Autom Softw Eng (2011) 18: 5–38
DOI 10.1007/s10515-010-0074-8

A model-based approach for multiple QoS in scheduling: from models to implementation

Christos Kloukinas · Sergio Yovine

Received: 11 March 2010 / Accepted: 16 August 2010 / Published online: 4 September 2010

Abstract Meeting multiple Quality of Service (QoS) requirements is an important factor in the success of complex software systems. This paper presents an automated, model-based *scheduler synthesis* approach for scheduling application software tasks to meet multiple QoS requirements. As a first step, it shows how designers can meet deadlock-freedom and timeliness requirements, in a manner that (i) does not overprovision resources, (ii) does not require architectural changes to the system, and that (iii) leaves enough degrees of freedom to pursue further properties. A major benefit of our synthesis methodology is that it increases traceability, by linking each scheduling constraint with a specific pair of *QoS property* and underlying *platform execution model*, so as to facilitate the *validation* of the scheduling constraints and the *understanding* of the overall system behaviour, required to meet further QoS properties.

The paper shows how the methodology is applied in practice and also presents a prototype implementation infrastructure for executing an application on top of common operating systems, without requiring modifications of the latter.

Keywords CASE · Model checking · Process management · Real-time systems and embedded systems

1 Introduction

The importance of QoS and, in general *quantitative non-functional* requirements, e.g., computational power, memory size, power consumption, etc., is increasing every

C. Kloukinas (✉)
City University London, London, EC1V 0HB, UK
e-mail: C.Kloukinas@soi.city.ac.uk

S. Yovine
CONICET and Universidad de Buenos Aires, Buenos Aires, Argentina
e-mail: syovine@dc.uba.ar

day, as computer systems move far away from the scientific and office applications of the past. Designers face these QoS requirements in many domains—from mobile embedded systems running on batteries to big server farms, like those of Google (Barroso et al. 2003) and of Second Life, whose avatars are estimated to consume "considerably more" electricity than people in developing countries do (Carr 2006). As sometimes these QoS requirements are conflicting, it is imperative to develop analysis methods that enable designers to tackle complex systems and their multiple QoS requirements (Fohler and Buttazzo 2002; Henzinger and Sifakis 2007), without imposing artificial constraints on either the design or the implementation.

One such type of requirement is real-time (R-T), supported by specialised OS like VxWorks and commodity OS, up to commercial, large-scale Java VMs like Sun RTS and IBM WebSphere. From embedded systems, such as the anti-lock breaking system in automobiles, to big distributed systems, such as those in the financial market, designers need to guarantee timeliness for the various tasks. Some of these systems are *mission critical*—performing an action at an unsuitable time can lead to a big financial loss, as is the case with the financial markets, where "A common Wall Street belief is that for every millisecond an investment bank can beat the market, it has the potential to earn an additional $100 million per year" (Bruno 2007). Even worse, others are *safety critical*—an untimely computation can lead to loss of human life. For this reason, R-T is one of the most important QoS requirements. However, system design and validation remains a difficult and error-prone task. Often, designers have to be overly pessimistic in estimating the demands on system resources, which leads to systems that have fewer capabilities and are more expensive than need be.

Classic R-T scheduling approach The general scheduling problem cannot be solved in polynomial time (Garey and Johnson 1977; Baruah et al. 1990). In order to have techniques that can be applied in practice (and in some cases at run-time), the classic approach to R-T scheduling abstracts the complexity of a software system's tasks into a model that is simple enough to permit an analysis that is of polynomial complexity and extremely fast. However, these classic (and still widely used) scheduling analyses for R-T, such as Rate Monotonic Analysis (RMA) or Earliest Deadline First (EDF) (Liu and Layland 1973) that are typically combined with the Priority Inheritance/Ceiling synchronisation Protocols (PIP/PCP) (Sha et al. 1990), make assumptions that are hardly realistic nowadays. For example, they assume that task periods equal their deadlines, that task importance is directly associated with its deadline, that non-periodic tasks are not critical, or that the goal is to derive a deterministic scheduling policy (which greatly hinders further optimisation). At the same time, these analyses are flow-insensitive and they ignore the order in which shared resources are acquired. Furthermore, the loss of detail in the models means that more resources are needed for guaranteeing timeliness, even though this is unnecessary. This is also true for extensions handling task dependencies, such as the Harbour-Klein-Lehoczky (HKL) response-time analysis (Harbour et al. 1994). For instance, Tripakis (2002) reported that the (manually performed) HKL-based analysis failed to safely schedule an automated vehicle control software, whereas Tripakis and Yovine (2001) succeeded using the automata-theoretic Model-Driven Engineering (MDE) framework TAXYS (Closse et al. 2001).

Another problem is that it is not obvious how to extend classic scheduling analyses, so as to *easily* meet further QoS requirements, because these analyses are *rigid*. While the response-time analysis of HKL allows the use of (fixed) task priorities for meeting other system concerns in principle, the majority of the schedulability analysis theory is concerned with how to assign priorities for meeting deadlines. By focusing only on deadlines and by deriving deterministic schedulers, they remove all degrees of freedom that could be used to control for other QoS. Thus, classic scheduling analyses effectively require designers to either restructure their system or to try to work against the analysis method, by artificially changing some of the inputs it considers, e.g., the worst case execution time (WCET) or periodicity of a task. Restructuring is not an easy task—not only because of the transformation steps themselves but also because it is not clear what the transformation goal should be for a given system. Indeed, designers are called upon to correctly identify which analysis theory among the massive (and ever-growing) body of knowledge is able to faithfully analyse the systems of their domain (Bordin et al. 2008). This is a big problem because violating some of the analysis assumptions or constraints may lead to results that are overly pessimistic. Worse even, invalidating the analysis assumptions may lead to incorrect results and a system that behaves differently from the analysis prediction. Given that systems are becoming heterogeneous, trying to continuously adapt the basic theory to accommodate each and every new sub-case is a Sisyphean task—asking software engineers to search among these cases for the one that best matches their needs and master it so that they can apply it in practice is simply unrealistic. Indeed, even the simplest of the classic techniques, RMA, needs to be applied with care—a book for practitioners that was published in 1993 is 712 pages long (Klein et al. 1993).

A model-based approach to scheduling This situation calls for a more automated framework based on a generic solution instead of a case-by-case one. An appealing approach in this direction is to *automatically synthesise* a scheduler (Altisen et al. 2002) for a *model* of the system given in an automata-based formalism. This solution can be part of a MDE framework (Schmidt 2006) and can greatly reduce the effort required by designers. Thanks to the ongoing advances in model-checking, this alternative is becoming more and more interesting for real-world systems. Indeed, scheduler synthesis is now tractable for many systems and, as such, one can forgo the limited models of the classic schedulability theory and consider far richer models that capture more details of a system. Here, we show how one can perform such a *fine-grained* analysis of systems and implement these in a way that *guarantees* safety properties (i.e., deadlocks, deadlines) and that can furthermore be *easily extended* to support other quality aspects of the system (e.g., jitter, memory, energy). We show how one can achieve these goals through a new methodology that increases the applicability and benefits of scheduler synthesis (Kloukinas and Yovine 2003), which has two goals. First, it reduces the state space scheduler synthesis has to explore so that the inherent *complexity* of synthesis does not render it impractical. Second, it links better the synthesised scheduler constraints with the properties that they are guaranteeing, so as to achieve *traceability* of the constraints to the properties. It achieves so by synthesising successive scheduler layers for guaranteeing different QoS requirements, considering a number of system models and platform execution policies. Thus, each synthesised scheduler is linked directly to a *specific QoS property* and *platform execution policy*, making it easier to *understand* and *validate* the schedulers themselves, as well as the system behaviour under various operational conditions. Our system analysis and scheduler synthesis methods do not make any assumptions on the tasks comprising the system, their periodicity, their synchronisation patterns, etc. Thus, they are easily applicable to systems where classic scheduling analysis proves to be problematic. Indeed, one of the main advantages of scheduler synthesis is that the application does not need to be restructured to facilitate the analysis and control of the system, nor does it require designers to make error-prone choices, such as where to enable PIP, that can lead to problems like those faced by the Mars Pathfinder robot (Reeves 1997).

Structure of the paper In the following, we present our system and scheduler architecture and a simple case study that we use to illustrate the various notions introduced. Then we detail our approach for modelling the QoS (here R-T-related) requirements of a system and for synthesising a scheduler for it. We follow with an in-depth presentation of scheduler synthesis in practice, showing how one can do such a task gradually, in order to better understand the resulting schedulers, analyse the system under different assumptions/conditions and better tolerate the inherent state-space explosion problem at the same time. We then introduce a more complex case study, on which we apply our scheduler synthesis approach. This is followed by the description of the implementation of a library of synchronisation and communication primitives, which allows the use of synthesised schedulers in currently available OS. We then consider the subject of robustness, i.e., what happens when our modelling assumptions do not hold, for example when an operation finishes executing later (or indeed earlier) than expected. Even though this is a subject that has to do with how the system is modelled and not with scheduler synthesis per se, it is very important to know how a system will behave when the modelling assumptions are invalid and what can be done to render it more robust. Finally, we compare our work with other related approaches, before finishing with a concluding discussion.

2 Overall system architecture

The overall system architecture we consider is depicted in Fig. 1(a), where arrows show the interactions among the different system components. The application code is *instrumented* so as to be able to *observe and control* it. The instrumentation code keeps track of the state of the application and intercepts application requests for lower level mechanisms of interest for the scheduler (e.g., synchronisation, communication). The intercepted requests are redirected to a subsystem which is responsible for controlling the system. This *scheduler subsystem* uses a number of *application specific scheduler constraints* to make a decision about whether it should block the application requests, when they may lead to an unsafe (or suboptimal) system state, or forward them to the underlying OS. Finally, the OS primitives are effectively our means to *observe and interact* with the environment and the application. This is a

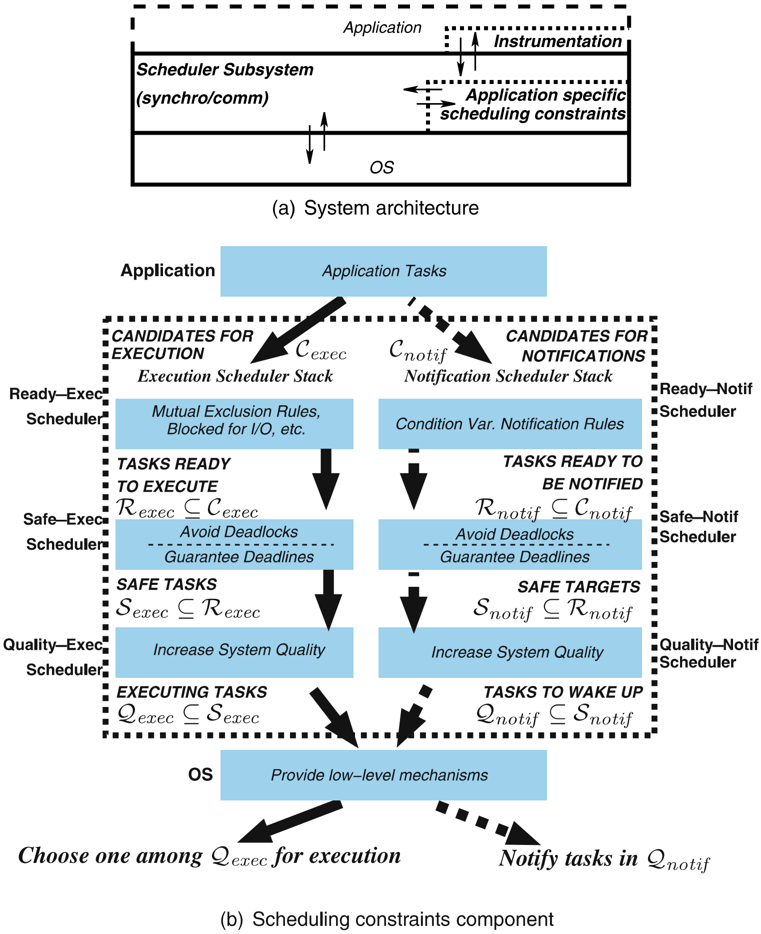


Fig. 1 Architecture of the system and the scheduler

general system architecture for uni-processor reactive systems and closed-loop control. In the RMA/PCP framework, the "instrumented" synchronisation, etc. primitives use a set of scheduler constraints that together form the RMA/PCP priorities. In our context, the scheduling constraints are automatically synthesised from an automata based model of the application and its environment.

The controllable primitives (e.g., request and release of shared resources) are the synchronisation by means of monitors (**`monitorEnter`**, **`monitorExit`**) and the communication by means of condition variables through notification, broadcasting, waiting for a notification and waiting for a notification until some timeout (**`notify`**, **`notifyAll`**, **`wait`**, **`timed_wait`**), with the well-known POSIX (IEEE. POSIX.1. 2001) or Java (Joy et al. 2000) semantics. Finally, **`awaitPeriod`** causes tasks to wait for their next period.

2.1 Scheduler architecture

The architecture of the component called *application-specific scheduling constraints* in Fig. 1(a) is depicted in Fig. 1(b). As shown there, the application tasks make some request that is forwarded to one of two scheduler stacks. The right stack elects a task as a target for a pending signal/notification and is executed when a task performs a **`notify`**, while the left scheduler stack is responsible for electing some application task for execution and is executed when a task performs any other scheduling primitive. Both of these stacks have the same structure; they are effectively subdivided into three main layers. The topmost scheduler layers (*Ready-Exec*, resp. *Ready-Notif*) identify application tasks which are eligible either for execution ($\mathcal{R}_{exec}$) or for notification ($\mathcal{R}_{notif}$). They effectively model in user-space the ready queue of the OS and its waiting-for-notification object queues respectively. The middle layers (*Safe-Exec*, resp. *Safe-Notif*) are the most important in a critical system. They elect among the eligible tasks those that will not lead the system to a bad state (task sets $\mathcal{S}_{exec}$ and $\mathcal{S}_{notif}$ respectively). That is, the middle layers are responsible for guaranteeing the *safety* properties of the system (e.g., deadlock-freedom, meeting deadlines). Finally, the lower layers (*Quality-Exec*, resp. *Quality-Notif*) are responsible for imposing further constraints, which are needed for guaranteeing other QoS, e.g., jitter minimisation, power consumption minimisation, etc. The sets of safe tasks meeting these further quality constraints ($\mathcal{Q}_{exec}$, resp. $\mathcal{Q}_{notif}$) form the final output of the application dependent scheduler constraints subsystem. The scheduler subsystem passes them to the OS, which chooses tasks for execution, resp. notification, using some OS dependent rule. From the point of view of the scheduler, the OS choice is *non-deterministic*. It is exactly this non-determinism that allows designers to easily explore further scheduling strategies for extra QoS requirements. In fact, the scheduler subsystem does not even assume that the OS choice is fair. For tasks with deadlines, fairness will be imposed by the scheduling constraints for timeliness. For tasks with no deadlines, fairness (if required) can be imposed by the use of some QoS scheduling policy, e.g., Round-Robin.

Our scheduler architecture has two different stacks for execution and communication so as to explicitly control task communication as well. This is an aspect which is usually not considered by other approaches, since it is assumed that the system designers have already solved all communication problems. Nevertheless, we believe that, given its complexity, the scheduler should explicitly cover this aspect as well.

The two scheduler stacks in Fig. 1(b) are *exclusive* iff *we are interested in deadlock-freedom*, where notifications are handled by the right stack alone. This is because the notified task will not be executable, since it must reenter the monitor that is still occupied by the notifier. However, the left stack needs to be given control after notifications when scheduling for deadlines and other QoS, so as to ensure some hard to meet deadline/constraint by preempting the notifying task.

2.1.1 Increasing system quality

As aforementioned, the bottom *Quality-Exec* and *Quality-Notif* layers of the scheduler, allow designers to easily experiment with and introduce additional constraints for increasing the quality of the system. Software engineers control the complexity of these layers directly and can employ a *best-effort* policy or a more contract-like QoS one, where specific bounds for certain values of the system state must be guaranteed. In the latter case, the QoS policy must be verified as a safety policy, to ensure that the system will never break its QoS contract.

A simple example of a (best-effort) quality policy is the *local minimisation of context switches* (LMCS), in order to speed-up the execution and (hopefully) minimise cache misses/flushes and, thus, also power consumption. This policy can be implemented quite easily, by examining whether the currently executing task, t_i, is in the set $\mathcal{S}_{exec}$ of tasks which are safe to execute next. If this is the case, then we can let it continue its execution, by setting the set $\mathcal{Q}_{exec}$ equal to the singleton $\{t_i\}$. Note, that LMCS differs from a non-preemptive platform execution policy, since LMCS allows preemption when the currently executing task is not in the safe set.

3 A simple system

This section introduces the simple system of Fig. 2(a), to be used for illustrating the various notions through concrete examples. The system consists of the Writer, User and Refresher tasks. The Writer produces values for variable V continuously (e.g., by reading a sensor or retrieving a stock price), which the periodic task User consumes. The production of these values takes place in the transition `W1->W2`, where the `write` computation is performed. As indicated by the interval `[2,3]`, this computation takes between 2 and 3 time units. However, User needs the values of V to be "fresh", i.e., they must have been produced recently and as such represent the current state of the environment. For that, the Refresher task uses an auxiliary variable L, to distinguish values of V that are too old, from these that are fresh enough for User. It does so by marking the current value of V as not fresh and then waiting for 13 time units. If the Writer produces a new value for V during that time, the freshness of V will be true, otherwise it will be false.

Notice that, as they obtain V and L in the opposite order, there is a potential deadlock between Writer and Refresher that arises when the Writer is at state W2 and the Refresher at state R3.

Remark In our prototype tool, *the control-flow diagrams* as the one in Fig. 2(a) are expressed in the simple language of the dot graph drawing tool from the Graphviz package (Gansner and North 2000). The main reason for this is that we do not expect our language to be used by software engineers directly, but automatically generated from application software code in the context of a full-fledged MDE framework. This was done in Kloukinas et al. (2003) and in Assayad et al. (2005) for annotated Java and C programs, respectively. At the same time, having a very simple language means that designers who wish to quickly explore an initial design can do it easily, without having to master a new complex language. Design changes at the program code level would be in general far more expensive—that level is better suited for fine-tuning through the use of extra information, such as additional application variables that could be taken into consideration by the scheduler (e.g., type of data to be processed).

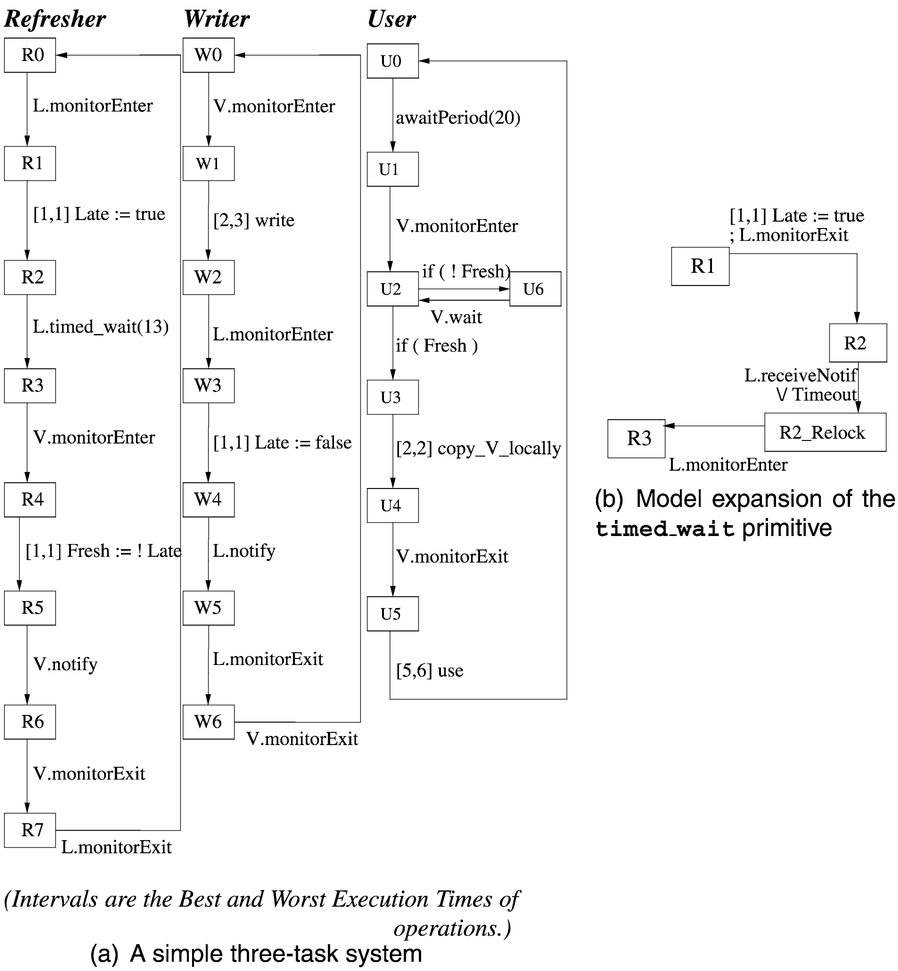


Fig. 2 A simple system model and the internal model expansion for one of its waiting primitives

4 System modelling

This section presents our modelling of a system through *discrete-time stopwatch automata*. Stopwatch automata (SA), called *integration graphs* in Kesten et al. (1999), allow for fine-grained modelling, thus permitting us to synthesise a flow-sensitive and not over-constraining scheduler, which needs fewer resources to meet requirements. The discrete time SA we are using are normal finite-state automata, where certain variables serve as *discrete time* clocks. The difference between SA and timed automata (Alur and Dill 1994) is that SA can stop certain clocks (without resetting them) and restart them later on. Thus, SA can model preemption. Since continuous-time SA are known to be undecidable in general (Kesten et al. 1999), in this work we resort to discrete-time SA.

4.1 Application modelling

As aforementioned, we consider that the application comprises a set of concurrently executing asynchronous tasks, $\mathcal{T} = \{t_i\}_{i \in I}$, where I is the set of task indexes. Tasks can synchronise through monitors, communicate through condition variables, wait for their next period or perform a computation.

Each task uses two clocks to model time-related behaviour. The first clock, SW_i, models the duration of computations of task t_i and so *is stopped* when the computation is preempted. SW_i is also used when a task performs a **`timed_wait`**, to measure the distance till the timeout. The second clock, C_i^{Periodic}, measures the time remaining until the next period (or deadline) of a task and *is never stopped*; it is only *reset* at each new period.

4.1.1 Expansion of application models

One thing that is not shown in the input model is the meaning of the **`wait`** and **`timed_wait`** primitives. These are directly linked with the reservation and release of shared resources, so we need to model them carefully. These communication primitives (notify/wait) must, by definition, be used inside a critical section/monitor. So, in order to notify some task that resource r has been modified, the notifying task must enter the monitor of r (r.**`monitorEnter`**), notify tasks interested in events about this resource (r.**`notify/All`**) and subsequently leave the monitor (r.**`monitorExit`**). Tasks interested on events for resource r, must enter its monitor, wait for an event (r.**`wait/timed_wait`**), treat the event in an application specific manner and then leave the monitor. Note that **`wait`** primitives force waiting tasks out of the corresponding monitor, so as to allow notifying tasks to enter it, and then attempt to re-enter the corresponding monitor once the task has been notified.

This behaviour is critical for the scheduler, since waiting on a notification releases and then re-acquires a lock on a shared resource. So, **`wait`** primitives are expanded to two states (R2, R2_Relock), as Fig. 2(b) shows using part of the Refresher's model from Fig. 2(a). The transition from the previous state to the first one (R1 → R2) also causes the task to leave the monitor, after having executed the action the program was performing there (`[1,1] Late := true`), as shown in Fig. 2(b). Then, the transition from the first to the second wait states (R2 → R2_Relock in Fig. 2(b)) waits for a notification (or a timeout if it is a **`timed_wait`**). Once a task is notified, it attempts to fire the transition from the second wait state to the subsequent program state (R2_Relock → R3 in Fig. 2(b)), so as to reenter the monitor and continue its execution.

4.2 System state

The system state model comprises: (i) an *abstract* program counter (PC_i) for each of the application tasks; (ii) a stopwatch (SW_i) for each task; (iii) N periodic clocks (C_i^{Periodic}), for the N periodic tasks, taking values over the interval $[0, P_i)$, where P_i is the period of the task; (iv) N Boolean variables (`task`_Alarm), for dissociating the cases "start of period" and "deadline/end of period", since for some tasks we may have $D_i = P_i$ (see Sect. 7); (v) a variable (T_{Exec}) for the currently executing task or IDLE when no task is executing; (vi) a 4-valued variable (`mode`) controlling which of the **`SchedExec`**, **`SchedNotif`**, **`Timeout`**, or one of the **`Application`** automata should execute in the current step (these automata are described in the following section); and (vii) the *Boolean* variables of the application guarding waiting statements and branches, if we wish to model them.

Example 1 For the system of Fig. 2(a), the variables are:
$\mathrm{PC}_{\mathrm{Writer}} \in \{\mathrm{W0}, \mathrm{W1}, \mathrm{W2}, \mathrm{W3}, \mathrm{W4}, \mathrm{W5}, \mathrm{W6}\}$,
$\mathrm{PC}_{\mathrm{Refresher}} \in \{\mathrm{R0}, \mathrm{R1}, \mathrm{R2}, \mathrm{R3}, \mathrm{R2\_Relock}, \mathrm{R4}, \mathrm{R5}, \mathrm{R6}, \mathrm{R7}\}$,
$\mathrm{PC}_{\mathrm{User}} \in \{\mathrm{U0}, \mathrm{U1}, \mathrm{U2}, \mathrm{U3}, \mathrm{U6}, \mathrm{U4}, \mathrm{U5}, \mathrm{U6\_Relock}\}$,
$SW_{\mathrm{Writer}} \in [0, 6]$, $SW_{\mathrm{Refresher}} \in [0, 13]$, $SW_{\mathrm{User}} \in [0, 6]$, $C_{\mathrm{User}}^{\mathrm{Periodic}} \in [0, 19]$,
$\mathrm{User\_Alarm} \in \{\mathrm{false}, \mathrm{true}\}$, $T_{Exec} \in \{\mathrm{IDLE}, \mathrm{Writer}, \mathrm{Refresher}, \mathrm{User}\}$,
$\mathrm{mode} \in \{\mathrm{Sched\text{-}Exec/Notif}, \mathrm{Timeout}, \mathrm{Application}\}$,
$\mathrm{Late} \in \{\mathrm{false}, \mathrm{true}\}$, $\mathrm{Fresh} \in \{\mathrm{false}, \mathrm{true}\}$.

4.3 Model structure and execution modes

The system model we construct is the synchronous composition of automata with shared execution of commonly labelled transitions of:

- the *Timeout* automaton which fires timeouts,
- the *Execution* and *Notification* Scheduler automata, and
- one automaton for each of the application tasks.

Application automata are derived from control-flow diagrams describing the application tasks.

The system operates in three modes, as can be seen in Fig. 3(a). In **`Timeout`** mode the *Timeout* automaton (shown in Fig. 3(b)) is the only one enabled in the system. It can fire one or more timeouts (corresponding to a **`timed_wait`** or **`awaitPeriod`** expiring) if any is enabled currently. When a timeout is fired, mode changes to "Schedulers Only" (where `mode` = **`SchedExec`**), so that our scheduler can handle it. If there is no timeout to be fired then the mode changes to "Application" (where `mode` = **`Application`**). At this mode, the automaton of the T_{Exec} application task becomes enabled. If the T_{Exec} task needs to execute a time guarded action (i.e., a computation), then it causes time to advance by performing a tick (i.e., a time step). The tick action causes all periodic clocks (C_i^{Periodic}) to advance at the same time. It also causes local stopwatches (SW_i) to advance, if the respective task is executing, i.e., $T_{Exec} = t_i$, or if it is performing a **`timed_wait`**. Ticks change mode back to **`Timeout`**, so as to check for new timeouts. If, however, T_{Exec} needs to perform an action which causes re-scheduling, then it passes control back to the schedulers, i.e., mode becomes "Schedulers Only" (`mode` $\in$ {**`SchedExec`**, **`SchedNotif`**}). Initially `mode` is **`Timeout`**, for periodic tasks to start their first period.

Remark It should be noted that the only activity that can cause time to advance is a computation by the application or waiting for an event. Indeed, the model assumes that scheduling decisions (i.e., the transitions of the scheduler automaton) take no

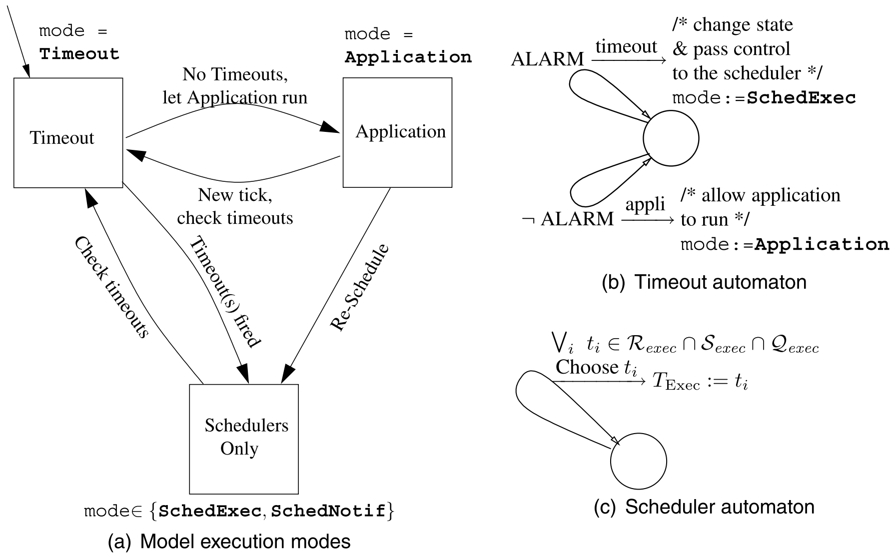


Fig. 3 Execution modes, Timeout and Execution Scheduler automata

time at all. This means that the (rather small) time needed for scheduling needs to be budgeted within the duration of computations. Another alternative modelling strategy would be to explicitly account for the execution time of the scheduler in the model. This could be achieved by a simple modification of the scheduler automaton by adding explicit time intervals to its transition.

5 Scheduler synthesis

Scheduler synthesis is a two-player game (Asarin et al. 1995). For each scheduler action (i.e., selection of an application task), there is a sequence of actions of its adversaries (i.e., timeout and application automata), and so on. In this game, scheduler synthesis amounts to finding a *winning strategy* for each state where it is the turn of the scheduler to act, if any such strategy exists indeed. That is, whenever the scheduler is called to perform a *controllable* action, it must have a plan that informs it which future *uncontrollable* actions of its adversaries it should render impossible, in order for the system to remain in a safe state. Thus, the scheduler synthesis problem can briefly be stated as "for each control state, find the environment actions that must be rendered impossible for the system to always remain in a safe (optimal) state." In our case, we have two layers which are needed for guaranteeing safety—the top one (*Ready-Exec* & *Ready-Notif*) and the middle one (*Safe-Exec* & *Safe-Notif*).

5.1 Synthesis of the ready task layer

We synthesise the top layer through a simple static analysis of the task control-flow graphs. This assigns to each of the task states, $N(t)$, the resources it holds and the ones it wishes to lock, constructing two different sets for each task: one stating when task t wants to lock a resource r, $\mathrm{WR}(r,t)$, and another one stating when the task has the resource locked, $\mathrm{LR}(r,t)$:

$$\mathrm{WR}(r,t) = \{n \in N(t) \mid \exists n_1 \in N(t) \,.\, n \xrightarrow{r.\texttt{monitorEnter}} n_1\} \tag{1}$$

$$\begin{aligned}\mathrm{LR}(r,t) = \{n \in N(t) \mid\ & \exists n_1, n_2 \in N(t) \\ & .\ (n_1 \xrightarrow{r.\texttt{monitorEnter}} n_2 \to^* n) \\ & \wedge (\nexists n_3, n_4 \in N(t) \,.\, n_2 \to^* n_3 \wedge n_4 \to^* n \\ & \wedge n_3 \xrightarrow{r.\texttt{monitorExit}} n_4)\}\end{aligned} \tag{2}$$

States with a **wait** transition are expanded as in Fig. 2(b). For the WR and LR sets, these are equivalent to a **monitorExit** and then a **monitorEnter** on the resource. The $i \times j$ sets produced, for the i program counters and j resources, inform us whether a task is blocked or not, which is needed for the top *Ready-Exec* and *Ready-Notif* scheduler layers.

Example 2 For the system of Fig. 2(a), these sets are:
$\mathrm{WR}(V, \mathrm{Writer}) := \{W0\}$, $\mathrm{WR}(L, \mathrm{Writer}) := \{W2\}$,
$\mathrm{LR}(V, \mathrm{Writer}) := \{W1, W2, W3, W4, W5, W6\}$, $\mathrm{LR}(L, \mathrm{Writer}) := \{W3, W4, W5\}$,
$\mathrm{WR}(V, \mathrm{Refresher}) := \{R3\}$, $\mathrm{WR}(L, \mathrm{Refresher}) := \{R0, R2\_Relock\}$,
$\mathrm{LR}(V, \mathrm{Refresher}) := \{R4, R5, R6\}$, $\mathrm{LR}(L, \mathrm{Refresher}) := \{R1, R3, R4, R5, R6, R7\}$,
$\mathrm{WR}(V, \mathrm{User}) := \{U1, U6\_Relock\}$, $\mathrm{LR}(V, \mathrm{User}) := \{U2, U3, U4\}$,
$\mathrm{WR}(L, \mathrm{User}) := \mathrm{LR}(V, \mathrm{User}) := \emptyset$.
Potential deadlocks are easily identified by considering the intersection of the sets WR and LR. Indeed, there is a *potential* deadlock at states (W2, R3,*) because we have that:
$\mathrm{WR}(L, \mathrm{Writer}) \cap \mathrm{LR}(V, \mathrm{Writer}) = \{W2\}$ and
$\mathrm{WR}(V, \mathrm{Refresher}) \cap \mathrm{LR}(L, \mathrm{Refresher}) = \{R3\}$.

Sets WR and LR also show which are the tasks that cannot be involved in a deadlock, e.g., here the User task.

5.1.1 Potential vs real deadlocks and over-constraining

Of course one cannot be certain that *potential* deadlock states identified through the WR and LR sets are *real*, until they are shown to be *reachable*. Attempting to render them unreachable by enclosing the corresponding critical regions inside a new monitor (e.g., enclose each of $R1$–$R7$ and $W0$–$W6$ inside a monitor on a new resource D) will certainly remove any chance for *that* deadlock but will unnecessarily decrease the degree of concurrency in the system, especially so if the deadlock is unreachable. In fact, for the system of Fig. 2(a), such a solution would be inadvisable as it is too constraining. Indeed, if we attempted such a solution, then variable Fresh would never become true and User would *never* finish its period (states U3, U4, and U5 would become unreachable). The reason for this is that whenever Refresher gains access to D, it will set Fresh to false and remain in the monitor of D while waiting, thus not allowing Writer to reset Fresh. This is like a financial system ignoring all stock values as too old or a web server dropping all client requests. Thus, we can see that this solution, advocated in Wang et al. (2009) for its simplicity, can break the application logic itself by removing too many valid execution traces. The problem identified with the approach of Wang et al. (2009) is a more general one—constraining against deadlocks (or for some other reason) can cause other properties to become invalid. For example, general liveness properties may no longer hold, i.e., a task may never advance. In our case, the introduction of timing constraints such as deadlines, make liveness properties to be bounded in time, and therefore reduce to safety ones. Indeed, one no longer is concerned whether the system will eventually react to a stimulus but whether it will do so before a deadline. As such, we do not have to check for liveness properties explicitly. This simplifies our synthesis tool since the synthesis procedure for liveness properties is different to that for safety ones, almost its dual (Tripakis and Altisen 1999). However, a designer should always at some point verify that both general liveness properties and any other application-specific properties hold, e.g., that Fresh will be true at some point when the User task checks it. Working in a model-based setting allows designers to easily verify such properties through model-checking—something which the classic schedulability theory cannot do. This is one of the benefits of an automata-theoretic model-based approach.

5.2 Synthesis of the safe task layer

The basic method for synthesising the *Safe-Exec* and *Safe-Notif* scheduler layer, starts by first constructing the set of reachable states and, thus, identifying the bad states. These are the states where the application tasks are deadlocked, or the states where some task has missed its deadline. Having bad states means that the current set $\mathcal{S}_{exec}$, of tasks that are safe to execute at a state s, needs to be constrained. The only *controllable* actions that can be constrained in the system are the transitions of the scheduler automata, shown in Fig. 3(c). $\mathcal{S}_{exec}$ is initially **`true`**, thus accepting all tasks in the set $\mathcal{R}_{exec}$ as safe. Having obtained the bad states, we do a backwards traversal of the state space starting from the bad states, until we reach a state, s, which corresponds to a *controllable* choice of one of the scheduler automata. There, we identify the controllable transition a outgoing from s which sets T_{Exec} to be task t_a, effectively enabling the path leading to a bad state, and create a new constraint for the layer *Safe-Exec* at state s for the controllable transition a. The constraint is constructed by changing the set $\mathcal{S}_{exec}$ to be:

$$\mathcal{S}'_{exec}(s) := \mathcal{S}_{exec}(s) \setminus \{t_a\} \tag{3}$$

If at some point we find that $\mathcal{S}'_{exec}(s)$ becomes equal to the empty set after constraining it, that is, if there is no safe task to execute at state s, then we also mark the state s as bad and continue the synthesis procedure.

So, the set of states where a task t is unsafe to execute is:

$$\text{Unsafe}(t) = \{s | t \in \mathcal{R}_{exec}(s) \wedge \neg \mathcal{S}_{exec}(s)\} \tag{4}$$

Table 1 Refresher timeliness constraints

(a) Refresher constraints when observing clocks and allowing preemption

$$
\begin{array}{l}
\texttt{LET } \mathbf{A} = (PC_{User}{=}\texttt{U5} \ \wedge\ C_{User}^{Periodic}{=}\texttt{13})\\
\texttt{IN } (\mathbf{A} \wedge \quad (PC_{Refresher}{=}\texttt{R0} \ \wedge \quad (PC_{Writer}{=}\texttt{W0})\\
\qquad\qquad\qquad\qquad\qquad \vee\ (PC_{Writer}{=}\texttt{W2} \ \wedge\ ((\quad T_{Exec}{=}PC_{Refresher}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee\ T_{Exec}{=}PC_{Writer})))\\
\qquad\qquad\qquad\qquad\qquad \vee\ (PC_{Writer}{=}\texttt{W6} \ \wedge\ T_{Exec}{=}PC_{Writer}))\\
\qquad \vee\ (PC_{Refresher}{=}\texttt{R3} \ \wedge \quad PC_{Writer}{=}\texttt{W0})
\end{array}
$$

(b) Refresher constraints when not observing clocks

$$(PC_{Refresher}{=}\texttt{R7} \ \wedge\ PC_{User}{=}\texttt{U1} \ \wedge\ PC_{Writer}{=}\texttt{W1} \ \wedge\ T_{Exec}{=}PC_{Writer})$$

Example 3 This set is expressed as a predicate over model variables. Table 1(a) shows the synthesised predicate for Refresher for ensuring the timeliness property of User (i.e., its period is never violated). The constraints essentially forbid Refresher from executing when User is about to miss its deadline (e.g., at $U1 \wedge C_{User}^{Periodic} = 11$), since Refresher would consume computational resources and/or invalidate the current value of V, in which case the User would need to wait for a new fresh value to be produced.

5.2.1 Partial state observability

In reality, the scheduler cannot observe the full state of the system. That is, the scheduler uses an observation function, obs, presenting it with a partial view of the current system state. Our default assumption is that the scheduler sees at most the values of the task program counters, PC_i, and those of the clocks, i.e., SW_i and $C_i^{Periodic}$, along with the value of the last task that was executing, T_{Exec}. All other system variables are hidden to it. The scheduler can observe these variables only, so that the instrumentation of the application will be minimal and easy to perform in practice, though system designers are free to enlarge the observation set. So the scheduler synthesis procedure really uses (5) and (6), rather than (3) and (4):

$$\mathcal{S}'_{exec}(\text{obs}(s)) := \mathcal{S}_{exec}(\text{obs}(s)) \setminus \{t_a\} \tag{5}$$

$$\text{Unsafe}(t) = \{s \mid t \in \mathcal{R}_{exec}(\text{obs}(s)) \wedge \neg \mathcal{S}_{exec}(\text{obs}(s))\} \tag{6}$$

Example 4 Again for the system of Fig. 2(a), the constraints we synthesise to render the system deadlock-free, once we have applied the projection on the state variables are:

Unsafe(User) := FALSE (*i.e.,* User *is always safe*),
Unsafe(Writer) := $(PC_{Writer} = W0) \wedge (PC_{Refresher} = R3)$,
Unsafe(Refresher) := $(PC_{Refresher} = R2\_Relock) \wedge (PC_{Writer} \in \{W1, W2\})$.

Table 1(b) shows timeliness constraints for Refresher, when hiding clocks.

A consequence of the partial state observability is that the synthesised scheduler is not necessarily the *maximal* one. This is because the scheduler may apply more constraints than is absolutely required to some system state s, if these constraints are needed by states that are equal to s modulo the observation function.

5.2.2 Branching bisimulation equivalence reduction

In order to render synthesis more tractable, we reduce our models modulo the *branching bisimulation equivalence (bbe) reduction* (van Glabbeek and Weijland 1996). The bbe reduction eliminates actions we do not wish to observe, called τ actions. Here, τ actions are all the *uncontrollable* actions, i.e., those of the timeout and the application automata. Indeed, since our scheduler can only act whenever some *controllable* action is enabled, we do not gain anything by storing uncontrollable ones. Compared to other bisimulation reductions, bbe has the property that it removes τ actions, *only* if doing so does not change the branching structure of transition systems. Thus, the bbe-reduced system is *equivalent* to the original with respect to safety properties.

The synthesised scheduler for the bbe-reduced system will be exactly the same with the one we would have synthesised for the non-reduced system. This is because in our initial parallel automata model, it is always the case that either some state has outgoing τ transitions or transitions labelled by some non-τ scheduler action a. Indeed, note that when the mode variable equals **`SchedExec`** or **`SchedNotif`** the current state has only non-τ transitions enabled (those of the scheduler automata), while in modes **`Timeout`** and **`Application`** we can only perform τ transitions. So, it is *never* the case that a state, s, can do both a τ and an a transition, where $a \neq \tau$. As a consequence, after the bbe reduction on the initial state space graph, we obtain classes of equivalence, where, *if we can leave them with a transition a, then we cannot leave them with a transition τ and vice versa*. So, the *controllable* equivalence classes are characterised by their *frontier*, which is exactly the member states having non-τ transitions. So, we define the frontier of a class, c, of bbe-equivalent states as in (7), where enable() produces the set of states enabling a particular transition. Note that the frontier of an uncontrollable equivalence class is the empty set, $\emptyset$:

$$\text{frontier}(c) = c \cap \bigcap_{a \neq \tau} \text{enable}(a) \tag{7}$$

5.2.3 Synthesis procedure

The synthesis procedure has three steps. First, the bbe reduction is applied. Then, scheduling constraints are synthesised. This assigns to each branching bisimilar class c the set $\text{Bad}(c)$, i.e., the transitions the scheduler must not take in that class for the system to stay safe. If a τ action is a member of $\text{Bad}(c)$ then the whole class c is marked as unsafe. Otherwise, the constraints of c are assigned to its controllable member states, i.e., the states in c that have at least one non-τ transition. This effectively computes the set $\mathcal{S}_{exec}$. So, for all $s \in \text{frontier}(c)$, where $\text{Bad}(s) = \text{Bad}(c)$:

$$\mathcal{S}'_{exec}(s) := \mathcal{S}_{exec}(s) \setminus \{t_a | a \in \text{Bad}(s)\} \tag{8}$$

When using the observation function obs to project the states of the frontier to the observable system variables, we may cause classes to *share* projected states, i.e., there may be two classes, say c and c', such that $\text{obs}(s) = \text{obs}(s')$ for some $s \in \text{frontier}(c)$ and $s' \in \text{frontier}(c')$, or, equivalently: $\text{obs}(\text{frontier}(c)) \cap \text{obs}(\text{frontier}(c')) \neq \emptyset$.
This means that the scheduler cannot dissociate these states, so each *projected frontier* state is assigned the *union* of all the constraints of the bbe-equivalent classes it is a member of:

$$\mathcal{S}'_{exec}(\text{obs}(s)) := \mathcal{S}_{exec}(\text{obs}(s)) \setminus \{t_a | a \in \text{Bad}(\text{obs}(s))\} \tag{9}$$

$$\text{Bad}(\text{obs}(s)) = \text{Bad}(\{c | \text{obs}(s) \in \text{obs}(\text{frontier}(c))\}) \tag{10}$$

6 A methodology for synthesis

Despite the bbe reduction, the size of the state space can still be considerable. As will be seen later in Sect. 7, it can easily run up to tens of thousands of states, e.g., lines (10)–(15) of Table 3 for an already constrained system. Therefore, it is imperative that synthesis follows a methodology which reduces the state-space explosion problem. Another problem with scheduler synthesis is that the resulting scheduling constraints can be difficult to understand and relate to specific system properties.

Thus, the methodology for scheduler synthesis presented herein has a *dual* purpose. First, it reduces the size of the state space, by synthesising schedulers for successively more detailed models. In this way, more complex models are only considered when a safe scheduler has been synthesised already for a more constrained version of the model. Second, this methodology also has as a purpose (and advantage) to synthesise scheduler constraints that are more easily related to a specific safety property and platform execution model. So, it can be immediately identified which constraints are needed for avoiding deadlocks due to resource synchronisation, which ones for meeting deadlines when computations are not preemptable, etc. Thus, it is easier to understand the constraints themselves, as well as, the behaviour of the different system tasks and their importance as far as each safety property is concerned, leading to a better analysis of the system under scrutiny. This is advantageous both for validating the synthesised scheduling constraints and for discovering ways to optimise the system further (Kloukinas 2004).

Our methodology for scheduler synthesis considers four orthogonal aspects of the modelled system: (i) modelling of time, (ii) platform execution model, (iii) scheduling policies for overall system quality, and (iv) compositional analysis. We take advantage of these aspects by performing scheduler synthesis in four major steps.

6.1 Abstraction of time

First, we consider the issue of *time*, by examining the *untimed* model of the system and synthesising a scheduler to guarantee the *absence of deadlocks* due to synchronisation, when that is indeed possible. For the case study of Sect. 7, the reduction obtained is 97% of the full timed model (see line (2) of Table 3).

Example 5 Indeed, for the example of Fig. 2(a), the synthesised constraints on the untimed model remove the deadlocks due to the wrong synchronisation of Writer and Refresher.

While the system cannot deadlock anymore, there are still cases where User misses its period. Bad states representing these timeliness violations must be rendered unreachable through further constraints. Table 2 shows the results from the various synthesis stages for achieving this (lines (3), (5) and (7)), by synthesising 163 additional constraints.

Finding and removing *all* deadlocks in the untimed model means that the synchronisation and communication protocols used are now *logically* correct. That is, no deadlocks will ever occur, even if computation execution times have been wrongly estimated or they change later on, by changing implementations, porting to different hardware platforms, etc., as long as the basic assumption (execution on a uniprocessor system) holds. This is particularly important for product families, since in these the timing information differs for each family member (Coplien et al. 1998).

It is not always possible to synthesise an untimed model that fulfils the safety properties. If a scheduler cannot be synthesised it means that there is a trivial deadlock— a state with no outgoing transitions—that cannot be avoided, e.g., a task exits abruptly. If such a behaviour is indeed desirable and the task is not supposed to iterate, then the model could be extended with a self-loop transition at the end state to allow the synthesis procedure to generate a strategy. The only other case where a deadlock cannot be avoided is when a task waits for a notification that never comes in—this is a clearly wrong interaction protocol that needs to be redesigned. Otherwise, all deadlocks due to shared resources can be avoided—the synthesised scheduler will effectively avoid the situation where wait-for cycles are created, by disallowing tasks to execute and claim resources when that can lead to a wait-for cycle. This is indeed the case in the scheduling constraints for deadlock-freedom in Example 4.

Remark Of course, the constraints synthesised on the untimed model could be so strong that tasks may miss their deadlines. But they will be still less conservative constraints than those imposed by other techniques, as for example the PCP family of synchronisation protocols. In PCP a task always raises its priority to the “ceiling” of a shared resource, i.e., the priority of the highest priority task that uses that resource. In our case, priorities are raised only when there is indeed a danger of deadlock—otherwise all tasks have the same priority. We can achieve this because unlike PCP, which considers simply the information about which resource a task locks and which other tasks use that resource, we also consider the information about which other resources these other tasks have locked and which resources they may attempt to lock at the specific program state they are.

Having found all the deadlocks in the untimed system, we impose the synthesised $\mathcal{S}_{exec}$ and $\mathcal{S}_{notif}$ scheduler constraints upon the *timed* model, and search for *timeliness* constraints, so that all tasks will meet their deadlines. As aforementioned in Sect. 5.1, the scheduling constraints for deadlock freedom may constrain the system so much that liveness properties become invalid, i.e., they can introduce livelocks. These livelocks will usually appear as missed deadlines in the timed version of the model, so we do not check for them explicitly.

6.2 Platform execution model

Again, we do not attack the full timed model immediately but consider first a constrained version of it, where tasks execute under a *non-preemptive* execution model. The non-preemptive platform execution model reduces the state space by removing all cases where an interrupt suspends a task computation.

6.2.1 Non-preemption and scheduler synthesis

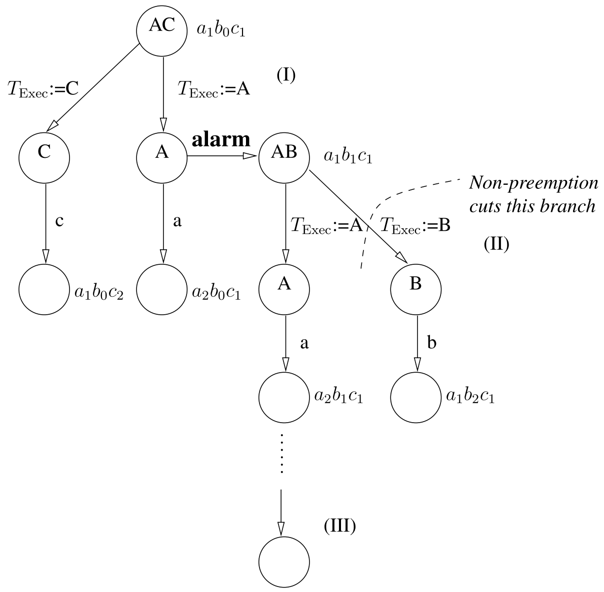
To better explain the benefits of examining the non-preemptive execution model first, let us consider the example in Fig. 4. As shown there, when imposing the non-preemptive execution model at state AB we are effectively cutting the branch AB $\rightarrow$ B, where the scheduler chose to preempt the execution of task A with task B after the alarm. This kind of reduction has a repercussion on the preemptive execution model we will examine subsequently. The result of examining the non-preemptive case first, depends on the kind of scheduler we will synthesise. If in the non-preemptive case we find that there is a winning strategy at point (III) and so we do not forbid branch AB $\rightarrow$ A, then adding preemption at the next stage will simply add branch AB $\rightarrow$ B. If, however, branch AB $\rightarrow$ A in the non-preemptive model is unsafe, then we will be obliged to constrain the system earlier on (since now branch AB $\rightarrow$ B is not available). If we needed to constrain the system at state AC, by cutting branch AC $\rightarrow$ A and selecting branch AC $\rightarrow$ C, then permitting preemption later on would mean that the whole sub-graph after branch AC $\rightarrow$ A will have been removed by the scheduler we synthesised for the non-preemptive execution model. Therefore, we have gained by being able to examine the inherent non-determinism of the scheduler synthesis problem, without being overwhelmed by the additional non-determinism introduced by the interrupts.

Once we can safely schedule the system for a non-preemptive execution model, we use the scheduling constraints to *reduce even further* the state space that we have to analyse, when we permit preemption. Observed reductions with the non-preemptive execution model and the bbe reduction ranged around 95% of the preemptive, unconstrained timed model (see lines (3) and (11) of Table 3).

The non-preemption of tasks is easily added to our models *through the use of a quality-level policy* that forbids the schedulers from choosing a task for execution, when another task is already in a state where it is computing:

$$\begin{aligned} \mathcal{Q}_{exec}(\mathrm{obs}(s)) := \{t \mid & \mathrm{computes}(t) \\ & \vee (t \in \mathcal{S}_{exec}(\mathrm{obs}(s)) \wedge \neg\exists t' \neq t \,.\, \mathrm{computes}(t'))\} \end{aligned} \tag{11}$$

Example 6 For the system of Fig. 2(a), the bbe-reduced non-preemptive system has 973 states (see line (3) of Table 2), while the bbe-reduced preemptive one has 804 states (using the constraints from the non-preemptive one, line (5)).



(Letters inside states denote tasks able to execute; vectors beside the states show possible values of the task PC*s.)*

Fig. 4 Preemption and state space size

It is still worthwhile to perform these separate synthesis steps even when the gains in state reduction are not spectacular, since it helps to understand the system behaviour better.

We should note here that we cannot safely schedule all systems when we do not allow tasks to be preempted. Indeed, in (11) we explicitly ignore the set of safe tasks ($\mathcal{S}_{exec}$) when some task is computing. For these systems we will not obtain any scheduling constraints and, therefore, will be obliged to examine the larger, unconstrained state space of the timed model, corresponding to a preemptive execution model.

6.3 Policies for overall system quality

Once we have synthesised a safe scheduler for deadlocks and deadlines, we can compose it with other policies to further constrain the set of safe states to those guaranteeing other QoS system requirements, e.g., memory or power consumption, jitter minimisation, etc. Designers can balance between the execution time and extra memory needed by these policies and the gains they offer to the overall system quality.

The aforementioned LMCS policy observes only the current system state, while more complex policies may examine application variables or the execution history. Such a policy, which also observes an application variable, is the optimisation policy of (12), which favours User to proceed if the current value is fresh. Multiple QoS policies can be applied as is shown in lines (13)–(14) of Table 2, where the policy of (12) has been applied to the safe system of line (8) and then the LMCS policy has been applied on top of it.

$$T_{Exec} := \begin{cases} \{\text{User}\} & \text{if Fresh} = \textbf{true} \wedge \text{User} \in \mathcal{S}_{exec} \\ & \quad \wedge \text{PC}_{\text{User}} = U1 \wedge \text{PC}_{\text{Writer}} = W0 \\ \{\text{Writer}\} & \text{if Fresh} = \textbf{false} \wedge \text{Writer} \in \mathcal{S}_{exec} \\ & \quad \wedge \text{PC}_{\text{User}} = U1 \wedge \text{PC}_{\text{Writer}} = W0 \\ T_{Exec} & \text{otherwise} \end{cases} \tag{12}$$

6.3.1 QoS conflicts

In most cases, QoS policies will be conflicting with each other. A designer would need to explore different combinations of these, e.g, LMCS first and then the policy of (12) or vice-versa, to better understand what their impact on the system is. As a general advice, one should order the policies in a way which follows the critical nature of the corresponding non-functional requirements and at the same time leaves enough freedom to apply further policies. So, if a requirement A is much more important than a requirement B then the policy of A should be applied first in most cases. At the same time, if a policy constraints the number of system states substantially then it might be a good idea to apply it at a later stage, so that other requirements can also be considered. Indeed, this is how our synthesis methodology has been structured—we start by the most important requirement (deadlock-freedom) and then try to successively constrain the system further but in a manner that leaves enough degrees of freedom for controlling according to subsequent QoS policies as well.

6.4 Compositional synthesis

Finally, designers can partition the system and independently synthesise constraints for subsystems. Then the synthesis algorithm is applied again on the parallel composition of the already constrained models, to obtain a scheduler guaranteeing the safety properties for the whole system.

Such a compositional synthesis allows designers to analyse bigger systems. Sometimes even ignoring a single task can make a great difference in the resulting state space—in our case study we observed a reduction of 82% by doing so (from 353730 down to 62137 states), as can be seen in Sect. 7.

Example 7 Table 2 shows the results of our methodology for the system of Fig. 2(a). As shown in line (15), without our methodology, one has to attack the full state space, which contains 21730 states (1338 after the bbe reduction), and will synthesise 56 constraints, instead of 163 that we had synthesised in line (6) for the same case (timed, preemptive execution).

Fewer constraints are synthesised without our methodology because the controller can be less conservative. That is, it ignores deadlocks hidden by time relations and deadline misses that occur only under a non-preemptive execution policy. Even if one

Table 2 Synthesis results for the system of Fig. 2(a)

*T/U: **Timed/Untimed** model, P/NP: **Preemption/No-Preemption***

	Model kind	States	Red.	Bad	Constraints Used/Prod.
	Synthesis Steps for the system of Fig. 2(a)				
(1)	T ***P, no bbe***	21730	0.00%	4	*N/A*
	U	2293	89.45%	24	*N/A*
	T ***NP***, No Deadlocks	14009	35.53%	4	*N/A*
	T ***NP***, Safe	7023	67.68%	0	*N/A*
	T ***P***	13228	39.13%	1	*N/A*
	T ***P***, Safe	11222	48.35%	0	*N/A*
	T ***P, Time Ind.***	2762	87.29%	5	*N/A*
	T ***P, Time Ind.***, Safe	2680	87.67%	0	*N/A*
	Synthesis Steps for the system of Fig. 2(a), with bbe				
(2)	U	279	98.72%	0	0/36
(3)	T ***NP***, No Deadlocks	973	95.52%	1	36/80
(4)	T ***NP***, Safe	574	97.36%	0	116/0
(5)	T ***P***	804	96.30%	1	116/47
(6)	T ***P***, Safe	702	96.77%	0	163/0
(7)	T ***P, Time Ind.***	282	98.70%	1	163/4
(8)	T ***P, Time Ind.***, Safe	285	98.69%	0	167/0
	System from line (8), with the LMCS QoS policy				
(9)	Model of (8), with LMCS	1406	93.53%	0	167/0
(10)	Model of (9), bbe	142	99.35%	0	167/0
	System from line (8), with the QoS policy of (12)				
(11)	(8) & (12)	2665	87.74%	0	167/0
(12)	Model of (11), bbe	285	98.69%	0	167/0
	QoS policy of (12) and then LMCS on system of (8)				
(13)	(8) & (12) & LMCS	1281	94.10%	0	167/0
(14)	Model of (13), bbe	130	99.40%	0	167/0
	Synthesis in one step, for preemption—compare with (6)				
(15)	T ***P, bbe***	1338	93.84%	1	0/56

would consider this as an advantage (we do not), there would still remain the problem of understanding why each constraint has been synthesised—to guard against a deadlock, a missed deadline or both? On the contrary, our step-by-step synthesis approach solves this issue. It also makes the constraints more robust, since now changes to timing relations cannot reveal previously hidden deadlocks.

6.4.1 Identifying system partitions

There are no general guidelines for deciding how to partition a system. Nevertheless, we believe that the first thing a designer should consider is the shared resources and what the tasks are trying to do with them. For example, one may consider first to partition according to how many resources the tasks use. For the case of Fig. 2(a), a possible partition would be to consider tasks Writer and Refresher first, since they use resources V and L, while task User only uses V. So one cannot expect User to be involved in a deadlock. Similarly, if a task is waiting at some point for a notification, then it does not make sense to consider it without the task that can notify it.

6.5 QoS for reducing the state-space

When the state-space is too big to be treated with our methodology, a designer might be able to employ QoS policies to control the state-space explosion. That is, do the synthesis with a policy that always gives priority to some tasks when that is possible, for example considers User a higher priority task than Refresher and Writer the least important task. Then use the synthesised constraints to constrain the system, remove the QoS policy and synthesise scheduling constraints for the general case, in a similar spirit that we have used non-preemption in our methodology.

7 Case study: a robotic arm

In this section we consider a case study based on a robotic arm system from Vicario (2001), shown in Fig. 5(a). The arm takes objects from a conveyor belt, stores them temporarily on a buffer shelf, and puts them into a basket. The arm is controlled by tasks running on a single processor.

Figure 5(a) shows the control-flow graphs of the tasks. `TrajectoryControl` reads commands from a shared buffer (`C`) and issues set-points (`L`) to the low-level arm `Controller`. If there are no commands (modelled by the predicate `T`) it holds, otherwise it reads the sensor value (`S`) and computes a new set-point. Its execution time is between 5 ms and 8 ms. There are two motion executors, `Lifter` and `Putter`. `Lifter` is activated periodically every 40 ms. It commands the arm to pick objects from the belt and place them into the buffer shelf. Upon termination, it issues a command to `TrajectoryControl` and activates `Putter`, sending it commands for moving the object from the shelf to the basket (predicate `P`). Its execution time is between 4 ms to 9 ms. `Putter` sends commands to move the object from the shelf into the basket. Its execution time is between 4 ms to 10 ms. The `SensorReader` task reads sensors every 24 ms. Its execution time is 1ms. Sensor readings are used by `TrajectoryControl`. `Controller` is a periodic task with a period of 16 ms.

7.1 Stopwatch automata model of the robotic arm

Figure 5(b) shows `Lifter`'s stopwatch automaton model. Note how the mode is changing—after each clock tick (e.g., $L_1 \rightarrow L_1$), which increases all running clocks/stopwatches, the mode changes to **`Timeout`**, so that we can check for deadlines/alarms. Mode changes to **`SchedExec`** before each **`monitorEnter`** ($L_1 \rightarrow L_2$),

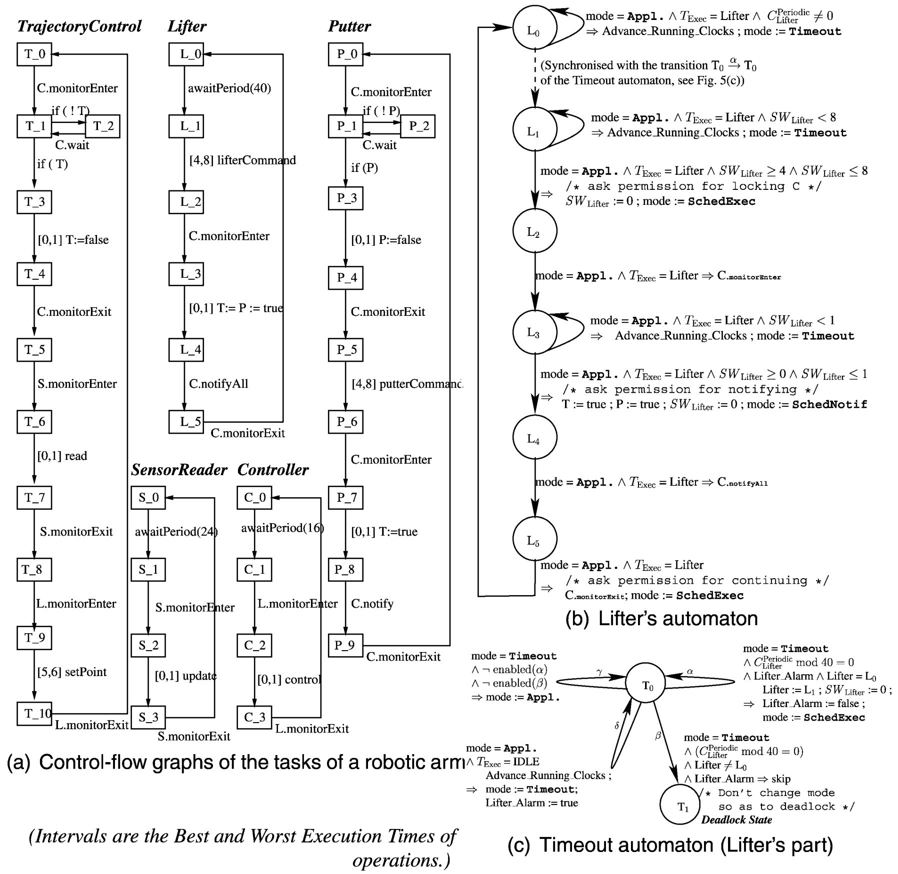


Fig. 5 A robotic arm case study

to ask the Execution scheduler for permission to enter the monitor. It also changes to **`SchedExec`** *after* each **`monitorExit`** ($L_5 \rightarrow L_0$), to get permission for continuing execution. Note finally, that before performing the **`notifyAll`** at state L_4, mode changed to **`SchedNotif`**, so that the Notification scheduler stack can decide what task(s), if any, should be notified.

Figure 5(c) shows the part of the Timeout automaton which is relative to `Lifter`. Transition $T_0 \xrightarrow{\alpha} T_0$ is used when `Lifter` is at its initial position and it should start a new period. So, its guard checks that `Lifter`'s clock has a value which is a multiple of its period. In this case, mode changes to **`SchedExec`** so that the Execution scheduler can respond to this "new task period" event. Transition $T_0 \xrightarrow{\gamma} T_0$ is used when there is no deadline/period to be signalled; we simply change mode back to **`Application`** to allow the application to continue. Transition $T_0 \xrightarrow{\delta} T_0$ is for the case where the scheduler had selected the IDLE task to execute; we just advance *all* running clocks/stopwatches (arming all alarms as a byproduct), waiting for a timeout. Finally, transition $T_0 \xrightarrow{\beta} T_1$ is when `Lifter` misses its deadline. In this case we move to a deadlock state and do not change mode; thus now the whole system becomes deadlocked. The Boolean variable Lifter_Alarm is used to dissociate between the cases $C_{\text{Lifter}}^{\text{Periodic}} = 0$ (start of period) and $C_{\text{Lifter}}^{\text{Periodic}} = \mathbf{40}$ (deadline). In the former case Lifter_Alarm is false and thus the deadlocking transition β is disabled, while in the latter case Lifter_Alarm is true and transition β is enabled. This variable starts with a value of true, gets disabled at each new period and *automatically becomes enabled by each tick*.

7.2 Applying scheduler synthesis

We decided to partition the application in two sub-systems, one comprising the 4 tasks `Lifter`, `Putter`, `SensorReader`, and `TrajectoryControl`, and one consisting solely of the `Controller` task. Table 3 shows the results obtained when applying our methodology on the case study. We started with the untimed model of the 4-task system, so as to check for deadlock states (see line (2) of Table 3). Not finding any, we used a non-preemptive execution policy to check the timed model of the system for states where deadlines are missed (line (3)). Such states indeed exist and we synthesised 103 scheduler constraints for avoiding them. In line (4) we see that when applying these constraints to the model, all deadline-miss states become unreachable (always assuming a non-preemptive task execution policy). Then, in line (5) we considered the timed model of the system under a preemptive execution policy. In this model, there are 15 more constraints we synthesise for avoiding the states where we can miss some deadline. When adding these 15 constraints to our scheduler we obtain a safe 4-task system, under both a non-preemptive and a preemptive execution policy, driven by a synthesised scheduler consisting of 118 constraints in total, as shown in line (6). In lines (7) and (8), we have attempted to synthesise constraints for the deadlines, when the scheduler is not allowed to observe the clock values. As can be seen, no extra constraints are needed, meaning that the 118-constraint scheduler from line (5) is already independent of time when the clock valuations are projected out of the constraints.

Having obtained a safe 4-task system, `Controller` is added to it to analyse the complete system. In line (10) of Table 3 we analysed the timed model of the system under a non-preemptive execution policy. We used as a scheduler the 118 constraints we had synthesised for the 4-task system, see line (8). As we can see, there were indeed new bad states where deadlines are missed and we synthesised 2325 constraints for avoiding them. Indeed, in line (11) where we applied these 2325 (plus $118 = 2443$) constraints to the system, all deadline-miss states have become unreachable. Then, in line (12), we examined the timed model under a preemptive execution policy, synthesising 976 new constraints. Using all the 3419 synthesised constraints, in line (13) we checked that they safely scheduled the system and then in line (14) we synthesised the final set of 71 constraints that are needed for a time-independent scheduler. The resulting scheduler (line (15)) has 3490 constraints, which keep the system in a safe state under both a non-preemptive and a preemptive execution policy, *without* observing the system clocks.

Finally, lines (16) and (17) apply the LMCS quality policy to the system of line (15), whose effect is to halve the number of states of the safe system. This shows that the preemptive, time-independent scheduler synthesised at line (15) does not

Table 3 Synthesis steps

	Model kind	States	Red.	Bad	Constraints Used/Prod.
	Synthesis Steps for the 4-task system, i.e., no Controller				
(1)	T ***P, no bbe***	62137	0.00%	7	*N/A*
	U, No Deadlocks	24597	60.41%	0	*N/A*
	T ***NP***, No Deadlocks	50139	19.31%	6	*N/A*
	T ***NP***, Safe	49054	21.06%	0	*N/A*
	T ***P***	61333	1.29%	4	*N/A*
	T ***P***, Safe	61051	1.75%	0	*N/A*
	T ***P, Time Ind.***	41574	33.09%	0	*N/A*
	T ***P, Time Ind.***, Safe	41574	33.09%	0	*N/A*
	Synthesis Steps for the 4-task system, with bbe reduction				
(2)	U, No Deadlocks	1553	97.50%	0	0/0
(3)	T ***NP***, No Deadlocks	2645	95.74%	1	0/103
(4)	T ***NP***, Safe	2605	95.81%	0	103/0
(5)	T ***P***	2740	95.59%	1	103/15
(6)	T ***P***, Safe	2702	95.65%	0	118/0
(7)	T ***P, Time Ind.***	2610	95.80%	0	118/0
(8)	T ***P, Time Ind.***, Safe	2610	95.80%	0	118/0
	Synthesis Steps for the 5-task system				
(9)	T ***P, no bbe***	353730	0.00%	176	*N/A*
	T ***NP***, (4-task Safe)	260020	26.49%	85	*N/A*
	T ***NP***, Safe	239501	32.29%	0	*N/A*
	T ***P***	337032	4.72%	113	*N/A*
	T ***P***, Safe	325460	7.99%	0	*N/A*
	T ***P, Time Ind.***	123514	65.08%	174	*N/A*
	T ***P, Time Ind.***, Safe	120449	65.95%	0	*N/A*
	Synthesis Steps for the 5-task system, with bbe reduction				
(10)	T ***NP***, (4-task Safe)	17604	95.02%	1	118/2325
(11)	T ***NP***, Safe	16476	95.34%	0	2443/0
(12)	T ***P***	23013	93.49%	1	2443/976
(13)	T ***P***, Safe	21913	93.81%	0	3419/0
(14)	T ***P, Time Ind.***	12313	96.52%	1	3419/71
(15)	T ***P, Time Ind.***, Safe	12428	96.49%	0	3490/0
	System line (15), with the LMCS QoS policy				
(16)	Before bbe	57003	83.89%	0	3431/0
(17)	After bbe	7858	97.78%	0	3431/0

over-constrain the system, thus allowing designers to effectively attack further quality properties.

8 Scheduler implementation

Once we have synthesised a scheduler we need to integrate it with the code of the application and the underlying OS. Time-independent schedulers can be easily implemented using widely available OS primitives, i.e., a preemptive, priority based FIFO scheduling policy, **`notify`**, **`notifyAll`**, **`wait`** and **`timed_wait`** on condition variables, and mutexes *without priority inheritance*. In addition to these, for time-dependent schedulers we need alarms and timers to measure how long a task operation has been executing. Almost all R-T OS provide access to alarms and timers—the problem is that the timers provided are usually measuring *response* time instead of pure *execution* time, i.e., they also measure the time an operation has been preempted. This creates a problem because in our model, duration intervals for operations are expressed as pure execution time, without considering the effects of preemption—`[2,3] write` means that `write` can take between 2 and 3 time units when executing without preemption, while its response time interval really depends on the possible preemptions it can suffer. So if one wants to keep these duration intervals and not replace them with intervals describing the best and worst case response times instead (which can be derived by considering the longest execution trace for an operation in the system space graph), then we need timers capable of measuring the exact execution time of computations, even in the presence of preemptions. Unfortunately, such timers are far from being widely available. For exactly this reason, our methodology produces time-independent schedulers at the last stage—to avoid requiring extremely reliable and precise timers. In fact, we also developed a version of our control subsystem for supporting synthesised schedulers on FAST-OS, the proprietary POSIX-compliant OS of Thalès Airborne Systems, for the PowerPC architecture. Unlike most OS, FAST-OS does not allow direct setting/observation of timers at all—a major reason behind our attempt to synthesise time-independent scheduling constraints (see Sect. 5.2.1, Table 1(b), and line (15) of Table 3). In the following, we present the core implementation of time-independent schedulers.

The code consists of two parts—the application code and the control subsystem (`U_Scheduler`). The application code is instrumented to call `U_Scheduler` when an application thread executes one of **`monitorEnter`**, **`monitorExit`**, **`notify`**, **`notifyAll`**, **`wait`**, **`timed_wait`** or **`awaitPeriod`**. In its turn, `U_Scheduler` evaluates the application-specific synthesised scheduling constraints corresponding to the different scheduler layers. The control subsystem is implemented as an accompanying library.

Our library uses a single mutex (`sched_mx`) and provides to each application thread a unique condition variable. These condition variables are all associated with the aforementioned mutex (a capability which exists in POSIX but not in Java). This construct is used simply for simulating the disabling of interrupts and can be used when our code needs to run in user space. Finally, we use three different priority levels, namely, **BLOCKED**, **EXECUTING** & **INTERRUPT** (from lowest to highest) and the `SCHED_FIFO` POSIX scheduling policy.

```
int interrupted_task = IDLE;

void U_Scheduler(int tc, bool in_notify, timespec &deadline) {
  bool finished = true, level_super = false;
  int  tn, i;
  // tc is the current thread (t_current) & tn the next one (t_next)
  do {
    // calculate Ready, Safe & Quality sets
    tn = Synthesized_Constraints(THREADS_TABLE);
    if (tn != tc) {
      if (in_notify) {
        if (-1 != tn) { // -1 means no thread is waiting
          // t_next has priority BLOCKED so it cannot preempt t_current
          notify(THREADS_TABLE[tn].cv);
        }
      } else { // ! in_notify
        if (!level_super) interrupted_task = tc;
        U_Set_Priority(tn, EXECUTING);
        THREADS_TABLE[tn].PC = THREADS_TABLE[tn].PC_Notif;
        notify(THREADS_TABLE[tn].cv);

        if (NULL == deadline){// Not a timed_wait or awaitPeriod
          // Release sched_mx here
          wait(THREADS_TABLE[tc].cv, sched_mx);
          // Here I have been signaled
        } else {                  // NULL != deadline
          U_Set_Priority(tc, INTERRUPT);//Release sched_mx
          timed_wait(THREADS_TABLE[tc].cv, sched_mx, deadline);
          /* Here I have been signaled or I have timed-out.
             Must re-schedule to be safe, if I timed-out. */
          if (THREADS_TABLE[tc].PC != THREADS_TABLE[tc].PC_Notif) {
            finished = false;
            THREADS_TABLE[tc].PC = THREADS_TABLE[tc].PC_Timeout;
          }
          level_super = true ; deadline = NULL;
        }
      }
    }
  } while (! finished);
  if (level_super) {
    U_Set_Priority(tc, EXECUTING);
    U_Set_Priority(interrupted_task, BLOCKED);
    interrupted_task = tc;
  }
}
void U_monitorEnter(obj o, int tc, int curr_pos, int next_pos) {
  lock(sched_mx);
  THREADS_TABLE[tc].PC = curr_pos;
  U_Scheduler(tc, false, NULL);
  lock(o.mutex); // We lock the object once we've got permission
  THREADS_TABLE[tc].PC = next_pos;
  unlock(sched_mx);
}
void U_monitorExit(obj o, int tc, int curr_pos, int next_pos) {
  lock(sched_mx);
  THREADS_TABLE[tc].PC = curr_pos;
  unlock(o.mutex);// Unlock the object before calling U_Scheduler
  THREADS_TABLE[tc].PC = next_pos;
  U_Scheduler(tc, false, NULL);
  unlock(sched_mx);
}
```

Fig. 6 Pseudo-code of the application scheduler

Figure 6 shows the pseudo-code of the implementation. Before calling procedure `U_Scheduler`, our **`monitorEnter`** locks the `sched_mx` variable and updates

Table 4 Timing primitives under eCos (*results in* μs)

Primitive	Min	Avg.	Max	Avg.-Dev.
Synthesised constraints	0.00	0.66	4.00	0.45
Context switch	0.00	0.77	1.00	0.35
Trylock (unlocked)	0.00	0.69	2.00	0.47
Unlock (locked)	0.00	0.75	3.00	0.47

the application task's position to be the same as in the model (lines 47–48). U_-Scheduler then calls Synthesized_Constraints (generated by the synthesis tool) in line 9, passing it the current task PC. If the thread to be executed next (t_{next}) is different from the current one ($t_{current}$) and $t_{current}$ is not doing a notification, t_{next}'s priority is set to **EXECUTING** (line 18), the condition variable ($cv_{t_{next}}$) of t_{next} is notified in line 20 and we finish by having $t_{current}$ wait on its own condition variable, $cv_{t_{current}}$, in line 24. This final action releases sched_mx just before blocking, thus allowing the notified thread t_{next} to resume execution. If t_{next} is the same as $t_{current}$, then the application scheduler returns normally and $t_{current}$ unlocks sched_mx.

The algorithm changes somewhat when calling the application scheduler through a **timed_wait** or an **awaitPeriod**. In this case, we also pass to our scheduler the time that the current thread should wait. The scheduler then performs a timed wait on $cv_{t_{current}}$ in line 28, using as timeout the *absolute* deadline argument, instead of doing a simple wait. It also increases the priority of $t_{current}$ to **INTERRUPT** just before performing the timed wait (line 27), so that $t_{current}$ gets the CPU when it timeouts. When $t_{current}$ timeouts, it re-evaluates the scheduler predicates (line 32), so as to find out if it is indeed safe to continue execution. Before calling U_Scheduler, functions U_timed_wait and U_wait (not shown in Fig. 6) set field PC_Notif to the label of to the internal state of the wait, where the thread has been notified but has not yet re-acquired the mutex of the object on which it was waiting. Similarly, functions U_timed_wait and U_wait_for_period set field PC_Timeout to the label of the internal state of the **timed_wait**, or the label of the first statement after a new period.

We have successfully executed our implementation over two different combinations of hardware architecture and embedded OS, namely an Intel Pentium II running eCos over Linux and a PowerPC simulator with FAST-OS. Experiments with eCos showed that the execution time of the synthesised predicates (i.e., function Synthesized_Constraints) is comparable to the execution time of locking an (unlocked) mutex, having a WCET in the order of 4 μs. Table 4 gives the results of our experiments under eCos. Experiments were run 1000 times on a 330 MHz Pentium II, where eCos was using the synthetic Linux hardware architecture, e.g., running over Linux as a user process. eCos had the highest real-time priority in SCHED_FIFO scheduling policy, thus running uninterrupted by *all* other processes. In addition, all memory pages of the eCos process were locked in RAM, so as to avoid paging from the OS.

The implementation pseudo-code shown in Fig. 6 refers to a POSIX-API implementation of this library. This implementation had to support FAST-OS that does not allow access to alarms. This is why timeouts (for U_**timed_wait** and U_**awaitPeriod**) were implemented with the **timed_wait** primitive. We also have a non-POSIX implementation over `eCos` that uses OS alarms and alarm handlers directly, giving us finer control over timeout events, since these are now treated by high priority interrupt handlers. In this way we can support deadline and period miss handlers as for example proposed in the RTSJ (Real-Time for Java Expert Group 2001).

9 Scheduler robustness

Synthesised schedulers can be intolerable to the wrong estimation of a computation's WCET. In fact, a computation should not finish earlier than its *Best Case Execution Time* (*BCET*) either; in both cases the system enters a state that was not in the model used to synthesise the scheduler. Since this state was not explored during synthesis, the scheduler does not have a strategy for it and thus can take an unsafe action. It should be noted that by unsafe we mean an action leading to a deadline miss, since deadlocks have been eliminated using the untimed model of the system, thus the deadlock-safety synthesised constraints are not sensitive to timing errors (indeed, our scheduler synthesis methodology was explicitly developed to guard against such a situation). Robustness is the reason for which RMA is sometimes preferred to EDF, since when the timing assumptions are invalid one still knows how the system will behave with RMA—the task with the smallest period will execute first and so on—while EDF's dynamic nature renders the system rather unpredictable. In our setting, the time-independent scheduler constraints offer the same level of robustness as RMA—they define the scheduling priorities independently of timing relations and therefore are robust to any changes to these.

However, knowing which task will execute when the timing relations have been estimated wrongly does not respond to the more important question of whether the system will behave in a desirable manner. To answer that question, one needs to explicitly analyse what the consequences of the invalid timing assumptions will be—will they lead to missed deadlines and, if so, is there a way to further constrain the system so as to render it safe even for these cases? The simplest solution for computations finishing before their BCET, is to *impose* it for each computation, by idling. This, however, implies that we either use a non-preemptive execution model, or that we have *execution time timers* (that do not measure the preemption time for a computation) so that we know how long the computation has executed. Unfortunately, such timers are not currently supported by many OS. Instead of imposing the BCET, we can explicitly verify whether the synthesised scheduler tolerates wrong estimations of it. To do so, we need to apply our synthesised scheduler to a model where all BCET are substituted by zero, thus exploring all possible cases of early completion of computations. If we do not need to synthesise any new constraints for keeping the system in a safe state, then our scheduler tolerates all the cases where a computation finishes earlier than expected. Otherwise, we can use the additional constraints synthesised in this step to render it safe anew. This step should evidently be performed last, since we need to explore a much bigger state space. In addition, by considering the question of tolerance to BCET estimations last, we can better identify the constraints which are needed explicitly for this case and keep them separate from the constraints needed for the case where our assumptions hold.

There are two different manners to establish tolerance of the scheduler to wrong estimations of the WCET of computations, similar to those for the BCET. If the OS supports execution time timers then we need do no further analysis. Indeed, it suffices to set alarms on these timers for the case where a computation exceeds its WCET. Otherwise, we need to translate each WCET into a *Worst Case Response Time* (*WCRT*) by taking into account all possible preemptions of this computation by other computations. In the state space graph we identify the WCRT as the longest path for each computation. Having done this, we need to verify again the model, using now the interval [BCRT = BCET, WCRT] as the execution time of a computation (since the underlying OS does not allow us to differentiate between execution and response time). The synthesised scheduler for this model can then be implemented along with watchdogs which guard against computations exceeding their WCRT. At the same time, we need to change the behaviour of the task stopwatches in the model so that they are no longer stopped when computations are preempted (otherwise we will be comparing *execution* versus *response* time). Another way of achieving this is by adding new clocks so as to be able to measure the preemption time of tasks in the model but then complexity goes up.

If the model using response times cannot be scheduled safely, then we need an OS with execution time timers, or a non-preemptive execution model to render the WCRT of *computations* equal to their WCET (since now computations cannot be preempted). So for the transition `W1->W2` in Fig. 2(a), the computation `write` will have a WCRT equal to its WCET, that is 3 time units, since it cannot be preempted once started. If this results in an unacceptably constrained system, we can break up computations to introduce explicit preemption points by introducing synchronisation constructs on new task-local objects. Thus, the deadlock-freedom of the system continues to hold (since the new synchronisation objects are local) and the scheduler has additional points where it can exert control.

10 Related work

Our methodology for building application-driven schedulers follows the controller synthesis paradigm (Wonham and Ramadge 1987) and builds upon (Altisen et al. 2002; Asarin et al. 1995). Controller synthesis for timed automata was also considered in Hoffmann and Wong Toi (1991), where the problem is reduced to the untimed framework of Wonham and Ramadge (1987) using the *region graph* construction that results in state space explosion. Wong Toi (1997) considers the more general setting of linear hybrid automata and presents a semi-decision procedure. The approach of Kwak et al. (1998) is also similar to ours since it uses an automata-based formalism (after translation from ACSR) but it relies on a different algorithm, based on weak bisimulation, and does not propose a particular scheduler architecture or implementation. A scheduler synthesis tool has also been described in Mok et al. (1996). It differs from ours in two major aspects: (i) it computes static cyclic schedules by sequencing events in a fixed time frame, whereas our algorithm produces dynamic (and not necessarily cyclic) schedules for an unbounded time frame; and (ii) it is restricted to deterministic execution times, while we can handle non-deterministic ones.

Task inter-dependencies due to resources are not considered in Isović and Fohler (2000), though applications are allowed to have heterogeneous task types. The advantage of our method is the handling of larger models than if we had tried to attack the original timed version of the model at once. In addition, following our method designers can better understand the behaviour of a system, since we successively drive them through: (i) states which cause a deadlock later on; and, (ii) states where a system is overloaded (and, thus, task preemption is needed). Our method can be applied to applications comprising any mix of periodic, aperiodic, etc. tasks sharing resources and communicating through condition variables.

A disadvantage of our method is that we must build the entire state space before synthesising a scheduler. It could be possible to adapt to our setting the on-the-fly synthesis algorithm proposed in Tripakis and Altisen (1999). Concerning state-space explosion, it is interesting to note Wang et al. (2009), who synthesise controllers for deadlock-freedom, using structural characteristics of Petri net models of the programs. This approach scales easily to very large programs, since it does not explore the full state space. It is similar to using the sets of task states where they hold or want to hold a resource (see Sect. 5.1), to identify *potential* deadlocks. Apart from the fact that not all potential deadlocks are real, the main problem with Wang et al. (2009) is the solution advocated—to add extra locks to render deadlocks impossible. As we have shown at the end of Sect. 5.1, this solution is rather Procrustean, since it greatly over-constrains the valid execution traces and can break the application logic (even in non-R-T systems). In fact, this is a problem that is shared by all approaches, the RMA family included—it is not known what are the repercussions of the constraints they impose on other properties of the system. Our methodology is best poised to deal with this problem for two reasons. First, by attempting to synthesise the maximal controller, it applies as few constraints as possible, so when it does change the application logic it is because that is the only possible way for safely controlling the system. So the designer has to accept this change, add extra control points and/or observation variables, or redesign the system completely—there is no other alternative. Second, since our methodology builds on model checking, software engineers can easily verify whether the synthesised scheduler respects basic application properties, which is not supported by the approach of Wang et al. (2009) or these based on RMA-type analyses. So for the system of Fig. 2(a) one can easily check whether the variable `Fresh` will ever become true so that the User task uses the value produced by the task Writer—indeed this is the case for our scheduling constraints but not for the solution of Wang et al. (2009). For this system the violation of this property will eventually show up as a missed deadline, since the User task cannot finish its iteration without reading a fresh value. But in other systems, that may not be the case—indeed, the User task could very simply have ignored old values and wait for its next period to treat a fresh value, thus never doing any real work. This could not have been identified easily without model-checking and designers should realise that meeting the deadlines does not necessarily mean that the program has done the work it needed to do.

Several have considered quality requirements for rate-monotonic scheduling. For instance, Dobrin and Fohler (2004) proposes a technique for reducing the number of preemptions, but at the cost of eventually having to increase the number of tasks by splitting some of the original ones. Flexible scheduling techniques (Fohler and Buttazzo 2002) consider the problem of scheduling together hard and so-called *soft* real-time tasks that are characterised by quality-of-service demands. However, they do not cope with quality requirements of hard real-time tasks, which our approach handles easily. Then, Combaz et al. (2008) handles hard deadlines together with specific quality properties, but only for video encoding/decoding. Besides, the major problem with such approaches is that each QoS property has to be tackled individually with a new algorithm and/or run-time system. In contrast, our methodology is able to handle QoS requirements by specifying the appropriate constraints at the *Quality-Exec* and *Quality-Notif* layers and possibly enriching the model, while the controller synthesis algorithm and the controller subsystem do not change.

Finally, compared to MDE approaches like Bordin et al. (2008) that are based on classic scheduling analyses, the general applicability of scheduler synthesis means that analysis tools do not need to be extended for each new analysis theory and that it can seamlessly support verification of general properties through model-checking, since it is based on automata.

11 Conclusions

Scheduling system tasks to meet multiple QoS requirements is an extremely difficult but at the same time very important task. We have introduced an automata theoretic model-based approach (based on scheduler synthesis) to achieve this, focusing as a start on meeting the most basic (i.e., deadlock freedom) and the most critical (i.e., timeliness) requirements, while showing how further QoS properties can be easily achieved as well (e.g., the reduction of context switches).

Our scheduler architecture and synthesis methodology allows to break the synthesised scheduling constraints into different parts, each representing some particular safety property and platform execution mode. This helps software engineers better understand the schedulers themselves and to get a better understanding of the behaviour and importance of the different tasks. Another advantage is that schedulers can be synthesised for larger systems by doing the synthesis successively, each time using a more detailed model of the system, after having applied to it the schedulers synthesised in previous steps.

Our approach does not impose restrictions on the type of tasks nor does it require that the system is restructured simply to facilitate analysis and control.

We have also performed a prototype validation of our scheduler synthesis methodology by using two OS (`eCos` and FAST-OS). On top of these we can execute an application controlled by a synthesised scheduler, through the use of a library we have developed to support application-specific synthesised schedulers. We have developed two versions of this library: one POSIX compliant (which works unchanged in both FAST-OS and `eCos`) that we have described herein, and a non-POSIX one that uses OS alarms and alarm handlers directly (for `eCos`). Our approach has been first integrated in an industry-strength RTSJ-compliant compilation infrastructure and run-time environment (Kloukinas et al. 2003): the model extraction and synthesis steps were interfaced with the Java-to-C TurboJ compilation chain (Weiss et al. 1998) and our controller subsystem was part of the Expresso executive (Gauthier and Richard-Foy 2002). More recently, it was used as part of an MDE framework for real-time embedded systems comprising the formal, model-transformation and code-generation tool *Jahuel* (Assayad et al. 2005) and the FlexCC2 compilation technology of STMicroelectronics (Bertin et al. 2002).

Acknowledgements C. Kloukinas would like to acknowledge the partial financial support provided by the EU project FP7-ICT-2009-257367 IoT@Work and the Royal Society, as well as the Verimag laboratory in France for its hospitality during the time when a large part of this paper was written.

S. Yovine would like to acknowledge the partial financial support provided by STIC-AmSud TAPIOCA, BID-ANPCyT PICTO-CRUP 31352, PICT-PAE 02287 and MinPlan GAMBETAS projects, and MINCyT "Cesar Milstein" grant.

References

Altisen, K., Gößler, G., Sifakis, J.: Scheduler modeling based on the controller synthesis paradigm. Real-Time Syst. **23**(1), 55–84 (2002)

Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994)

Asarin, E., Maler, O., Pnueli, A.: Symbolic controller synthesis for discrete and timed systems. In: HS-II. LNCS, vol. 999, pp. 1–20. Springer, Berlin (1995)

Assayad, I., Bertin, V., Defaut, F.-X., Gerner, Ph., Quevreux, O., Yovine, S.: Jahuel: A formal framework for software synthesis. In: ICFEM'05. LNCS, vol. 3785, pp. 204–218. Springer, Berlin (2005)

Barroso, L.A., Dean, J., Hölzle, U.: Web search for a planet: The Google cluster architecture. IEEE Micro **23**(2), 22–28 (2003)

Baruah, S.K., Howell, R.R., Rosier, L.: Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. Real-Time Syst. **2**, 301–324 (1990)

Bertin, V., Daveau, J.-M., Guillaume, P., Lepley, T., Pilat, D., Richard, C., Santana, M., Thery, T.: FlexCC2: An optimizing retargetable C compiler for DSP processors. In: EMSOFT'02, pp. 382–398 (2002)

Bordin, M., Panunzio, M., Vardanega, T.: Fitting schedulability analysis theory into model-driven engineering. In: ECRTS'08, pp. 135–144 (2008)

Bruno, E.: Java RTS real-time enables financial applications. (Online) www.devx.com/Java/Article/35246/. 23 August 2007

Carr, N.: Avatars consume as much electricity as Brazilians. (Online) http://www.roughtype.com/archives/2006/12/avatars_consume.php. 5 December 2006

Closse, E., Poize, M., Pulou, J., Sifakis, J., Venier, P., Weil, D., Yovine, S.: Taxys: a tool for the development and verification of real-time embedded systems. In: CAV'01. LNCS, vol. 2102. Springer, Berlin (2001)

Combaz, J., Fernandez, J.-C., Sifakis, J., Strus, L.: Symbolic quality control for multimedia applications. Real-Time Syst. **40**(1), 1–43 (2008)

Coplien, J., Hoffman, D., Weiss, D.: Commonality and variability in software engineering. IEEE Softw. **15**(6), 37–45 (1998)

Dobrin, R., Fohler, G.: Reducing the number of preemptions in fixed priority scheduling. In: ECRTS'04, pp. 144–152, June 2004

Fohler, G., Buttazzo, G.C.: Introduction to the special issue on flexible scheduling. Real-Time Syst. **22**(1/2), 5–7 (2002)

Gansner, E.R., North, S.C.: An open graph visualization system and its applications to software engineering. Softw. Pract. Exp. **30**(11), 1203–1233 (2000)

Garey, M.R., Johnson, D.S.: Two-processor scheduling with start-times and deadlines. SIAM J. Comput. **6**(3), 416–426 (1977)

Gauthier, L., Richard-Foy, M.: Expresso RNTL project—High Integrity Profile. www.irisa.fr/rntl-expresso/docs/hip-api.pdf (2002)

Harbour, M.G., Klein, M.H., Lehoczky, J.P.: Timing analysis for fixed-priority scheduling of hard real-time systems. IEEE Trans. Softw. Eng. **20**(1), 13–28 (1994)

Henzinger, T.A., Sifakis, J.: The discipline of embedded systems design. Computer **40**(10), 32–40 (2007)

Hoffmann, G., Wong Toi, H.: The input-output control of real-time discrete event systems. In: CDC'91 (1991)
IEEE. POSIX.1. IEEE Std 1003.1:2001: Standard for Information Technology—Portable Operating System Interface (POSIX). IEEE (2001)
Isović, D., Fohler, G.: Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints. In: RTSS'00, November 2000
Joy, B., Steele, G., Gosling, J., Bracha, G.: The Java Language Specification, 2nd edn. Addison-Wesley, Reading (2000)
Kesten, Y., Pnueli, A., Sifakis, J., Yovine, S.: Decidable integration graphs. Inf. Comput. **150**, 209–243 (1999)
Klein, M.H., Ralya, T., Pollak, B., Obenza, R., Harbour, M.G.: A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems. Kluwer Academic, Norwell (1993)
Kloukinas, C.: Data-mining synthesised schedulers for hard real-time systems. In: ASE'04, pp. 14–23, September 2004
Kloukinas, C., Yovine, S.: Synthesis of safe, QoS extendible, application specific schedulers for heterogeneous real-time systems. In: ECRTS'03, pp. 287–294, July 2003
Kloukinas, C., Nakhli, C., Yovine, S.: A methodology and tool support for generating scheduled native code for real-time Java applications. In: EMSOFT'03. LNCS, vol. 2855. Springer, Berlin (2003)
Kwak, H., Lee, I., Philippou, A., Choi, J., Sokolsky, O.: Symbolic schedulability analysis of real-time systems. In: RTSS'98, December 1998
Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM **20**(1), 46–61 (1973)
Mok, A.K., Tsou, D.C., Rooij, R.C.M.: The MSP.RTL real-time scheduler synthesis tool. In: RTSS'96, December 1996
Real-Time for Java Expert Group: The real-time specification for Java. Tech. report, RTJ.org, December 2001
Reeves, G.E.: What really happened on mars? (Online) http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/Authoritative_Account.html. December 1997
Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. Computer **39**(2), 25–31 (2006)
Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority inheritance protocols: An approach to real-time synchronization. IEEE Trans. Comput. **39**(9), 1175–1185 (1990)
Tripakis, S.: Description and schedulability analysis of the software architecture of an automated vehicle control system. In: EMSOFT'02, October 2002
Tripakis, S., Altisen, K.: On-the-fly controller synthesis for discrete and dense-time systems. In: FM'99. LNCS, vol. 1708. Springer, Berlin (1999)
Tripakis, S., Yovine, S.: Timing analysis and code generation of vehicle control software using Taxys. In: RV'01. El. Notes in Th. Comp. Sci., vol. 55(2), pp. 277–286 (2001)
van Glabbeek, R.J., Weijland, W.P.: Branching time and abstraction in bisimulation semantics. J. ACM **43**(3), 555–600 (1996)
Vicario, E.: Static analysis and dynamic steering of time-dependent systems. IEEE Trans. Softw. Eng. **27**(8), 728–748 (2001)
Wang, Y., Lafortune, S., Kelly, T., Kudlur, M., Mahlke, S.A.: The theory of deadlock avoidance via discrete control. In: POPL'09, pp. 252–263, January 2009
Weiss, M., de Ferrière, F., Delsart, B., Fabre, C., Hirsch, F., Johnson, E.A., Joloboff, V., Roy, F., Siebert, F., Spengler, X.: TurboJ, a Java bytecode-to-native compiler. In: LCTES'98. LNCS, vol. 1474, pp. 119–130. Springer, Berlin (1998)
Wong Toi, H.: The synthesis of controllers for linear hybrid automata. In: CDC'97, pp. 4607–4612 (1997)
Wonham, W.M., Ramadge, P.J.: On the supremal controllable sublanguage of a given language. SIAM J. Control Optim. **25**(3), 637–659 (1987)