Form Methods Syst Des (2013) 42:146–174
DOI 10.1007/s10703-012-0170-4

Code aware resource management

Krishnendu Chatterjee · Luca de Alfaro · Marco Faella · Rupak Majumdar · Vishwanath Raman

Published online: 16 August 2012

Abstract Multithreaded programs coordinate their interaction through synchronization primitives like mutexes and semaphores, which are managed by an OS-provided resource manager. We propose algorithms for the automatic construction of *code-aware* resource managers for multithreaded embedded applications. Such managers use knowledge about the structure and resource usage (mutex and semaphore usage) of the threads to guarantee deadlock freedom and progress while managing resources in an efficient way. Our algorithms compute managers as winning strategies in certain infinite games, and produce a compact code description of these strategies. We have implemented the algorithms in the tool CYNTHESIS. Given a multithreaded program in C, the tool produces C code implementing a code-aware resource manager. We show in experiments that CYNTHESIS produces compact resource managers within a few minutes on a set of embedded benchmarks with up to 6 threads.

Keywords Scheduling · Deadlock avoidance · Code analysis

K. Chatterjee
IST Austria (Institute of Science and Technology Austria), Klosterneuburg, Austria
e-mail: krishnendu.chatterjee@ist.ac.at

L. de Alfaro
Computer Science Department, University of California, Santa Cruz, USA
e-mail: luca@soe.ucsc.edu

M. Faella
Computer Science Division, Università di Napoli "Federico II", Naples, Italy
e-mail: mfaella@na.infn.it

R. Majumdar
Max Planck Institute for Software Systems, Saarbrücken, Germany
e-mail: rupak@mpi-sws.org

V. Raman (✉)
College of Engineering, Carnegie Mellon University, Moffett Field, USA
e-mail: vishwa.raman@west.cmu.edu

1 Introduction

Embedded and reactive software is often implemented as a set of communicating and interacting threads. The threads most commonly rely on primitives such as mutexes and counting semaphores to coordinate their interaction, to ensure the atomic execution of critical code regions, and to ensure that shared data structures are correctly accessed. These mutexes and semaphores (which we collectively term *resources*) are managed independently of the application code. In this paper, we propose the automated construction of *code-aware* managers for resources. Such managers use their knowledge of the thread structure and resource usage to manage resources in an efficient and deadlock-free fashion.

The simplest resource managers, found in the implementation of just about any thread library, use the most liberal of policies: grant a resource whenever it is available. The liberality of this policy creates the possibility of deadlocks: the classical example is when thread 1 requests (and is granted) a mutex A, and thread 2 requests (and is granted) a mutex B. If the next requests are for mutex B from thread 1, and for mutex A from thread 2, deadlock ensues. Writing software that is deadlock-free under such a simple resource management policy is a difficult and error-prone task [18, 33]. Monotonic locking [31] ensures deadlock freedom, at the price of imposing additional bookkeeping on the programmer. Monotonic locking also cannot be extended to counting semaphores, where there is no notion of a particular thread "holding" a resource. Priority ceiling uses information on the set of locks used by each thread to guarantee deadlock freedom [5]. Like monotonic locking, however, priority ceiling cannot cope with counting semaphores. Furthermore, in the setting that we study in this paper, when all threads have the same priority and need to get a fair share of CPU time, priority ceiling is a most restrictive policy: it allows at most one thread to hold mutexes at any given time. Other algorithms, such as the banker's algorithm [31], rely on a manual analysis of the resources needed for given tasks, and again do not cover code with semaphores.

We present an automatic static technique to synthesize code-aware resource managers for multithreaded embedded applications that guarantee deadlock freedom while managing resources in a liberal and efficient way. Rather than synthesizing the whole scheduler, we focus on the *resource policy*, i.e., the part of the scheduler responsible for granting resources, depending on the underlying OS scheduler to resolve the remaining scheduling choices. Our formulation does not require special programmer annotations or code structures, nor any change in programming style. Hence, it is directly applicable to existing bodies of code.

To illustrate the advantages of code-aware managers, consider the threads of Fig. 1. Thread 1 and Thread 2 can lead to a deadlock under a standard, most liberal resource manager. On the other hand, the code-aware manager we construct is able to differentiate, in Thread 1, between the requests for the mutex a occurring on the *then* and *else* branches of the *if* statement (during code analysis, information about the location of resource manager calls is added to the calls themselves). When Thread 1 holds mutex a, and Thread 2 requests mutex b, the request is granted if Thread 1 is in the *else* branch, and denied otherwise. Similarly, when Thread 2 holds the mutex b, and Thread 1 requests the mutex a, the request is granted if Thread 1 is in the *else* branch, and denied otherwise. In all cases, the code-aware manager guarantees deadlock freedom while managing resources in a fair and liberal manner.

We focus on the problem of ensuring fair, deadlock-free progress of all the threads composing the embedded application. We assume that threads are correct, except possibly for their resource interaction: for instance, we do not guarantee progress if a thread holding a

```
while (1) {
  if (exp) {
    mutex_lock(a);
    mutex_lock(b);
    // critical region
    mutex_unlock(b);
    mutex_unlock(a);
  } else {
    mutex_lock(a);
    mutex_lock(c);
    // critical region
    mutex_unlock(c);
    mutex_unlock(a);
  }
}
```

(a) Thread 1

```
while (1) {
  mutex_lock(b);
  mutex_lock(a);
  // critical region
  mutex_unlock(a);
  mutex_unlock(b);
}
```

(b) Thread 2

Fig. 1 Two fragments of C code

mutex enters an infinite loop (no resource manager guarantees progress under these conditions). In other words, we focus on the resource management problem, rather than on the software verification problem.

We formulate the scheduling problem as a game between the manager and the threads, where the goal for the manager is to avoid deadlocks while ensuring that all threads make progress. A winning strategy in this game provides a code-aware manager that guarantees progress for all threads at run time. In this game, the manager has two sources of antagonism: first, there is the non-determinism of each thread (such as the *if* of Thread 1); second, the OS scheduler chooses which thread to run when more than one is ready. Treating both sources of antagonism in a purely adversarial way would lead to the conclusion that most systems are doomed to starvation. Rather, we include a detailed analysis of what kind of fairness assumptions are needed to obtain a more realistic model of the system. This analysis is not present in some recent work on code-aware schedulers [27, 28], a circumstance that prevents those schemes from addressing the problem of progress (or absence of starvation), which is a major concern in the present paper. We argue that this analysis is also necessary to extend the scope of the synthesis to address quality of service concerns.

To achieve compact, yet fair, managers, we consider winning strategies that may be *randomized*, that is, scheduling decisions may use lotteries over available moves; the strategies ensure progress and fairness with probability 1. We provide efficient algorithms that compute winning strategies from the source code in quadratic time, while accounting for scheduler and thread fairness. We then take a closer look at the interaction between the resource manager and the underlying operating system scheduler, and we show how the standard strategy obtained by solving the game can be made more efficient in a real-world resource manager. We show how the strategies can be represented compactly using BDDs, and we discuss how to implement the resource manager so that it is compact in terms of code size as well as efficient to execute at run-time.

1.1 The tool Cynthesis

We have implemented these algorithms in the tool CYNTHESIS. Our tool takes as input a multithreaded application written in C, and produces code for a custom resource manager for the application. The CYNTHESIS tool flow is illustrated in Fig. 2. First, CYNTHESIS identifies the threads composing the embedded application, and extracts from each thread

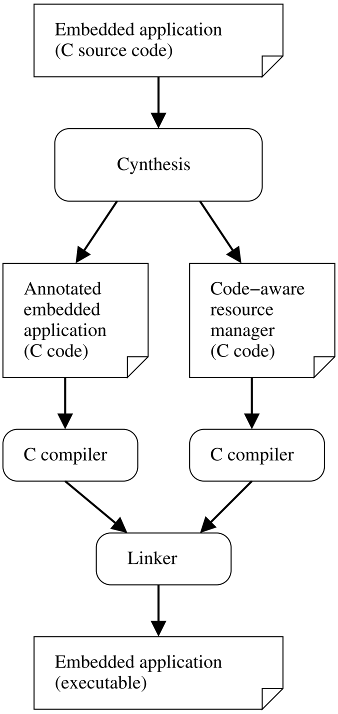


Fig. 2 CYNTHESIS tool flow: from the source code of an embedded application, to the executable application with its code-aware resource manager

a *resource interface* which summarizes the resource usage (mutexes, semaphores) of the thread. These resource interfaces are then merged into a joint interface, and game-theoretic methods are used to generate a code-aware resource manager from the joint interface; this code-aware resource manager also consists of C code. While generating the resource interfaces, CYNTHESIS annotates the code of the embedded application, so that it can communicate with the resource manager. The resulting annotated application, and resource manager, can then be compiled and linked to obtain the complete embedded application. Currently, CYNTHESIS produces code for the the eCos embedded operating system [17]; the tool can be easily retargeted to other operating systems.

We have applied the tool to a set of small multithreaded embedded applications with up to six threads. In each case, CYNTHESIS produced the custom resource manager within a few minutes, and the resource manager could be compactly represented using BDD-based data structures with a few hundred nodes. We have also applied CYNTHESIS to a larger case study, described in Sect. 5, consisting in a multi-threaded program implementing an ad-hoc network protocol for mobile robots. In this case study, CYNTHESIS correctly identified and prevented a subtle deadlock that was present in the original application.

1.2 Related work

In closely related work, [27, 28] study the synthesis of code-aware managers for Java. The focus is deadlock avoidance, and as mentioned earlier, the question of progress (absence of starvation) is not addressed.

The ongoing focus on multi-core hardware architectures has given rise to a wide array of formal techniques aimed at assisting the programmers in the difficult job of writing correctly-synchronized concurrent programs. Many of these techniques can be seen as forms of *partial synthesis*, as they "complete" a given program by filling in some critical or hard-to-get-right parts. For instance, Kuperstein et al. [29] propose a technique to automatically place a minimal number of memory fences in a concurrent program in order to guarantee a given safety property. Golan-Gueta et al. [22] show how to automatically synchronize concurrent programs which employ certain classes of dynamic data structures. Finally, program sketching [3, 34] is a general purpose partial synthesis approach that has also been applied to concurrent data structures [35]. In a recent paper, Cerny et al. [6] study the problem of lock synchronizers for concurrent data-structures in such a way that the resulting program satisfies a safety goal and is optimal w.r.t. a given performance model. As opposed to these works, we consider a stronger liveness goal and we allow each thread to retain some non-determinism, in order to over-approximate all branching and looping constructs. Our objective is to synthesize a resource manager that ensures all threads make progress as opposed to [6], that is concerned with the optimal placement of locks to meet safety and quantitative objectives.

The problem of deadlock prevention has been extensively studied in at least three different fields: databases, operating systems, and flexible manufacturing systems. In the latter field [1, 16, 19, 24, 25, 37], it is assumed that a Petri Net model is constructed by hand. In contrast, our approach and tool rely on the automated analysis of software, and we deal in detail with the issues arising from code abstraction and interaction with operating-system schedulers. Also, most of these works deal with processes that are terminating and/or deterministic. The work of [11] is amongst the first to consider the problem of synthesis given temporal logic specifications and much progress has been made since then [2, 21]. In [11], the authors describe synthesizing synchronization skeletons for concurrent programs from CTL specifications. A synchronization skeleton for a given process is a labeled graph, where nodes correspond to blocks of code that are atomic and the labels on edges correspond to the conditions under which the process can transition between nodes such that the CTL specification is satisfied for the program. If the specifications are satisfiable, then the skeletons are extracted from a finite model that satisfies the specification. Similar to this work our thread interfaces abstract each process to the level of its interactions with the resource manager, but in our case we explictly include the non-determinism introduced by the OS scheduler and the non-determinism inherent in each process due to conditional branch statements. Introducing these sources of non-determinism entail synthesizing policies for the resource manager against all possible, and hence adversarial, behaviors of the sources of non-determinism. Finally, the use of randomization to generate efficient resource managers has not been studied before in these works.

Static compiler techniques have been used in high performance thread packages to improve response time through better scheduling [38], however, the problem of resource interaction and deadlock has not been studied. Finally, deadlock detection and prevention methods from transactional databases do not apply in our setting, since our applications do not have transactional semantics and rollback.

1.3 Paper organization

In Sect. 2, we define thread resource interfaces and joint interfaces, and outline how such interfaces are extracted from the code of the embedded application. Section 3 covers the game-theoretical techniques used to generate code-aware resource managers. This section presupposes some knowledge of game theory, and may be skipped by readers interested in forming a general idea of the tool CYNTHESIS. Section 4 explains how to adapt the resource managers obtained via game-theoretical methods to the characteristics of the runtime environment of an embedded application, obtaining managers that are more efficient in practice. Finally, Sect. 5 describes the tool CYNTHESIS, as well as the examples and case studies that have been analyzed with it.

This paper is an improved and extended version of [14] with full proofs and examples that illustrate the rationale behind our synthesis objective. Moreover, we introduce finitary fairness and finitary progress objectives and argue that rather than progress under fairness assumptions as proposed in [14], the more appropriate synthesis objective should be finitary progress, under finitary fairness assumptions. The finitary progress objective, under finitary fairness assumptions, provides a bound on the number of times that a thread waiting for a resource can be bypassed. Interestingly, we show that the winning strategies we compute in [14] are also winning for the case of the new finitary progress objectives, thus strengthening the results we presented in [14].

2 Thread resource interfaces

In this section we introduce resources, thread interfaces and the systems that consist of resources and thread interfaces. Systems with semaphores may have an unbounded state space as semaphore values may grow beyond bounds. We conclude this section by showing that the problem of deciding whether or not the reachable state space of systems with semaphores is finite, is EXPSPACE-complete.

2.1 Resources

A *resource* is a non-sharable, reusable quantity. For our purposes, a resource x is an integer-valued variable together with a set of *actions* $\{w_x!, g_x?, r_x!\}$ on x. Intuitively, these actions correspond to communications between the threads that manipulate the resource and the resource manager, and have the following meaning:

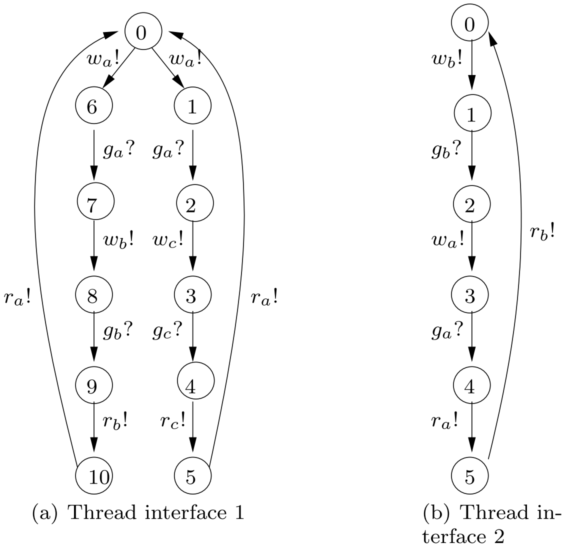
- $w_x!$: a thread requests the resource x (“want x”).
- $g_x?$: the resource manager grants the resource x to a thread (“get x”).
- $r_x!$: the thread releases the resource x (“release x”).

Given a set R of resources, the set of *actions on* R is $Acts[R] = \{w_x!, g_x?, r_x! \mid x \in R\} \cup \{\varepsilon\}$. The *output actions* over R are given by $Acts^O[R] = \{w_x!, r_x! \mid x \in R\} \cup \{\varepsilon\}$, and correspond to communication from the thread to the resource manager. In addition, we have a special action ε which is needed in Definition 3 below. The *input actions* over R are given by $Acts^I[R] = \{g_x? \mid x \in R\}$, and correspond to communication from the resource manager to the thread. We consider two types of resources: *mutexes* and (counting) *semaphores*. A mutex is a resource that takes value in $\{0, 1\}$ and starts from the initial value 1; a mutex can only be released by the same thread that acquired it (as in POSIX). A semaphore, on the other hand, can be initialized to any integer, and can be released and acquired without constraints, except that its value can never become negative.

2.2 Thread interfaces

We model the behavior of threads by *thread interfaces*. Thread interfaces model only the resource manipulation aspect of threads, and abstract out all data manipulation.

Fig. 3 The thread interfaces corresponding to the code in Fig. 1



(a) Thread interface 1

(b) Thread interface 2

Definition 1 A *thread interface* $I = (R, S, E, s^{\text{init}}, \lambda)$ consists of a set R of resources, a finite control-flow graph (S, E) with $E \subseteq S \times S$, an initial state $s^{\text{init}} \in S$, and an *action label* $\lambda : E \to Acts[R] \setminus \{\varepsilon\}$ mapping each edge to a resource action, such that

- each $w_x!$ edge leads to a state whose only outgoing edge is labeled with $g_x?$;
- each $g_x?$ edge starts from a state whose incoming edges are all labeled with $w_x!$.

Intuitively, the conditions on a thread interface guarantee that a "want" action is immediately followed by the corresponding "get" action; moreover, a "get" action has no siblings. We say that a state s is *final* if it has no successors. For $s \in S$, let $Isucc(s) = \{t \in S \mid (s, t) \in E \wedge \lambda(s, t) \in Acts^I[R]\}$ be the set of input successors of s, and let $Osucc(s) = \{t \in S \mid (s, t) \in E \wedge \lambda(s, t) \in Acts^O[R]\}$ be the set of output successors of s. We carry subscripts over to components, so that an interface I_i will consist of $(R_i, S_i, E_i, s_i^{\text{init}}, \lambda_i)$; similarly, we carry subscripts to *Isucc* and *Osucc*.

Example 1 Consider the POSIX interface for mutexes with functions `mutex_lock(x)` and `mutex_unlock(x)`. Each call `mutex_lock(x)` is represented by the pair of actions $w_x!$ and $g_x?$; a (nonblocking) call `mutex_unlock(x)` is represented by the action $r_x!$. Similarly, for a counting semaphore y, the function `sem_wait(y)` corresponds to the two actions $w_y!$ and $g_y?$, and the function `sem_post(y)` corresponds to the release action $r_y!$. For example, our tool extracts the resource interfaces of Fig. 3 from the code in Fig. 1.

2.3 Systems

2.3.1 Syntax

Given a set R of resources, a *resource valuation* is a function $\nu : R \mapsto \mathbb{N}$ mapping each resource to a natural number value. For a valuation ν and $x \in R$, we denote by $\nu[x := k]$ the valuation obtained from ν by assigning the value $k \in \mathbb{N}$ to x. A *system* is a set of resources, an initial resource valuation of the resources, and a tuple of (a fixed number of) thread interfaces.

Definition 2 A *system* is a tuple $\mathcal{I} = (R, \nu^0, (I_1, \ldots, I_n))$, consisting of a set R of resources, a mapping $\nu^0 : R \mapsto \mathbb{N}$ assigning an initial value to each resource, and of $n > 0$ thread interfaces $I_1, \ldots, I_n$. We require that $R_i \subseteq R$, for $1 \leq i \leq n$, and that if $x \in R$ is a mutex, $\nu^0(x) = 1$.

2.3.2 Semantics

Given a system, we can define its semantics using a *joint interface*, obtained by constructing the product of the interfaces, annotated with the values of the resources at the states. The joint interface models the execution of a multithreaded system on a single processor.

Definition 3 Given a system $\mathcal{I} = (R, \nu^0, (I_1, \ldots, I_n))$, its *joint interface* is a tuple $M_{\mathcal{I}} = (R, S, E, s^{\text{init}}, \lambda, \theta)$, where R is as in $\mathcal{I}$, and:

- $S = (\prod_i S_i) \times (R \mapsto \mathbb{N})$;
- $s^{\text{init}} = (s_1^{\text{init}}, \ldots, s_n^{\text{init}}, \nu^0)$;
- $E \subseteq S \times S$, and $\lambda : E \mapsto Acts[R]$, $\theta : E \mapsto \{0, \ldots, n\}$ are defined as follows. Let $s = (s_1, \ldots, s_n, \nu) \in S$; we have $(s, t) \in E$, $\lambda(s, t) = \alpha$, and $\theta(s, t) = i$ iff there is $s_i' \in S_i$ such that $(s_i, s_i') \in E_i$, $\lambda_i(s_i, s_i') = \alpha$, and for $t = (s_1, \ldots, s_{i-1}, s_i', s_{i+1}, \ldots, s_n, \nu')$ we have:

  [resource grant] if $\alpha = g_x?$, then $\nu(x) > 0$ and $\nu' = \nu[x := \nu(x) - 1]$;
  [resource request] if $\alpha = w_x!$, then $\nu' = \nu$; and
  [resource release] if $\alpha = r_x!$, then $\nu' = \nu[x := \nu(x) + 1]$; further, if x is a mutex, then $\nu(x) = 0$.

Moreover, let s be a state that has no successors according to the above rules. Then, we add a self-loop $(s, s) \in E$ and we set $\lambda(s, s) = \varepsilon$ and $\theta(s, s) = 0$.

Let $s \in S$ and $s = (s_1, \ldots, s_n, \nu)$; for all $i = 1, \ldots, n$, we set $loc_i(s) = s_i$. We let *Osucc*, *Isucc* refer to $M_{\mathcal{I}}$, and for $1 \leq i \leq n$, we let $Osucc_i$, $Isucc_i$ refer to I_i.

In $M_{\mathcal{I}}$, edges labeled with the special action ε are a technical addition, used to ensure that all finite paths can be extended to infinite ones. The portion of the joint interface $M_{\mathcal{I}}$ that is reachable from its initial state s^{init} may not be finite, as the value of resources could grow beyond bounds. Of course, if all resources are mutexes (which take values 0 and 1), the state space is finite. In general, we show that the problem is EXPSPACE-complete in the following theorem.

Theorem 1 *Let $M_{\mathcal{I}} = (R, S, E, s^{\text{init}}, \lambda, \theta)$ be the joint interface of a system $\mathcal{I}$. The problem of deciding whether the portion of S that is reachable in (S, E) is finite is EXPSPACE-complete.*

Proof The EXPSPACE upper bound follows from the Karp-Miller Coverability Tree algorithm [26] for Petri Nets that checks for boundedness. By modeling the tokens of the Petri Nets as resource values, we have a reduction of the boundedness problem of Petri Nets to our problem. This gives us the EXPSPACE lower bound. □

In the following, we only consider systems $\mathcal{I}$ such that the reachable portion of $M_{\mathcal{I}}$ is finite. In our tool CYNTHESIS we avoid solving the question of whether the portion of the joint interface reachable from the initial state is finite. Rather, we simply take as input the maximum value to consider for any semaphore; this value is usually well known to the programmer. If we find a reachable state where the value of a semaphore is greater than this maximum, we stop and report the problem.

3 The scheduling game

In this section, unless otherwise noted, we consider a fixed system $\mathcal{I} = (R, \nu^0, (I_1, \ldots, I_n))$, which gives rise to a joint interface $M_{\mathcal{I}} = (R, S, E, s^{\text{init}}, \lambda, \theta)$.

A joint interface evolves by the interaction between three entities: the threads, the resource manager, and the scheduler. From a given state, if there are any outgoing edges labeled by input actions, the resource manager can choose to follow one of them: this corresponds to granting a resource to a thread. Once the input edge has been followed (and the resource granted), the resource manager still retains control at the destination state. From a given state, if there are any edges labeled by output actions that leave the state, the resource manager can also decide to return control to the threads. At this point, which output action occurs next depends on two factors. The underlying operating-system scheduler, using its own policy (such as time-sharing with round robin), selects which of the ready threads execute on the CPU. In addition, each thread has its own internal non-determinism, which determines which output action the thread generates next. Thus, we identify three types of non-determinism in the joint interface.

1. *Resource manager non-determinism,* due to the resource manager choosing an input edge, or choosing to wait for an output action.
2. *Inter-thread non-determinism,* due to the operating-system scheduler resolving thread interleaving.
3. *Intra-thread non-determinism,* which determines which of several possible output actions a thread will do.

Resource manager The goal is to synthesize a resource manager that ensures that all threads make progress, unless they terminate. In order to define the goal, we introduce the following predicates over edges of $M_{\mathcal{I}}$: for $1 \leq i \leq n$, the predicate $\mathit{progress}_i$ is true over an edge $(s, t) \in E$ if $\theta(s, t) = i$, and the predicate final_i is true over an edge $(s, t) \in E$ if the thread i is in a final state in s. Notice that for all thread interfaces, the set of final states is absorbing. Therefore, final_i being true over an edge $(s, t) \in E$, implies that it remains true along all paths that originate at s; this means that $\Box \mathit{final}_i$ holds on all paths that originate at s. Using temporal logic notation, the goal can therefore be written as a *generalized Büchi* condition over the edges:

$$\phi_{\mathcal{I}}^{\mathit{goal}} = \bigwedge_{i=1}^{n} \Box \Diamond (\mathit{progress}_i \vee \mathit{final}_i).$$

Our aim is to synthesize a resource manager that satisfies the goal $\phi_{\mathcal{I}}^{\mathit{goal}}$. We first describe the two sources of non-determinism that the resource manager plays against.

Inter-thread non-determinism This non-determinism is due to the scheduler. If there are two or more threads that are waiting to issue output actions, then which thread gets to issue an output action depends on the underlying OS scheduler. If two threads want to get a resource, which thread gets to call the OS primitive to acquire the resource is decided by the

```
while (1) {
  if (exp) {
    mutex_lock(a);
    x = 1;
    // critical region
    mutex_unlock(a);
  } else {
    mutex_lock(b);
    x = 1;
    mutex_lock(c);
    // critical region
    mutex_unlock(b);
    while (x != 2)
      ;
    mutex_unlock(c);
  }
}
```

(a) Thread 1

```
while (1) {
  if (!exp) {
    mutex_lock(b);
    x = 2;
    mutex_lock(c);
    // critical region
    mutex_unlock(b);
    while (x != 1)
      ;
    mutex_unlock(c);
  } else {
    mutex_lock(d);
    x = 2;
    // critical region
    mutex_unlock(d);
  }
}
```

(b) Thread 2

```
while (1) {
  mutex_lock(b);
  mutex_lock(c);
  // critical region
  mutex_unlock(c);
  mutex_unlock(b);
}
```

(c) Thread 3

Fig. 4 A system of three threads to illustrate the assumptions on the sources of non-determinism and the goal of the resource manager

scheduler. Similarly, if a thread wants to release a resource, whether or not it gets to release the resource again depends on the scheduler.

Intra-thread non-determinism This non-determinism has two origins. The first is the *environment*: often, the behavior of a thread in an embedded system reacts to inputs (input timings, or input values) received from the environment. The second is *abstraction*: our thread interface is an abstraction of the actual thread behavior that disregards variable values. In particular, the outcome of control-flow statements such as loop tests, and if-then-else, is modeled as intra-thread non-determinism. Assuming that intra-thread non-determinism is resolved in an arbitrary way may easily lead to declaring the manager synthesis problem to be infeasible.[1] In fact, whenever a thread can execute a loop while holding a resource, the arbitrary resolution of intra-thread non-determinism introduces the possibility that the loop never terminates.

The synthesis objective We first show that the automatic synthesis of a resource manager given the goal $\phi_{\mathcal{I}}^{goal}$ will fail against arbitrary resolution of the inter-thread and intra-thread non-determinism. We then use fairness assumptions on inter-thread and intra-thread non-determinism and derive a synthesis objective that satisfies $\phi_{\mathcal{I}}^{goal}$, given these fairness assumptions. Consider the system of three threads in Fig. 4. If we assume that the scheduler never schedules Thread 3, then the $w_b!$ action from Thread 3 never takes place. In this case, irrespective of the resource manager policy, $\phi_{\mathcal{I}}^{goal}$ is not satisfied. We need to restrict the inter-thread non-determinism and we do so by placing a fairness assumption on the underlying

[1] Recall that our goal is to schedule *correct* software, rather than to perform software verification.

operating system scheduler: more precisely, if a thread is infinitely often ready to execute, it will make progress infinitely often. We introduce a predicate $ready_i$, for $1 \leq i \leq n$, which is true over an edge $(s,t) \in E$ iff (i) (s,t) is labeled with an output action, and (ii) there is $(s,t') \in E$ with $\theta(s,t') = i$. Intuitively, (i) means that the resource manager decided to let the scheduler schedule some thread, and (ii) means that thread i was among the threads that could have generated the next output. With this notation, the fairness assumption on the scheduler is:

$$\varphi_{\mathcal{I}}^{inter} = \bigwedge_{i=1}^{n} (\Box \diamond ready_i \Rightarrow \Box \diamond progress_i).$$

We now show that even if we assume that inter-thread non-determinism is resolved satisfying $\varphi_{\mathcal{I}}^{inter}$, the goal of the resource manager can still be violated: more precisely, the resource manager cannot ensure that $\varphi_{\mathcal{I}}^{inter} \Rightarrow \phi_{\mathcal{I}}^{goal}$. Assume that the conditional expression *exp* in Thread 1 and Thread 2 is always false. Thread 1 releases resource b and waits till Thread 2 acquires it before releasing resource c. Similarly, Thread 2 releases resource b and waits till Thread 1 acquires it before releasing resource c. There is no way to resolve intra-thread non-determinism in this case to ensure that Thread 3 makes progress for any policy followed by the resource manager; either Thread 3 is starved or deadlock ensues. Notice that even if we assume that the scheduler is fair and that Thread 3 is scheduled infinitely often it cannot make progress because either the system is in a deadlock or one of Thread 1 or Thread 2 always hold resource b, thus starving Thread 3. Therefore the resource manager cannot ensure that $\varphi_{\mathcal{I}}^{inter} \Rightarrow \phi_{\mathcal{I}}^{goal}$.

We need to restrict intra-thread non-determinism and we do so by placing a fairness constraint on intra-thread non-determinism: if each choice is presented infinitely often, then each choice outcome is followed infinitely often. For all threads $1 \leq i \leq n$, all $u, v \in S_i$, and all $(s,t) \in E$, we introduce the predicates $from_i^u(s,t) \overset{\text{def}}{=} (loc_i(s) = u)$ and $take_i^{u,v}(s,t) \overset{\text{def}}{=} ((loc_i(s) = u) \wedge (loc_i(t) = v))$. The fairness assumption for intra-thread non-determinism can then be written as

$$\varphi_{\mathcal{I}}^{intra} = \bigwedge_{i=1}^{n} \bigwedge_{u \in S_i} \bigwedge_{v \in Osucc_i(u)} \bigl(\Box \diamond from_i^u \Rightarrow \Box \diamond take_i^{u,v}\bigr).$$

This entails that the conditional expression *exp* takes both values infinitely often. With this assumption, Thread 1 (Thread 2) will enter the *then* (*else*) branch of the conditional statement infinitely often. This implies that either Thread 1 is holding resource a or Thread 2 is holding resource d infinitely often. The resource manager strategy is as follows:

- If both Thread 1 and Thread 2 are holding resources a and d, then a winning strategy for the resource manager would be to assign resources b and c to Thread 3.
- If Thread 1 is holding resource a and Thread 2 is holding resources b and c. Then a winning strategy for the resource manager would be to wait till Thread 2 releases both b and c and then allocate these resources to Thread 3, thus ensuring that Thread 3 enters its critical region.
- If Thread 2 is holding resource d and Thread 1 is holding resources b and c, then a strategy similar to the one above will ensure that Thread 3 enters its critical region.

Therefore, the objective for resource manager synthesis requires fairness assumptions on both inter-thread and intra-thread non-determinism. Formally, the objective for the resource manager is:

$$\phi^2 = \bigl(\varphi_{\mathcal{I}}^{inter} \wedge \varphi_{\mathcal{I}}^{intra}\bigr) \Rightarrow \phi_{\mathcal{I}}^{goal}. \tag{1}$$

Finitary progress The progress objective $\phi_{\mathcal{I}}^{goal}$ states that each thread that is ready makes progress eventually, but the "eventual" time to make progress can be unbounded. A stronger and more desirable notion of progress is that of finitary progress, which states that each ready thread makes progress within bounded time. Let $\sigma \in S^{\omega}$ be an infinite path that can be taken in a joint interface $M_{\mathcal{I}}$; we take $\sigma[j]$ for $j = (0, 1, 2, \ldots)$ as the sequence of states in the path σ. Let $progress_i(s, t)$ be the predicate that is true for an edge (s, t) if $\theta(s, t) = i$, and the predicate $final_i$ is true over an edge $(s, t) \in E$ if the thread i is in a final state in s. The finitary progress goal $\phi_{\mathcal{I},f}^{goal}$ can be defined as follows:

$$\phi_{\mathcal{I},f}^{goal} = \bigcap_{i=1}^{n} \bigl(\diamond final_i \cup \bigl\{\sigma \in S^{\omega} \mid \exists b \in \mathbb{N}, \forall j \geq 0, \exists l \leq j,$$
$$\bigl(progress_i\bigl(\sigma[l], \sigma[l+1]\bigr) \wedge \bigl(j < l \leq (j + b)\bigr)\bigr)\bigr\}\bigr).$$

Intuitively, the winning set of paths for the resource manager is the set of paths such that in each path for every thread i, $progress_i(s, t)$ is true over edges that are never more than b apart. We now show that the fairness assumption on inter-thread and intra-thread non-determinism is not sufficient to ensure finitary progress; we need finitary fairness assumptions on the sources of non-determinism.

Consider again the example in Fig. 4. From our earlier analysis of the example, the resource manager can give resources b and c to Thread 3 only when either Thread 1 is in its *then* branch or Thread 2 is in its *else* branch. As long as Thread 1 and Thread 2 are in their *else* and *then* branches respectively, the resource manager does not have a strategy to ensure that Thread 3 enters its critical region. A fair strategy to resolve intra-thread non-determinism is as follows. The strategy is played in rounds. In round i, Thread 1 and Thread 2 collude such that Thread 1 is in the *else* branch of its conditional statement and Thread 2 is in the *then* branch of its conditional statement for at least i executions of the while loop. Thread 1 then enters its *then* branch or Thread 2 enters its *else* branch once before proceeding to round $i + 1$. For example, let the conditional expression exp be $power\_of\_2(y)$ where y is a variable shared by Thread 1 and Thread 2. The variable y is initially 0 and is incremented by 1 in Thread 1 each time the while loop executes. The function $power\_of\_2(y)$ returns 1 if y is a power of 2 and 0 otherwise. For any bound $\beta > 0$, there exists a $y > 0$ with $2^{y-1} \leq \beta < 2^{y}$ such that the resource manager has no strategy to allocate resources b and c to Thread 3 for $2^{y} > \beta$ executions of the loop in Thread 1 and Thread 2. It follows that $\varphi_{\mathcal{I}}^{inter} \wedge \varphi_{\mathcal{I}}^{intra} \Rightarrow \phi_{\mathcal{I},f}^{goal}$ fails. On the other hand, if Thread 1 and Thread 2 satisfy the stronger notion of finitary fairness, where both branches of the conditional statement will be executed within a bound $\beta > 0$, then as soon as Thread 1 enters its *then* branch or Thread 2 enters its *else* branch, the resource manager has a strategy to allocate b and c to Thread 3 and ensure that Thread 3 makes progress within bound β thus satisfying its goal $\phi_{\mathcal{I},f}^{goal}$. We now formulate the finitary fairness assumption on intra-thread non-determinism as:

$$\varphi_{\mathcal{I},f}^{intra} = \bigcap_{i=1}^{n} \bigl\{\sigma \in S^{\omega} \mid \exists \beta \in \mathbb{N}, \forall j \geq 0, \forall u \in S_i, \forall v \in Osucc_i(u), \exists l \in \mathbb{N},$$
$$from_i^{u}\bigl(\sigma[j], \sigma[j+1]\bigr) \Rightarrow take_i^{u,v}\bigl(\sigma[l], \sigma[l+1]\bigr) \wedge \bigl(j < l \leq (j + \beta)\bigr)\bigr\}.$$

Intuitively, the finitary assumption $\varphi_{\mathcal{I},f}$ is the set of paths such that in each path, if a thread visits a state where it has multiple output successors, then each output successor can be ignored at most a bounded number of times. A similar definition applies to the finitary fairness assumption $\varphi^{inter}_{\mathcal{I},f}$ on inter-thread non-determinism. The amended objective for the automatic synthesis of resource managers is then:

$$\phi^2_f = \left(\varphi^{inter}_{\mathcal{I},f} \wedge \varphi^{intra}_{\mathcal{I},f}\right) \Rightarrow \phi^{goal}_{\mathcal{I},f}. \tag{2}$$

3.1 Stochastic games

We base the synthesis of the resource manager on *stochastic games*. As we will see in detail later, we use probabilities both to approximate the above types of non-determinism, and to be able to generate manager strategies that are memoryless, but that may require randomization [8]. Given a finite set A, we denote by $\mathrm{Dist}(A)$ the set of probability distributions over A. For $d \in \mathrm{Dist}(A)$ we let $\mathrm{Supp}(d) = \{a \in A \mid d(a) > 0\}$. Given $a \in A$ we denote by $\delta(a) \in \mathrm{Dist}(A)$ the probability distribution that associates probability 1 with a, and 0 to all other elements of A. We also denote by $\mathit{Uniform}(A)$ the probability distribution that associates probability $1/|A|$ to every element of A.

Definition 4 A *two-player game* structure $G = (S, \mathit{Moves}, \Gamma_1, \Gamma_2, \tau)$ consists of a set of states S, of a set of moves Moves, of two mappings $\Gamma_1, \Gamma_2 : S \mapsto 2^{\mathit{Moves}} \setminus \emptyset$ associating to each state s and player $i \in \{1, 2\}$ the set of moves $\Gamma_i(s)$ that player i can play at s, a (probabilistic) destination function $\tau : S \times \mathit{Moves}^2 \mapsto \mathrm{Dist}(S)$, which associates with each $s \in S$ and $m_1 \in \Gamma_1(s)$, $m_2 \in \Gamma_2(s)$, a probability distribution $\tau(s, m_1, m_2)$ over the successor state.

For $i \in \{1, 2\}$, we say that G is an *i-Markov decision process* (i-MDP) [15] if $|\Gamma_{3-i}(s)| = 1$ at all $s \in S$; 1-MDPs are also called simply MDPs. A *strategy* for player $i \in \{1, 2\}$ in a game $G = (S, \mathit{Moves}, \Gamma_1, \Gamma_2, \tau)$ is a mapping $\pi_i : S^+ \mapsto \mathrm{Dist}(\mathit{Moves})$, such that for all $\sigma \in S^*$ and $s \in S$, we have $\pi_i(\sigma s)(m) > 0$ implies $m \in \Gamma_i(s)$. We denote by Π_1, Π_2 the set of strategies for players 1 and 2 respectively. Once the strategies π_1 and π_2 are fixed, the game is reduced to an ordinary stochastic process, and the probabilities of all measurable events (which include all ω-regular properties [36]) are defined (see e.g. [20]). A the winning condition φ is a measurable subset of S^ω. We say that a state $s \in S$ is *winning* if there is $\pi_1 \in \Pi_1$ such that, for all $\pi_2 \in \Pi_2$, we have $\mathrm{Pr}^{\pi_1,\pi_2}_s(\varphi) = 1$. As we use randomized strategies, winning with probability 1 is the natural notion of winning. Given a game structure G and a winning condition φ we denote by $\mathit{Win}(G, \varphi)$ the set of winning states. A *winning strategy* is a strategy that wins from all winning states, that is, a strategy $\pi_1 \in \Pi_1$ such that, for all $s \in \mathit{Win}(G, \varphi)$ and all $\pi_2 \in \Pi_2$, we have $\mathrm{Pr}^{\pi_1,\pi_2}_s(\varphi) = 1$. The *size* of a game is defined by $|G| = \sum_{s\in S} \sum_{m_1\in\Gamma_1(s)} \sum_{m_2\in\Gamma_2(s)} |\mathrm{Supp}(\tau(s, m_1, m_2))|$.

3.2 The scheduling game

Since our aim is to derive strategies that resolve resource manager non-determinism, we formulate the resource manager synthesis problem as a game played on the joint interface by the resource manager against a team consisting of the threads and the scheduler. Again, unless otherwise noted, we refer to a system $\mathcal{I} = (R, \nu^0, (I_1, \ldots, I_n))$ which gives rise to a joint interface $M_{\mathcal{I}} = (R, S, E, s^{\mathrm{init}}, \lambda, \theta)$.

Definition 5 Given a game structure G corresponding to a system $\mathcal{I}$, depending on whether the objective is progress as defined in (1) or finitary progress as defined in (2), we get two versions of the *scheduling game*. The scheduling game for progress is defined as the tuple

$G^2 = (G, \phi^2)$, where ϕ^2 corresponds to the objective (1). The scheduling game for finitary progress is defined as the tuple $G^2_f = (G, \phi^2_f)$, where ϕ^2_f corresponds to the objective (2). In a scheduling game, the sets of moves for player 1 (representing the resource manager) and player 2 (representing the inter and intra-thread non-determinism) are as follows, for all $s \in S$:

- If $\mathit{Osucc}(s) \neq \emptyset$, then $\Gamma_1(s) = \mathit{Isucc}(s) \cup \{\bot\}$ and $\Gamma_2(s) = \mathit{Osucc}(s)$.
- If $\mathit{Osucc}(s) = \emptyset$, then $\Gamma_1(s) = \mathit{Isucc}(s)$ and $\Gamma_2(s) = \{\bot\}$.

The destination function is given by the following rules, where $*$ represents a wild-card, and $s \in S$:

- For $t \in \mathit{Isucc}(s)$, we have $\tau(s, t, *) = \delta(t)$;
- for $t \in \mathit{Osucc}(s)$, we have $\tau(s, \bot, t) = \delta(t)$.

The manager synthesis problem can thus be phrased as the problem of finding a winning strategy in G^2_f. We say that the system is *schedulable* if $s^{\mathrm{init}} \in \mathit{Win}(G, \phi^2_f)$. The winning condition ϕ^2_f has a finitary Streett assumption implying a finitary liveness guarantee. For such winning conditions, finite memory winning strategies exist for player 1, the resource manager, from the result of [7].

3.3 Theoretical solution of the scheduling game

In this section we present theoretical solutions for computing winning strategies in G^2 and G^2_f. We first note that the objectives in G^2 and G^2_f are different. In the finitary progress objective ϕ^2_f, the assumption $\varphi^{\mathit{inter}}_{\mathcal{I},f} \wedge \varphi^{\mathit{intra}}_{\mathcal{I},f}$ is stronger than the assumption $\varphi^{\mathit{inter}}_{\mathcal{I}} \wedge \varphi^{\mathit{intra}}_{\mathcal{I}}$ in objective ϕ^2 but the guarantee $\phi^{\mathit{goal}}_{\mathcal{I},f}$ in ϕ^2_f is also stronger than the guarantee $\phi^{\mathit{goal}}_{\mathcal{I}}$ in ϕ^2. Thus in general there is no relation between the objective ϕ^2 and ϕ^2_f. In the following theorem we show that in the special case of scheduling games that we consider, the set of winning states in G^2 and G^2_f are the same and that further, a winning strategy for G^2 remains winning for G^2_f and vice-versa.

Theorem 2 *For all scheduling game structures G, given objectives ϕ^2 and ϕ^2_f, the following assertions hold*:

1. $\mathit{Win}(G, \phi^2) = \mathit{Win}(G, \phi^2_f)$.
2. *If $\pi_1 \in \Pi_1$ is a winning strategy in G^2, then it is also winning in G^2_f and vice-versa.*

Proof We prove assertion (2) and assertion (1) is an easy consequence of the proof.

– Assume that given a scheduling game G and the objective ϕ^2 the resource manager has a winning strategy. Since ϕ^2 is an ω-regular objective, it follows from [23] that finite memory winning strategies exist for the objective ϕ^2 (the finite memory winning strategy implements the latest appearance record data structure required for winning in ω-regular games). Fix such a finite memory winning strategy π_1. Given the strategy π_1, the game reduces to the special class of games, where there is only one player (the opponent). Assume towards a contradiction that the opponent can falsify ϕ^2_f: it follows from the results of [7] that the opponent then has a finite memory strategy to do so. Fix such a finite memory strategy π_2. Consider the unique play arising from π_1 and π_2: since π_2 is a witness for violating ϕ^2_f against π_1 it follows that the play does not satisfy ϕ^2_f. Since for finite deterministic systems (both player strategies are fixed), ϕ^2 and ϕ^2_f coincide, it follows that π_2 is a witness for violating ϕ^2. This contradicts our assumption that π_1 is winning for ϕ^2.

– In the other direction, consider a scheduling game G and objective ϕ^2_f. We note that for a scheduling game, the resource manager can never violate the objectives $\varphi^{inter}_{\mathcal{I},f}$ and $\varphi^{intra}_{\mathcal{I},f}$; in general, violating $\varphi^{inter}_{\mathcal{I},f} \wedge \varphi^{intra}_{\mathcal{I},f}$ may require infinite memory. Therefore, given that in a scheduling game player 1 cannot violate $\varphi^{inter}_{\mathcal{I},f}$ and $\varphi^{intra}_{\mathcal{I},f}$, and can satisfy ϕ^2_f, it follows that there is a finite memory winning strategy π_1 for ϕ^2_f. We argue that π_1 is winning for ϕ^2 as well. Assume towards a contradiction that π_1 is not winning for ϕ^2. Then there is a counter strategy for the opponent to violate ϕ^2. Since ϕ^2 is ω-regular, there is a finite memory witness strategy π_2 that violates ϕ^2. As above, since π_1 and π_2 are both finite memory, if ϕ^2 is violated, then so is ϕ^2_f. This contradicts our assumption that π_1 is winning for ϕ^2_f. This completes the proof.

The desired result follows. □

Given a game structure G with $|S|$ states and $|E|$ edges we have the following complexity results:

1. *The algorithm for fairness*. Given a fairness objective φ with d fairness constraints, the algorithm to compute the winning set $Win(G, \varphi)$ has time complexity $\boldsymbol{O}(|E| \cdot |S|^d \cdot d!)$ [32]. The algorithm is a classical recursive algorithm to solve games with fairness objectives.
2. *The algorithm for finitary fairness*. Given a finitary fairness objective φ_f with d finitary fairness constraints, the algorithm to compute the winning set $Win(G, \varphi_f)$ has time complexity $\boldsymbol{O}(2^d \cdot |E|^2 \cdot |S|)$ [10]. The key intuition to obtain the algorithm is a reduction to a game of size 2^d times the size of the original game structure with a generalized Büchi objective.

For a scheduling game structure $G = (S, Moves, \Gamma_1, \Gamma_2, \tau)$, we have $|E| \le |G|$, and $|S| \le |G|$, where $|G|$ is the size of the game. If there are n threads and at most m conditional branches in each thread then there are n fairness assumptions on inter-thread nondeterminism and $2.n.m$ fairness assumptions on intra-thread non-determinism. Further, for the special case of scheduling games, by Theorem 2 we have $Win(G, \phi^2) = Win(G, \phi^2_f)$. Since the algorithm for solving finitary fairness has better time complexity as compared to fairness, choosing the algorithm to compute finitary fairness with objective ϕ^2_f for a scheduling game G, we get the following complexity result.

Theorem 3 *Given a scheduling game G with size $|G|$, n threads and at most m conditional branches in each thread, computing $Win(G, \phi^2_f)$ has time complexity $\boldsymbol{O}(2^{2\cdot n\cdot m+n} \cdot |G|^3)$.*

3.4 Practical solution of the scheduling game

Theorem 2 shows that the winning strategies in games G^2 and G^2_f are identical. We can therefore compute a winning strategy in G^2_f with an algorithm that can solve finitary Streett games with complexity given in Theorem 3. Instead, we show that we can exploit the special structure of the joint interface and solve the synthesis problem in a more efficient way, consisting of two steps. We first consider two simplified versions of G^2:

1. A game $G^{2.5}$, resulting from resolving all intra-thread non-determinism in G^2 in a purely randomized fashion.

2. An MDP $G^{1.5}$, resulting from resolving both the intra-thread and the inter-thread nondeterminism in G^2 in a purely randomized fashion.

Given G^2, we show that we can construct in quadratic time in $|G|$ a winning strategy for the MDP $G^{1.5}$ which is also a winning strategy of the game $G^{2.5}$. We show that this winning strategy, under many cases of practical importance, is also a winning strategy for the original game G^2, and hence by Theorem 2 winning in G^2_f. In all cases, we show that it is possible to check efficiently whether the strategy for game $G^{2.5}$ also works for G^2—and in our experience, this has always been the case in the examples we have studied so far.

Definition 6 Given a game structure $G = (S, \mathit{Moves}, \Gamma_1, \Gamma_2, \tau)$ and the scheduling game $G^2 = (G, \varphi_{\mathcal{I}})$, the games $G^{2.5} = (G', \phi^{2.5})$ and $G^{1.5} = (G'', \phi^{1.5})$ are obtained as follows. We take $G' = (S, \mathit{Moves}', \Gamma_1, \Gamma'_2, \tau')$ and $G'' = (S, \mathit{Moves}, \Gamma_1, \Gamma''_2, \tau'')$ We have $\mathit{Moves}' = \mathit{Moves} \cup \{1, \dots, n\}$, $\phi^{2.5} = \varphi^{\mathit{inter}}_{\mathcal{I}} \Rightarrow \phi^{\mathit{goal}}_{\mathcal{I}}$, and $\phi^{1.5} = \phi^{\mathit{goal}}_{\mathcal{I}}$. The functions Γ'_2, τ' and Γ''_2, τ'' coincide with Γ_2, τ, except that:

- For all $s \in S$ such that $|\mathit{Osucc}(s)| > 1$, we let $\Gamma'_2(s) = \{i \mid \exists t \in \Gamma_2(s) \,.\, \theta(s,t) = i\}$, and for $i \in \Gamma'_2(s)$, we let $\tau'(s, \bot, i) = \mathit{Uniform}(\{t \in \Gamma_2(s) \mid \theta(s,t) = i\})$.
- For all $s \in S$, we let $\Gamma''_2 = \{\bot\}$, and we let $\tau''(s, \bot, \bot) = \mathit{Uniform}(\mathit{Osucc}(s))$.

Given $G^{2.5}$ and $G^{1.5}$, let $G^{2.5}_f = (G', \phi^{2.5}_f)$ and $G^{1.5}_f = (G'', \phi^{1.5}_f)$ be the corresponding simplified finitary versions of G^2_f, where for the finitary objectives we require the expected number of steps to visit each winning state to be bounded. First, we show how to construct the most liberal winning strategy for game $G^{1.5}$; informally, this is the strategy that, among the winning ones, plays with positive probability the largest possible sets of moves. We then prove that a winning strategy in $G^{1.5}$ is also winning in $G^{1.5}_f$.

A memoryless strategy $\pi \in \Pi_1$ gives rise to a graph (S, E_π), where $E_\pi = \{(s,t) \mid \pi(s)(t) > 0 \text{ or } \pi(s)(\bot) > 0 \text{ and } \lambda(s,t) \in \mathit{Acts}^O[R]\}$. A *maximal end component* (MEC) of $G^{1.5}$ is a maximal subgraph (C, F) of (S, E) such that: there is a memoryless strategy π such that C is closed, with no outgoing edge, and is a strongly connected component of (S, E_π), and such that $F = \{(s,t) \in E_\pi \mid s \in C\}$ [12]. We say that thread k is *finished* in a state s if $\mathit{loc}_k(s)$ is final in I_k. Notice that if a thread k is finished at some state of a MEC, it is finished at all states of the MEC. We say that a MEC (C, F) is *fair* iff, for every thread $1 \le k \le n$, either k is finished in C, or there is $(s,t) \in F$ with $\theta(s,t) = k$. Let W be the union of all sets of states belonging to fair end components. It can be shown that a state is winning in $G^{1.5}$ iff it can reach W with probability 1 [8]; we denote by $\mathit{Win}(G^{1.5})$, the set $\mathit{Win}(G'', \phi^{1.5})$ of winning states of game $G^{1.5}$. By the results of [12, 13], this set can be computed in time quadratic in $|G|$.

The *most liberal winning strategy* π^* for $G^{1.5}$ is the strategy that selects uniformly at random among moves of player 1 that lead only to winning states. Precisely, for $s \in \mathit{Win}(G'', \phi^{1.5})$, we let $\pi^*(s) = \mathit{Uniform}(\{m \in \Gamma_1(s) \mid \forall t \in S.(\tau''(s, m, \bot)(t) > 0 \Rightarrow t \in \mathit{Win}(G^{1.5}))\})$. π^* is arbitrarily defined on states $s \in S \setminus \mathit{Win}(G'', \phi^{1.5})$.

Theorem 4 *For all scheduling games the following assertions hold,*

1. *the strategy π^* is winning in $G^{1.5}$,*
2. *the strategy π^* is winning in $G^{1.5}_f$ and*
3. *π^* can be computed in time $\boldsymbol{O}(|G|^2)$.*

Proof Notice that in $G^{1.5}$ the objective $\phi^{1.5}$ is a generalized Büchi objective. Since π^* chooses moves that lead to winning states with positive probability and the set of winning states is finite and closed, every state in $Win(G'', \phi^{1.5})$ is eventually visited with probability 1 [8]. This proves assertion (1). We now show that if π^* is winning in $G^{1.5}$ then it is also winning in $G^{1.5}_f$ and vice-versa. In one direction, it is easy to see that since $\phi^{1.5}_f \Rightarrow \phi^{1.5}$, if π^* is winning in $G^{1.5}_f$ then it is also winning in $G^{1.5}$. In the other direction, we show that following π^* in G'', there exists a bound $\beta \in \mathbb{N}$ such that the expected number of steps to visit every state in $Win(G'', \phi^{1.5})$ is at most β with probability 1, which would imply that π^* is also winning in $G^{1.5}_f$. Fix the memoryless randomized strategy π^* in G''. This gives us a Markov chain. Further, the Markov chain is closed and recurrent, which implies bounded expectation on the visit time to every state [20]. Therefore, there exists a bound $\beta \in \mathbb{N}$ such that the expected number of steps to visit every state in $Win(G'', \phi^{1.5})$ within bound β is probability 1, thus completing the proof. The third assertion follows from the results of [12, 13]. □

In Theorem 2 and Theorem 4 we have shown that strategies that are winning in the non-finitary scheduling games are also winning in their respective finitary versions. Given that the winning strategies coincide for the finitary and the non-finitary objectives, we do not consider the finitary objectives in the sequel. In the following we present properties of scheduling game structures that we exploit to compute winning strategies in time quadratic in the size of the game structures. We show that these strategies suffice in almost all practical scenarios and fail only in contrived examples.

3.5 Properties

In order to argue that π^* is winning not only in $G^{1.5}$, but also in $G^{2.5}$, we need to develop some properties of π^* and $M_{\mathcal{I}}$. First, we state a simple property of $M_{\mathcal{I}}$.

Lemma 1 *In $M_{\mathcal{I}}$, there is no loop made entirely of input edges, and there is no loop made entirely of output edges.*

Proof The first statement is due to the fact that each input edge decreases the value of a resource. The second statement is due to the fact that resource requests ($w_x!$) are immediately followed by an input edge, and resource releases ($r_x!$) increase the value of a resource. □

We now show that, in $M_{\mathcal{I}}$, input and output moves commute, as they are independent. In the following, we write $s \xrightarrow[i]{x} t$ to signify that $(s, t) \in E$, $\lambda(s, t) = x$ and $\theta(s, t) = i$.

Lemma 2 *For all $s, s_1, s_2 \in S$, if $s \xrightarrow[i]{\alpha!} s_1$ and $s \xrightarrow[j]{\beta?} s_2$, then there is $t \in S$ such that $s_2 \xrightarrow[i]{\alpha!} t$ and $s_1 \xrightarrow[j]{\beta?} t$.*

Proof First, notice that $i \neq j$, as input edges have no siblings in their respective thread (see Definition 1). Second, the value of each resource in s_1 is at least as much as it is in s. Thus, there is a state t such that $s_1 \xrightarrow[j]{\beta?} t$. In s_2, the value of a certain resource is lower than it is s. However, output edges are not affected by the value of the resources, so there is a state t' such that $s_2 \xrightarrow[i]{\alpha!} t'$, and by construction of $M_{\mathcal{I}}$, we have $t = t'$. □

The following lemma states an equivalent commutativity property for outputs belonging to different threads.

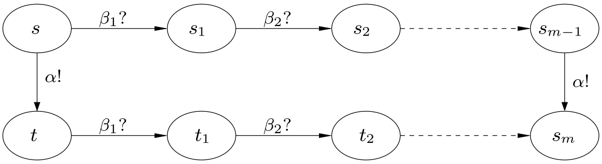


Fig. 5 Outputs cannot link winning states to losing ones

Lemma 3 *For all $s, s_1, s_2 \in S$, if $s \underset{i}{\overset{\alpha!}{\rightarrow}} s_1$ and $s \underset{j}{\overset{\beta!}{\rightarrow}} s_2$, with $i \neq j$, then there is $t \in S$ such that $s_2 \underset{i}{\overset{\alpha!}{\rightarrow}} t$ and $s_1 \underset{j}{\overset{\beta!}{\rightarrow}} t$.*

Proof Since output edges can either decrease resource usage (in the case of resource release actions), or leave resource usage unchanged (in the case of resource request actions), $\alpha!$ will still be enabled from s_2, and $\beta!$ will be enabled from s_1; moreover, by construction of $M_{\mathcal{I}}$, we have $s_2 \underset{i}{\overset{\alpha!}{\rightarrow}} t$ and $s_1 \underset{j}{\overset{\beta!}{\rightarrow}} t$ for the same t. □

The following lemma shows that, in $G^{1.5}$, an edge labeled with an output cannot connect a winning state to a losing state.

Lemma 4 *Let $s \in Win(G^{1.5})$ and $s \underset{i}{\overset{\alpha!}{\rightarrow}} t$. Then, $t \in Win(G^{1.5})$.*

Proof Suppose that, starting from s, we keep following winning inputs, as long as there is a winning input in the current state. By Lemma 1, we must eventually reach a state s_{m-1} that has no winning inputs. By repeated applications of Lemma 2, the output $\alpha!$ is still enabled in s_{m-1}.

Summarizing, as illustrated in Fig. 5, we can find a path $\sigma = ss_1 \dots s_m$ such that (i) all states in σ are winning, (ii) all edges in σ except the last one are labeled with inputs, and (iii) the last edge (s_{m-1}, s_m) is labeled with $\alpha!$.

Again by repeated applications of Lemma 2, from t we can mimic the path σ, by taking similar input edges, finally reaching s_m. We obtain the conclusion that t can reach the winning state s_m be means of input edges only. So, t itself is a winning state. □

In the following, we say that a path is *in $Win(G^{1.5})$* to mean that it is a path in $G^{1.5}$ made entirely of winning states. We now introduce a binary relation "$\sqsubseteq$" over the set of winning states of $G^{1.5}$. For all $s, s' \in Win(G^{1.5})$, let $s \sqsubseteq s'$ if and only if there is a path σ in $Win(G^{1.5})$ that goes from s to s' using only output edges. The following lemma shows that if $s \sqsubseteq s'$ and an input edge is winning from s, the corresponding input edge from s' is also winning.

Lemma 5 *Let $s \sqsubseteq s'$. For all $t \in Win(G^{1.5})$ such that $s \underset{i}{\overset{\alpha?}{\rightarrow}} t$ there is $t' \in Win(G^{1.5})$ such that $s' \underset{i}{\overset{\alpha?}{\rightarrow}} t'$ and $t \sqsubseteq t'$.*

Proof Let σ be a path from s to s' in $Win(G^{1.5})$ that contains only outputs edges. By repeated applications of Lemma 2, we can take a similar path σ' from t, leading to a state t' such that $t \sqsubseteq t'$. Moreover, by construction $s' \underset{i}{\overset{\alpha?}{\rightarrow}} t'$. By applying Lemma 4 to all edges in σ' we obtain that, since t is winning, t' is also winning. □

The following lemma will be instrumental in showing that π^* is a winning strategy also in $G^{2.5}$.

Lemma 6 *There is $p > 0$ such that, for all $s \in Win(G^{1.5})$, if in $Win(G^{1.5})$ there is an acyclic path from s to a state s', then using π^* in $G^{2.5}$, for all player 2 strategies, with probability at least p, starting from s the game reaches a state t' such that $s' \sqsubseteq t'$.*

Proof Let ρ be the path from s to s'; the proof is by induction on the length of ρ. Fix an arbitrary strategy of player 2. For $|\rho| = 0$, the result trivially holds. As induction hypothesis, assume that there is a path ρ from s to s' in $Win(G^{1.5})$, and assume that using π^* in $G^{2.5}$ we can reach from s a state t' such that $s' \sqsubseteq t'$ with positive probability. Let σ be the sequence of output actions leading from s' to t', and let θ be the path from s to t'. We will show that, if we prolong ρ by one step, reaching s'', then we can prolong θ by 0 or more steps, obtaining a path θ'' to t'', such that $s'' \sqsubseteq t''$, and such that θ'' is followed with positive bounded probability in $G^{2.5}$. Notice that, due to Lemma 3, outputs of different threads commute. Hence, we can consider the ordering in σ restricted to outputs belonging to the same thread. Equivalently, rather than σ, we can reason about the collection of sequences of output actions $\{\sigma_i\}_{i=1..n}$, where σ_i represents the sequence of actions of thread i along σ. There are then three cases, depending on the step $s's''$:

- Assume that $s' \underset{i}{\xrightarrow{\alpha?}} s''$, for some α and $i \in \{1, \dots, n\}$. By Lemma 5, there is also a winning step $t' \underset{i}{\xrightarrow{\alpha?}} t''$, and a path from s'' to t'' that uses the sequence of output actions σ. As π^* takes this step with positive probability, this leads to the result.
- Assume that $s' \underset{i}{\xrightarrow{\alpha!}} s''$, for some α and $i \in \{1, \dots, n\}$; assume also that α does not appear in σ_i. By Lemma 3, from t', the same output α is enabled, so that π^* will play with positive probability action $\perp$, and in $G^{2.5}$ some output β will occur. If β belongs to thread i, then with positive probability (according to the randomized resolution of intra-thread non-determinism) it must be $\beta = \alpha$, and the destination state t'' will be related to s'' again by σ. If β does not belong to thread i, we add β to σ. By Lemma 3 we have that output α is still enabled from the destination state after β, so that π^* will again play $\perp$ from the destination with positive probability. Eventually, an output belonging to thread i will occur, as by Lemma 1 there cannot be an infinite path consisting entirely of output actions.
- Assume that $s' \underset{i}{\xrightarrow{\alpha!}} s''$, for some α and $i \in \{1, \dots, n\}$; assume also that α appears in σ_i. Then, with positive probability (due to the resolution of inter-thread non-determinism), α will be the first action of σ_i. We remove α from σ_i, obtaining a shorter σ'; we have that $s'' \sqsubseteq t'$, and s'' and t' are related by σ'.

The existence of a constant bound $p > 0$ derives from the fact that the length of ρ, and the size of σ, are bounded, as is the number of ways in which intra-thread non-determinism can be resolved. □

3.6 Comparing games

We now proceed to prove that the strategy π^* is also a winning strategy for $G^{2.5}$.

Theorem 5 *The strategy π^* is winning in game $G^{2.5}$, and $Win(G^{1.5}) = Win(G^{2.5})$.*

Proof For $i \in \{1, \dots, n\}$ and $s \in Win(G^{1.5})$, we say that thread i is enabled in s if there is an edge $(s,t) \in E$ such that $\theta(s,t) = i$ and $t \in Win(G^{1.5})$. Note that this definition is correct, as by Lemma 4 output edges are always winning.

For $i \in \{1, \dots, n\}$ and $s^* \in Win(G^{1.5})$, we have to prove that, using π^* in $G^{2.5}$ and starting from s^*, with positive probability a state is reached where thread i is enabled. Since this is true of every winning state s^*, and since the game stays forever in the set of winning states, it follows that the probability of enabling thread i infinitely often, ensuring that it is also taken infinitely often, is in fact 1.

If in s^* the next action of thread i is an output, then by Lemma 4 it is available directly from s^*. Thus, assume in the following that the next action of thread i in s^* is an input. Since s^* is winning in $G^{1.5}$, there is a path in $Win(G^{1.5})$ from s^* to a state t^* where thread i is enabled. By applying Lemma 6 to states $s = s^*$ and $s' = t^*$, we obtain that in $G^{2.5}$ from s^* with positive probability a state t' is reached such that $t^* \sqsubseteq t'$, and therefore thread i is enabled in t'. □

The previous result, which depends in a crucial way on the structural properties of $G^{2.5}$ (it is certainly not valid for an arbitrary two-person game), enables us to compute in quadratic time a winning strategy for game $G^{2.5}$. We now show how to use this result for $G^{2.5}$ also for our original problem G^2.

Our first result concerns systems where all resources are mutexes (called *mutex-only systems*), and where the threads satisfy the *periodically mutex-free* (PMF) assumption. Informally, this assumption states that, if the intra-thread non-determinism is resolved in a fair fashion, then the thread is infinitely often not holding any mutex. In practice, threads in mutex-only systems invariably satisfy the PMF assumption. To make this precise, consider a fixed thread interface $I_i = (R_i, S_i, E_i, s_i^{\text{init}}, \lambda_i)$, for $1 \leq i \leq n$. A *path* in I_i is a path in the graph (S_i, E_i). We say that an infinite path is *fair* iff it satisfies $\bigwedge_{u \in S_i} \bigwedge_{v \in Osucc_i(u)} \Box\Diamond from_i^u \Rightarrow \Box\Diamond take_i^{u,v}$. Moreover, for a finite path σ and a resource $x \in R$, let $decr(x,\sigma) = |\{(s,t) \in \sigma \mid \lambda_i(s,t) = g_x?\}|$, $incr(x,\sigma) = |\{(s,t) \in \sigma \mid \lambda_i(s,t) = r_x!\}|$, and $balance(x,\sigma) = incr(x,\sigma) - decr(x,\sigma)$. We say that I_i is *mutex-correct* if for all finite traces σ and all mutexes $x \in R_i$, it holds $balance(x,\sigma) \in \{-1, 0\}$.

Definition 7 We say that a thread is *periodically mutex free* (PMF) if it only uses mutexes, it is mutex-correct, and in all fair paths σ, there exist infinitely many prefixes σ' of σ that satisfy $balance(x,\sigma') = 0$ for all mutexes x.

For mutex-only systems consisting of threads satisfying the PMF assumption (called, for short, *PMF systems*), the strategy π^* is winning also in G^2. Hence, for PMF systems we can derive resource managers in time quadratic in $|G^2|$.

Theorem 6 *For PMF systems, π^* is winning in game G^2, and $Win(G^{1.5}) = Win(G^2)$.*

Proof By Theorem 5, we have that π^* is winning in game $G^{2.5}$. The difference between $G^{2.5}$ and G^2 is in the resolution of intra-thread non-determinism; $\varphi_{\mathcal{I}}^{intra}$ may not hold in G^2, implying that some conditional branches may never be taken. Towards the proof, we first show that $Win(G^{2.5}) = Win(G^2)$. In one direction, if a state $s \in Win(G^2)$ then $s \in Win(G^{2.5})$; if a state s is winning against arbitrary resolution of intra-thread non-determinism, then it must be the case that s is winning if $\varphi_{\mathcal{I}}^{intra}$ holds. Therefore, $Win(G^2) \subseteq Win(G^{2.5})$. In the other direction, consider $s \in Win(G^{2.5})$ but $s \notin Win(G^2)$. By Lemma 4, since output edges never lead to losing states, it must be the case that there exists $s' \in Win(G^2)$ with

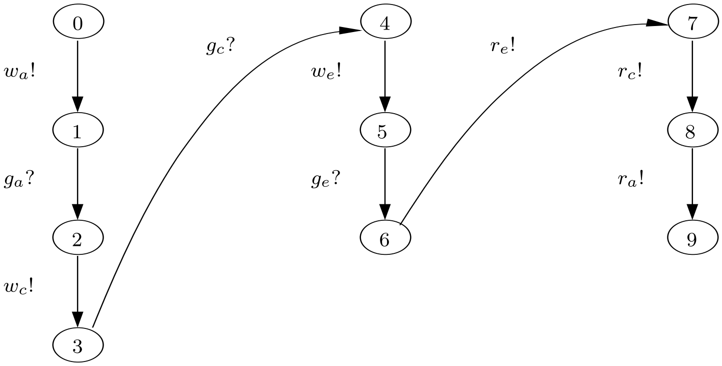


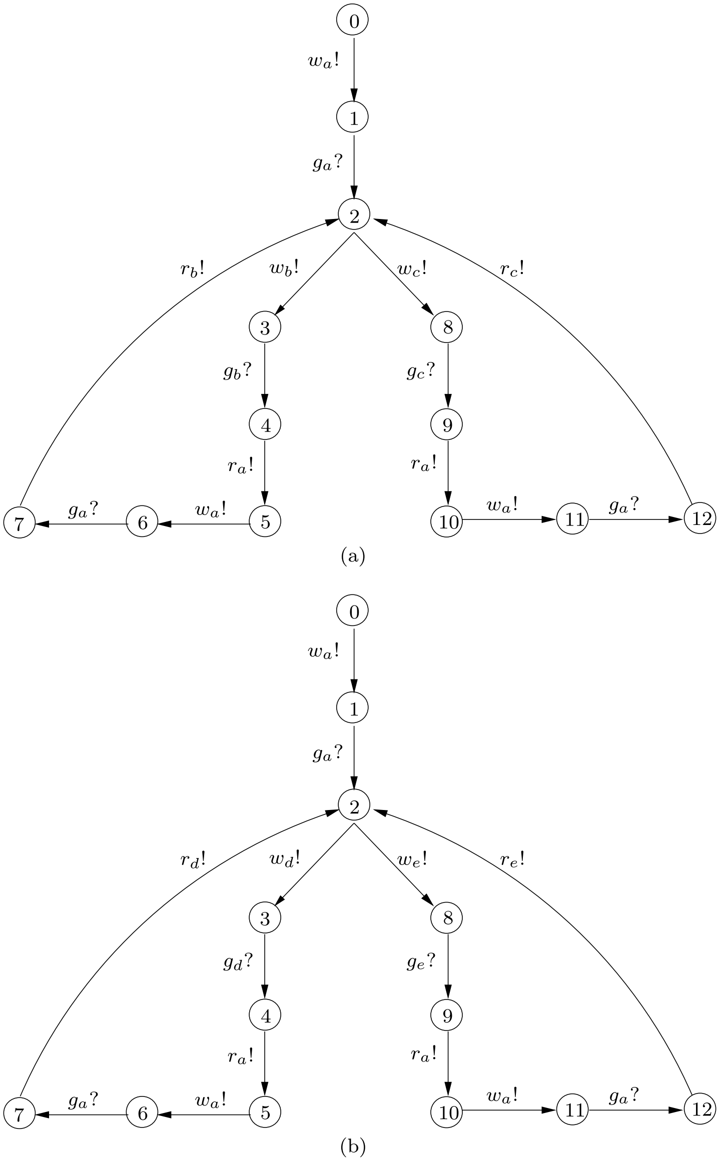
Fig. 6 Thread interface from Example 2

$s' \xrightarrow[i]{\alpha?} s$ for some thread i and input action α; specifically, a resource was granted to thread i in state s' leading to state s from which the resource was never released as a conditional branch was never taken. But this can never happen, given the system is PMF, as along all fair paths, there occur infinitely many states where thread i is not holding any resource for all threads $i \in \{1, \dots, n\}$. Therefore, it must be the case that $s \in \mathit{Win}(G^2)$ and hence $\mathit{Win}(G^{2.5}) \subseteq \mathit{Win}(G^2)$ leading to $\mathit{Win}(G^{2.5}) = \mathit{Win}(G^2)$. Further, given π^* is winning in $G^{2.5}$, $\mathit{Win}(G^{2.5}) = \mathit{Win}(G^2)$ and using π^*, the game forever remains in the set of winning states, we have that for PMF systems π^* is winning in G^2. Finally, given $\mathit{Win}(G^{1.5}) = \mathit{Win}(G^{2.5})$ by Theorem 5 and $\mathit{Win}(G^{2.5}) = \mathit{Win}(G^2)$, we conclude $\mathit{Win}(G^{1.5}) = \mathit{Win}(G^2)$. □

The next example shows that π^* may not be winning in G^2, when the system is not PMF. Notice that a rather special thread structure is required for this to happen.

Example 2 Consider the 5-mutex, 3-thread system $(\{a, b, c, d, e\}, \nu^0, (I_1, I_2, I_3))$ where I_1 is as in Fig. 7(a), I_2 is as in Fig. 7(b), and I_3 is as in Fig. 6. First, at all times after thread 1 reaches state 2, it will always own at least one mutex among $\{a, b, c\}$. Similarly, thread 2 will always own at least one of $\{a, d, e\}$. For this reason, the system is not PMF. However, the initial state $(0, 0, 0, \nu^0)$ of $G^{1.5}$ is winning. Clearly, threads 1 and 2 can make infinite progress, since they only share mutex a, and they both release said mutex periodically. It remains to show that under the most general winning strategy π^*, thread 3 is allowed to perform its critical region (i.e. state 6) with probability 1. In $G^{1.5}$ (and $G^{2.5}$) the non-determinism that threads 1 and 2 exhibit in state 2 is resolved by a uniform distribution. So, while making infinite progress, with probability 1 those threads will acquire mutexes b and d at the same time, thus leaving mutexes c and e free. At that point, as soon as mutex a is released, thread 3 can safely execute its critical region, by acquiring mutexes a, c, e.

On the other hand, in game G^2 threads 1 and 2 can cooperate in order to never release both c and e at the same time. When thread 1 is in state 2, thread 2 can only be in state 6 or 11 (because those are the only states where thread 2 does not hold a). So, player 2 can choose to acquire c when thread 2 is in 6 (thus holding d) and acquire b when thread 2 is in 11 (thus holding e). This ensures that c and e are never free at the same time. Now, consider a state where a is free. Giving a to thread 3 inevitably leads to a deadlock, because thread 3 needs c and e before releasing a, and either of them is currently owned and will not be released before a is.



(a)

(b)

Fig. 7 Thread interfaces from Example 2

Our next result, useful for threads that may use semaphores, enables us to establish whether the strategy π^* is winning also for G^2. To develop the result, note that the game G^2, once player 1 fixes strategy π^*, is a 2-MDP. For such 2-MDPs, we can compute in polynomial time the set of winning states for player 2 with respect to the complementary goal $\neg\phi^2$ using an algorithm that is a modified version of the algorithm proposed in [9] for Streett MDPs. This leads to the following result.

Theorem 7 *We can check in time $O(|G|^2 \cdot n \cdot \sum_{i=1}^{n} |E_i|)$ whether the strategy π^* is winning in G^2.*

In our experience, the strategy π^* is almost invariably winning in G^2; indeed, the only counterexamples we have been able to construct are based on threads with fairly special structure, where inter-thread communication can be used to synchronize the usage of resources by threads in particular ways. Therefore, we claim that in most cases, we can construct a resource manager strategy in time quadratic in $|G^2|$.

4 Towards efficient resource managers

The strategy π^*, even when winning, may not be an efficient strategy in practice. According to it, the resource manager would issue $\perp$ (wait for a resource request or release) with positive probability when there are input moves that are available and winning. First, this potentially reduces CPU utilization. In fact, other things being equal, it is better to grant immediately as many resource requests as possible: this ensures that the OS scheduler has the widest choice of threads to execute on the CPU, helping to avoid idle time when all available threads are blocked, e.g., waiting for I/O. More importantly, as a consequence of how we abstract thread interfaces, there is no guarantee that a thread whose next action is an output will issue that output within a short amount of time. For instance, the next resource request may be issued only after some user input has occurred.

In this section, we propose several improvements to π^*, aimed at reducing the number of times when the manager issues $\perp$ when input actions are available.

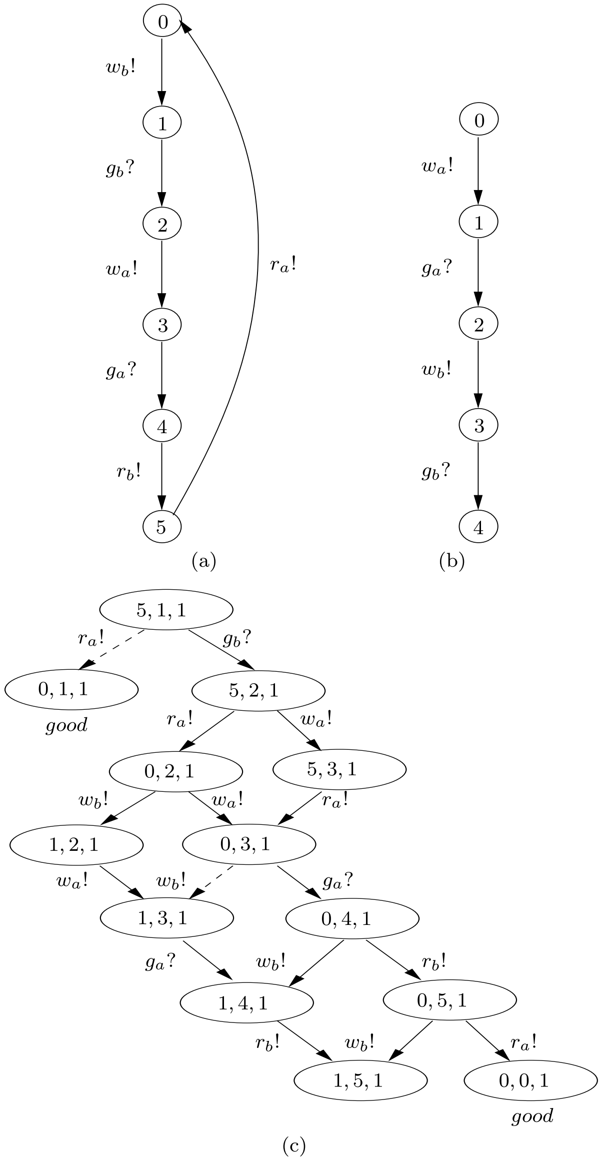
4.1 Maximal progress and critical progress strategies

The simplest idea consists in issuing $\perp$ only in the states $S^! = \{s \in S \mid \pi^*(s)(\perp) = 1\}$ where $\perp$ is the only winning move: this corresponds to waiting for output moves only when no resource can be granted. This idea leads to the *maximal progress strategy* π^{p}, defined by $\pi^{\mathrm{p}}(s) = \delta(\perp)$ for $s \in S^!$, and $\pi^{\mathrm{p}}(s) = \mathit{Uniform}(\mathrm{Supp}(\pi^*(s)) \setminus \{\perp\})$ otherwise. Unfortunately, the maximal progress strategy is not always winning, as the following example demonstrates.

Example 3 Consider the 3-thread system $(\{a, b\}, \{a \mapsto 1, b \mapsto 1\}, (I_1, I_2, I_3))$ where I_1 and I_2 are as in Fig. 8(a), while I_3 is as in Fig. 8(b). Figure 8(c) shows a fragment of the corresponding joint interface. Let us analyze this fragment as part of G^2, and assume that player 1 employs π^{p}. One can check that, starting from the initial state $(0, 0, 0, \nu^0)$, player 2 can steer the game to state $(5, 1, 1, \nu)$, where $\nu = \{a \mapsto 0, b \mapsto 1\}$. At this point, all of the edges, except for the dashed ones, can be taken under π^{p}. The objective for the player 1 is to reach one of the states labeled as “good”, as in those states thread 3 can make progress without risking a deadlock. However, player 2 can steer the game away from the two good states, thus reaching $(1, 5, 1, \nu)$ with certainty. Since $(1, 5, 1, \nu)$ is symmetrical w.r.t. $(5, 1, 1, \nu)$, this strategy enables player 2 to keep thread 3 starving forever. Thus, π^{p} is not a winning strategy in this game. The same applies to $G^{2.5}$, since the threads under consideration have no inter-thread non-determinism.

It should be noted that the situation is different in $G^{1.5}$. Since all output edges happen uniformly at random, π^{p} is winning in this case, as state $(0, 0, 1, \nu^0)$ is eventually reached with probability 1.

Fig. 8 A system where the maximal progress strategy is not winning



The example above suggests that sometimes, as in state $(5, 1, 1, \nu)$, it is necessary to wait for output actions, even when there are resources that are ready to be granted. The problem of waiting for outputs, as mentioned earlier, is that in general there is no guarantee that the outputs will be generated in a timely fashion. However, in mutex-only systems, we can assume that when a thread holds a mutex it will generate an output in a timely fashion, either to release the mutex, or to request another mutex. This captures the idea that, in well-written code, critical regions have short durations. Based on this idea, we let S^c be the set of states of a mutex-only system where there is some thread holding a mutex, and we propose a strategy that waits for outputs only in S^c. We define the *critical progress strategy* π^c by letting, for all $s \in S$, $\pi^{\mathrm{c}}(s) = \pi^*(s)$ if $s \in S^{\mathrm{c}}$ or $s \in S^!$, and $\pi^{\mathrm{c}}(s) = \mathit{Uniform}(\mathrm{Supp}(\pi^*(s)) \setminus \{\bot\})$ otherwise. The following result shows that, for PMF systems, π^{c} is an efficient resource manager strategy.

Theorem 8 *In a PMF system, π^{c} is winning for G^2.*

Proof For all states $s \in S^{\mathrm{c}} \cup S^!$, since $\pi^{\mathrm{c}}(s) = \pi^*(s)$, given π^* is winning in G^2 for PMF systems by Theorem 6, we have π^{c} is winning in G^2. For all states $s \in S \setminus (S^{\mathrm{c}} \cup S^!)$, given π^* is winning in G^2, all moves in $\mathrm{Supp}(\pi^*(s))$ are winning. As none of the threads are holding a resource at s by the definition of S^{c}, and choosing $\bot$ is necessary only when some thread is holding a resource, we have $\pi^{\mathrm{c}}(s) = \mathit{Uniform}(\mathrm{Supp}(\pi^*(s)) \setminus \{\bot\})$ is winning in G^2. □

4.2 Efficient strategies for systems with semaphores

A natural extension of π^{c} to systems with semaphores is a strategy that waits for outputs only when there is at least one thread waiting for a resource that is not available (so that another thread must be holding a resource, and it may be reasonable to expect an output action in a timely manner). Unfortunately, there are examples showing that such an extension is not winning in general. We discuss two related strategies that are winning, and efficient, for systems with semaphores.

To obtain our first strategy, we reason as follows. Once a memoryless strategy $\pi \in \Pi_1$ is fixed, the game G^2 is equivalent to a 2-MDP $G^2(\pi)$. If an end-component in this 2-MDP is not fair, that is, if there is a thread k that is neither finished, nor progresses in the end component, then it can be seen that thread k must be stuck waiting for an input (a resource) at all states of the end component. This suggests to skip $\bot$ (waiting for outputs) only when no thread is blocked: in this way, if the strategy differs from π^* by cutting $\bot$, it can do so only in a winning component. Precisely, for $s \in S$ we let $\mathit{Succ}(s, \pi^*) = \{t \in S \mid \exists m_1 \in \Gamma_1(s). \exists m_2 \in \Gamma_2(s). (\pi^*(m_1) > 0 \wedge \tau(s, m_1, m_2)(t) > 0)\}$ be the set of possible successors of s according to π^*, and we let $S^{\mathrm{b}} = \{s \in S \mid \exists k \in [1..n]. \forall t \in \mathit{Succ}(s, \pi^*). \theta(s, t) \neq k\}$ be the set of states where some thread is blocked. For $s \in S$, we then define π^{b} by $\pi^{\mathrm{b}}(s) = \pi^*(s)$ if $s \in S^{\mathrm{b}} \cup S^!$, and $\pi^{\mathrm{b}}(s) = \mathit{Uniform}(\mathrm{Supp}(\pi^*(s)) \setminus \{\bot\})$ otherwise.

Theorem 9 *The strategy π^{b} is winning in G^2 iff π^* is winning in G^2.*

Proof In one direction, if π^{b} is winning in G^2, then not skipping $\bot$ in states where none of the threads are holding a resource is also winning in G^2 and hence π^* is winning in G^2. In the other direction, similar to the proof of Theorem 8, if π^* is winning in G^2, then cutting $\bot$ in states where no thread is holding a resource is winning in G^2. Hence π^{b} is winning in G^2. □

Finally, we can obtain an efficient strategy *with memory* as follows. We say that a thread k is *bypassed* whenever it is waiting for an input, and the scheduling strategy does not give that input. Then, given a *bypass bound* $M \in \mathbb{N}$, we can construct a strategy π^{p}_M as follows. For each thread $k \in [1..n]$, π^{p}_M keeps track of the number b_k of times for which thread k has been consecutively bypassed. As long as $b_k \leq M$ for all $1 \leq k \leq n$, the strategy π^{p}_M behaves like π^{p}. When $b_k > M$ for some $k \in [1..n]$, on the other hand, π^{p}_M reverts to behave like π^*, thus sometimes waiting for outputs when there are input actions (resource grants) that could be taken. The idea, informally, is as follows: if a thread is bypassed for a large number of consecutive times, it means that some other threads may be holding the resources it needs to proceed. Favoring output actions (among which are resource releases) enables the system to reach a state where the bypassed thread can be finally granted the resource it needs.

Theorem 10 *For all $M \in \mathbb{N}$, we have that π^{p}_M is winning in G^2 iff π^* is winning in G^2.*

Proof In one direction, consider an arbitrary fixed bound $M \in \mathbb{N}$ and the resulting strategy π^{p}_M that is winning in G^2 from a starting state s^*. We show that π^* is winning in G^2. For all states $s \in S^!$, where the only winning move is $\bot$, since π^{p}_M and π^* will choose $\bot$, π^* is winning at s. If $b_i \le M$ for all threads $i \in \{1, \dots, n\}$ for all paths starting at s^*, then given π^{p}_M is winning, we can always reach a state t^*_i where thread i is enabled with positive probability. This implies using π^*, which only differs from π^{p}_M by playing $\bot$ with positive probability, we can again reach t^* with positive probability. Therefore, we have π^* is winning. If $b_k > M$ for some thread $k \in \{1, \dots, n\}$ for some path starting at s^*, then as π^{p}_M reverts to π^* and π^{p}_M is winning, π^* is winning as well.

In the other direction, given π^* is winning in G^2 from a starting state s^*, if $M = 0$, then as π^{p}_M is the same as π^*, we have π^{p}_M is winning in G^2. Consider an arbitrary fixed $M > 0$. The strategy π^{p}_M differs from π^* by cutting $\bot$ in states $Win(G^2) \setminus S^!$ as long as $b_i \le M$ for all threads $i \in \{1, \dots, n\}$. Since the game always remains in $Win(G^2)$ and π^{p}_M reverts to π^* when $b_k > M$ for some thread $k \in \{1, \dots, n\}$, given π^* is winning, we have π^{p}_M is also winning in G^2. □

5 The tool

We have developed a prototype tool called CYNTHESIS that realizes the theory hereby presented. The tool takes as input a C program, and it either produces a warning that the system is not schedulable (according to the definition in Sect. 3.2), or it outputs a custom resource manager encoded as a C program that can be compiled and linked to the original program. The result is an executable that is deadlock-free whenever the OS scheduler is fair, and the threads do not block for reasons other than resources (such as infinite loops). The tool is currently tailored to the eCos embedded OS [17], but it can be easily modified to work with another OS.

To extract thread interfaces, the tool uses the CIL library [30] to build a control-flow graph (CFG) for each thread. For the purpose of this graph, function calls are treated as inlined. While building the CFG, each time a synchronization primitive is detected, edges labeled with the appropriate action are added to the thread interface, as follows: (i) calls to `mutex_unlock(x)` and `sem_post(x)` are represented by an edge labeled $r_x!$, and (ii) calls to `mutex_lock(x)` and `sem_wait(x)` are represented by a sequence of two edges labeled with $w_x!$ and $g_x?$ respectively. The original calls are also automatically annotated with location information, to allow the resource manager to distinguish them at run-time. The graph is then minimized to remove transitions that do not involve resources.

In order for the tool to correctly identify resources, they must be declared as global variables and then used by their original names; we are working to add alias analysis to the tool to overcome this limitation. Once the thread interfaces are extracted, the tool solves the game $G^{1.5}$ and it outputs a custom resource manager in the form of compilable C code. The resource manager behaves like the strategy π^*, or optionally like one of the other winning strategies discussed in Sect. 4. In order to simulate the behavior of a strategy, the custom manager needs to know which winning moves are available at any given decision point. In turn, this means that it has to know in which state of the joint interface the system currently

Table 1 Experiments

# threads	$\lvert M_{\mathcal{I}} \rvert$	# bad states	# BDD nodes	Size of BDD (kbytes)	Time (seconds)
2	37	3	45	0.5	0.04
3	171	18	113	1.3	0.05
4	13905	580	181	2.2	0.6
6	17496	2592	267	3.2	12
6	33120	5490	1084	13	150

is, and what are the winning moves from that state. Rather than keeping a copy of the joint interface, which can be of exponential size in the number of threads, the manager keeps separate copies of the individual thread interfaces, along with the value of the resources. With this information, the manager is aware of all moves; all that remains to encode are the moves that are *not* part of the winning strategy: to do this, it suffices to store the set of losing states. As the number of losing states can grow exponentially with the number of threads, we encode the losing states using a BDD [4], leading to a very compact representation. In Table 1, we report the result of some experiments, all run on a 3.4 GHz AMD Phenom II machine with 4 GB of memory. The threads involved in the test give rise to thread interfaces having between 5 and 12 states; apart from the resource primitives, the size of the source code of the threads has a negligible effect on the running time of the tool, and it is irrelevant to the size of the synthesized manager and the BDD. The second column reports the number of states in the joint interface, and the last column reports the total time needed to synthesize the manager.

5.1 A case study

We conducted a more extensive test, consisting in analyzing a multi-threaded program implementing an ad-hoc network protocol for Lego robots. As illustrated in Fig. 9, the program is composed of five threads, represented by ovals in the figure, that manage four message queues, represented as boxes in the figure.

Threads *user* and *generator* add packets to the *input* queue. The *router* thread removes packets from the *input* queue, and dispatches them to the other queues. Packets in the *user* queue are intended for the local node, so they are consumed by the *user* thread. Packets in the *broadcast* queue are intended for broadcast, and they are moved to the *output* queue by the *delay* thread, after a random delay, intended to avoid packet collisions during broadcast propagations. Packets in the *output* queue are in transit to another node, so they are treated by the *sender* thread. Notice that if the *sender* fails to send a packet on the network, it puts it in the *broadcast* queue (even if it is not a broadcast packet), so that it will be re-sent after a delay.

Each queue is protected by a mutex, and two semaphores that count the number of empty and free slots, respectively. Altogether, the program employs 6 mutexes and 8 semaphores. By restricting all queues to having 1 slot, the resulting joint interface contains 200,000 states, and the tool terminates its analysis in under 30 seconds. The BDD which encodes the set of deadlock states occupies 16 kB.

The tool found a deadlock that corresponds to the following situation. Suppose that queues *output* and *broadcast* are both full. Suppose also that the *sender* thread extracts a packet from *output* and tries to send it on the network. If the send fails, the thread will try to insert the packet in the *broadcast* queue. Since the latter is full, the *sender* thread will

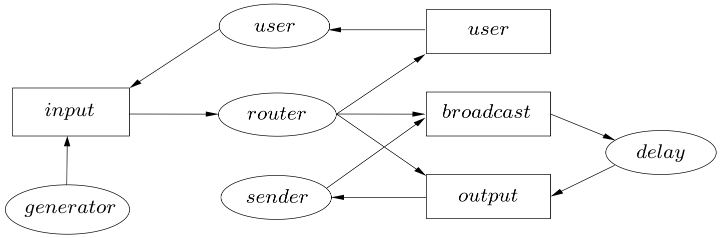


Fig. 9 Scheme of an ad-hoc network protocol implementation

hang on a semaphore, waiting for an empty slot in *broadcast*. However, the only way a slot in *broadcast* can be emptied is for the *delay* thread to move a packet to *output*, which is still full. Therefore, the *sender* will hang forever, and the whole system will consequently block.

Interestingly, the tool reports that there is a winning strategy in this situation. The strategy consists in "slowing down" the router, preventing it from adding packets to *broadcast* if *output* is full, and vice-versa.

Acknowledgements This research was supported in part by the National Science Foundation CAREER award CCR-0132780, by the ONR grant N00014-02-1-0671, by the National Science Foundation grants CCR-0427202 and CCR-0234690, and by the ARP award TO.030.MM.D.

References

1. Banaszak ZA, Krogh BH (1990) Deadlock avoidance in flexible manufacturing systems with concurrently competing process flows. IEEE Trans Rob Autom 6(6):724–734
2. Bloem R, Jobstmann B, Piterman N, Pnueli A, Sa'ar Y (2012) Synthesis of reactive(1) designs. J Comput Syst Sci 78(3):911–938
3. Bodík R (2012) Compiling what to how: technical perspective. Commun ACM 55(2):102
4. Bryant RE (1986) Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers 35:677–691
5. Buttazzo GC (2004) Hard real-time computing systems: predictable scheduling algorithms and applications. Real-time systems series. Springer, Santa Clara
6. Cerný P, Chatterjee K, Henzinger TA, Radhakrishna A, Singh R (2011) Quantitative synthesis for concurrent programs. In: CAV 11: proc of 23rd conf on computer aided verification, pp 243–259
7. Chatterjee K, Henzinger TA (2006) Finitary winning in omega-regular games. In: TACAS, pp 257–271
8. Chatterjee K, de Alfaro L, Henzinger TA (2004) Trading memory for randomness. In: QEST 04: proceedings of the first international conference on quantitative evaluation of systems. IEEE Comput Soc, New York, pp 206–217
9. Chatterjee K, de Alfaro L, Henzinger TA (2005) The complexity of stochastic rabin and streett games. In: Proc 32nd int colloq aut lang prog. Lect notes in comp sci, vol 3580. Springer, Berlin, pp 878–890
10. Chatterjee K, Henzinger TA, Horn F (2009) Stochastic games with finitary objectives. In: MFCS, pp 34–54
11. Clarke EM, Emerson EA (1981) Design and synthesis of synchronization skeletons using branching time temporal logic. In: Proc workshop on logic of programs. Lect notes in comp sci, vol 131. Springer, Berlin, pp 52–71
12. de Alfaro L (1997) Formal verification of probabilistic systems. PhD thesis, Stanford University. Technical Report STAN-CS-TR-98-1601
13. de Alfaro L, Henzinger TA, Kupferman O (1998) Concurrent reachability games. In: Proc 39th IEEE symp found of comp sci. IEEE Comput Soc, New York, pp 564–575
14. de Alfaro L, Faella M, Majumdar R, Raman V (2005) Code aware resource management. In: EMSOFT 05: 5th intl. ACM conference on embedded software. ACM, New York, pp 191–202
15. Derman C (1970) Finite state Markovian decision processes. Academic Press, San Diego
16. Devillers R (1977) Game interpretation of the deadlock avoidance problem. Commun ACM 20(10):741–745

17. ecos homepage. http://ecos.sourceware.org/
18. Engler DR, Ashcraft K (2003) RacerX: effective, static detection of race conditions and deadlocks. In: SOSP 03: symposium on operating systems principles. ACM, New York, pp 237–252
19. Ezpeleta J, Colom JM, Martínez J (1995) A petri net based deadlock prevention policy for flexible manufacturing systems. IEEE Transactions on Robotics and Automation, N 2(11):173–184
20. Filar J, Vrieze K (1997) Competitive Markov decision processes. Springer, Berlin
21. Filiot E, Jin N, Raskin J-F (2010) Compositional algorithms for ltl synthesis. In: ATVA, pp 112–127
22. Golan-Gueta G, Grasso Bronson N, Aiken A, Ramalingam G, Sagiv M, Yahav E (2011) Automatic fine-grain locking using shape properties. In: 26th ACM SIGPLAN conf on object-oriented programming, systems, languages, and applications (OOPSLA), pp 225–242
23. Gurevich Y, Harrington L (1982) Trees, automata, and games. In: Proc 14th ACM symp theory of comp. ACM, New York, pp 60–65
24. Hsieh FS, Chang SC (1992) Deadlock avoidance controller synthesis for flexible manufacturing systems. In: Proc of 3rd int conf on comp integrated manufacturing, pp 252–261
25. Iordache MV, Moody J, Antsaklis PJ (2002) Synthesis of deadlock prevention supervisors using petri nets. IEEE Trans on Robotics and Automation 18:59–68
26. Karp RM, Miller RE (1969) Parallel program schemata. J Comput Syst Sci 3(2):147–195
27. Kloukinas C, Yovine S (2003) Synthesis of safe, qos extendible, application specific schedulers for heterogeneous real-time systems. In: ECRTS, pp 287–294
28. Kloukinas C, Nakhli C, Yovine S (2003) A methodology and tool support for generating scheduled native code for real-time java applications. In: EMSOFT, pp 274–289
29. Kuperstein M, Vechev MT, Yahav E (2010) Automatic inference of memory fences. In: 10th int conf on formal methods in computer-aided design (FMCAD), pp 111–119
30. Necula GC, McPeak S, Rahul SP, Weimer W (2002) Intermediate language and tools for analysis and transformation of C programs. In: Proceedings of conference on compiler construction (CC)
31. Peterson JL, Silberschatz A (1988) Operating system concepts. Addison-Wesley, Reading
32. Piterman N, Pnueli A (2006) Faster solutions of rabin and streett games. In: LICS, pp 275–284
33. Savage S, Burrows M, Nelson CG, Sobalvarro P, Anderson TA (1997) Eraser: a dynamic data race detector for multithreaded programs. ACM Transactions on Computer Systems 15(4):391–411
34. Solar-Lezama A, Arnold G, Tancau L, Bodík R, Saraswat VA, Seshia SA (2007) Sketching stencils. In: PLDI, pp 167–178
35. Solar-Lezama A, Jones CG, Bodík R (2008) Sketching concurrent data structures. In: ACM SIGPLAN conf on programming language design and implementation (PLDI), pp 136–148
36. Thomas W (1990) Automata on infinite objects. In: van Leeuwen J (ed) Handbook of theoretical computer science, vol B. Elsevier/North-Holland, Amsterdam, pp 135–191. Chapter 4
37. Toshimi M (1982) Deadlock avoidance revisited. J ACM 29(4):1023–1048
38. von Behren JR, Condit J, Zhou F, Necula GC, Brewer EA (2003) Capriccio: scalable threads for internet services. In: SOSP 03: symposium on operating systems principles. ACM, New York, pp 268–281