that accepts $\{w \in S^* : |w|_s \ge f(s)$ for all $s \in S\}$ and whose size is exponential in $|S|$ but, for fixed $|S|$, polynomial in $max_{s \in S} f(s)$ (length of binary encoding).*

PROOF. Let $k = max_{s \in S} f(s)$ (which must exist as S is finite). Then, $\mathcal{F}_f$ is the finite automaton $\langle Q, S, q^0, \Delta, q^f \rangle$ where: (i) $Q = \{0 \ldots k\}^S$ is the set of states, interpreted as S-indexed vectors of values in $0 \ldots k$; (ii) S is the alphabet; (iii) $q_0 = f$ is the initial state; and (iv) q^f is the final state, with $q^f(s) = 0$ for all $s \in S$. The transitions of $\mathcal{F}_f$ are defined as follows: $(q, s, q') \in \Delta$ iff $q'(s) = \max\{0, q(s) - 1\}$ and $q'(t) = q(t)$ for all $t \neq s$. Thus, each transition labeled by s decrements the $q(s)$ *counter* if it is not zero, and does not modify it otherwise. Observe that $\mathcal{F}_f$ is *deterministic* and *complete*. Obviously, the size of $\mathcal{F}_f$ respects the constraints of the Lemma. Let us now show that $L(\mathcal{F}_f) = \{w \in S^* \mid$ for all $s \in S : |w|_s \ge f(s)\}$. Let $w = a_1 \ldots a_n \in \mathcal{L}(\mathcal{F}_f)$ be a word accepted by the run $q_0 a_1 q_1 \ldots a_n q_n$ where $q_0 = q^0$ and $q_n = q^f$. By definition of Δ, it is clear that $|w|_s \ge q_0(s) = f(s)$ for all $s \in S$. Symmetrically, let w be word that is not in $\mathcal{L}(\mathcal{F}_f)$, and let $q_0 a_1 q_1 \ldots a_n q_n$ be the unique run of $\mathcal{F}_f$ in w. Let $s \in S$ be s.t. $q_n(s) > 0$. Such an s must exist since $w \notin \mathcal{L}(\mathcal{F}_f)$ and thus $q_n \neq q^f$. Since each transition (q_{i-1}, a_i, q_i) ensures that $q_{i-1}(s) \ge q_i(s)$, we conclude that $|w|_s < f(s)$. □

This Parikh condition recognizer can then be directly included in the PDS.

PROPOSITION 4.4. *Given a synchronous* QDAS $\mathcal{A}$ *with set of states* S *and a function* $f : S \to \mathbb{N}$*, one can build a* PDS $\mathcal{P}_{\mathcal{A},f}$ *whose size is exponential in the size of* $\mathcal{A}$ *and a state* s *of* $\mathcal{P}_{\mathcal{A},f}$*, such that* $\mathcal{P}_{\mathcal{A},f}$ *reaches* s *iff* f *is Parikh coverable in* $\mathcal{A}$*.*

PROOF. First, we construct the PDS with data $\mathcal{P}_{\mathcal{A}}$ and states S as in the proof of Proposition 4.2. Then, we translate the PDS with data to a PDS without data $\widehat{\mathcal{P}}_{\mathcal{A}} = \langle \widehat{Y}, \widehat{y^0}, \widehat{\Phi}, \widehat{\Sigma}, \widehat{\Delta} \rangle$ by encoding the variable valuations into the states of $\widehat{\mathcal{P}}_{\mathcal{A}}$. More precisely, $\widehat{Y} = S \times \mathbb{D}^{\mathcal{X}}$, and the transition relation is adapted accordingly. For all $y = (s, v) \in \widehat{Y}$, we let $S(y) = s$, i.e. $S(y)$ is the state of $\mathcal{P}_{\mathcal{A}}$ correponding to y. Observe that the size of $\widehat{\mathcal{P}}_{\mathcal{A}}$ is at most exponential in the size of $\mathcal{P}_{\mathcal{A}}$ and this construction does not change the pushdown system's behavior with respect to the stack but only internal actions. Second, from the function f, we construct the automaton $\mathcal{F}_f = \langle Q, S, q^0, \Delta_{\mathcal{F}}, q^f \rangle$ as in the proof of Lemma 4.3.

Finally, we define the PDS $\mathcal{P}_{\mathcal{A},f} = \langle Y, y^0, \Phi, \Sigma, \Delta_{\mathcal{A},f} \rangle$ as follows: (i) $Y = \widehat{Y} \uplus Q$ is the set of states; (ii) $y^0 = \widehat{y^0}$ is the initial state; (iii) $\Phi = \widehat{\Phi} = S$ is the stack alphabet; and (iv) $\Sigma = \widehat{\Sigma} \cup \{\tau\}$. Moreover, there is a transition (y, a, y') in $\Delta_{\mathcal{A},f} \subseteq Y \times \Sigma \times Y$ iff one of the following holds. Either (i) $(y, a, y') \in \widehat{\Delta}$ (all transitions of $\widehat{\mathcal{P}}_{\mathcal{A}}$ are transitions of $\mathcal{P}_{\mathcal{A},f}$); or (ii) there is $s \in \Phi$ s.t. $(q, s, q') \in \Delta_{\mathcal{F}}$ and $a = \mathtt{pop}(s)$ (the transitions of $\mathcal{P}_{\mathcal{A},f}$ include all transitions of $\mathcal{F}_f$ where the s label has been replaced by a pop(s) action); or (iii) $y \in \widehat{Y}$, $a = \mathtt{push}(S(y))$ and $y' = q^0$ (from all states in $\widehat{Y}$ there is a transition to the initial state of $\mathcal{F}_f$ that pushes the current state $S(y)$). Note that the size of $\mathcal{P}_{\mathcal{A},f}$ is exponential in the sizes of the instance of the Parikh coverability problem $(\mathcal{A}, f)$, because $\mathcal{P}_{\mathcal{A},f}$ amounts, roughly speaking to a serial composition of $\widehat{\mathcal{P}}_{\mathcal{A}}$, and (a modified version of) $\mathcal{F}_f$. The aim of this construction is that $\mathcal{P}_{\mathcal{A},f}$ first simulates of run of $\mathcal{A}$ in its '$\widehat{\mathcal{P}}_{\mathcal{A}}$' part, and then uses its '$\mathcal{F}_f$' part to check that the Parikh image of the corresponding QDAS configuration (stored on the stack) covers f.

Formally, let $\rho = x_0, a_1, x_1, \ldots, a_k, x_k, a_{k+1}, x_{k+1}, a_{k+2}, \ldots, a_n, x_n$ be a run of $\mathcal{P}_{\mathcal{A},f}$ that reaches q^f where $x_i = (y_i, w_i)$ for all i. Let us show that there exists a configuration $c = (G, \mathbb{d})$ of $\mathcal{A}$ such that $f \preceq \mathsf{Parikh}(G)$. Because of the definition of $\mathcal{P}_{\mathcal{A},f}$, ρ has a prefix that is a run of $\widehat{\mathcal{P}}_{\mathcal{A}}$ and thus visits only states of the form (y, w) with $y \in \widehat{Y}$. Let $x_0, a_1, \ldots, x_k$ be this prefix. Hence, $a_{k+1} = \mathtt{push}(S(y_k))$ and the suffix visits only states from Q: $\{y_{k+1}, \ldots, y_n\} \subseteq Q$ Since, $x_0, a_1, \ldots, x_k$ is a run of $\widehat{\mathcal{P}}_{\mathcal{A}}$, there is, by Proposition 4.2 a run of $\mathcal{A}$ that reaches $c = (G, \mathbb{d})$ with $G \triangleright y_k \cdot S(y_k)$. Then, the transition $(x_k, \mathtt{push}(S(y_k)), x_{k+1})$ transfers $w_{k+1} = y_k \cdot S(y_k)$ on the stack (recall that, intuitively, w_{k+1} encodes G). Now, by Lemma 4.3 the suffix $x_{k+1}, a_{k+2}, \ldots, a_n, x_n$ leading to the final state of $\mathcal{F}_f$ ensures that $|w_{k+1}|_s \geq f(s)$ for all $s \in S$. Hence, $f \preceq \mathsf{Parikh}(G)$ with $c = (G, \mathbb{d}) \in \textit{Reach}(\mathcal{A})$, and f is thus Parikh-coverable in $\mathcal{A}$.

The other direction of the proof follows from a similar argument. If f is Parikh-coverable in $\mathcal{A}$, there exists a run of $\mathcal{A}$ that reaches some configuration $c = (G, \mathbb{d})$ s.t. $f \preceq \mathsf{Parikh}(G)$. By proposition 4.2, this run can be simulated by the '$\widehat{\mathcal{P}}_{\mathcal{A}}$' part of $\mathcal{P}_{\mathcal{A},f}$ that will reach a configuration with stack content w s.t. $G \triangleright w$. Then, the '$\mathcal{F}_f$' part of $\mathcal{P}_{\mathcal{A},f}$ will check that $f \preceq \mathsf{Parikh}(G)$, and accept. □

From PDS to QDAS: Given a PDS $\mathcal{P}$, we construct a synchronous QDAS $\mathcal{A}_{\mathcal{P}}$ as shown in Figure 5. The underlying idea is symmetric to the reduction from QDAS to PDS: we simulate a $\mathtt{push}(\phi)$ action by a synchronous dispatch of a block ϕ, and a $\mathtt{pop}(\phi)$ action by the termination of a ϕ task. Hence, the content of the stack is encoded in the CTG at all times. The control state of the PDS is stored in the `state` global variable. Each block selects non-deterministically a possible transition of $\mathcal{P}$ (line 5) and simulates it by means of a synchronous dispatch or task termination (lines 8 and 11), as described above. The `state` is updated accordingly.

```
global state := $x^0$
global c_queue q

def $\phi$(): // for each $\phi \in \Phi \uplus \{main\}$
  while(true):
    select $(s,a,s') \in \Delta_{\mathcal{P}}$ where state=$s$
    if $a = \text{push}(\phi')$ :
      state := $s'$
      dispatch_s(q,$\phi'$)
    if $a = \text{pop}(\phi)$ and $\phi = \phi'$ :
      state := $s'$
      terminate
```

CTG in *Reach*$(\mathcal{A}_{\mathcal{P}})$:

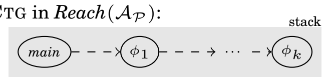


Fig. 5. From a pushdown system to a QDAS: *main* and ϕ for $\phi \in \Phi$

As before, synchronous dispatch calls ensure there is at most one active task at all times. Let $c = (G, \mathbb{d})$ be a configuration from *Reach*$(\mathcal{A}_{\mathcal{P}})$ and $y = (x, w) \in X \times \Phi^*$ be a reachable configuration of $\mathcal{P}$ with $w = w_1 \dots w_k$. Recall that G must be a path of vertices $v_0 v_1 \dots v_k$, because c is a reachable configuration in $\mathcal{A}_{\mathcal{P}}$. Then, we say that c is represented by y, written $c \triangleright y$, iff $\mathbb{d}(\texttt{state}) = x$; $\lambda(v_0) =$ *main* and $\lambda(v_i) = w_i$ for all $1 \leq i \leq k$. Observe that the empty stack is represented by a CTG contain a single *main* vertex labeled. The following proposition can be shown by induction on the runs of $\mathcal{A}_{\mathcal{P}}$:

PROPOSITION 4.5. *For all* PDS $\mathcal{P}$, *we can build a synchronous* QDAS $\mathcal{A}_{\mathcal{P}}$ *such that:*

(*1*) *for all run* $\pi = y_0 a_1 y_1 \dots a_n y_n$ *of* $\mathcal{P}$ *there exists a run* $\rho = c_0 a_1 c_1 \dots a_n c_n$ *of* $\mathcal{A}_{\mathcal{P}}$ *such that* $c_i \triangleright x_i$ *for all* $0 \leq i \leq n$*; and*
(*2*) *for all run* $\rho = c_0 a_1 c_1 \dots a_n c_n$ *of* $\mathcal{A}_{\mathcal{P}}$ *there exists a run* $\pi = y_0 a_1 y_1 \dots a_n y_n$ *of* $\mathcal{P}$ *such that* $c_i \triangleright x_i$ *for all* $0 \leq i \leq n$

Complexity of the Parikh Coverability Problem: Testing emptiness of a PDS without data is PTIME-C [Bouajjani et al. 1997]. Hence, by Proposition 4.4, the Parikh coverability problem is in EXPTIME for *synchronous* QDAS (with both types of queues). Proposition 4.5 and Proposition 2.3 allow to obtain a matching lower bound. Observe that the reduction we have used to prove Proposition 4.5 above requires only one *concurrent* queue, so the Parikh reachability problem is EXPTIME-hard already for *synchronous concurrent* QDAS:

THEOREM 4.6. *The Parikh coverability problem is* EXPTIME-C *for synchronous and for synchronous concurrent* QDAS*.*

Let us take a closer look at the dispatches that happen in runs of synchronous QDAS that have only *serial* queues. Here, each task except the `main` task blocks the queue it is started from. Hence, any other block dispatched to these queues will deadlock, and all reachable CTG have at most $|SQID| + 2$ vertices. Applying the reduction to PDS described above, we obtain a PDS with bounded stack height, for which reachability can be tested in PSPACE. The lower bound can be derived from Proposition 4.1.

THEOREM 4.7. *The Parikh coverability problem is* PSPACE-C *for serial synchronous* QDAS*.*

4.3. Concurrent asynchronous QDAS

Let us now establish a relationship between *concurrent asynchronous* QDAS and Petri nets to prove that the Parikh coverability problem is EXPSPACE-complete.

From QDAS to Petri Nets:
Given a concurrent asynchronous QDAS $\mathcal{A}$, we build a Petri net $N_{\mathcal{A}}$ that simulates it. The places of $N_{\mathcal{A}}$ are $(\mathcal{X} \times \mathbb{D}) \cup S$. Each place $s \in S$ counts how many blocks are currently running and are in state s. Each place (x, d) encodes the fact that the variable

x contains value d in the current valuation. The transitions of $N_{\mathcal{A}}$ are defined to simulate $\mathcal{A}$, according to this encoding. Formally, assuming $\mathcal{A}$ is a concurrent asynchronous QDAS with set of variables $\mathcal{X}$, set of locations S, set of transitions Δ and data domain $\mathbb{D}$, we let $N_{\mathcal{A}}$ be the PN $\langle P, T, m_0 \rangle$ where $P = (\mathcal{X} \times \mathbb{D}) \cup S$, $m_0(s^0_\gamma) = 1$, for all $x \in \mathcal{X}$: $m_0(x, d_0) = 1$, $m_0(p) = 0$ for all other places p and $t = (I_t, O_t) \in T$ iff one of the following holds:

Async. dispatch: there is $(s, (\mathtt{dispatch}_{\mathtt{a}}, q, \gamma), s') \in \Delta$ s.t. $I_t(p) = 1$ iff $p = s$ and $O_t(p) = 1$ iff $p \in \{s', s^0_\gamma\}$.
Test: there is $(s, x = d, s') \in \Delta$ with $x = d \in \mathsf{guards}(\mathcal{X})$ s.t. $I_t(p) = 1$ iff $p \in \{(x, d), s\}$ and $O_t(p) = 1$ iff $p \in \{(x, d), s'\}$.
Assignment: there are $(s, a, s') \in \Delta$ with $a = x \leftarrow v \in \mathsf{assign}(\mathcal{X})$ and $v' \in \mathbb{D}$ s.t. $I_t(p) = 1$ iff $p \in \{(x, v'), s\}$ and $O_t(p) = 1$ iff $p \in \{(x, v), s'\}$.
Task termination: there is $\gamma \in \Gamma$ s.t. $I_t(p) = 1$ iff $p = f_\gamma$ and for all $p \in P$: $O_t(p) = 0$.

Note that we have no place encoding the contents of the queue, as the dispatch of block γ directly creates a new token in s^0_γ. This encoding is, however, sound with respect to the *Parikh coverability problem*, as $\mathsf{Parikh}(G)$ does not distinguish between a block γ that is waiting in a queue, and a task executing γ in its initial state. The soundness of this construction is given by the following proposition:

PROPOSITION 4.8. *For all concurrent asynchronous* QDAS $\mathcal{A}$ *with set of locations* S, *we can build, in polynomial time, a Petri net* $N_{\mathcal{A}}$ *s.t.* $f : S \to \mathbb{N}$ *is Parikh-coverable in* $\mathcal{A}$ *iff* $m \in \mathit{Cover}(N_{\mathcal{A}})$, *where* m *is the marking s.t. for all* $s \in S$: $m(s) = f(s)$ *and for all* $p \in P \setminus S$: $m(p) = 0$.

The proof of the proposition relies on the following lemma, showing that $N_{\mathcal{A}}$ can simulate precisely the sequence of Parikh images that are reachable in $\mathcal{A}$. Let $(G, \mathbb{d})$ be a configuration of $\mathcal{A}$, and let m be marking of $N_{\mathcal{A}}$. We say that m *encodes* $(G, \mathbb{d})$, written $m \rhd (G, \mathbb{d})$ iff: (i) for all $x \in \mathcal{X}$: $m(x, \mathbb{d}(x)) = 1$; (ii) for all $x \in \mathcal{X}$: for all $d \in \mathbb{D} \setminus \{\mathbb{d}(x)\}$: $m(x, d) = 0$; and (iii) for all $s \in S$ $m(s) = \mathsf{Parikh}(G)(s)$. Then:

LEMMA 4.9. *Let* $\mathcal{A}$ *be a concurrent asynchronous* QDAS *with set of variables* $\mathcal{X}$ *and set of locations* S, *and let* $N_{\mathcal{A}}$ *be its associated* PN. *Then, for all* $(G, \mathbb{d}) \in \mathit{Reach}(\mathcal{A})$ *there is* $m \in \mathit{Reach}(N_{\mathcal{A}})$ *s.t.* $m \rhd (G, \mathbb{d})$ *and for all* $m \in \mathit{Reach}(N_{\mathcal{A}})$, *there is* $(G, \mathbb{d}) \in \mathit{Reach}(\mathcal{A})$ *s.t.* $m \rhd (G, \mathbb{d})$.

PROOF. Let $(G, \mathbb{d})$ be a configuration in $\mathit{Reach}(\mathcal{A}_N)$, and let $(G_0, \mathbb{d}_0)a_0(G_1, \mathbb{d}_1)a_1 \cdots \cdots a_{n-1}(G_n, \mathbb{d}_n)$ be a run s.t. $(G, \mathbb{d}) = (G_n, \mathbb{d}_n)$. We can thus inductively build a run $m_0 m_1 \cdots m_k$ of $N_{\mathcal{A}}$ s.t. $m_k \rhd (G, \mathbb{d})$, whose correctness follows by induction on the length n of the QDAS run.

Now, let $m_0 m_1 \cdots m_n$ be a run of $N_{\mathcal{A}}$ and let us build, inductively, a run $(G_0, \mathbb{d}_0)a_0(G_1, \mathbb{d}_1)a_1 \cdots a_{k-1}(G_k, \mathbb{d}_k)$ s.t. $m_n \rhd (G_k, \mathbb{d}_k)$ *and* where all the queues are empty in G_n, whose correctness follows by induction on the length n of the PN run. □

We are now ready to prove Proposition 4.8:

PROOF OF PROPOSITION 4.8. It is easy to check that the construction of $N_{\mathcal{A}}$, as described above, is polynomial. Then, assume f is Parikh coverable in $\mathcal{A}$, i.e. there is $(G, \mathbb{d}) \in \mathit{Reach}(\mathcal{A})$ s.t. $f \preceq \mathsf{Parikh}(G)$. By Lemma 4.9, there is $m' \in \mathit{Reach}(N_{\mathcal{A}})$ s.t. $m' \rhd (G, \mathbb{d})$, which means that $m'(s) = \mathsf{Parikh}(G)(s)$ for all $s \in S$. So, for all $s \in S$: $m(s) = f(s) \leq \mathsf{Parikh}(G)(s) = m'(s)$ and thus, $m \preceq m'$, since $m(p) = 0$ for all $p \notin S$. Finally, $m' \in \mathit{Reach}(N_{\mathcal{A}})$ implies that $m \in \mathit{Cover}(N_{\mathcal{A}})$.

On the other hand, assume $m \in \mathit{Cover}(N_{\mathcal{A}})$, with $m(p) = 0$ for all $p \notin S$, and let f be s.t. for all $s \in S$: $f(s) = m(s)$. Let $m' \in \mathit{Reach}(N_{\mathcal{A}})$ be a marking s.t. $m \preceq m'$. Such

```
1  def main():
2    for each $p \in P$:
3      $v_p$ := 0
4      select $k_p \in \{0, \ldots, m_0(p)\}$
5      for i = 1...$k_p$:
6        dispatch_a($C$, p())
7    dispatch_a($C$, trans())
8    while(true): do nothing

9  block p(): // For all $p \in P$
10   while($v_p = 0$): do nothing
11   $v_p$ := 0
```

```
12 block trans():
13   while(true):
14     select $t = (I_t, O_t) \in T$
15     for each $p \in P$ s.t. $I_t(p) = 1$:
16       $v_p$ := 1
17     while($\exists p \in P$: $v_p = 1$): do nothing
18     for each $p \in P$ s.t. $O_t(p) = 1$:
19       dispatch_a($C$, p())
```

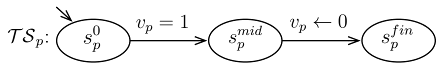


Fig. 6. Encoding of Petri net coverability $\langle P, T, m_0\rangle$ by a QDAS / LTS $\mathcal{TS}_p$ of block p

a marking exists since $m \in \mathit{Cover}(N_{\mathcal{A}})$. By Lemma 4.9, there is $(G, \mathbb{d}) \in \mathit{Reach}(\mathcal{A})$ s.t. $m' \rhd (G, \mathbb{d})$. Thus, by definition of $\rhd$, $m'(s) = \mathsf{Parikh}(G)(s)$ for all $s \in S$. Since $m \preceq m'$ and by definition of f, we conclude that $f(s) = m(s) \leq m'(s) = \mathsf{Parikh}(G)(s)$ for all $s \in S$, and f is Parikh-coverable in $\mathcal{A}$. $\square$

From Petri Net Coverability to QDAS: Let us now reduce the Petri net coverability problem to the Parikh coverability problem for QDAS (this construction is an extension of a construction found in [Ganty and Majumdar 2012]). Let $N = \langle P, T, m_0\rangle$ be a Petri net and let us consider the concurrent asynchronous QDAS $\mathcal{A}_N = \langle CQID, \emptyset, \Gamma, \mathtt{main}, \mathcal{X}, \Sigma, (\mathcal{TS}_\gamma)_{\gamma\in\Gamma}\rangle$, on the finite domain $\mathbb{D} = \{0, 1\}$, where $CQID = \{C\}$ and $\Gamma = \{main, trans\} \cup P$, $\mathcal{X} = \{v_p \mid p \in P\}$. The transition systems of the blocks $(\mathcal{TS}_\gamma)_{\gamma\in\Gamma}$ are given in Fig. 6. For convenience, we express $\mathcal{TS}_{\mathtt{main}}$ and $\mathcal{TS}_{\mathtt{trans}}$ using pseudo-code while each $\mathcal{TS}_p$ is given by the automaton with states $s^0_{\mathtt{p}}$, $s^{mid}_{\mathtt{p}}$ and $s^{fin}_{\mathtt{p}}$ depicted in the figure. We assume that, for $\gamma \in \{\mathtt{trans}, \mathtt{main}\}$, s^ℓ_γ is the location of $\mathcal{TS}_\gamma$ that is reached when the control reaches line ℓ. Let $G = \langle V, E, \lambda, \mathit{queue}, \mathit{state}\rangle$ be a CTG of $\mathcal{A}_N$, and let m be a marking of N. Then, we say that G *encodes* m, written $G \rhd m$ iff (i) $\mathsf{Parikh}(G)(s^{13}_{\mathtt{trans}}) = \mathsf{Parikh}(G)(s^8_{\mathtt{main}}) = 1$; (ii) for all $p \in P$: $\mathsf{Parikh}(G)(s^0_{\mathtt{p}}) = m(p)$ (remember that $s^0_{\mathtt{p}}$ is the initial location of $\mathcal{TS}_p$); and (iii) for all $p \in P$, for all $s \in S_p \setminus \{s^0_{\mathtt{p}}\}$: $\mathsf{Parikh}(G)(s) = 0$. Thus, intuitively, a CTG G encodes a marking m iff main is at line 8, trans is at line 13, $m(p)$ counts the number of p blocks that are either in C or executing but at their initial state, and there are no p blocks that are in states $s^{mid}_{\mathtt{p}}$ or $s^{fin}_{\mathtt{p}}$.

The intuition behind the construction is as follows. Each run of $\mathcal{A}_N$ starts by an initialization phase, where main sets all the v_p variables to 0 and dispatches, for all $p \in P$, k_p blocks p with $k_p \leq m_0(p)$, then dispatches a call to trans. At that point, the only possible action is that the scheduler dequeues all the blocks. All the p tasks are then blocked, as they need that $v_p = 1$ to proceed and terminate. Then, trans non-deterministically picks a transition t, sets to 1 all the variables v_p s.t. t consumes a token in p, and waits that all the v_p variables return to 0. This can only happen because *at least* $I_t(p)$ p tasks have terminated, for all $p \in P$. So, when trans reaches line 19, the encoded marking has been decreased by *at least* I_t. Note that more than $I_t(p)$ p tasks could terminate, as they run concurrently, and the lines 10 and 11 do not execute atomically. Then, for all $p \in P$, trans dispatches one new p block iff t produces a token in p. This increases the encoded marking by O_t, so the effect of one iteration of the main while loop of trans is to simulate the effect of t, plus a possible token loss. Hence, the resulting marking is guaranteed to be in $\mathit{Cover}(N)$ (but maybe not in $\mathit{Reach}(N)$). This is formalized by the following proposition:

PROPOSITION 4.10. *For all Petri nets N, we can build, in polynomial time, a concurrent asynchronous* QDAS $\mathcal{A}_N$ *s.t.:* $m \in \mathit{Cover}(N)$ *iff there are a marking* m' *and a configuration* $(G, \mathbb{d}) \in \mathit{Reach}(\mathcal{A}_N)$ *with* $G \rhd m'$ *and* $m \preceq m'$.

The proof of Proposition 4.10 is split into two lemmata, given hereunder. They rely on a different characterization of $\mathit{Cover}(N)$. That is, $m \in \mathit{Cover}(N)$ iff m is reachable by a so-called *lossy* run of N, i.e. a sequence of markings $m'_0 m'_1 \cdots m'_n$ s.t. $m'_0 \preceq m_0$ and for all $0 \leq i \leq n-1$: there is $\overline{m}_{i+1}$ and a transition t_i s.t. $m'_i \xrightarrow{t_i} \overline{m}_{i+1}$ and $m'_{i+1} \preceq \overline{m}_{i+1}$. Intuitively, each step in a lossy run corresponds to firing a transition of the PN, and then spontaneously losing some tokens. The proof of these lemmata also assumes that for each $p \in P$, the corresponding LTS $\mathcal{TS}_p = \langle \{s^0_P, s^{mid}_p, s^{fin}_p\}, s^0_p, \Sigma, \Rightarrow \rangle$ is as in Fig. 6.

LEMMA 4.11. *Let* $N = \langle P, T, m_0 \rangle$ *be a* PN, *and let* $\mathcal{A}_N = \langle \mathit{CQID}, \emptyset, \Gamma, \mathtt{main}, \mathcal{X}, \Sigma, (\mathcal{TS}_\gamma)_{\gamma \in \Gamma} \rangle$ *be its corresponding* QDAS. *If* $m \in \mathit{Cover}(N)$ *then there are a marking* m' *and* $(G, \mathbb{d}) \in \mathit{Reach}(\mathcal{A}_N)$ *s.t.* $G \rhd m'$ *and* $m \preceq m'$.

PROOF. Let m be a marking from $\mathit{Cover}(N)$ and let $m_0 m_1 \cdots m_n$ be a (non-lossy) PN run s.t. $m \preceq m_n$. Such a run exists by definition of $\mathit{Cover}(N)$. It is easy to build, by induction on the length of this run, a run in the QDAS that simulates it precisely. That is, that for all $0 \leq i \leq n$, there is $(G_i, \mathbb{d}_i) \in \mathit{Reach}(\mathcal{A}_N)$ (with $G_i = \langle V^i, E^i, \lambda^i, \mathit{queue}^i, \mathit{state}^i \rangle$) s.t. $E^i = \emptyset$; $\mathbb{d}_i(v_p) = 0$ for all $p \in P$; and $G_i \rhd m_i$. $\square$

LEMMA 4.12. *Let* $N = \langle P, T, m_0 \rangle$ *be a* PN, *and let* $\mathcal{A}_N = \langle \mathit{CQID}, \emptyset, \Gamma, \mathtt{main}, \mathcal{X}, \Sigma, (\mathcal{TS}_\gamma)_{\gamma \in \Gamma} \rangle$ *be its corresponding* QDAS. *For all* $(G, \mathbb{d}) \in \mathit{Reach}(\mathcal{A}_N)$ *and all markings* m *it holds that* $G \rhd m$ *implies* $m \in \mathit{Cover}(N)$.

PROOF. For a CTG G of $\mathcal{A}_N$ with set of vertices V, we denote by $M(G)$ the marking of N s.t. for all $p \in P$: $M(G)(p) = |\{v \in V \mid \mathit{state}(v) = s^0_{\mathtt{p}}\}|$. Thus, in the case where G encodes a configuration s.t. `trans` is at line 13, `main` is at line 8, and all the `p` blocks are in their initial state $s^0_{\mathtt{p}}$, then $G \rhd M(G)$.
To establish the lemma, one can prove the following stronger statement: for all reachable configurations $(G, \mathbb{d})$ s.t. `trans` is at line 13: $M(G) \in \mathit{Cover}(N)$. This is easily established by induction on the length of the run. $\square$

Remark 4.13. Observe that in the above reduction, we achieved to simulate *lossy Petri net runs*. This stems from the fact that lines 10 and 11 – or the two transition in $\mathcal{TS}_p$, respectively – (see Fig. 6) do not execute atomically, and thus, more than one p block can terminate concurrently when a transition consumes a single token in p, which accounts for the loss. This situation must be contrasted with the sequential asynchronous systems of [Ganty and Majumdar 2012] that can simulate (non-lossy) *Petri net runs*.

Decidability of Parikh Coverability: Combining Proposition 4.8 and Proposition 4.10 together with the known EXPSPACE-C result on Petri net coverability [Rackoff 1978] leads to our main result for concurrent asynchronous QDAS.

THEOREM 4.14. *The Parikh coverability problem is* EXPSPACE-C *for concurrent asynchronous* QDAS.

5. UNDECIDABILITY OF THE PARIKH COVERABILITY PROBLEM

5.1. Asynchronous Serial QDAS

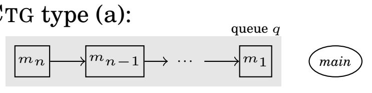
Let us show that for the class of QDAS with one serial queue, and where asynchronous dispatches are allowed, the Parikh coverability problem is *undecidable*. We establish

```
1   global state, head
2   global s_queue q

3   def main():
4     state := $s_F^0$
5     head := $\varepsilon$
6     dispatch_a(q, $\varepsilon$)
7     while(true): do nothing

8   def m(): //for all $m \in M \cup \{\varepsilon\}$
9     if (head$\neq m$): goto 19
10    while(true):
11      if (state = $c$): goto 20
12      select $(s, a, s') \in \Delta_F$
13      if ($s \neq$state): goto 19
14      state := $s'$
15      if ($a = !n$): dispatch_a(q, $n$)
16      else if ($a = ?n$):
17        head := $n$
18        terminate
19    while(true): do nothing // wrong guess
20    while(true): do nothing // $c$ is reached
```

CTG type (a):



CTG type (b):

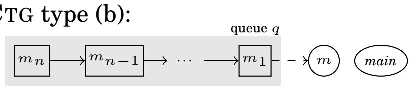


Fig. 7. Encoding of a FIFO system in a serial asynchronous QDAS, and the two possible types of CTG.

this by a reduction from the control state reachability problem of FIFO systems which is known to be undecidable [Brand and Zafiropulo 1983].

Let $F = \langle S_F, s_F^0, M, \Delta_F\rangle$ be a FIFO system and let $c \in S_F$ be a state whose reachability has to be tested. We build the asynchronous serial QDAS $\mathcal{A}_F = \langle \emptyset, \{q\}, \Gamma, \mathtt{main}, \mathcal{X}, \Sigma, (\mathcal{TS}_\gamma)_{\gamma\in\Gamma}\rangle$ on domain $\mathbb{D} = M \cup S_F \cup \{\varepsilon\}$, where $\Gamma = M \cup \{\varepsilon, \mathtt{main}\}$, $\mathcal{X} = \{\mathtt{state}, \mathtt{head}\}$ and the $\mathcal{TS}_\gamma$ are given by the pseudo code in Fig. 7.

Intuitively, runs of $\mathcal{A}_F$ simulate the runs of F, by encoding the current state of F in the state variable and the content of the queue in the content of the serial queue q. It easy to check that, once main has reached line 7, all the CTG that are reached in $\mathcal{A}_F$ are of either shapes given in Fig. 7 (with $\{m_1, \ldots, m_n, m\} \subseteq M \cup \{\varepsilon\}$). That is, there are at most two running tasks: main and possibly one task running an m block (with $m \in M \cup \{\varepsilon\}$), that has to terminate to allow a further dequeue from q (q is a serial queue and all the dispatches are asynchronous). When the CTG is of shape (b), the duty of the running m block is to simulate a transition of F. To do so, m executes an infinite while loop (line 10 onwards – ignore the test at line 11 for the moment), that (i) tests whether c has been reached (line 11) and jumps to line 20 if it is the case; (ii) guesses a transition (s, a, s') of F; and (iii) checks that the guessed transition is fireable from the current configuration (line 13). If not, the block jumps to the infinite loop of line 19, which blocks the simulation. Otherwise, the transition is simulated.

Note that the simulation of a receive of m from q is a bit involved, as only the scheduler can decide to dequeue a block from q, and this can happen only if the current running block terminates (line 18). Still, we have to check that message m is indeed in the head of q. This is achieved by setting global variable head to m, and letting the next dequeued block check that it is of type head (line 9). If this test is not satisfied, the block jumps to the infinite loop of line 19, and the simulation is blocked.

Note that, since the queue is initially empty, this process works only when at least one task executes, i.e., at least one message has been sent to the channel. To bootstrap this process, we rely on the ε block, which is dispatched by main at the beginning of the execution. Since head is initialized to ε, the test in line 9 will evaluate to true in the task running ε, and this task will be able to simulate the first transitions of F.

Thus, in all reachable configurations of $\mathcal{A}_F$, a block m (with $m \in M \cup \{\varepsilon\}$) will reach line 20 iff c is reachable in F. This effectively reduces the control location reachability of FIFO systems to the Parikh coverability problem of serial asynchronous QDAS. Let us formalise these ideas.

For all $\gamma \in \Gamma$, we denote by s_γ^ℓ the location of $\mathcal{TS}_\gamma$ that corresponds to line ℓ in Fig. 7. Then, we say that a configuration $(G, \mathbb{d})$ of $\mathcal{A}_F$ encodes a configuration (s, w) of

F, written $(G, \mathbb{d}) \rhd (s, w)$ iff: (i) $s = \mathbb{d}(\mathtt{state})$; (ii) G is of either shapes in Fig. 7 with $w = m_1 m_2 \cdots m_n$; (iii) $\mathsf{Parikh}(G)(s^7_{\mathtt{main}}) = 1$; and (iv) there exists $\mathtt{m} \in M \cup \{\varepsilon\}$ such that $\mathsf{Parikh}(G)(s^{11}_{\mathtt{m}}) = 1$. That is, s and w are encoded as described above, `main` is at line 7, and the running `m` block is at line 11. Then by induction on the structure of the runs of the FIFO system and the corresponding QDAS we can show the following:

LEMMA 5.1. *Let F be a* FIFO *system, let c be a configuration of F, and let $\mathcal{A}_F$ be its associated* QDAS. *For all runs $(s_0, w_0)(s_1, w_1) \cdots (s_n, w_n)$ of F s.t. for all $0 \leq i < n$: $s_i \neq c$, there exists $(G, \mathbb{d}) \in$ Reach$(\mathcal{A}_F)$ s.t. $(G, \mathbb{d}) \rhd (s_n, w_n)$. Moreover, for all $(G, \mathbb{d}) \in$ Reach$(\mathcal{A}_F)$ and for all configurations (s, w) of F: $(G, \mathbb{d}) \rhd (s, w)$ implies $(s, w) \in$ Reach(F).*

We can thus reduce the control location reachability problem of FIFO systems to the Parikh coverability problem of serial asynchronous QDAS (using only one serial queue) as follows. For all $m \in M \cup \{\varepsilon\}$, we let f_m be the Parikh image s.t. $f_m(s^{20}_{\mathtt{m}}) = 1$ and $f_m(s) = 0$ for all $s \neq s^{20}_{\mathtt{m}}$. Note that there are only finitely many such f_m. Then, we show that c is reachable in F iff there exists $m \in M \cup \{\varepsilon\}$ s.t. f_m is Parikh-coverable in $\mathcal{A}_F$. By Lemma 5.1, we can deduce the Parikh coverability of a QDAS configuration where $\mathbb{d}(\mathtt{state}) = c$ from the reachability of a control state in F. For the reverse direction assume there is $m \in M \cup \{\varepsilon\}$ that is Parikh-coverable in $\mathcal{A}_F$. By Lemma 5.1, this entails the existence of $(c, w) \in$ *Reach*(F), and thus c is reachable in F. Since the control location/state reachability problem for FIFO systems is undecidable [Brand and Zafiropulo 1983], this reduction shows that:

THEOREM 5.2. *The Parikh coverability problem is undecidable for asynchronous* QDAS *with at least one serial queue.*

5.2. Concurrent QDAS

Let us show that, once we allow both synchronous and asynchronous dispatches in a *concurrent* QDAS, the Parikh coverability problem becomes undecidable by reduction from the reachability problem for two counters systems.

From a 2CM $\mathcal{P} = \langle S_{\mathcal{P}}, s^0_{\mathcal{P}}, \Delta_{\mathcal{P}} \rangle$, we build the QDAS $\mathcal{A}_{\mathcal{P}} = \langle CQID, \emptyset, \Gamma, \mathtt{main}, \mathcal{X}, \Sigma, (\mathcal{TS}_\gamma)_{\gamma \in \Gamma} \rangle$ with only one concurrent queue: $CQID = \{q\}$, with set of blocks $\Gamma = (\{null, one\} \times \{1, 2\}) \cup \{main\}$, and set of variables $\mathcal{X} = \{\mathtt{state}, \ell^1_1, \ell^1_2, x^1, \ell^2_1, \ell^2_2, x^2\}$ where `state` takes values in $S_{\mathcal{P}}$, all ℓ^j_i are Boolean variables and x^1, x^2 range over the domain $\{\mathtt{incr}, \mathtt{decr}, \mathtt{is\_zero}, \mathtt{ack}\}$ (the purpose of these variables will be made clear later). The respective transition systems of the blocks are given in Fig. 8. Let us explain how this construction works.

First, let us explain how the counters are encoded. Our construction ensures that all reachable CTG are of the form depicted in Fig. 8, i.e., for each counter i there is always one unblocked block, which is either $one(i)$ (counter i is not null) or $null(i)$ (counter i is null). More precisely, in such a CTG, the value of counter i is given by the number of $one(i)$ blocks, so counters are encoded in a fashion similar to the encoding of stacks we have used in section 4.2. The two special blocks $null(i)$ are always present and are unblocked iff counter i is null. Finally, the control location of $\mathcal{P}$ is recorded in global variable `state`.

It is the duty of the *main* task to simulate the actions of the 2CM. To achieve this, the *main* task runs an infinite `while` loop (line 12 onwards) that consists of guessing a transition (s, a, s') of F and synchronising, via *rendezvous* with the relevant $null$ or one unblocked task, to let it execute the operation on the counter. Recall that rendezvous synchronisations are not a feature of QDAS, yet we argue hereunder that they can be implemented in a concurrent QDAS. So, let us assume for now that the tasks can execute instructions of the form $i!m$ or $i?m$ meaning respectively that the tasks send or

```
global state
global $\ell_1^1$, $\ell_2^1$, $x^1$ // rdvz channel 1
global $\ell_1^2$, $\ell_2^2$, $x^2$ // rdvz channel 2
global c_queue q

def main():
  foreach i in {1,2}:
    dispatch_a(q, $null$(i))
    i?ack // rendezvous synchronisation
  state := $x^0$
  while(true):
    select $(s,a,s') \in \Delta_{\mathcal{P}}$ where state=$s$
    if $a = \mathtt{incr}(1)$ :
      1!incr
      1?ack
      state:=$s'$
    \\ other actions analogous
    ...
```

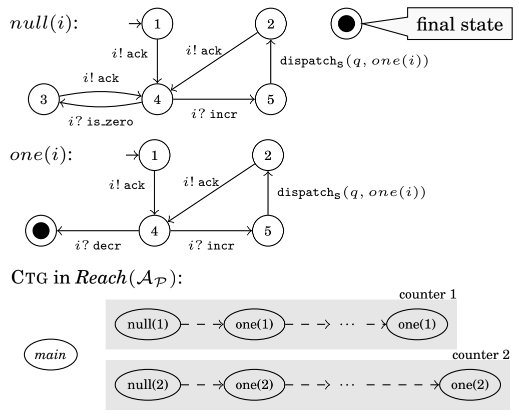


Fig. 8. From a two counter system to a QDAS with rendezvous: *main* and $null(i), one(i)$ for $i = 1, 2$

read message m to (from) a communication channel i (both operations being *blocking* in order to achieve true rendezvous). In the reduction of Fig. 8, two channels (1 and 2) are considered, and *main* can send messages of the form $i!\,$incr, $i!\,$decr, or $i!\,$is_zero to signal that an increment (respectively decrement, test-for-zero) must be performed on counter i. These messages will be received by the unblocked $one(i)$ or $null(i)$ blocks. In addition an $i?\,$ack message can be received by *main* to acknowledge the execution of the counter operation.

It is thus the duty of the $one(i)$ and $null(i)$ blocks to perform the actual counter updates. When a $null(i)$ or $one(i)$ receives an incr message, it performs an asynchronous dispatch of $one(i)$ into q to increment counter i, and acknowledges the operation to *main*, thanks to message ack. When an one block receives a decr message, it terminates, which decrements the counter. $null$ blocks cannot receive decr messages, so, if *main* requests a decr operation when the counter is zero, *main* gets blocked. This means that the guessed transition was not fireable in the currently simulated two-counter system configuration, and ends the simulation. Finally, only $null$ blocks can receive and acknowledge is_zero messages, so, again, *main* is blocked after sending is_zero to a non-zero counter.

Note that the reduction needs both *asynchronous* calls to start two counters in parallel, and *synchronous* calls to encode the counter values.

To finish the reduction, we need to explain how the $i!m$ and $i?m$ operations can be implemented in QDAS. We achieve this by proper use of global variables: for each channel i we need two global Boolean variables ℓ_1^i and ℓ_2^i and an additional variable x^i taking value in M, where M is the finite set of messages that can be exchanged (in our case $M = \{\mathtt{incr}, \mathtt{decr}, \mathtt{is\_zero}, \mathtt{ack}\}$. Let γ_1 and γ_2 be two blocks whose LTS are:

$$\gamma_1: \begin{tikzcd} \arrow[r] & s_0 \arrow[r, "\ell_1 = 1"] & \circ \arrow[r, "x \leftarrow m"] & \circ \arrow[r, "\ell_2 \leftarrow 1"] & \circ \arrow[r, "\ell_1 = 0"] & \circ \arrow[r, "\ell_2 \leftarrow 0"] & s_5 \end{tikzcd} \qquad \text{(for } m \in M\text{)}$$

$$\gamma_2: \begin{tikzcd} \arrow[r] & s'_0 \arrow[r, "\ell_1 \leftarrow 1"] & \circ \arrow[r, "\ell_2 = 1"] & \circ \arrow[r, "x = m"] & \circ \arrow[r, "\ell_1 \leftarrow 0"] & \circ \arrow[r, "\ell_2 = 0"] & s'_5 \end{tikzcd}$$

Assume a configuration c where $\ell_1 = \ell_2 = 0$ and where two distinct tasks are running γ_1 and γ_2, are unblocked, and are in s_0 and s'_0 respectively. Assume that no other task can access ℓ_1, ℓ_2 and x. It is easy to check that, from c, there is only one possible interleaving of the transitions of γ_1 and γ_2. So if γ_2 reaches s'_5 from c, then γ_1 must have reached s_5, and the $x = m$ test in γ_1 has been fired *after* the $x \leftarrow m$ assignment in γ_2. This achieves a rendezvous synchronisation between γ_1 and γ_2, with the passing of message m. In our encoding (see Fig. 8), we need two such channels, so we have six global variables (see lines 2 and 3).

By induction on the runs of $\mathcal{P}$ and $\mathcal{A}_\mathcal{P}$, it is easy to show that $\mathcal{P}$ reaches a state $s \in S_\mathcal{P}$ iff f is coverable in $\mathcal{A}_\mathcal{P}$ where $f(s) = 1$ and $f(s') = 0$ for all $s' \neq s$. Hence:

THEOREM 5.3. *The Parikh coverability problem is undecidable for concurrent* QDAS *that use both synchronous and asynchronous dispatches.*

6. THE TERMINATION PROBLEM

The proofs of undecidability for the Parikh coverability problem extend straightforwardly to termination. The connection between QDAS and PDS (with data) yields an EXPTIME algorithm for the termination problem, using the emptiness testing of a PDS with Büchi condition [Esparza et al. 2000] (for this problem the best lower bound is PSPACE-hard, as far as we know). Finally, the complexity of termination for asynchronous concurrent QDAS follows from the complexity of termination for Petri nets [Lipton 1976; Rackoff 1978]. Hence:

THEOREM 6.1. *The termination problem is* PSPACE-C *for synchronous serial* QDAS*, it is in* EXPTIME *and* PSPACE*-hard for synchronous* QDAS*, and it is* EXPSPACE-C *for asynchronous concurrent* QDAS*. It is undecidable for asynchronous serial* QDAS*, and* QDAS *that use both synchronous and asynchronous dispatches.*

7. EXTENDING QDAS WITH FORK/JOIN

Although the QDAS model we have discussed so far allows us to capture many useful features of GCD, it does not allow us to model all of its constructs. A notable example is the fork-join primitive which is present in GCD and in many other concurrent programming frameworks.

As applying fork/join is usually sensible in the context of asynchronous dispatches to concurrent queues, we restrict ourselves in this section to extending asynchronous and concurrent QDAS. This also allows us to avoid the undecidability results proved in the previous sections.

Let us now extend the basic syntax of QDAS with a `forkjoin`(q, γ, p) action, where q is a queue, γ is a block name, and p is (for now) a natural number. The semantics of this action is to dispatch simultaneously p copies of the block γ to the queue q (the so-called *"fork"* of the fork-join). This dispatch is blocking for the caller, which must then wait for the termination of all the p blocks before carrying on its own execution (*"join"*).

Observe that we could apply the fork-join primitive to directly create all the `one_cell` tasks in the matrix multiplication example (Example 2.1) without the need for the semaphore `count`. However, the number of forked `one_cell` tasks depends on the size of the input matrices, thus it would be crucial to verify the algorithm *for any possible size of the input*. Thus, we permit the parameter p of `forkjoin` to take the value "$*$" where $*$ represents 'any non-negative value'. That is, the execution of `forkjoin`$(q, \gamma, *)$ dispatches a *non-deterministically* chosen number of blocks γ into q.

7.1. QDAS extended by fork/join

Let us first introduce our extension of asynchronous and concurrent QDAS to handle fork and joins. A QDAS *extended by fork/join (*EQDAS*)* is a tuple $\langle CQID, \emptyset, \Gamma, main, \mathcal{X}, \Sigma, (\mathcal{TS}_\gamma)_{\gamma\in\Gamma}\rangle$ that is defined as an *asynchronous* QDAS where the possible actions are extended by: $\{$`forkjoin`$\}\times CQID \times \Gamma \times (\mathbb{N} \cup \{*\})$. Note that $*$ stands for "any number". The *parameter* of a `forkjoin` action is the last value of the tuple. An EQDAS is $*$*-free* iff none of its transition systems $\mathcal{TS}_\gamma$ contains a `forkjoin` action with parameter $*$.

The semantics of EQDAS extends the semantics of QDAS: Dispatches, tests, assignments and scheduler actions are handled as in the case of (plain) QDAS. However, there

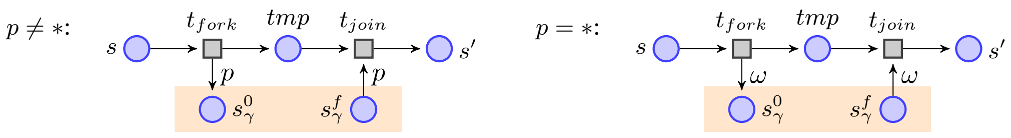


Fig. 9. The two ω-PN gadgets to simulate the EQDAS action $(s, \texttt{forkjoin}(q, \gamma, p), s')$.

is a transition $\big((G, \mathbb{d}), \texttt{forkjoin}(q, \gamma, p), (G', \mathbb{d}')\big)$ in the transition system of an EQDAS iff $\mathbb{d}' = \mathbb{d}$ and there are $\delta = (s, \texttt{forkjoin}(q, \gamma, p), s') \in \Delta$, and $G'' \in \mathsf{step}(\delta)(G)$ such that: $G' = G''_n$ where (i) $p = *$ implies $n \geq 0$; (ii) $p \neq *$ implies $n = p$; (iii) $G''_0 = G''$; and (iv) for all $0 \leq i < n$, $G''_{i+1} = \mathsf{letwait}(v, v'_{i+1})\big(\mathsf{enqueue}(q, \gamma_i)(G''_i)\big)$ where v is the node whose *state* has changed during the step operation, and v'_{i+1} is the fresh node that has been created by the enqueue operation. Intuitively, a $\texttt{forkjoin}(q, \gamma, p)$ action inserts a sequence of p blocks γ to the queue q and creates a wait edge from the task executing the forkjoin to each newly created node (recall that $p = *$ means 'a non-deterministically chosen number'). Hence, forkjoin is blocking for the caller. The actual *join* is thus implicit: the task that has executed the forkjoin will be blocked until all the tasks created during the *fork* terminate their execution.

As before, we are interested in the verification problems of Parikh coverability and termination for EQDAS, which are defined as before in the case of QDAS.

Since a $\texttt{forkjoin}(q, \gamma, p)$ action with parameter $p = 1$ is equivalent to a synchronous dispatch of γ to q, already $*$-free EQDAS can simulate concurrent QDAS (with both synchronous and asynchronous dispatch). Hence, by Theorem 5.3, we get:

THEOREM 7.1. *The* EQDAS *Parikh coverability and termination problem are undecidable, even for $*$-free* EQDAS.

To circumvent this undecidability result in practice, we propose, in the next subsection, a technique to over-approximate an EQDAS by means of an extended Petri net.

7.2. Over-approximations of EQDAS

To recover a practical procedure for analysing EQDAS, we propose to use a recently introduced extension of Petri nets [Geeraerts et al. 2013b], called ω-Petri net (ω-PN for short). ω-PN transitions with an arbitrary but fix number of tokens, and extend classical Petri nets by permitting arcs to be labeled by ω. Formally, in an ω-PN, each transition is a pair $t = (I_t, O_t)$, where I_t and O_t are both functions from P to $\mathbb{N} \cup \{\omega\}$. The semantics of $O_t(p) = \omega$ (resp. $I_t(p) = \omega$) is that the firing of t produces (consumes) a non-deterministically chosen, yet finite, number of tokens in (from) p. The coverability and the termination problems are both decidable for ω-PN [Geeraerts et al. 2013b].

We now encode the behavior of an EQDAS $\mathcal{A}$ into an ω-PN $N^*_{\mathcal{A}}$, that over-approximates its behavior, by an extension of the translation of concurrent asynchronous QDAS into PN from Section 4.3: we encode dispatches, tests, assignments and scheduler actions as before, and we handle a $\big(s, \texttt{forkjoin}(q, \gamma, p), s'\big)$ transition as depicted in Fig. 9 (by means of two extra transitions t_{fork} and t_{join}, and an extra place tmp). In this figure, the places s^0_γ and s^f_γ are, as before, the places encoding respectively the initial and the final states of γ.

Let us now explain why these constructions establish a behavioral *over-approximation*: Consider a transition $\delta = \big(s, \texttt{forkjoin}(q, \gamma, p), s'\big)$, with $p \neq *$, of $\mathcal{A}$, and consider a configuration $(G, \mathbb{d})$ of $\mathcal{A}$ such that $\mathsf{Parikh}(G)(s^f_\gamma) = p$ and $\mathsf{Parikh}(G)(s) = 1$, and $\mathsf{Parikh}(G)(s') = 0$, i.e., there is one task in location s, no task in s' and p tasks already in location s^f_γ. Then executing δ from $(G, \mathbb{d})$ yields a configuration $(G', \mathbb{d})$ $\mathsf{Parikh}(G')(s) = 1$ and $\mathsf{Parikh}(G')(s') = 0$, and the block that executes the forkjoin will

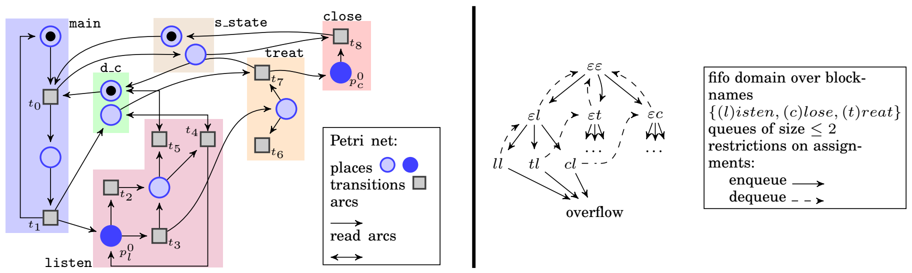


Fig. 10. Petri net translation of asynchronous server and FIFO data domain

remain blocked in location s until the p forked tasks, which each execute γ, terminate. This is not the case in the ω-PN $N^*_{\mathcal{A}}$: the marking m corresponding to $(G, \mathbb{d})$ satisfies $m(s) = 1, m(s^f_\gamma) = p$ and $m(s') = 0$. From m, one can fire the t_{fork} transition corresponding to δ, and immediately afterwards the t_{join} transition, because s^f_γ still contains p tokens. Thus, the join can occur even if the p tokens created by t_{fork} have not reached s^f_γ yet. Still, the ω-PN can simulate faithfully the behavior of the original EQDAS as far as Parikh coverability and termination are concerned. A similar phenomenon occurs for $p = *$. We conclude that the constructed ω-PN $N^*_{\mathcal{A}}$ *over-approximates* the original EQDAS $\mathcal{A}$:

PROPOSITION 7.2. *Let $\mathcal{A}$ be an* EQDAS *with set of locations $\mathcal{S}$, and let f be a Parikh image of $\mathcal{A}$. Then, we can build, in polynomial time, an ω-Petri net $N^*_{\mathcal{A}}$ s.t.* if *f is Parikh-coverable in $\mathcal{A}$,* then *$m \in Cover(N^*_{\mathcal{A}})$, where m is the marking s.t. for all $s \in \mathcal{S}$: $m(s) = f(s)$ and for all $p \in P \setminus \mathcal{S}$: $m(p) = 0$. Moreover, if $N^*_{\mathcal{A}}$ terminates, then $\mathcal{A}$ terminates too.*

Over-approximations are useful for program verification when proving *safety* or *termination* properties: if the ω-PN is *safe*, i.e., does not reach a bad state, then so is the EQDAS (and similarly for termination).

8. A PRAGMATIC APPROACH TO VERIFY GCD PROGRAMS IN PRACTICE

In this section, we show that simple yet realistic examples of QDAS, that are *not* in one of the decidable classes, can nevertheless be analysed automatically, in practice.

Let us return to our initial examples of matrix multiplication (Example 2.1) and asynchronous server (Example 2.2). The correctness of both relies on the usage of a *serial* queue, and of *asynchronous* and *synchronous* calls. By Theorem 5.2, these examples fall in an undecidable class, yet proving correctness of such programs is an important issue in practice. Let us show that an extension of the constructions from Section 7 allows us to verify meaningful properties on our examples, by means of carefully crafted Petri nets abstractions. (The code of these examples can be found at [QDAS].)

In the following, we consider the following properties: On Example 2.1, we want to verify whether there is never more than one block `one_cell` incrementing the `count` variable at the same time. On Example 2.2 we want to verify whether the following three properties hold for our asynchronous server: (1) at most one `close` task is running at a time, (2) the last task on a queue, before it is reassigned to a new connection, is `close`, (3) while the first running task of the queue is `treat`, the last task is `close`.

Over-approximation: First, observe that replacing all serial queues by concurrent ones, and all synchronous calls by asynchronous ones yields an *over-approximation* of the original QDAS. Moreover, the resulting QDAS falls in a class for which the coverability problem is decidable (Theorem 4.14). Applying this procedure to Example 2.2 produces the Petri net given at the left-hand side of Fig. 10 (in this model, we only

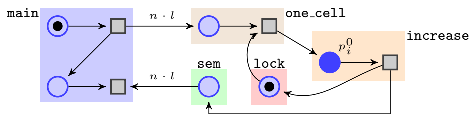


Fig. 11. Petri Net translation of matrix multiplication example

remember the state of the socket by the places in `s_state`). Using the [mist2] tool—an automatic safety checker for Petri net coverability—, we managed to prove that property (1) holds by checking that no marking m with $m(p_c^0) \geq 1$ is reachable in the Petri net of Fig. 10 (an instance of the *coverability problem*).

Observe that, by applying the construction described above, our over-approximation loses two important properties of the *serial queues*. First, the *mutual exclusion* (i.e., two blocks from the same serial queue cannot execute at the same time), does not hold because we replace serial queues by concurrent queues. Second, the FIFO *ordering* of the blocks is lost in the counting abstraction that produces the Petri net. Note that the *mutual exclusion* property can be recovered in the Petri net model, by associating, to each queue q, a place $lock_q$ that acts as a global lock, from which blocks dispatched to q have to consume the token before starting to execute. Applying this refinement to the matrix multiplication example yields the Petri net in Fig. 11. It allows us to prove that two blocks `increase` cannot modify the `count` variable at the same time (a property that we formalize as the reachability of a marking m with $m(p_i^0) \geq 2$).

Yet, this refinement of the over-approximation does not allow to prove property (2) (formalized as the reachability of a marking m with $m(p_c^0) \geq 1$ and $m(p_l^0) \geq 1$) on the server example (Fig. 2). Running `mist2` on the Petri net in Fig. 10, augmented with a lock for queue `q`, produces the counterexample t_0, t_1, t_3, t_4, t_7. This trace is spurious as it demands that the dispatch of `close` blocks overtakes a `listen` block dispatched.

Under-approximation: We end this section by showing how *under-approximations* allow us to verify properties (2) and (3) in a mechanical way, *and* to detect a potential deadlock. We under-approximate the semantics of QDAS by bounding the size of the queues by a length $k \in \mathbb{N}$. Now, one can represent the queues as global variables over a finite domain, e.g., in our case $\mathbb{D} = (\{\texttt{listen,treat,close,}\} \cup \{\varepsilon\})^k \cup \{\mathit{overflow}\}$, with additional restrictions on assignments given by the FIFO behavior (Fig. 10 (right)). Encoding the FIFO behavior of those bounded queues in the Petri net is then straightforward. We also block the execution when trying to dispatch to a queue that already contains k elements, and mark the content of the queue as $\mathit{overflow}$.

As anticipated, this encoding allows us to prove, with `mist2`, that properties $(2, 3)$ hold in the under-approximation (with a bound $k = 2$ for the queues). Moreover, we can prove that this under-approximation is *complete*, in the sense that it is not possible to reach a configuration where `q` contains more than two blocks in the QDAS of Fig. 2. This can be established by proving, again thanks to the help of `mist2`, that no marking where `q` contains $\mathit{overflow}$ can be reached. We have thus proved properties $(2, 3)$.

Finally, our under-approximation is sufficient to detect that a task `close` can be *synchronously* dispatched to the serial queue `q` by a `treat` block which has been dequeued from `q` too (in line 13 of Fig. 2). We can thus detect automatically this *deadlock*.

Note that further restricting the model by bounding the number of concurrently running tasks yields a *finite state under-approximation* that can be analysed against LTL properties thanks to the [SPIN] model checker, for instance.

9. CONCLUSION & OUTLOOK

In this paper we have introduced QDAS, the first (as far as we know) formal model that captures the main features of the GCD programming framework. We have precisely characterised the decidability and complexity of two natural problems on QDAS and some of their subclasses: the Parikh coverability problem and the termination problem. From a more practical point of view, we have shown how careful abstractions can be used to analyse examples of QDAS that are not in a decidable class.

Future works include designing finer over- and under-approximation techniques that can be applied to practical examples; extending the range of GCD primitives that the QDAS model can handle (for instance, adding task groups, priorities and timer events); and implementing a prototypical verification tool.

REFERENCES

Apple. 2009. DispatchWebServer in Mac Developper Library. (2009). available online at https://developer.apple.com/library/mac/.

Apple. 2010. *Grand Central Dispatch (GCD) Reference*. Technical Report.

Apple. 2011. *Concurrency Programming Guide*. Technical Report.

A. Bouajjani and M. Emmi. 2012. Analysis of Recursively Parallel Programs. In *Proc. of POPL'12*. 203–214.

A. Bouajjani, J. Esparza, and O. Maler. 1997. Reachability Analysis of Pushdown Automata: Application to Model-Checking. In *Proc. of CONCUR'97 (LNCS)*, Vol. 1243. Springer, 135–150.

D. Brand and P. Zafiropulo. 1983. On Communicating Finite-State Machines. *J. ACM* 30, 2 (1983), 323–342.

P. Cousot and R. Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77*. ACM, New York, NY, USA, 238–252. DOI:http://dx.doi.org/10.1145/512950.512973

J. Esparza. 1998. Decidability and complexity of Petri net problems — an introduction. In *Lectures on Petri nets I*. LNCS, Vol. 1491. Springer.

J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. 2000. Efficient Algorithms for Model Checking Pushdown Systems. In *Proc. of CAV'00 (LNCS)*, Vol. 1855. Springer, 232–247.

P. Ganty and R. Majumdar. 2012. Algorithmic Verification of Asynchronous Programs. *TOPLAS* 34, 1 (2012).

P. Ganty, R. Majumdar, and A. Rybalchenko. 2009. Verifying liveness for asynchronous programs. In *Proc. of POPL'09*. ACM Press, 102–113.

G. Geeraerts, A. Heußner, M. Praveen, and J.-F. Raskin. 2013b. ω-Petri nets. In *Proceedings of Petri Nets 2013 (LNCS)*, Vol. 7927. Springer, 49–69.

G. Geeraerts, A. Heußner, and J.-F. Raskin. 2013a. Queue-Dispatch Asynchronous Systems. In *Proc. of ACSD'13*. IEEE, 150–159.

A. Heußner, J. Leroux, A. Muscholl, and G. Sutre. 2012. Reachability Analysis of Communicating Pushdown Systems. *Logical Methods in Computer Science* 8, 3 (2012).

S. Schwoon J. Esparza, A. Kučera. 2003. Model checking LTL with regular valuations for pushdown systems. *Information and Computation* 186, 2 (2003), 355–376.

R. Jhala and R. Majumdar. 2007. Interprocedural analysis of asynchronous programs. In *Proc. of POPL'07*. ACM Press, 339–350.

J. Kochems and C.-H. L. Ong. 2013. Safety Verification of Asynchronous Pushdown Systems with Shaped Stacks. In *Proc. of CONCUR'13 (LNCS)*, Vol. 8052. SV, 288–302.

S. La Torre, P. Madhusudan, and G. Parlato. 2008. Context-Bounded Analysis of Concurrent Queue Systems. In *Proc. of TACAS'08 (LNCS)*, Vol. 4963. Springer, 299–314.

libdispatch. 2013. Project web page. (2013). http://libdispatch.macosforge.org/.

R. Lipton. 1976. *The Reachability Problem Requires Exponential Space*. Technical Report 62. Yale University.

M. Minsky. 1967. *Computation: Finite and Infinite Machines*. Prentice Hall Int.

mist2. 2013. Tool repository and web page. (2013). https://github.com/pierreganty/mist.

QDAS. 2013. `mist2` Example Code. (2013). http://www.swt-bamberg.de/aheussner/research/qdas.html.

C. Rackoff. 1978. The Covering and Boundedness Problem for Vector Addition Systems. *Theoretical Computer Science* 6 (1978), 233–231.

K. Sen and M. Viswanathan. 2006. Model Checking Multithreaded Programs with Asynchronous Atomic Methods. In *Proc. of CAV'06 (LNCS)*, Vol. 4144. 300–314.

SPIN. 2013. Project web page. (2013). http://www.spinroot.com.

Received ; revised ; accepted