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:
DOI:http://dx.doi.org/10.1145/0000000.000000

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
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 asyn-
chronous 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 fork/join 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 \( \text{EXPSPACE} \)-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 \( \text{EXPSPACE}-c \); boundedness (i.e., is the number of pending calls bounded) is decidable in \( \text{EXPSPACE} \), 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.
global int const l,m,n
global int [l][m] matrix1, int[m][n] matrix2, int[l][n] res
global c_queue workqueue, s_queue semaphore, int count
def block increase():
    count = count + 1
def block one_cell(int i, int j):
    for k in range (m):
        res[i][j] += matrix1[i][k] * matrix2[k][j]
        dispatch_s(semaphore, increase())
def main():
    // read input matrix1, matrix2
    count = 0
    for i in range (l):
        for j in range (n):
            dispatch_a(workqueue, one_cell(i,j))
    wait(count = l*n)
    // 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 dispatch_a and dispatch_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.
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 remains 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 “⋆”. 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., \(s \in \prod_{i \in I} S_i\). The i-component of s is written \(s_i \in S_i\), and we identify s with the indexed family of elements \((s_i)_{i \in I}\). We use \(\uplus\) to denote the disjoint union of sets. An alphabet \(\Sigma\) is a finite set of letters. We write \(\Sigma^*\) for the set of all finite words, over \(\Sigma\) and denote the empty word by \(\varepsilon\). 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 \(\sigma\) 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 $d : \mathcal{X} \to \mathbb{D}$. 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 guards $(\mathcal{X})$, assign $(\mathcal{X})$ and $\text{vals}(\mathcal{X})$ 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 $d$ of $\mathcal{X}$ and all $g \in \text{guards}(\mathcal{X})$, we write $d \models g$ iff $d$ 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 $\Delta_F \subseteq S_F \times (\{a, ?a | a \in M\} \cup \{\varepsilon\}) \times S_F$ 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 head 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 $P = \langle S_P, s_P^0, \Delta_P \rangle$ where $S_P$ is a finite set of states, $s_P^0 \in S_P$ is the initial state and $\Delta_P \subseteq S_P \times \{\text{incr}(i), \text{decr}(i), \text{is}\_\text{zero}(i) | i = 1, 2\} \times S_P$ 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 $P = \langle Y, \mathcal{X}, y^0, \Gamma, \Delta \rangle$, 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, $\Gamma$ is the stack alphabet, and $\Delta \subseteq Y \times S \times Y$ is the transition relation, with $\Sigma = (\{\text{push}, \text{pop}\} \times \Gamma) \cup \{\emptyset\} \cup \text{guards}(\mathcal{X}) \cup \text{assign}(\mathcal{X})$. We rely on the classical semantics of PDS with data. In particular, a configuration of a PDS with data is a pair $(s, w, d)$ where $s \in Y$ is a control state, $w \in \Gamma^*$ is the stack content, and $d$ is a valuation of the variables. Clearly, each PDS with data $P = \langle Y, \mathcal{X}, y^0, \Gamma, \Delta \rangle$ can be translated into an equivalent (as far as reachability is concerned) PDS without variables $P' = \langle Y \times \mathbb{D}^{|\mathcal{X}|}, \emptyset, (y^0, d_0, \ldots, d_n), \Gamma, \Delta' \rangle$, 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 $I_t : P \to \{0, 1\}$ 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

\footnote{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 discussing 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) \text{ for all } p \in P. \text{ Given a marking } m, \text{ a transition } t = (I_t, O_t) \text{ is enabled in } m \text{ iff } m(p) \geq I_t(p) \text{ for all } p \in P. \text{ When } t \text{ is enabled in } m, \text{ one can fire the transition } t \text{ in } m, \text{ which produces a new marking } m' \text{ s.t. } m'(p) = m(p) - I_t(p) + O_t(p) \text{ for all } p.

This is denoted } m \rightarrow m', \text{ or simply } m \rightarrow m' \text{ when the transition identity is irrelevant.}

A execution \text{ of a PN is either a sequence } m_0m_1 \ldots m_n \text{ s.t. for all } 1 \leq i \leq n: m_{i-1} \rightarrow m_i \text{ or an infinite sequence } m_0m_1 \ldots m_j \ldots \text{ s.t. for all } i \geq 1: m_{i-1} \rightarrow m_i. \text{ For a PN } N, \text{ we denote by } \text{Reach}(N) \text{ (resp. } \text{Cover}(N)) \text{ the reachability (coverability) set of } N, \text{i.e. the set of all markings } m \text{ s.t. there exists a run } m_0m_1 \ldots m_n \text{ of } N \text{ with } m = m_n (m \leq m_n). \text{ The coverability problem asks, given a PN } N \text{ and a marking } m, \text{ whether } m \in \text{Cover}(N). \text{ 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 $\mathcal{D}$ be a finite data domain containing an initial value $d_0$. A queue-dispatch asynchronous system (QDAS) $A$ is a tuple $\langle CQID, SQID, \Gamma, main, X, \Sigma, (TS_\gamma)_{\gamma \in \Gamma} \rangle$ where:

- $CQID$ and $SQID$ are sets of (c)oncurrent and (s)erial queues respectively;
- $\Gamma$ is a finite set of blocks and $main \in \Gamma$ is the initial block. A block $\gamma \in \Gamma$ is a tuple $TS_\gamma = (S_\gamma, S_0, \beta_\gamma, f_\gamma, \Delta_\gamma, \Sigma_\gamma)$ where $(S_\gamma, S_0, \Sigma_\gamma)$ is an LTS with final state $f_\gamma \in S_\gamma$;
- $X$ is a finite set of variables, taking value in $\mathcal{D}$;
- $\Sigma = \Sigma_d \cup \text{guards}(X) \cup \text{assign}(X)$ is the set of actions, where $\Sigma_d$ is the set of possible dispatch actions: $\Sigma_d = \{\text{dispatch}_a, \text{check}_a\} \times (CQID \cup SQID) \times (\Gamma \setminus \{main\})$.

We assume that $SQID, CQID, \Gamma, X,$ and all $S_\gamma$ for $\gamma \in \Gamma$ are disjoint from each other. Let $S = \bigcup_{\gamma \in \Gamma} S_\gamma$, $F = \bigcup_{\gamma \in \Gamma} f_\gamma$, $\Delta = \bigcup_{\gamma \in \Gamma} \Delta_\gamma$, and $QID = SQID \cup CQID \cup \{1\}$ (where $1 \notin SQID \cup CQID$). We further assume that there is a symbol $\tau$ 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 $\text{queue}(v)$ of the queue that contains it (for call vertices) or that contained it (for task vertices). Task vertices are labeled by their current state $\text{state}(v)$ (for convenience, we also label call vertices by the initial state of their respective blocks – not shown in the figure).
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 $γ_2γ_3γ_2$, and $q_1$ is parallel with content $γ_1γ_2$. There are 4 active tasks, 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 $A = \langle CQID, SQID, Γ, main, X, Σ, (TS)_{γ∈Γ} \rangle$, a call-task graph over $A$ is a tuple $G_A = \langle V, E, λ, queue, state \rangle$ where: $V = V_C \cup V_T$ 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; $λ : V → Γ$ labels each vertex by a block; $queue : V → QID \cup \{\} \setminus \{\} \cup \{\} \cup \{\}$ associates each vertex to a queue identifier (or $q$ is empty); and $state : V → S$ associates each vertex to an LTS state. For each $v \in QID$, let $V_q = \{ v ∈ V | queue(v) = q \}$. We let $E_q = \bigcup_{q ∈ QID} E_q$ be the set of queue edges, with $E_q = E \setminus (V_q \times V_q)$ for all $q ∈ QID$; and let $E_w = E \setminus E_q$.

A CTG is empty iff $V = \emptyset$. A path (of length $n$) in $G_A$ is a sequence of vertices $v_0, v_1, \ldots, v_n$ s.t. for all $1 ≤ i ≤ n$; $(v_{i−1}, v_i) \in E$. Such a path is simple iff $v_i ≠ v_j$ for all $1 ≤ i < j ≤ n$. The restriction of $G_A$ to $V' \subseteq V$ is the CTG $G'_{A} = \langle V', E', λ', queue', state' \rangle$, 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 $v ∈ V_T$: $state(v) ∈ S_{λ(v)}$ where $S_{λ(v)}$ are the states of $TS_{λ(v)}$.
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 $q ∈ QID$, the restriction of $G_A$ 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 $q ∈ SQID$, there is at most one task vertex $v$ s.t. $queue(v) = q$. This ensures that serial queues force the serial execution of the block they contain.

For convenience, we also introduce the following notations: Let $G_A$ be a CTG, and let $q$ be a queue identifier of $A$. Then, head$\langle q, G_A \rangle$ and tail$\langle q, G_A \rangle$ denote respectively the head and the tail of $q$ in the configuration described by $G_A$. That is, tail$\langle q, G_A \rangle$ is the call vertex $v \in V_q$ that has no incoming queue edge, or $⊥$, if such a vertex does not exist; and head$\langle q, G_A \rangle$ is the call vertex $v \in V_q$ that has no outgoing queue edge (but possibly an outgoing wait edge), or $⊥$, 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) $state(v) = f_{λ(v)}$ (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 $A$ be a QDAS and $G_A = \langle V, E, λ, queue, state \rangle$ be a CTG for $A$.
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 \( q \in QID \), enqueue\((q,\gamma))(G,A)\) is the CTG \((V', E', \lambda', \text{queue}', \text{state}')\) where: \( V' = V \cup \{v'\}, v' \) is a fresh call vertex, \( \lambda(v') = \gamma \), \( \text{queue}'(v') = q \), \( \text{state}'(v') = s_0' \), and for all \( v \in V: \lambda'(v) = \lambda(v), \text{queue}'(v) = \text{queue}(v), \text{state}'(v) = \text{state}(v) \). Finally, \( E' = E \cup E_1 \cup E_2 \), where: (i) \( E_1 = \{(v', \text{tail}(G,A,q))\} \) if \( \text{tail}(G,A,q) \neq \bot \), and \( E_1 = \emptyset \) otherwise, and (ii) if \( v \in V \) is a task node s.t. \( \text{queue}(v) = q \in SQID \) and \( \text{tail}(G,A,q) \neq \bot \), then \( E_2 = \{(v', v)\} \), otherwise \( E_2 = \emptyset \).

Intuitively, this operation inserts a call to \( \gamma \) 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 \( \text{head}(q) \) is different from \( \bot \) and unblocked, then dequeue\((q)(G,A)\) is the CTG \((V', E', \lambda, \text{queue}, \text{state})\) where \( V' = V \setminus \{\text{head}(q)\} \) and \( E' = E \cup \{(\text{head}(q), v')\} \). Otherwise, \( \text{head}(q) = \bot \) and dequeue\((q)(G,A)\) is undefined. Intuitively, this operation removes the first (with respect to the FIFO ordering) block from \( q \) and turns the corresponding call vertex \( \text{head}(q) \) into a task vertex, meaning that the block is now running as a task.

**Executing one step in a task:** for all \( \delta = (s, a, s') \in \Delta \), step\((\delta)(G,A)\) is a set of CTG defined as follows. \((V, E, \lambda, \text{queue}, \text{state})\) is a fresh node that \( step(\delta)(G,A) \) if there exists an unblocked \( v \in V \setminus \{s\} \) s.t. \( \text{state}(v) = s \), \( \text{state}'(v) = s' \) and for all \( v' \neq v \): \( \text{state}'(v') = \text{state}(v') \). Note that \( step(\delta)(G,A) \) can be empty. Intuitively, each graph in \( step(\delta)(G,A) \) 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 \( v' \in V: \text{letwait}(v,v')(G,A) \) is either the CTG \( G,A \) if \( v = \bot \), or the CTG \((V, E \cup \{v,v'\}, \lambda, \text{queue}, \text{state})\) 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 \( A = (CQID, SQID, \Gamma, \text{main}, \mathcal{X}, \Sigma, (TS,)_\gamma \) be a QDAS. A configuration is a pair \((G,d)\), where \( G \) is a CTG of \( A \) and \( d \in \text{vals}(\mathcal{X}) \). The semantics of \( A \) is the labeled transition system \([A] = (C, d^0, \Sigma, \Rightarrow)\) where: (i) \( C \) contains all the pairs \((G,d)\) where \( d \in \text{vals}(\mathcal{X}) \), and \( G \) is a CTG of \( A \). (ii) \( d^0 = (G^0, d^0) \) with \( d^0(x) = d_0 \) for all \( x \in \mathcal{X} \), and \( G^0 = (\{v^0\}, s_0, \lambda, \text{queue}, \text{state}) \), where \( v^0 \) is a task node, \( \lambda(v^0) = \text{main}, \text{state}(v^0) = s_{\text{main}} \) and \( \text{queue}(v^0) = i \). (iii) \( \Sigma = \Sigma \cup \{\tau\} \) and (iv) \((G,d), (a, (G,d')) \) \( \in \Rightarrow \) iff one of the following cases holds:

**Asynchronous dispatch:** \( a = \text{dispatch}_a(q, \gamma), d' = d, \) and there are \( \delta = (s, a, s') \in \Delta \) and \( G'' \in \text{step}(\delta)(G) \) s.t.: \( G'' = \text{enqueue}(q, \gamma)(G'') \).

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

**Test:** \( a = g \in \text{guards}(\mathcal{X}), d' = d, d \mid\mid g, \) and there is \( \delta = (s, a, s') \in \Delta \) s.t. \( G'' \in \text{step}(\delta)(G) \).

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

**Scheduler action:** \( a = \tau, d' = d \) and either
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 \((s, d)\), where \(s\) is a state of main, and \(d\) is a valuation of the variables. Hence queueless QDAS are essentially LTS with variables over a finite data domain, and thus:
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 \((G, d)\) 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 \(TG\) either.

\[ G ⊩ w \]

\[ A \quad \Delta \quad y = ε \]

\[ \text{From QDAS to PDS:} \quad \text{Let} \quad A = \langle \emptyset, SQID, Γ, main, X, Σ, (TS, γ)_{γ ∈ Γ} \rangle \quad \text{be a synchronous QDAS with set of locations} \quad S, \text{set of transitions} \quad Δ, \text{set of final states} \quad F. \text{Let} \quad G \quad \text{be a CTG of one of the forms given in Fig. 4, and let} \quad w = w_0w_1 \cdots w_n \quad \text{be a word in} \quad S^*. \] Then, \( G \quad \text{is encoded by} \quad w \), written \( G \triangleright w \), iff either \( G \) is empty and \( w = ε \), or \( G \) is not empty and \( w_i = state(v_i) \) for all \( 0 ≤ i ≤ n \). From \( A \), we build a pushdown system with data \( \mathcal{P}_A = \langle Y, X, y^0, S, Δ_P \rangle \) where:

- the set of states is \( Y = S \cup \{τ\} \) and the initial state is \( y^0 = s^0_{main} \)
- a tuple \((y, a, y') \in Δ_P\) iff one of the following holds: (i) either \( a \in \text{guards}(X) \) and \( (y, a, y') \in Δ \) (ii) or \( a = \text{push}(s'), (s, \text{dispatch}_a(q, γ), s') \in Δ \) and \( y' = s_i^0 \)
- (iii) or \( a = \text{pop}(s), y ∈ F \) and \( y' = s \) (iv) or \( a = \text{empty}? \), \( y = f_{main} \), and \( y' = τ \).

Thus, at all times, the current location of \( \mathcal{P}_A \) encodes the current location of the (single) running block in \( A \), and the stack content records the sequence of synchronous dispatches, as described above. A guard or assignment in \( A \) is kept as is in \( \mathcal{P}_A \). A synchronous dispatch \((s, \text{dispatch}_a(q, γ), s') \) in \( A \) 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 \( \mathcal{P}_A \) to the initial state of \( γ \). The termination of a block is simulated by a pop (and we use the empty? action for the termination of \( \text{main} \)).

**Proposition 4.2.** For all synchronous QDAS \( A \), we can construct a pushdown system with data \( \mathcal{P}_A \) such that:

(i) for all runs \( ρ = (G_0, d_0)a_1(G_1, d_1) \ldots a_n(G_n, d_n) \) of \( A \), there exists a run \( π = (s_0, w_0, d_0)a_1(s_1, w_1, d_1') \ldots a_n(s_n, w_n, d_n') \) of \( \mathcal{P}_A \) with \( d_i = d_i' \) and \( G_i \triangleright w_i \) for all \( 0 ≤ i ≤ n \) and, symmetrically

(ii) for all runs \( π = (s_0, w_0, d_0)a_1(s_1, w_1, d_1') \ldots a_n(s_n, w_n, d_n') \) of \( \mathcal{P}_A \), there exists a run \( ρ = (G_0, d_0)a_1(G_1, d_1) \ldots a_n(G_n, d_n) \) of \( A \) with \( d_i = d_i' \) and \( G_i \triangleright w_i \) for all \( 0 ≤ i ≤ n \)

**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 \((y, w, d) ∈ Y \times S^* \times D^X\), 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 $\mathcal{P}_A$, and $d \in \mathbb{D}^X$ as before.

Let $p = (G_0, d_0)a_1(G_1, d_1)a_2 \ldots a_n(G_n, d_n)$ be a run of $A$. Let us build by induction a corresponding run $(x_0, d_0)a_1(x_1, \hat{d}_1)a_2 \ldots a_n(x_n, \hat{d}_n)$ of $\mathcal{P}_A$ with $x_i = w_i \cdot y_i \in S^* \cdot (S \cup \{\tau\})$ and $\hat{d}_i \in \mathbb{D}^X$ for $1 \leq i \leq n$. The base case holds by definition of $\mathcal{P}_A$. Let us assume that $(x_0, d_0) \ldots (x_j, d_j)$ is a run of $\mathcal{P}_A$ s.t. $d_i = d_i^i = \hat{d}_i$ and $G_i \triangleright w_i$ for all $0 \leq i \leq j$ and let us show how to extend it. Since $p$ 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:** $a_{j+1} = \text{dispatch}_{\alpha}(\gamma, q)$. In this case, $d_j = d_{j+1}$ and $G_{j+1}$ is a path graph $v_0v_1 \ldots v_nv_{n+1}$ where $v_{n+1}$ is a fresh vertex with state $(v_{n+1}) = v_0^0$. In the run of $\mathcal{P}_A$, we let $\hat{d}_j = \hat{d}_{j+1}$ and $x_{j+1} = x_j \cdot s_i^0$ (i.e. $\mathcal{P}_A$ pushes the current state on the stack, jumps to state $s_i^0$).

**Test/Assignment:** In this case, $G_{j+1}$ equals $G_j$ except for state $j(v_n) = s$ and state $j+1(v_n) = s'$ and a possible change of $d_{j+1}$ if $a_{j+1}$ is an assignment. We simulate this action in $\mathcal{P}_A$ by letting $\hat{d}_{j+1} = \hat{d}_{j+1}$ 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 state $j(v) \in F$,

$G_{j+1} = G_j \setminus v$, and $d_j = d_{j+1}$ (note that $G_{j+1}$ could be empty). We let $\hat{d}_j = \hat{d}_{j+1}$.

To define $x_{j+1}$, we consider two further cases: (i) Either $x_j = w \cdot y_j$ with $w_j \neq \varepsilon$. In this case, we use a pop action in $\mathcal{P}_A$, and let $x_{j+1} = w$ (ii) Or $x_j = y_j \in S$ (i.e., the stack is empty). In this case, $x_j = f_{\text{main}}, G_j$ is a path of length 1 and $G_{j+1}$ is empty. We thus execute the empty? action in $\mathcal{P}_A$, and obtain $x_{j+1} = \tau$.

Clearly, in all those cases, $G_{j+1} \triangleright w_{j+1}$ and $d_{j+1} = \hat{d}_{j+1}$. 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 $\{w \in S^* : |w|_s \geq f(s) \text{ for all } s \in S \}$ 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 $(Q, S, q_0, \Delta, q_f)$ 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 $L(\mathcal{F}_f) = \{w \in S^* \mid |w|_s \geq f(s) \}$ for all $s \in S$. Let $w = a_1 \ldots a_n \in L(\mathcal{F}_f)$ be a word accepted by the run $q_0 a_1 a_2 \ldots a_n q_n$ where $q_0 = q_0^0$ and $q_n = q_f$. By definition of $\Delta$, it is clear that $|w|_s \geq q_0(s) = f(s)$ for all $s \in S$. Symmetrically, let $w$ be word that is not in $L(\mathcal{F}_f)$, and let $q_0 a_1 a_2 \ldots a_n q_n$ 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 $w \notin L(\mathcal{F}_f)$ and thus $q_n \neq q_f$. Since each transition ($q_i-1, a_i, q_i$) ensures that $q_i-1(s) \geq q_i(s)$, 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 $A$ with set of states $S$ and a function $f : S \to \mathbb{N}$, one can build a PDS $P_{A,f}$ whose size is exponential in the size of $A$ and a state $s$ of $P_{A,f}$, such that $P_{A,f}$ reaches $s$ iff $f$ is Parikh coverable in $A$.

Proof. First, we construct the PDS with data $P_A$ and states $S$ in the proof of Proposition 4.2. Then, we translate the PDS with data to a PDS without data $P_A = \langle \hat{Y}, \hat{y}^0, \hat{\phi}, \hat{\Sigma}, \hat{\Delta} \rangle$ by encoding the variable valuations into the states of $P_A$. More precisely, $\hat{Y} = S \times \mathbb{N}^A$, and the transition relation is adapted accordingly. For all $y = (s,v) \in \hat{Y}$, we let $S(y) = s$, i.e. $S(y)$ is the state of $P_A$ corresponding to $y$. Observe that the size of $P_A$ is at most exponential in the size of $P_A$ 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 $F_f = \langle Q, S, q^0, \Delta_f, q_f \rangle$ as in the proof of Lemma 4.3.

Finally, we define the PDS $P_{A,f} = \langle Y, y^0, \Phi, \Sigma, \Delta_{A,f} \rangle$ as follows: (i) $Y = \hat{Y} \cup Q$ is the set of states; (ii) $y^0 = \hat{y}^0$ is the initial state; (iii) $\Phi = \hat{\Phi} = S$ is the stack alphabet; and (iv) $\Sigma = \hat{\Sigma} \cup \{ \tau \}$. Moreover, there is a transition $(y,a,y')$ in $\Delta_{A,f} \subseteq Y \times \Sigma \times Y$ iff one of the following holds. Either (i) $(y,a,y') \in \Delta$ (all transitions of $P_A$ are transitions of $P_{A,f}$); or (ii) there is $s \in \Phi$ s.t. $(q,s,q') \in \Delta_f$ and $a = \text{pop}(s)$ (the transitions of $P_{A,f}$ include all transitions of $F_f$ where the $s$ label has been replaced by a $\text{pop}(s)$ action); or (iii) $y \in \hat{Y}$, $a = \text{push}(S(y))$ and $y' = q^0$ (from all states in $\hat{Y}$ there is a transition to the initial state of $F_f$ that pushes the current state $S(y)$). Note that the size of $P_{A,f}$ is exponential in the sizes of the instance of the Parikh coverability problem $(A,f)$, because $P_{A,f}$ amounts, roughly speaking to a serial composition of $P_A$, and a (modified version of) $F_f$. The aim of this construction is that $P_{A,f}$ first simulates of run of $A$ in its $P_A$ part, and then uses its $F_f$ 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 $P_{A,f}$ that reaches $q^0$ where $x_i = (y_i, w_i)$ for all $i$. Let us show that there exists a configuration $c = (G,d)$ of $A$ such that $f \preceq \text{Parikh}(G)$. Because of the definition of $P_{A,f}$, $\rho$ has a prefix that is a run of $P_A$ and thus visits only states of the form $(y,w)$ with $y \in \hat{Y}$. Let $x_0, a_1, \ldots, x_k$ be this prefix. Hence, $a_{k+1} = \text{push}(S(y_k))$ 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 $P_A$, there is, by Proposition 4.2 a run of $A$ that reaches $c = (G,d)$ with $G \triangleright y_k \ast S(y_k)$. Then, the transition $(x_k, \text{push}(S(y_k)), x_{k+1})$ transfers $w_{k+1} = y_k \ast S(y_k)$ 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 $F_f$ ensures that $|w_{k+1}| \geq f(s)$ for all $s \in S$. Hence, $f \preceq \text{Parikh}(G)$ for $c = (G,d) \in \text{Reach}(A)$, and $f$ is thus Parikh-coverable in $A$.

The other direction of the proof follows from a similar argument. If $f$ is Parikh-coverable in $A$, there exists a run of $A$ that reaches some configuration $c = (G,d)$ s.t. $f \preceq \text{Parikh}(G)$. By proposition 4.2, this run can be simulated by the $\langle P_A, \rangle$ part of $P_{A,f}$ that will reach a configuration with stack content $w$ s.t. $G \triangleright w$. Then, the $\langle F_f, \rangle$ part of $P_{A,f}$ will check that $f \preceq \text{Parikh}(G)$, and accept. \hfill $\square$

From PDS to QDAS: Given a PDS $P$, we construct a synchronous QDAS $A_P$ as shown in Figure 5. The underlying idea is symmetric to the reduction from QDAS to PDS: we simulate a push($\phi$) action by a synchronous dispatch of a block $\phi$, and a pop($\phi$) action by the termination of a $\phi$ task. Hence, the content of the stack is encoded in the Cfg 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 $P$ (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.
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 $A$, we build a Petri net $N_A$ that simulates it. The places of $N_A$ are $(X \times D) \cup S$. 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 $x$ has $d$ elements.
there is
 Async. dispatch: there is \((s,(\text{dispatch}_a,q,\gamma),s')\) \(\in \Delta\) \(s.t.\) \(I_t(p) = 1\) \(i f f\) \(p = s\) and \(O_t(p) = 1\) \(i f f\) \(p \in \{s',s_0\}\).

**Test:** there is \((s,x = d,s')\) \(\in \Delta\) with \(x = d\) \(i n\) guards \((\mathcal{X})\) \(s.t.\) \(I_t(p) = 1\) \(i f f\) \(p \in \{(x,d),s\}\) and \(O_t(p) = 1\) \(i f f\) \(p \in \{(x,d),s'\}\).

**Assignment:** there are \((s,a,s')\) \(\in \Delta\) with \(a = x \leftarrow v \in \text{assign}(\mathcal{X})\) and \(v' \in \mathcal{D}\) \(s.t.\) \(I_t(p) = 1\) \(i f f\) \(p \in \{(x,v'),s\}\) and \(O_t(p) = 1\) \(i f f\) \(p \in \{(x,v),s'\}\).

**Task termination:** there is \(\gamma \in \Gamma\) \(s.t.\) \(I_t(p) = 1\) \(i f f\) \(p = f_\gamma\) 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 \(\gamma\) directly creates a new token in \(s_0\). This encoding is, however, sound with respect to the Parikh coverability problem, as Parikh(G) does not distinguish between a block \(\gamma\) that is waiting in a queue, and a task executing \(\gamma\) in its initial state. The soundness of this construction is given by the following proposition:

**Proposition 4.8.** For all concurrent asynchronous QDAS A with set of locations S, we can build, in polynomial time, a Petri net \(N_A\) \(s.t.\) \(f : S \rightarrow \mathbb{N}\) is Parikh-coverable in A \(i f f\) \(m \in \text{Cover}(N_A)\), 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 \(N_A\) can simulate precisely the sequence of Parikh images that are reachable in \(A\). Let \((G,d)\) be a configuration of \(A\), and let \(m\) be marking of \(N_A\). We say that \(m\) encodes \((G,d)\), written \(m \triangleright (G,d)\) \(i f f:\) (i) for all \(x \in \mathcal{X}: m(x,dl(x)) = 1\); (ii) for all \(x \in \mathcal{X}:\) for all \(d \in \mathcal{D}\) \(\setminus \{dl(x)\}: m(x,d) = 0\); and (iii) for all \(s \in S\) \(m(s) = \text{Parikh}(G)(s)\).

**Lemma 4.9.** Let \(A\) be a concurrent asynchronous QDAS with set of variables \(\mathcal{X}\) and set of locations \(\mathcal{S}\), and let \(N_A\) be its associated PN. Then, for all \((G,d) \in \text{Reach}(A)\) there is \(m \in \text{Reach}(N_A) \setminus m \triangleright (G,d)\) and for all \(m \in \text{Reach}(N_A)\), there is \((G,d) \in \text{Reach}(A)\) \(s.t.\) \(m \triangleright (G,d)\).

**Proof.** Let \((G,d)\) be a configuration in \(\text{Reach}(A)\), and let \((G_0,d_0)\equiv (G_1,d_1)\equiv \cdots \equiv (G_n,d_n)\) be a run \(s.t.\) \((G_n,d_n)\). We can thus inductively build a run \(m_0m_1 \cdots m_n\) of \(N_A\) \(s.t.\) \(m_k \triangleright (G,d)\), whose correctness follows by induction on the length \(n\) of the QDAS run.

Now, let \(m_0m_1 \cdots m_n\) be a run of \(N_A\) and let us build, inductively, a run \((G_0,d_0)\equiv (G_1,d_1)\equiv \cdots \equiv (G_k,d_k)\) \(s.t.\) \(m_{n} \triangleright (G_k,d_k)\) 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 \(N_A\), as described above, is polynomial. Then, assume \(f\) is Parikh coverable in \(A\), i.e. there is \((G,d) \in \text{Reach}(A)\) \(s.t.\) \(f \leq \text{Parikh}(G)\). By Lemma 4.9, there is \(m' \in \text{Reach}(N_A)\) \(s.t.\) \(m' \triangleright (G,d)\), which means that \(m'(s) = \text{Parikh}(G)(s)\) for all \(s \in S\). So, for all \(s \in S:\) \(m(s) = f(s) \leq \text{Parikh}(G)(s) = m'(s)\) and thus, \(m \leq m'\), since \(m(p) = 0\) for all \(p \notin S\).

Finally, \(m' \in \text{Reach}(N_A)\) implies that \(m \in \text{Cover}(N_A)\).

On the other hand, assume \(m \in \text{Cover}(N_A)\), 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 \(m' \in \text{Reach}(N_A)\) be a marking \(s.t.\) \(m \leq m'\). Such
a marking exists since \( m \in \text{Cover}(N_A) \). By Lemma 4.9, there is \((G, a) \in \text{Reach}(A) \) s.t. \( m' > (G, a) \). Thus, by definition of >, \( m'(s) = \text{Parikh}(G)(s) \) for all \( s \in S \). Since \( m \leq m' \) and by definition of \( f \), we conclude that \( f(s) = m(s) \leq m'(s) = \text{Parikh}(G)(s) \) for all \( s \in S \), and \( f \) is Parikh-coverable in \( A \). 

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 \( N = \langle P, I, m_0 \rangle \) be a Petri net and let us consider the concurrent asynchronous QDAS \( A_N = \langle \text{CQID}, \emptyset, \Gamma, \text{main}, \Sigma, (TS_p)_{p \in \Gamma} \rangle \), on the finite domain \( \mathbb{D} = \{0,1\} \), where \( \text{CQID} = \{C\} \) and \( \Gamma = \{\text{main}, \text{trans}\} \cup P \), \( \Sigma = \{v_p \mid p \in P\} \). The transition systems of the blocks \((TS_p)_{p \in \Gamma} \) are given in Fig. 6. For convenience, we express \( TS_{\text{main}} \) and \( TS_{\text{trans}} \) using pseudo-code while each \( TS_p \) is given by the automaton with states \( s^0_p, s^\text{mid}_p, s^\text{fin}_p \) depicted in the figure. We assume that, for \( \gamma \in \{\text{trans}, \text{main}\} \), \( s^\gamma_p \) is the location of \( TS_p \) that is reached when the control reaches line \( \ell \). Let \( G = \langle V, E, \lambda, \text{queue}, \text{state} \rangle \) be a CTG of \( A_N \), and let \( m \) be a marking of \( N \). Then, we say that \( G \) encodes \( m \), written \( G > m \) if (i) \( \text{Parikh}(G)(s^{\text{trans}}_{13}) = \text{Parikh}(G)(s^\text{main}_8) = 1 \); (ii) for all \( p \in P \): \( \text{Parikh}(G)(s^0_p) = m(p) \) (remember that \( s^0_p \) is the initial location of \( TS_p \)); and (iii) for all \( p \in P \), for all \( s \in S_p \setminus \{s^0_p\} \): \( \text{Parikh}(G)(s) = 0 \). Thus, intuitively, a CTG \( G \) encodes a marking \( m \) if \( \text{main} \) is at line 8, \( \text{trans} \) 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 \( s^\text{mid}_p \) or \( s^\text{fin}_p \).

The intuition behind the construction is as follows. Each run of \( A_N \) 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 \( \text{trans} \). 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, \( \text{trans} \) 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 \( \text{at least} \ I_t(p) \) tasks have terminated, for all \( p \in P \). So, when \( \text{trans} \) reaches line 19, the encoded marking has been decreased by \( \text{at least} \ I_t \). 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 \( p \in P \), \( \text{trans} \) 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 \( \text{trans} \) is to simulate the effect of \( t \), plus a possible token loss. Hence, the resulting marking is guaranteed to be in \( \text{Cover}(N) \) (but maybe not in \( \text{Reach}(N) \)). This is formalized by the following proposition:

---

1. def main():
   2.     for each p ∈ P:
   3.         vp := 0
   4.     select kp ∈ {0, ..., m0(p)}
   5.     for i := 1...kp:
   6.         dispatch_s(C, p())
   7.     dispatch_s(C, trans())
   8.     while(true): do nothing
   9.     block p(): // For all p ∈ P
  10.     if vp = 0: do nothing
  11.     vp := 1

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

Fig. 6. Encoding of Petri net coverability \( \langle P, I, m_0 \rangle \) by a QDAS / LTS \( TS_p \) of block \( p \)
Proposition 4.10. For all Petri nets $N$, we can build, in polynomial time, a concurrent asynchronous QDAS $A_N$ s.t.: $m \in \text{Cover}(N)$ iff there are a marking $m'$ and a configuration $(G, d) \in \text{Reach}(A_N)$ with $G \triangleright m'$ and $m \preceq m'$.

The proof of Proposition 4.10 is split into two lemmas, given hereunder. They rely on a different characterization of $\text{Cover}(N)$. That is, $m \in \text{Cover}(N)$ 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 $m_{i+1}$ and a transition $t_i$ s.t. $m'_i \xrightarrow{t_i} m_{i+1}$ and $m_{i+1} \preceq m_{i+1}$. 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 lemmas also assumes that for each $p \in P$, the corresponding LTS $TS_p = \{s_0, s_{p,mid}, s_{p,fin}\} \cup \{s_{p}, \Sigma, \Rightarrow\}$ is as is in Fig. 6.

Lemma 4.11. Let $N = \langle P, T, m_0 \rangle$ be a PN, and let $A_N = \langle CQID, 0, \Gamma, \text{main}, X, \Sigma, (TS_p)_{p \in \Gamma} \rangle$ be its corresponding QDAS. If $m \in \text{Cover}(N)$ then there are a marking $m'$ and $(G, d) \in \text{Reach}(A_N)$ s.t. $G \triangleright m'$ and $m \preceq m'$.

Proof. Let $m$ be a marking from $\text{Cover}(N)$ 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 $\text{Cover}(N)$. 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 $(G_i, d_i) \in \text{Reach}(A_N)$ (with $G_i = \langle V^i, E^i, \lambda^i, \text{queue}^i, \text{state}^i \rangle$) s.t. $E^i = \emptyset; d_i(v_p) = 0$ for all $p \in P$; and $G_i \triangleright m_i$. □

Lemma 4.12. Let $N = \langle P, T, m_0 \rangle$ be a PN, and let $A_N = \langle CQID, 0, \Gamma, \text{main}, X, \Sigma, (TS_p)_{p \in \Gamma} \rangle$ be its corresponding QDAS. For all $(G, d) \in \text{Reach}(A_N)$ and all markings $m$ it holds that $G \triangleright m$ implies $m \in \text{Cover}(N)$.

Proof. For a CTG $G$ of $A_N$ with set of vertices $V$, we denote by $M(G)$ the marking of $N$ s.t. for all $p \in P$: $M(G)(p) = | \{ v \in V \mid \text{state}(v) = s_0^p \} |$. Thus, in the case where $G$ encodes a configuration s.t. trans is at line 13, $\text{main}$ is at line 8, and all the $p$ blocks are in their initial state $s_0^p$, then $G \triangleright M(G)$.

To establish the lemma, one can prove the following stronger statement: for all reachable configurations $(G, d)$ s.t. trans is at line 13: $M(G) \in \text{Cover}(N)$. This is easily established by induction on the length of the run. □

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

Let $F = (S_F, s^0_F, M, \Delta_F)$ 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 $A_F = (\emptyset, \{q\}, \Gamma, \text{main}, \Sigma, (TS)_\gamma) | \gamma \in \Gamma)$ on domain $\mathbb{D} = M \cup S_F \cup \{\varepsilon\}$, where $\Gamma = M \cup \{\varepsilon, \text{main}\}$, $\Sigma = \{\text{state}, \text{head}\}$ and the $TS_\gamma$ are given by the pseudo code in Fig. 7.

Intuitively, runs of $A_F$ 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 is easy to check that, once $\text{main}$ has reached line 7, all the CTG that are reached in $A_F$ 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: $\text{main}$ and possibly one task running an $m$ block (with $m \in M \cup \{\varepsilon\}$), that has to terminate to allow a further dequeue from $q$ (if $q$ 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 $\varepsilon$ block, which is dispatched by main at the beginning of the execution. Since head is initialized to $\varepsilon$, the test in line 9 will evaluate to true in the task running $\varepsilon$, and this task will be able to simulate the first transitions of $F$.

Thus, in all reachable configurations of $A_F$, 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 $s^\ell_\gamma$ the location of $TS_\gamma$ that corresponds to line $\ell$ in Fig. 7. Then, we say that a configuration $(G, d)$ of $A_F$ encodes a configuration $(s, w)$ of
F, written \((G,d) \triangleright (s,w)\) iff: (i) \(s = \text{d(state)}\); (ii) \(G\) is of either shapes in Fig. 7 with \(w = m_1 m_2 \ldots m_n\); (iii) \(\text{Parikh}(G)(s_{a_{\text{main}}}) = 1\); and (iv) there exists \(a \in M \cup \{\varepsilon\}\) such that \(\text{Parikh}(G)(s_{a_{\text{main}}}) = 1\). That is, \(s\) and \(w\) are encoded as described above, \text{main} is at line 7, and the running \(a\) 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 \(A_F\) be its associated QDAS. For all runs \((s_0,w_0)(s_1,w_1)\ldots(s_n,w_n)\) of \(F\) s.t. for all \(0 \leq i < n\): \(s_i \neq c\), there exists \((G,d) \in \text{Reach}(A_F)\) s.t. \((G,d) \triangleright (s_n,w_n)\). Moreover, for all \((G,d) \in \text{Reach}(A_F)\) and for all configurations \((s,w) \in F\): \((G,d) \triangleright (s,w)\) implies \((s,w) \in \text{Reach}(F)\).

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. \(f_m(s^{(20)}_a) = 1\) and \(f_m(s) = 0\) for all \(s \neq s^{(20)}_a\). 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 \(A_F\).

By Lemma 5.1, we can deduce the Parikh coverability of a QDAS configuration where \(\text{d(state)} = c\) 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 \(A_F\). By Lemma 5.1, this entails the existence of \((c,w) \in \text{Reach}(F)\), 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 \(P = (S_P, s^0_P, \Delta_P)\), we build the QDAS \(A_P = (CQID, 0, \Gamma, \text{main}, \mathcal{X}, \Sigma, (TS_M)_{\gamma \in \Gamma})\) with only one concurrent queue: \(CQID = \{q\}\), with set of blocks \(\Gamma = \{(\text{null}, \text{one}) \times \{1,2\}\} \cup \{\text{main}\}\), and set of variables \(\mathcal{X} = \{\text{state}, \ell^1, \ell^2, x^1, \ell^3, \ell^2, x^2\}\) where state takes values in \(S_P\), all \(\ell^i\) are Boolean variables and \(x^1, x^2\) range over the domain \{\text{incr, decr, is_zero, 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 \(\text{one}(i)\) (counter \(i\) is not null) or \(\text{null}(i)\) (counter \(i\) is null). More precisely, in such a CTG, the value of counter \(i\) is given by the number of \(\text{one}(i)\) 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 \(\text{null}(i)\) are always present and are unblocked if counter \(i\) is null. Finally, the control location of \(P\) is recorded in global variable state.

It is the duty of the \text{main} task to simulate the actions of the 2CM. To achieve this, the \text{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 \text{null} or \text{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 \(\text{if} m \text{ or } i? m\) meaning respectively that the tasks send or
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 \( \text{main} \) can send messages of the form \( !\text{incr} \), \( !\text{decr} \), or \( !\text{is\_zero} \) to signal that an increment (respectively decrement, test-for-zero) must be performed on counter \( i \). These messages will be received by the unblocked \( \text{one}(i) \) or \( \text{null}(i) \) blocks. In addition an \( i?\text{ack} \) message can be received by \( \text{main} \) to acknowledge the execution of the counter operation.

It is thus the duty of the \( \text{one}(i) \) and \( \text{null}(i) \) blocks to perform the actual counter updates. When a \( \text{null}(i) \) or \( \text{one}(i) \) receives an \text{incr} message, it performs an asynchronous dispatch of \( \text{one}(i) \) into \( q \) to increment counter \( i \), and acknowledges the operation to \( \text{main} \), thanks to message \text{ack}. When an \( \text{one} \) block receives a \text{decr} message, it terminates, which decrements the counter. \( \text{null} \) blocks cannot receive \text{decr} messages, so, if \( \text{main} \) requests a \text{decr} operation when the counter is zero, \( \text{main} \) 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 \( \text{null} \) blocks can receive and acknowledge \text{is\_zero} messages, so, again, \( \text{main} \) is blocked after sending \text{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 \( i\text{lm} \) and \( i?\text{m} \) 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 \( \ell_1 \) and \( \ell_2 \) 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 \( M = \{\text{incr}, \text{decr}, \text{is\_zero}, \text{ack}\} \)). Let \( \gamma_1 \) and \( \gamma_2 \) be two blocks whose LTS are:

\[
\gamma_1:\hspace{1cm} (\text{for } m \in M)
\]

\[
\gamma_2:
\]

Assume a configuration \( c \) where \( \ell_1 = \ell_2 = 0 \) and where two distinct tasks are running \( \gamma_1 \) and \( \gamma_2 \), and are in \( s_0 \) and \( s_0' \) respectively. Assume that no other task can access \( \ell_1 \), \( \ell_2 \) and \( x \). It is easy to check that, from \( c \), there is only one possible interleaving of the transitions of \( \gamma_1 \) and \( \gamma_2 \). So if \( \gamma_2 \) reaches \( s_0' \) from \( c \), then \( \gamma_1 \) must have reached \( s_0 \), and the \( x = m \) test in \( \gamma_1 \) has been fired after the \( x \leftarrow m \) assignment in \( \gamma_2 \). This achieves a rendezvous synchronisation between \( \gamma_1 \) and \( \gamma_2 \), 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 \( P \) and \( A_P \), it is easy to show that \( P \) reaches a state \( s \in S_P \) iff \( f \) is coverable in \( A_P \) 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 [fork/join](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 \( \text{count} \). 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 fork/join to take the value “\( * \)” where \( * \) represents “any non-negative value”. That is, the execution of fork/join(\( q, γ, * \)) 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 \( 
\langle CQID, \emptyset, \Gamma, \text{main, } \Sigma, (TS_{\gamma})_{\gamma \in \Gamma} \rangle 
\) that is defined as an asynchronous QDAS where the possible actions are extended by: \( \{\text{fork/join}\} \times CQID \times \Gamma \times (\mathbb{N} \cup \{*\}) \). Note that \( * \) stands for “any number”. The parameter of a fork/join action is the last value of the tuple. An EQDAS is \( * \)-free if none of its transition systems \( TS_{\gamma} \) contains a fork/join 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
is a transition \((G, d), \forkjoin(q, \gamma, p), (G', d'))\) in the transition system of an EQDAS iff \(d' = d\) and there are \(\delta = (s, \forkjoin(q, \gamma, p), s')\) \(\in \Delta\), and \(G'' \in \text{step}(\delta)(G)\) such that: \(G' = G''\) where (i) \(p = *\) implies \(n \geq 0\); (ii) \(p \neq *\) implies \(n = p\); (iii) \(G_0'' = G''\); and (iv) for all \(0 \leq i < n\), \(G_{i+1}'' = \text{letwait}(v, v_{i+1}'')(\text{enqueue}(q, \gamma))(G_i'')\) 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 \(\forkjoin(q, \gamma, p)\) action inserts a sequence of \(p\) blocks \(\gamma\) 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 \(\forkjoin(q, \gamma, p)\) action with parameter \(p = 1\) is equivalent to a synchronous dispatch of \(\gamma\) 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 \(\omega\)-Petri net (\(\omega\)-PN for short). \(\omega\)-PN transitions with an arbitrary but fix number of tokens, and extend classical Petri nets by permitting arcs to be labeled by \(\omega\). Formally, in an \(\omega\)-PN, each transition is a pair \(t = (I_t, O_t)\), where \(I_t\) and \(O_t\) are both functions from \(P \rightarrow N \cup \{\omega\}\). 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 \(\omega\)-PN [Geeraerts et al. 2013b].

We now encode the behavior of an EQDAS \(A\) into an \(\omega\)-PN \(N_{\lambda t}\), 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 \((s, \forkjoin(q, \gamma, p), s')\) 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 \(s^0_i\) and \(s^f_i\) are, as before, the places encoding respectively the initial and the final states of \(\gamma\).

Let us now explain why these constructions establish a behavioral over-approximation: Consider a transition \(\delta = (s, \forkjoin(q, \gamma, p), s')\), with \(p \neq *\), of \(A\), and consider a configuration \((G, d)\) of \(A\) such that Parikh\((G)(s^f_i) = p\) and Parikh\((G)(s) = 1\), and Parikh\((G)(s') = 0\), i.e., there is one task in location \(s\), no task in \(s'\) and \(p\) tasks already in location \(s^f_i\). Then executing \(\delta\) from \((G, d)\) yields a configuration \((G', d')\) where \(\text{Parikh}(G')(s) = 1\) and \(\text{Parikh}(G')(s') = 0\), and the block that executes the \(\forkjoin\) will
remain blocked in location $s$ until the $p$ forked tasks, which each execute $\gamma$, terminate. This is not the case in the $\omega$-$\text{PN}$ $N^*_A$: the marking $m$ corresponding to $(G, d)$ satisfies $m(s) = 1$, $m(s') = p$ and $m(s') = 0$. From $m$, one can fire the $t_{\text{fork}}$ transition corresponding to $\delta$, and immediately afterwards the $t_{\text{join}}$ transition, because $s'\delta$ still contains $p$ tokens. Thus, the join can occur even if the $p$ tokens created by $t_{\text{fork}}$ have not reached $s'\delta$ yet. Still, the $\omega$-$\text{PN}$ 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 $\omega$-$\text{PN}$ $N^*_A$ over-approximates the original EQDAS $A$:

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

Over-approximations are useful for program verification when proving safety or termination properties: if the $\omega$-$\text{PN}$ 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
remember the state of the socket by the places in 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 \( m(p_0^0) \geq 1 \) 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 increment cannot modify the count variable at the same time (a property that we formalize as the reachability of a marking \( m \) with \( m(p_0^i) \geq 2 \)).

Yet, this refinement of the over-approximation does not allow to prove property (2) (formalized as the reachability of a marking \( m \) with \( m(p_0^0) \geq 1 \) and \( m(p_0^1) \geq 1 \)) 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 \( D = \{\text{listen,treat,close,} \} \cup \{\varepsilon\}^k \cup \{\text{overflow}\} \), 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 \( \text{overflow} \).

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 \( \text{overflow} \) 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


Received : revised : accepted

25