Code Aware Resource Management*

Luca de Alfaro
Vsihwanath Raman
UC Santa Cruz, USA

Marco Faella
UC Santa Cruz, USA
Università di Napoli
"Federico II", Italy

Rupak Majumdar
UC Los Angeles, USA

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.

Categories and Subject Descriptors

D.4.1 [**Software**]: Operating Systems—*process management*

General Terms

Theory, Reliability, Algorithms.

Keywords

Scheduling, Deadlock Avoidance, Code analysis.

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

EMSOFT'05, September 19–22, 2005, Jersey City, New Jersey, USA.

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 [21, 11]. Monotonic locking [20] 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 [3]. 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 [20], 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

```
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

Figure 1: Two fragments of C 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 Figure 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; priorities will be dealt with in future work. We assume that threads are correct, except possibly for their resource interaction: for instance, we do not guarantee progress if a thread holding a 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 [17, 16], a circumstance that prevents those schemes from addressing the problem of progress (or absence of starvation),

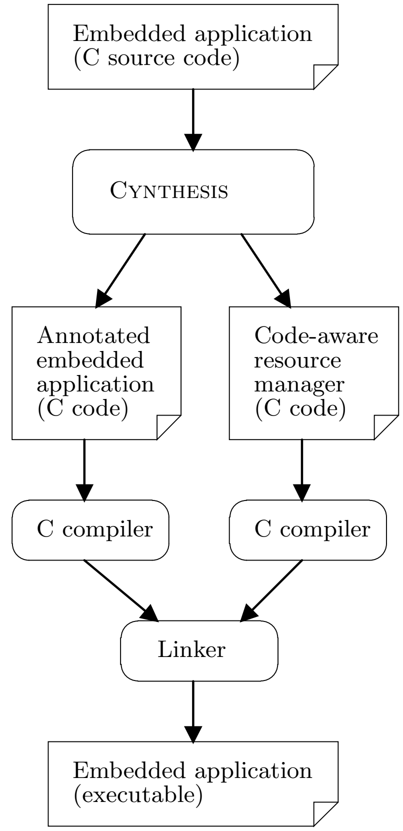


Figure 2: Cynthesis **tool flow: from the source code of an embedded application, to the executable application with its code-aware resource manager.**

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.

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 Figure 2. First, Cynthesis identifies the threads composing the embedded application, and extracts from each thread 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 [10]; 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 Section 5, consisting in a multithreaded 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.

Related work. In closely related work, [17, 16] 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 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 [9, 18, 1, 14, 12, 15], it is assumed that a Petri Net model is constructed by hand. Also, most of these works deal with processes that are terminating and/or deterministic. 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. Further, the use of randomization to generate efficient schedulers has not been studied.

Static compiler techniques have been used in high performance thread packages to improve response time through better scheduling [24], 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.

Paper organization. In Section 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, Section 5 describes the tool CYNTHESIS, as well as the examples and case studies that have been analyzed with it.

2. THREAD RESOURCE INTERFACES

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

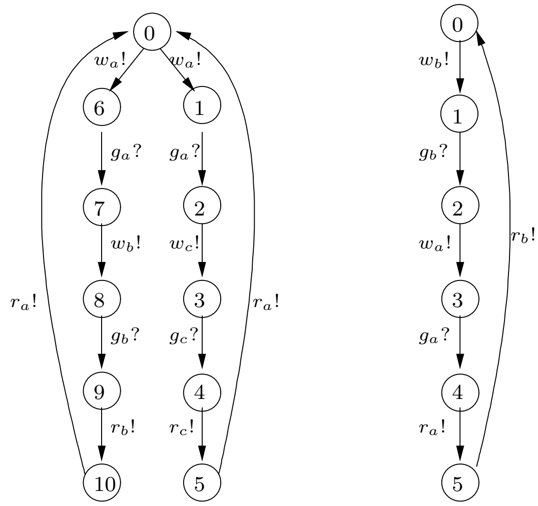
We model the behavior of threads by *thread interfaces.* Thread interfaces model only the resource manipulation aspect of threads, and abstract out all data manipulation.

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



(a) Thread interface 1 (b) Thread interface 2

Figure 3: The thread interfaces corresponding to the code in Figure 1.

$g_x?$; 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 Figure 3 from the code in Figure 1. ∎

2.3 Systems

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 $1 \le i \le n$, and that if $x \in R$ is a mutex, $\nu^0(x) = 1$.*

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:*

- $S = \left(\prod_i S_i\right) \times (R \mapsto \mathbb{N})$;
- $s^{\text{init}} = (s_1^{\text{init}}, \ldots, s_n^{\text{init}}, \nu^0)$;
- *$E \subseteq S \times S$, 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 $1 \le i \le n$, 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, a coverability tree algorithm for Petri nets can check for boundedness, but this check is expensive.

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-hard.*

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 elicit 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 nondeterminism, which determines which output action the thread generates next. Thus, we identify three types of nondeterminism in the joint interface.

1. *Resource manager nondeterminism,* due to the resource manager choosing an input edge, or choosing to wait for an output action.

2. *Inter-thread nondeterminism,* due to the operating-system scheduler resolving thread interleaving.

3. *Intra-thread nondeterminism,* which determines which of several possible output actions a thread will do.

Resource manager. The goal of the resource manager is to ensure that all threads 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. Using temporal logic notation, and considering that final_i is equivalent to $\Box\mathit{final}_i$, the goal can be written as a *generalized Büchi* condition over the edges:

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

Our aim is to synthesize a resource manager that satisfies this goal. In order to model accurately the resource manager synthesis problem, we make the following fairness assumptions over the other two types of nondeterminism.

Inter-thread nondeterminism. We assume that the underlying operating system scheduler is fair: more precisely, we assume that, 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:

$$\phi^{inter}_{\mathcal{I}} = \bigwedge_{i=1}^{n} (\Box\Diamond\mathit{ready}_i \rightarrow \Box\Diamond\mathit{progress}_i).$$

Intra-thread nondeterminism. Assuming that intra-thread nondeterminism is resolved in an arbitrary way may easily lead to declaring the manager synthesis problem to be infeasible. In fact, whenever a thread can execute a loop while holding a resource, the arbitrary resolution of intra-thread nondeterminism introduces the possibility that the loop never terminates. In practice, a reasonable assumption is that intra-thread nondeterminism is resolved in a (strongly) fair fashion: if each choice is presented infinitely often, each choice outcome will follow infinitely often. Such fairness entails loop termination.[1] For all threads $1 \leq i \leq n$, all $u, v \in S_i$, and all $(s,t) \in E$, we introduce the predicates $\mathit{from}^u_i(s,t) \stackrel{\text{def}}{=} (\mathit{loc}_i(s) = u)$ and $\mathit{take}^{u,v}_i(s,t) \stackrel{\text{def}}{=} \big((\mathit{loc}_i(s) = u) \wedge (\mathit{loc}_i(t) = v)\big)$. The fairness assumption for intra-thread nondeterminism can then be written as

$$\phi^{intra}_{\mathcal{I}} = \bigwedge_{i=1}^{n} \bigwedge_{u \in S_i} \bigwedge_{v \in Osucc_i(u)} (\Box\Diamond\mathit{from}^u_i \rightarrow \Box\Diamond\mathit{take}^{u,v}_i).$$

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

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 nondeterminism, and to be able to generate manager strategies that are memoryless, but that may require randomization [4]. Given a finite set A, we denote by $\mathrm{Distr}(A)$ the set of probability distributions over A. For $d \in \mathrm{Distr}(A)$ we let $\mathrm{Supp}(d) = \{a \in A \mid d(a) > 0\}$. Given $a \in A$ we denote by $\delta(a) \in \mathrm{Distr}(A)$ 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 $G = (S, \mathit{Moves}, \Gamma_1, \Gamma_2, \tau, \phi)$ 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 $\tau : S \times \mathit{Moves}^2 \mapsto \mathrm{Distr}(S)$, 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. Finally, the winning condition ϕ is a measurable subset of S^ω.*

For $i \in \{1,2\}$, we say that G is an *i-Markov decision process* (i-MDP) [8] 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 $\pi_i : S^+ \mapsto \mathrm{Distr}(\mathit{Moves})$, 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 [22]) are defined (see e.g. [13]). 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 $\mathrm{Pr}^{\pi_1,\pi_2}_s(\phi) = 1$. As we use randomized strategies, winning with probability 1 is the natural notion of winning. We denote by $\mathit{Win}(G)$ 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 $s \in \mathit{Win}(G)$ and all $\pi_2 \in \Pi_2$, we have $\mathrm{Pr}^{\pi_1,\pi_2}_s(\phi) = 1$. 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 nondeterminism, 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 $M_{\mathcal{I}} = (R, S, E, s^{\text{init}}, \lambda, \theta)$.

Definition 5. *The* two-player game *corresponding to a system $\mathcal{I}$ consists of a tuple $G^2 = (S, \mathit{Moves}, \Gamma_1, \Gamma_2, \tau, \phi^2)$,*

where Moves $= S \cup \{\bot\}$ *and* $\phi^2 = (\phi_{\mathcal{I}}^{inter} \wedge \phi_{\mathcal{I}}^{intra}) \rightarrow \phi_{\mathcal{I}}^{goal}$. *The sets of moves for player 1 (representing the resource manager) and player 2 (representing the inter and intra-thread nondeterminism) are as follows, for all* $s \in S$*:*
- *If* $Osucc(s) \neq \emptyset$*, then* $\Gamma_1(s) = Isucc(s) \cup \{\bot\}$ *and* $\Gamma_2(s) = Osucc(s)$.
- *If* $Osucc(s) = \emptyset$*, then* $\Gamma_1(s) = Isucc(s)$ *and* $\Gamma_2(s) = \{\bot\}$.

The destination function is given by the following rules, where $*$ *represents a wildcard, and* $s \in S$*:*
- *For* $t \in Isucc(s)$*, we have* $\tau(s, t, *) = \delta(t)$*;*
- *for* $t \in Osucc(s)$*, 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. We say that the system is *schedulable* if $s^{\text{init}} \in Win(G^2)$. One can see that this goal is *upward-closed,* so that memoryless, but randomized, strategies suffice to win the game [4].

3.3 Practical Solution of the Scheduling Game

The best known algorithms to compute a winning strategy in G^2 take time exponential in the winning condition, and in our case, the size of the winning condition is proportional to the sum of the sizes (numbers of states) of all thread interfaces in $\mathcal{I}$ [23]. Thus, this approach leads to an inefficient algorithm. 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 consider two simplified versions of G^2:

1. A game $G^{2.5}$, resulting from resolving all intra-thread nondeterminism 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.

We show that we can construct in quadratic time in $|G^2|$ 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. In all cases, we show that it is possible to check efficiently whether the strategy for game $G^{2.5}$ works also for G^2 — and in our experience, this has been always the case in the examples we have studied so far.

Definition 6. *Given the game* $G^2 = (S, Moves, \Gamma_1, \Gamma_2, \tau, \phi^2)$*, the games* $G^{2.5} = (S, Moves', \Gamma_1, \Gamma_2', \tau', \phi^{2.5})$ *and* $G^{1.5} = (S, Moves, \Gamma_1, \Gamma_2'', \tau'', \phi^{1.5})$ *are obtained as follows. We have* $Moves' = Moves \cup \{1, \ldots, n\}$, $\phi^{2.5} = \phi_{\mathcal{I}}^{inter} \rightarrow \phi_{\mathcal{I}}^{goal}$*, and* $\phi^{1.5} = \phi_{\mathcal{I}}^{goal}$. *The functions* Γ_2', τ' *and* Γ_2'', τ'' *coincide with* Γ_2, τ*, except that:*

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

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.

A memoryless strategy $\pi \in \Pi_1$ gives rise to a graph (S, E_π), where $E_\pi = \{(s,t) \mid \pi(s)(t) > 0 \text{ or } \pi(s)(\bot) > 0 \text{ and } \lambda(s,t) \in Acts^O[R]\}$. 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 a closed (no outgoing edge) and strongly connected component of (S, E_π), and such that $F = \{(s,t) \in E_\pi \mid s \in C\}$ [6]. We say that thread k is *finished* in a state s if $loc_k(s)$ 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 $1 \leq k \leq n$, 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 [4]; we denote by $Win(G^{1.5})$ the set of winning states of $G^{1.5}$. By the results of [6, 7], this set can be computed in time quadratic in $|G^{1.5}|$.

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 $s \in Win(G^{1.5})$, we let $\pi^*(s) = Uniform(\{m \in \Gamma_1(s) \mid \forall t \in S.(\tau''(s, m, \bot)(t) > 0 \rightarrow t \in Win(G^{1.5}))\})$. π^* is arbitrarily defined on states $s \in S \setminus Win(G^{1.5})$.

Theorem 2. *The strategy* π^* *is winning in* $G^{1.5}$*, and can be computed in time* $O(|G^{1.5}|^2)$.

In the next section, we present some technical lemmas, that are later used to show the relationships between the different versions of the scheduling game.

3.4 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.

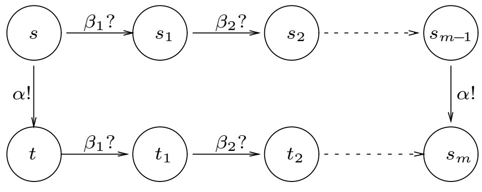


Figure 4: Outputs cannot link winning states to losing ones.

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

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 $s_2 \xrightarrow[i]{\alpha!} t$ and $s_1 \xrightarrow[j]{\beta!} t$ 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 $s \xrightarrow[i]{\alpha!} t$. 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 Figure 4, we can find a path $\sigma = ss_1 \ldots s_m$ 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 $s \xrightarrow[i]{\alpha?} t$ there is $t' \in Win(G^{1.5})$ such that $s' \xrightarrow[i]{\alpha?} t'$ 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 $s' \xrightarrow[i]{\alpha?} t'$. 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 $s' \xrightarrow[i]{\alpha?} s''$, for some α and $i \in \{1, \ldots, n\}$.

  By Lemma 5, there is also a winning step $t' \xrightarrow[i]{\alpha?} t''$, 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 $s' \xrightarrow[i]{\alpha!} s''$, for some α and $i \in \{1, \ldots, n\}$; 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 nondeterminism) 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 $s' \xrightarrow[i]{\alpha!} s''$, for some α and $i \in \{1, \ldots, n\}$; assume also that α appears in σ_i. Then, with positive probability (due to the resolution of inter-thread nondeterminism), 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 nondeterminism can be resolved. ∎

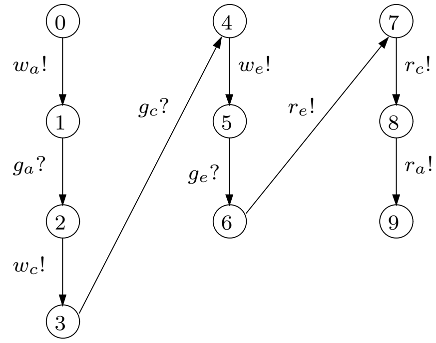


Figure 5: Thread interface from Example 2.

3.5 Comparing Games

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

Theorem 3. *The strategy π^* is winning in game $G^{2.5}$, and $Win(G^{1.5}) = Win(G^{2.5})$.*

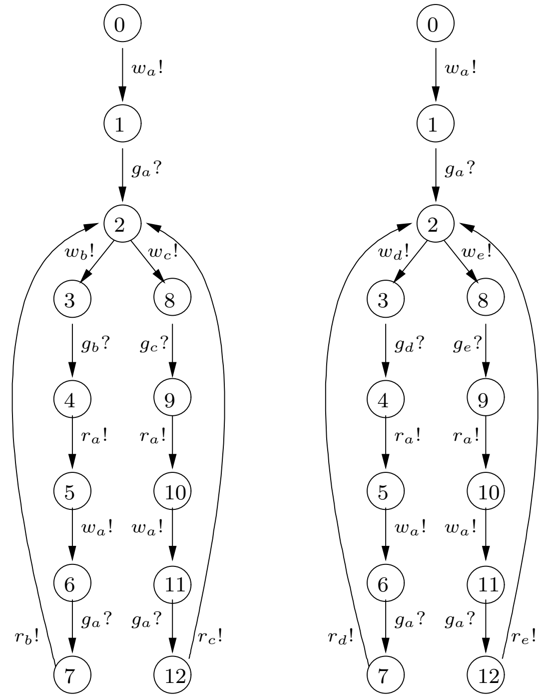
PROOF. For $i \in \{1, \ldots, n\}$ and $s \in Win(G^{1.5})$, we say that thread i is enabled in s if there is an edge $(s,t) \in E$ such that $\theta(s,t) = i$ and $t \in Win(G^{1.5})$. Note that this definition is correct, as by Lemma 4 output edges are always winning.

For $i \in \{1, \ldots, n\}$ 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 $s = s' = s^*$ and $t = t^*$, we obtain that in $G^{2.5}$ 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 nondeterminism 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 $1 \le i \le n$. 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 $\bigwedge_{u \in S_i} \bigwedge_{v \in Osucc_i(u)} \Box\Diamond \mathit{from}_i^u \rightarrow \Box\Diamond \mathit{take}_i^{u,v}$. 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 $balance(x,\sigma) \in \{-1, 0\}$.



(a) (b)

Figure 6: Thread interfaces from Example 2.

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

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 Figure 6(a), I_2 is as in Figure 6(b), and I_3 is as in Figure 5. 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 nondeterminism 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. ∎

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 [5] for Streett MDPs. This leads to the following result.

Theorem 5. *We can check in time $O\big(|G^2|^2 \cdot n \cdot \sum_{i=1}^{n} |E_i|\big)$ 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.

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 Figure 7(a), while I_3 is as in Figure 7(b). Figure 7(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. ∎

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 $\pi^{\mathrm{c}}(s) = \mathit{Uniform}(\mathrm{Supp}(\pi^*(s)) \setminus \{\perp\})$ otherwise. The following result shows that, for PMF systems, π^{c} is an efficient resource manager strategy.

Theorem 6. *In a PMF system, π^{c} is winning for G^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

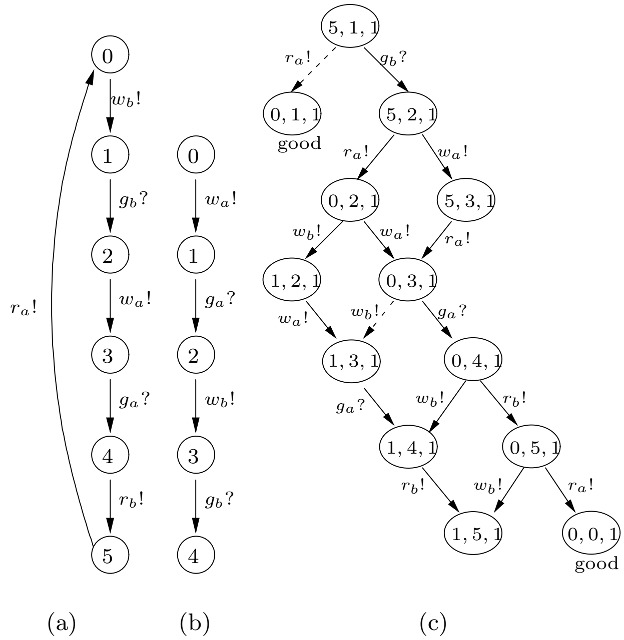


Figure 7: A system where the maximal progress strategy is not winning.

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 $\perp$ (waiting for outputs) only when no thread is blocked: in this way, if the strategy differs from π^* by cutting $\perp$, it can do so only in a winning component. Precisely, for $s \in S$ we let $Succ(s, \pi^*) = \{t \in S \mid \exists m_1 \in \Gamma_1(s).\exists m_2 \in \Gamma_2(s).(\pi^*(m_1) > 0 \wedge \tau(s, m_1, m_2)(t) > 0)\}$ be the set of possible successors of s according to π^*, and we let $S^{\mathrm{b}} = \{s \in S \mid \exists k \in [1..n].\forall t \in Succ(s, \pi^*).\theta(s,t) \neq k\}$ 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 $\pi^{\mathrm{b}}(s) = \mathit{Uniform}(\mathrm{Supp}(\pi^*(s)) \setminus \{\perp\})$ otherwise.

Theorem 7. *The strategy π^{b} is winning in G^2 iff π^* 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 8. *For all $M \in \mathbb{N}$, we have that π^{p}_M is winning in G^2 iff π^* is 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 Section 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 [10], but it can be easily modified to work with another OS.

To extract thread interfaces, the tool uses the CIL library [19] 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.

Currently, 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 Section 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 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 [2], leading to a

n	$\lvert M_{\mathcal{I}} \rvert$	# bad states	BDD nodes	time (sec.)
2	37	3	15	0.05
3	171	18	30	0.07
6	17496	2592	62	39
6	33120	5490	211	334

Table 1: Experiments.

very compact representation. In Table 1, we report the result of some experiments, all run on a 2.4GHz Pentium 4 machine with 512Mb 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.

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 Figure 8, 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 7 mutexes and 8 semaphores. By restricting all queues to having 1 slot, the resulting joint interface contains 400,000 states, and the tool terminates its analysis in about 7 minutes.

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 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 viceversa.

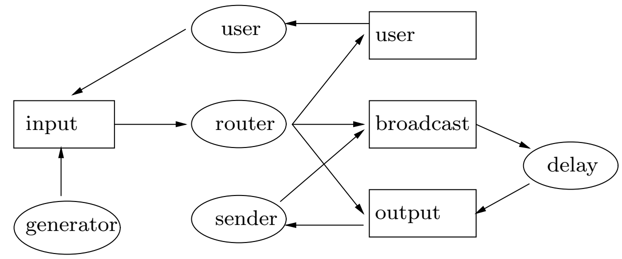


Figure 8: Scheme of an ad-hoc network protocol implementation.

6. REFERENCES

[1] Z.A. Banaszak and B.H. Krogh. Deadlock avoidance in flexible manufacturing systems with concurrently competing process flows. *IEEE Transactions on Robotics and Automation*, 6(6):724–734, 1990.

[2] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[3] G.C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer, 1997.

[4] K. Chatterjee, L. de Alfaro, and T.A. Henzinger. Trading memory for randomness. In QEST 04: *Proceedings of the First International Conference on Quantitative Evaluation of Systems*, pages 206–217. IEEE Computer Society Press, 2004.

[5] K. Chatterjee, L. de Alfaro, and T.A. Henzinger. The complexity of stochastic rabin and streett games. In *ICALP 05: Automata, Languages, and Programming*, Lect. Notes in Comp. Sci. Springer-Verlag, 2005.

[6] L. de Alfaro. *Formal Verification of Probabilistic Systems*. PhD thesis, Stanford University, 1997. Technical Report STAN-CS-TR-98-1601.

[7] L. de Alfaro, T.A. Henzinger, and O. Kupferman. Concurrent reachability games. In *Proc. 39th IEEE Symposium on the Foundations of Computer Science*, pages 564–575. IEEE Computer Society Press, 1998.

[8] C. Derman. *Finite State Markovian Decision Processes*. Academic Press, 1970.

[9] R. Devillers. Game interpretation of the deadlock avoidance problem. *Communications of the ACM*, 20(10):741–745, 1977.

[10] ecos homepage. http://ecos.sourceware.org/.

[11] D.R. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *SOSP 03: Symposium on Operating Systems Principles*, pages 237–252. ACM, 2003.

[12] J. Ezpeleta, J. M. Colom, and J. Martnez. A petri net based deadlock prevention policy for flexible manufacturing systems. *IEEE Transactions on Robotics and Automation., N, 2*, 11:173–184, 1995.

[13] J. Filar and K. Vrieze. *Competitive Markov Decision Processes*. Springer-Verlag, 1997.

[14] F. S. Hsieh and S. C. Chang. Deadlock avoidance controller synthesis for flexible manufacturing systems. *Proc. of 3rd Int. Conf. on Comp. Integrated Manufacturing*, pages 252–261, 1992.

[15] M.V. Iordache, J. Moody, and P.J. Antsaklis.

Synthesis of deadlock prevention supervisors using petri nets. *IEEE Transactions on Robotics and Automation*, 18:59–68, 2002.

[16] C. Kloukinas, C. Nakhli, and S. Yovine. A methodology and tool support for generating scheduled native code for real-time java applications. In *EMSOFT 03: 3rd Intl. Workshop on Embedded Software*, volume 2855 of *Lect. Notes in Comp. Sci.*, pages 274–289. Springer-Verlag, 2003.

[17] C. Kloukinas and S. Yovine. Synthesis of safe, qos extendible, application specific schedulers for heterogeneous real-time systems. In *ECRTS 03: Euromicro Conference on Real-Time Systems*, pages 287–294. IEEE Computer Society Press, 2003.

[18] Toshimi Minoura. Deadlock avoidance revisited. *Journal of the ACM*, 29(4):1023–1048, 1982.

[19] G.C. Necula, S. McPeak, S.P. Rahul, and W. Weimer. Intermediate language and tools for analysis and transformation of C programs. In *Proc. Conference on Compiler Construction (CC)*, 2002.

[20] J.L. Peterson and A. Silberschatz. *Operating System Concepts*. Addison-Wesley, 1985.

[21] S. Savage, M. Burrows, C.G. Nelson, P. Sobalvarro, and T.A. An derson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[22] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 4, pages 135–191. Elsevier Science Publishers (North-Holland), Amsterdam, 1990.

[23] W. Thomas. On the synthesis of strategies in infinite games. In *Proc. of 12th Annual Symposium on Theoretical Aspects of Computer Science*, *Lect. Notes in Comp. Sci.*, pages 1–13. Springer-Verlag, 1995.

[24] J.R. von Behren, J. Condit, F. Zhou, G.C. Necula, and E.A. Brewer. Capriccio: scalable threads for internet services. In *SOSP 03: Symposium on Operating Systems Principles*, pages 268–281. ACM, 2003.