Kloukinas, C. & Yovine, S. (2011). A model-based approach for multiple QoS in scheduling: from models to implementation. Automated Software Engineering, 18(1), pp. 5-38. doi: 10.1007/s10515-010-0074-8

**Noname manuscript No.**
(will be inserted by the editor)

# A Correct-by-Construction MDE Approach for QoS-Aware Scheduling: From Models to Implementation

**Christos Kloukinas · Sergio Yovine**

Received: date / Accepted: date

**Abstract** Meeting multiple Quality of Service (QoS) requirements is now an important factor for the success of complex software systems. This paper presents a correct-by-construction, automated, model-driven *scheduler synthesis* approach for scheduling system tasks so as to meet multiple QoS requirements. As a first step, it shows how software engineers can meet deadlock-freedom and timeliness requirements, in a manner that *(i)* does not over-provision resources, *(ii)* does not require architectural changes to the system, and that *(iii)* leaves enough degrees of freedom to pursue further properties. The synthesis methodology directly associates each scheduler with a specific pair of *QoS property* and underlying *platform execution model*, so as to facilitate their *validation* 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 the implementation infrastructure needed for executing an application on top of common operating systems, without requiring modifications of the operating system.

**CR Subject Classification** D.2.2.a. CASE · D.2.4.e. Model Checking · D.4.1. Process Management · D.4.7.e. Real-time systems and embedded systems

## 1 Introduction

The importance of *Quality of Service (QoS)* and, in general *quantitative non-functional* requirements, is increasing every day, as computer systems move far away from the scientific and office applications of the past. Nowadays most aspects of our lives depend on the correct behaviour of computerised systems, which have a number of, sometimes quite stringent, QoS requirements, e.g., computational power, memory size, power consumption, etc. Software engineers face these QoS requirements in different application domains - from mobile embedded systems running on batteries to big server farms, like those of Google [6] and of Second Life, whose avatars are estimated to consume "considerably more" electricity than people at developing countries do [10]. As sometimes these QoS requirements are conflicting, it becomes imperative to develop analysis methods that enable software engineers to tackle complex systems and their multiple QoS requirements [15,21], 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's like VxWorks and commodity OS's like Solaris, AIX or Linux, up to commercial, large-scale Java VMs like Sun RTS and IBM WebSphere. From embedded systems, such as the anti-lock breaking system (ABS) in automobiles, to games and big distributed systems, such as those responding to financial market events, software engineers need to guarantee timeliness for the various computational 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." [9]. Even worse, others like the ABS 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, software engineers have to be overly pessimistic in estimating the demands on system re-

C. Kloukinas
City University London, London EC1V 0HB, UK.
E-mail: C.Kloukinas@soi.city.ac.uk

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

sources, which leads to systems that have fewer capabilities and are more expensive than need be.

Classic scheduling analyses for *R-T* (e.g., RMA/EDF [31], used with the PIP/PCP [37] synchronisation protocols), make assumptions that are hardly realistic nowadays. Furthermore, since they use only a very small part of the overall system state, much detail is lost and additional resources are needed for guaranteeing task timeliness, even though this is not really necessary. This is also true for extensions handling task dependencies, such as *HKL* analysis [17]. For instance, [38] reported that *HKL*-based analysis failed to schedule an automated vehicle control software safely, whereas an automata-theoretic, tool-assisted Model-Driven Engineering (MDE) framework succeeded [40].

Another problem is that it is not obvious how to extend classic scheduling analyses, so as to *easily* meet other QoS requirements such as memory minimisation or power consumption, because these analyses are *rigid*. That is, they remove all possible choices in scheduling the tasks, demanding either to fix task priorities according to their periods or dynamically allocate priorities according to deadlines alone, without taking into account other system aspects. By removing all degrees of freedom in scheduling tasks, classic scheduling analyses effectively require software engineers 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 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, software engineers are called upon to correctly identify which case among the massive (and ever-growing) body of analysed systems better fits their own system [8]. 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 is simply unrealistic.

This picture calls for a more automated framework based on a generic solution instead of a case-by-case one. Automation liberates software engineers from the pressure of mastering a sophisticated and ever-expanding body of analyses, allowing them to concentrate instead on designing the system correctly and expressing its QoS requirements. From the analysis tool vendor's viewpoint, the applicability of a generic solution means that the tool need not be constantly extended each time research delivers another specialised analysis for some case that had not been considered so far. On the contrary, tool evolution can be devoted now to improving performance and usability.

An appealing approach in this direction is to *automatically synthesise* a scheduler [1] for a *model* of the system given in an automata-based formalism. This solution fits into the more general paradigm of MDE [36], leading to systems that are *correct by construction*, while greatly reducing the effort required from software engineers. Due to the ongoing advances in model-checking, this alternative is becoming more and more interesting for real-world systems. Here, we show how one can use it to perform a *fine-grain* analysis of systems and implement these in a way *guaranteeing* 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, which increases the applicability and benefits of scheduler synthesis [29]. Our system analysis and scheduler synthesis methods do not make any assumptions on the tasks comprising the system, their periodicity (or lack thereof), 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 software engineers to make error-prone choices, such as where to enable PIP, that can lead to problems like those faced by Mars Pathfinder [35].

Our methodology synthesises successive scheduler layers for guaranteeing different QoS requirements, by 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.

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*) requirements of a system and for synthesising a scheduler for it. We follow with an in-depth presentation of our methodology for applying 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-explosion problem at the same time. We then introduce a more complex case study, on which we apply our scheduler synthesis methodology. This is followed by the description of the implementation of a library of synchronisation/communication primitives, which allows the use of synthesised schedulers in currently available *OS*'s, and a discussion on the robustness of the synthesised schedulers. Finally, we compare our work with other related approaches, before finishing with a concluding discussion.

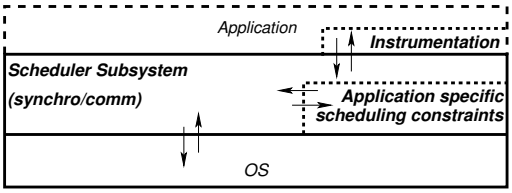



**Fig. 1** System architecture

## 2 Overall System Architecture

The overall system architecture we consider is depicted in Fig. 1. 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 *scheduler constraints*, which are *application specific*, to make a decision. These decisions are 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 general system architecture for 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 dynamic, application-specific scheduling constraints are automatically synthesised from a stopwatch automata model of the application and its environment.

The primitives of interest for control 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 [23] or Java [25] semantics. Finally, with `waitForPeriod` periodic application tasks wait for the arrival of their next period.

### 2.1 Scheduler Architecture

The architecture of the *application-specific scheduling constraints* themselves is depicted in Fig. 2. As shown there, the application tasks make some request that is forwarded to one of two scheduler stacks. The left scheduler stack is responsible for electing some application task for execution, whereas the right stack elects a task as a target for a pending signal/notification. 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 system requirements, e.g., jitter minimisation, energy 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 the extra QoS requirements.

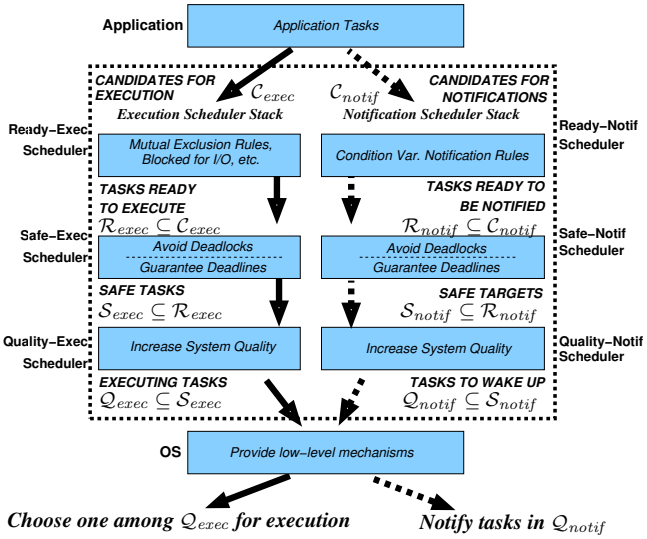



**Fig. 2** Scheduler architecture

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. 2 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 energy 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. 3, 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. 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 doing a timed wait 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.

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

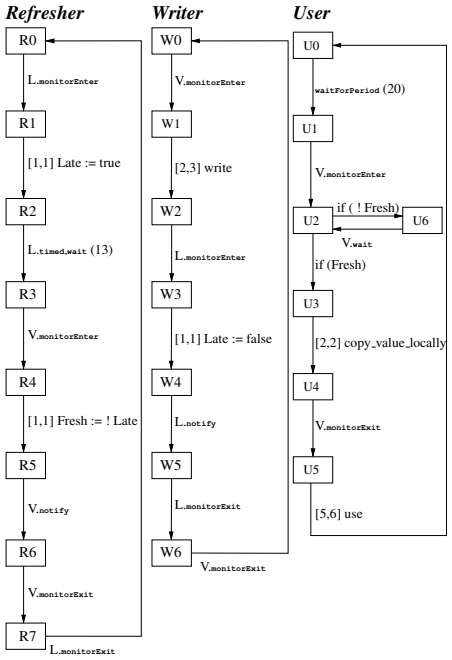



**Fig. 3** A simple three-task system
*(Clock variables are omitted for readability – each computation is annotated with its duration interval. )*

## 4 System Modelling

This section presents our modelling of a system through *discrete-time stopwatch automata*. Stopwatch automata allow for fine grain modelling, thus permitting us to synthesise a flow-sensitive and not over-constraining scheduler, which needs fewer resources to meet requirements. The discrete-time stopwatch automata we are using are normal finite-state automata, where certain variables serve as *discrete time* clocks. The difference between the stopwatch automata we use and the (discrete) timed automata of [2,20,3] is that we can stop certain clocks (without resetting them) and restart them later on. Thus, we can easily model preemption. The difference with stopwatch automata [32,26,11] is that our clocks take discrete and not continuous values. As a consequence, reachability is decidable for discrete-time stopwatch automata, while it is not in general for continuous-time [26, 19].

### 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.

Communication primitives 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

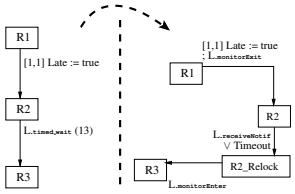



**Fig. 4** Modelling of the **wait** primitive

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. So, **wait** primitives are modelled by two states (R2, R2_Relock), as Fig. 4 shows using part of the Refresher's model from Fig. 3. The transition from the previous state to the first one (R1 → R2) makes the task leave the monitor, after having executed the action the program was performing there (`[1,1] Late := true`). The transition from the first to the second wait states (R2 → R2_Relock) 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), so as to reenter the monitor and continue its execution.

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, $C_i^{\text{Periodic}}$, 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.

## 4.2 System State

The system state model comprises: *(i)* an *abstract* program counter ($\text{PC}_i$) for each of the application tasks; *(ii)* a stopwatch ($SW_i$) for each task; *(iii)* $N$ periodic clocks ($C_i^{\text{Periodic}}$), 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 section 7); *(v)* a variable ($T_{\text{Exec}}$) 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.

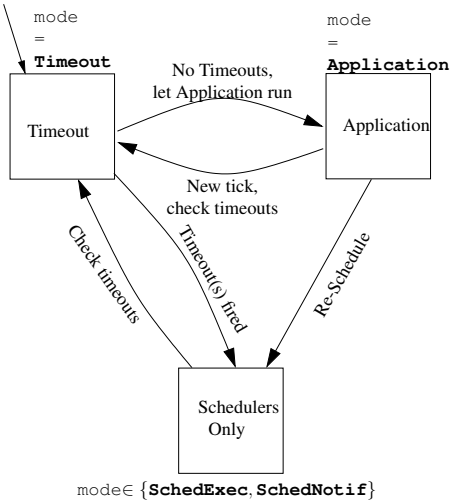



**Fig. 5** Model execution modes

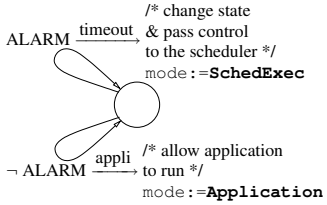



**Fig. 6** Timeout automaton

*Example 1* For the system of Fig. 3, the variables are:
$\text{PC}_{\text{Writer}} \in$ {W0,W1,W2,W3,W4,W5,W6},
$\text{PC}_{\text{Refresher}} \in$ {R0,R1,R2,R3,R2_Relock,R4,R5,R6,R7},
$\text{PC}_{\text{User}} \in$ {U0,U1,U2,U3,U6,U4,U5,U6_Relock},
$SW_{\text{Writer}} \in$ [0,6], $SW_{\text{Refresher}} \in$ [0,13],
$SW_{\text{User}} \in$ [0,6], $C_{\text{User}}^{\text{Periodic}} \in$ [0,19],
User_Alarm ∈ {false,true}, $T_{\text{Exec}} \in$ {IDLE,Writer,Refresher,User},
mode ∈ {Sched-Exec/Notif,Timeout,Application},
Late ∈ {false,true}, Fresh ∈ {false,true} ▲

## 4.3 Model Structure and Execution Modes

The system model we construct is the parallel composition 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. These are annotated with the timing constraints modelling the execution times of the corresponding code. These models can also be extracted automatically from application code, as was done in [28] and in [5] for annotated Java and C programs, respectively.

The system operates in three modes, as shown in Fig. 5. In **Timeout** mode the *Timeout* automaton (shown in Fig. 6) is the only one enabled in the system. It can fire one or

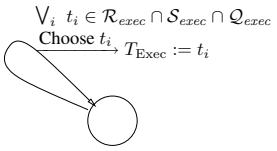



**Fig. 7** Execution Scheduler automaton

more timeouts (corresponding to a **timed_wait** or **waitForPeriod** 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 $T_{\mathrm{Exec}}$ application task becomes enabled. If the $T_{\mathrm{Exec}}$ 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 ($C_i^{\mathrm{Periodic}}$) to advance at the same time. It also causes local stopwatches ($SW_i$) to advance, if the respective task is executing, i.e., $T_{\mathrm{Exec}} = t_i$, or if it is performing a **timed_wait**. Ticks change mode back to **Timeout**, so as to check for new timeouts. If, however, $T_{\mathrm{Exec}}$ 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.

## 5 Scheduler Synthesis

This is essentially a two player game. 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 [4]. 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, $\mathrm{LR}(r,t)$:

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

$$\mathrm{LR}(r,t) = \{n \in \mathrm{N}(t)\ \mid \exists n_1, n_2 \in \mathrm{N}(t)\ .\ (n_1 \xrightarrow{r.\texttt{monitorEnter}} n_2 \rightarrow^* n) \wedge (\nexists n_3, n_4 \in \mathrm{N}(t).\ n_2 \rightarrow^* n_3 \wedge n_4 \rightarrow^* n \wedge n_3 \xrightarrow{r.\texttt{monitorExit}} n_4)\} \quad (2)$$

States with a **wait** transition are expanded as in Fig. 4. 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. 3, 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
$\mathrm{WR}(V, \mathrm{Refresher}) \cap \mathrm{LR}(L, \mathrm{Refresher}) = \{R3\}$ ▲

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. 3, such a solution is *wrong*. Indeed, variable Fresh will never become true and User will *never* finish its period (states U3, U4, and U5 are 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 [43] for its simplicity, can break the application logic itself by removing too many valid execution traces.

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

## 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. 7. $\mathcal{S}_{exec}$ 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 $T_{\mathrm{Exec}}$ 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:

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

*Example 3* This set is expressed as a predicate over model variables. Table 1(a) shows the synthesised predicate $\mathrm{Unsafe}($ $\mathrm{Refresher})$ for ensuring the timeliness property of $\mathrm{User}$ (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 $U1 \wedge C^{\mathrm{Periodic}}_{\mathrm{User}} = 11$), 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, $\mathrm{obs}$, 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, $\mathrm{PC}_i$, and those of the clocks, i.e., $SW_i$ and $C^{\mathrm{Periodic}}_i$, along with the value of the last task that was executing, $T_{\mathrm{Exec}}$. 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):

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

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

*Example 4* Again for the system of Fig. 3, the constraints we synthesise to render the system deadlock-free, once we have applied the projection on the state variables are:

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

Table 1(b) shows timeliness constraints for $\mathrm{Refresher}$, 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* [41]. The *bbe* reduction eliminates actions we do not wish to observe, called $\tau$ actions. Here, $\tau$ 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 $\tau$ 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.

**Table 1** Refresher timeliness constraints

(a) Refresher constraints when observing clocks and allowing preemption

$$\begin{aligned}&\texttt{LET } \mathbf{A} = (\mathrm{PC}_{\mathrm{User}}\texttt{=U5} \wedge C^{\mathrm{Periodic}}_{\mathrm{User}}\texttt{=13})\\ &\texttt{IN } (\mathbf{A} \wedge (\mathrm{PC}_{\mathrm{Refresher}}\texttt{=R0} \wedge (\mathrm{PC}_{\mathrm{Writer}}\texttt{=W0})\\ &\qquad\qquad \vee (\mathrm{PC}_{\mathrm{Writer}}\texttt{=W2} \wedge ((T_{\mathrm{Exec}}=\mathrm{PC}_{\mathrm{Refresher}} \vee T_{\mathrm{Exec}}=\mathrm{PC}_{\mathrm{Writer}})))\\ &\qquad\qquad \vee (\mathrm{PC}_{\mathrm{Writer}}\texttt{=W6} \wedge T_{\mathrm{Exec}}=\mathrm{PC}_{\mathrm{Writer}}))\\ &\quad \vee (\mathrm{PC}_{\mathrm{Refresher}}\texttt{=R3} \wedge \mathrm{PC}_{\mathrm{Writer}}\texttt{=W0})\end{aligned}$$

(b) Refresher constraints when not observing clocks

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

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 $\tau$ transitions or transitions labelled by some non-$\tau$ scheduler action $a$. Indeed, note that when the mode variable equals **SchedExec** or **SchedNotif** the current state has only non-$\tau$ transitions enabled (those of the scheduler automata), while in modes **Timeout** and **Application** we can only perform $\tau$ transitions. So, it is *never* the case that a state, $s$, can do both a $\tau$ 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 $\tau$ and vice versa*. So, the *controllable* equivalence classes are characterised by their *frontier*, which is exactly the member states having non-$\tau$ transitions. So, we define the frontier of a class, $c$, of *bbe*-equivalent states as in (7), where $\mathrm{enable}()$ produces the set of states enabling a particular transition. Note that the frontier of an uncontrollable equivalence class is the empty set, $\emptyset$:

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

### *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 $\mathrm{Bad}(c)$, i.e., the transitions the scheduler must not take in that class for the system to stay safe. If a $\tau$ action is a member of $\mathrm{Bad}(c)$ 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-$\tau$ transition. This effectively computes the set $\mathcal{S}_{exec}$. So, for all $s \in \mathrm{frontier}(c)$, where $\mathrm{Bad}(s) = \mathrm{Bad}(c)$:

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

When using the observation function $\mathrm{obs}$ 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 $\mathrm{obs}(s) = \mathrm{obs}(s')$ for some $s \in \mathrm{frontier}(c)$ and $s' \in \mathrm{frontier}(c')$, or, equivalently: $\mathrm{obs}(\mathrm{frontier}(c)) \cap \mathrm{obs}(\mathrm{frontier}(c')) \neq \emptyset$.
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:

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

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

## 6 A Methodology for Synthesis

Despite the *bbe* reduction, the size of the state space can still be considerable. 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 [27].

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. For the case study of section 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. 3, 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 protocols used are *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, using more processors, etc. This is particularly important for product families, since there the timing information differs for each family member [13].

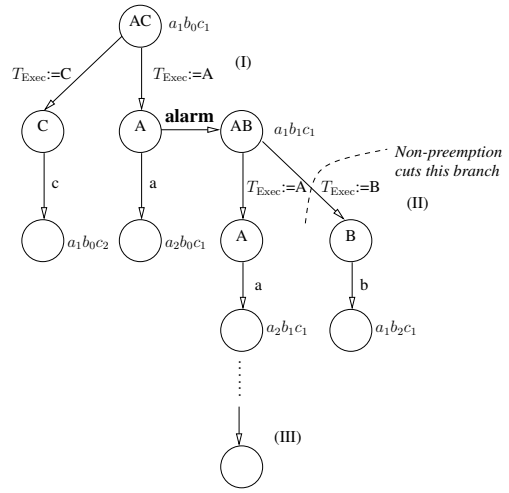
Having found all the deadlocks in the untimed system, we impose the synthesised $\mathcal{S}_{exec}$ and $\mathcal{S}_{notif}$ scheduler constraints upon the *timed* model, and search for *timeliness* constraints, so that all tasks will meet their deadlines.

### 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. 8. As shown there, when imposing the non-preemptive execution model at state AB we are effectively cutting the branch AB → B, where the scheduler chose to preempt the execution of task A with task B after the alarm. This kind of




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

**Fig. 8** Preemption and state space size

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 → A, then adding preemption at the next stage will simply add branch AB → B. If, however, branch AB → A in the non-preemptive model is unsafe, then we will be obliged to constrain the system earlier on (since now branch AB → B is not available). If we needed to constrain the system at state AC, by cutting branch AC → A and selecting branch AC → C, then permitting preemption later on would mean that the whole sub-graph after branch AC → 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:

$$\mathcal{Q}_{exec}(\text{obs}(s)) := \{t \mid \text{computes}(t) \vee (t \in \mathcal{S}_{exec}(\text{obs}(s)) \wedge \neg\exists t' \neq t \,.\, \text{computes}(t'))\} \quad (11)$$

*Example 6* For the system of Fig. 3, 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)*). ▲

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 eq. (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 energy 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 eq (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 eq. (12) has been applied to the safe system of line *(8)* and then the LMCS policy has been applied on top of it.

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

### 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), see section 7.

**Table 2** Synthesis results for the system of Fig. 3

*T/U: **T**imed/**U**ntimed model, P/NP: **P**reemption/**N**o-**P**reemption*

| | Model kind | States | Red. | Bad | Constraints Used/Prod. |
|---|---|---|---|---|---|
| | ***Synthesis Steps for the system of Fig. 3.*** | | | | |
| *(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. 3, 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 eq.* (12).** | | | | |
| *(11)* | *(8)* & eq. (12) | 2665 | 87.74% | 0 | 167 / 0 |
| *(12)* | Model of *(11)*, *bbe* | 285 | 98.69% | 0 | 167 / 0 |
| | ***QoS policy of eq.* (12) *and then LMCS on system of (8) .*** | | | | |
| *(13)* | *(8)* & eq. (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 |

*Example 7* Table 2 shows the results of our methodology for the system of Fig. 3. 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 167. ▲

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 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.

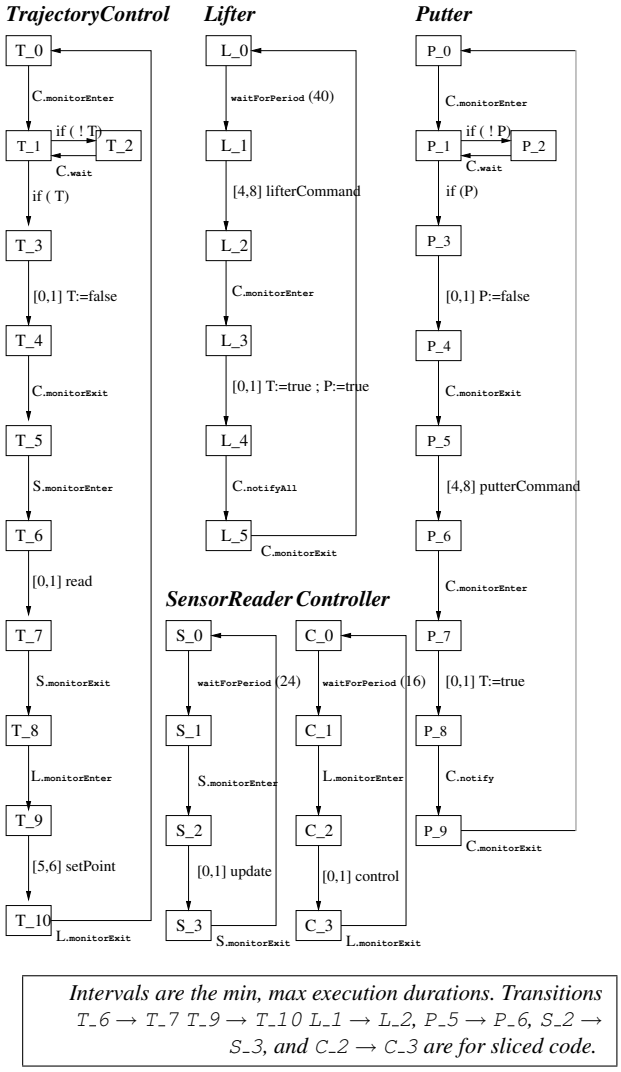



**Fig. 9** Control-flow graphs of the tasks of a robotic arm

## 7 Case Study: A Robotic Arm

In this section we consider a case study based on a robotic arm system from [42], shown in Fig. 9. The arm takes objects from a conveyor belt, stores them temporarily in a buffer shelf, and puts them into a basket. The arm is controlled by tasks running on a single processor.

Fig. 9 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 5ms and 8ms. There are two motion executors, `Lifter` and `Putter`. `Lifter` is activated periodically every 40ms. 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 4ms to 9ms. `Putter` sends commands to move the object from the shelf into the basket. Its execution time is between 4ms to 10ms. The `SensorReader` task reads sensors every 24ms. Its execution time is 1ms. Sensor readings are used by `TrajectoryControl.Controller` is a periodic task with a period of 16ms.

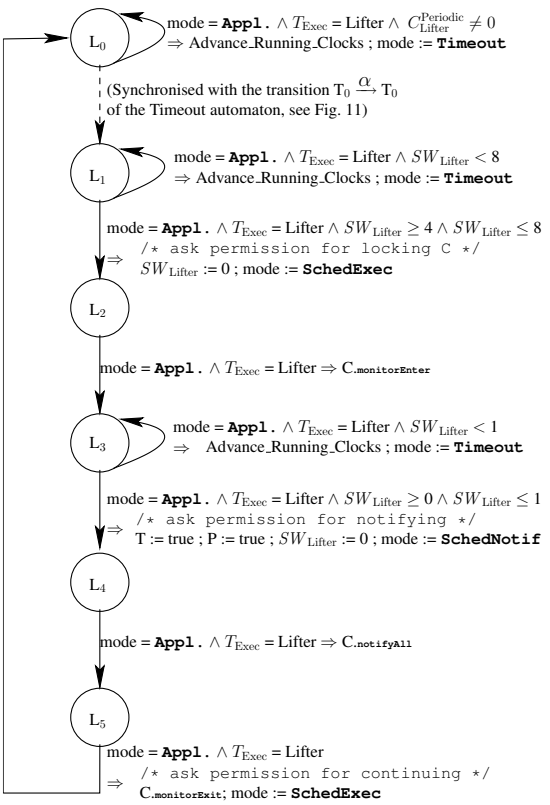



**Fig. 10** Lifter's automaton

### 7.1 Stopwatch Automata Model of the Robotic Arm

Fig. 10 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`** ($L_1 \rightarrow L_2$), 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.

Fig. 11 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

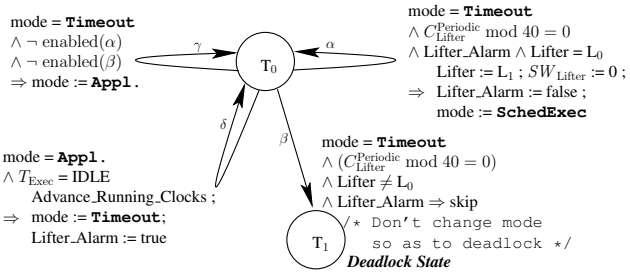



**Fig. 11** Timeout automaton (Lifter's part)

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 $C^{\text{Periodic}}_{\text{Lifter}} = 0$ (start of period) and $C^{\text{Periodic}}_{\text{Lifter}} = \mathbf{40}$ (deadline). In the former case Lifter_Alarm is false and thus the deadlocking transition $\beta$ is disabled, while in the latter case Lifter_Alarm is true and transition $\beta$ 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 4 tasks, namely, `Lifter`, `Putter`, `SensorReader`, and `TrajectoryControl`, and another 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,

**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 |

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 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*. Time-dependent schedulers need in addition alarms and *response* time timers, if the deadlines on the computations have been transformed to response times. Such timers are available in almost all *OS*'s. If, however, deadlines refer to *execution* times, then we need timers capable of measuring the exact execution time of computations, even in the presence of preemptions, which are not 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*'s, FAST-OS does not allow direct setting/observation of timers at all. In the following, we present the core implementation of time-independent schedulers.

The generated 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 **`waitForPeriod`**. 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.

```
1  int interrupted_task = IDLE;
2
3  void U_Scheduler(int tc, bool in_notify, timespec &deadline) {
4    bool finished = true, level_super = false;
5    int  tn, i;
6    // tc is the current thread (t_current) & tn the next one (t_next)
7    do {
8      // calculate Ready, Safe & Quality sets
9      tn = Synthesized_Constraints(THREADS_TABLE);
10     if (tn != tc) {
11       if (in_notify) {
12         if (-1 != tn) { // -1 means no thread is waiting
13           // t_next has priority BLOCKED so it cannot preempt t_current
14           notify(THREADS_TABLE[tn].cv);
15         }
16       } else { // ! in_notify
17         if (!level_super) interrupted_task = tc;
18         U_Set_Priority(tn, EXECUTING);
19         THREADS_TABLE[tn].PC = THREADS_TABLE[tn].PC_Notif;
20         notify(THREADS_TABLE[tn].cv);
21
22         if (NULL == deadline){// Not a waitTimed or waitForPeriod
23           // Release sched_mx here
24           wait(THREADS_TABLE[tc].cv, sched_mx);
25           // Here I have been signaled
26         } else {                  // NULL != deadline
27           U_Set_Priority(tc, INTERRUPT);//Release sched_mx
28           timed_wait(THREADS_TABLE[tc].cv, sched_mx, deadline);
29           /* Here I have been signaled or I have timed-out.
30              Must re-schedule to be safe, if I timed-out. */
31           if (THREADS_TABLE[tc].PC != THREADS_TABLE[tc].PC_Notif) {
32             finished = false;
33             THREADS_TABLE[tc].PC = THREADS_TABLE[tc].PC_Timeout;
34           }
35           level_super = true ; deadline = NULL;
36         }
37       }
38     }
39   } while (! finished);
40   if (level_super) {
41     U_Set_Priority(tc, EXECUTING);
42     U_Set_Priority(interrupted_task, BLOCKED);
43     interrupted_task = tc;
44   }
45 }
46 void U_monitorEnter(obj o, int tc, int curr_pos, int next_pos) {
47   lock(sched_mx);
48   THREADS_TABLE[tc].PC = curr_pos;
49   U_Scheduler(tc, false, NULL);
50   lock(o.mutex); // We lock the object once we've got permission
51   THREADS_TABLE[tc].PC = next_pos;
52   unlock(sched_mx);
53 }
54 void U_monitorExit(obj o, int tc, int curr_pos, int next_pos) {
55   lock(sched_mx);
56   THREADS_TABLE[tc].PC = curr_pos;
57   unlock(o.mutex);// Unlock the object before calling U_Scheduler
58   THREADS_TABLE[tc].PC = next_pos;
59   U_Scheduler(tc, false, NULL);
60   unlock(sched_mx);
61 }
```

**Fig. 12** Pseudo-code of the application scheduler

**Table 4** Timing primitives under `eCos` *(results in $\mu$s)*

| 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 |

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.

Fig. 12 shows the pseudo-code of the implementation. Before calling `U_Scheduler`, our **`monitorEnter`** locks `sched_mx`

and updates the application task's position to be the same as in the model (lines 47–48). U_Scheduler calls Synthesized_Constraints (generated by the synthesis tool) in line 9, passing it the current task PC's. 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 a **waitForPeriod**. 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/U_wait (not shown in Fig. 12) 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/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*'s, namely an Intel Pentium II (333MHz) 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$\mu$s. 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. 12 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**/U_**waitForPeriod**) 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 proposed by RTSJ [34].

## 9 Scheduler Robustness

Synthesised schedulers can be intolerable to wrong estimation of computation WCET's. In fact, a computation should not finish earlier than its 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 to 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 is explicitly meant to guard against such a situation).

The simplest solution for 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* so that we know how long the computation has executed. Unfortunately, such timers are not currently supported by many *OS*'s. 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's 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 equal to the WCET. 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 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 [46] and builds upon [1,4]. Controller synthesis for timed automata was also considered in [22], where the problem is reduced to the untimed framework of [46] using the *region graph* construction that results in state space explosion. [45] considers the more general setting of linear hybrid automata and presents a semi-decision procedure. The approach of [30] 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 [33]. 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 [24], 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 [39]. Concerning state-space explosion, it is interesting to note Wang et al. [43], 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, to identify *potential* deadlocks. Apart from the fact that not all potential deadlocks are real, the main problem with [43] is the solution advocated - to add extra locks to render deadlocks impossible. As we have shown at the end of section 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. 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 [43] or these based on RMA-type analyses.

Our approach fits into the schedule carrying code paradigm proposed for Giotto [18]. However, our solution is based on controller synthesis, while [18] relies on RMA/EDF, thus suffering their problems. Also, the Giotto compiler must be extended for each different scheduling theory, while our compilation infrastructure remains unchanged.

Several have considered quality requirements for rate-monotonic scheduling. For instance, [14] 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 [15] 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, [12] 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 in-

dividually 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 [8] 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 system case that gets analysed.

## 11 Conclusions

Scheduling system tasks so as to meet multiple QoS requirements is an extremely difficult but at the same time very important task. We have introduced a correct-by-construction 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 scheduler synthesis methodology allows to break the synthesised schedulers 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*'s (`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 [28]: the model extraction and synthesis steps were interfaced with the Java-to-C TurboJ compilation chain [44] and our controller subsystem was part of the Expresso executive [16]. 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* [5] and STMicroelectronics's FlexCC2 compilation technology [7].

## References


1. K. Altisen, G. Gößler, and J. Sifakis. Scheduler modeling based on the controller synthesis paradigm. *R-T Sys.*, 23(1):55–84, July 2002.
2. R. Alur, C. Courcoubetis, and D. L. Dill. Model checking in dense real time. *Inf. and Comp.*, 104(1):2–34, 1993.
3. R. Alur and D. L. Dill. A theory of timed automata. *Th. Comp. Sci.*, 126(2):183–235, April 1994.
4. E. Asarin, O. Maler, and A. Pnueli. Symbolic controller synthesis for discrete and timed systems. In *HS-II*, volume 999 of *LNCS*, pages 1–20, 1995.
5. I. Assayad, V. Bertin, F.-X. Defaut, Ph. Gerner, O. Quevreux, and S. Yovine. Jahuel: A formal framework for software synthesis. In *ICFEM'05*, volume 3785 of *LNCS*, pages 204–218, November 2005.
6. Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, March/April 2003.
7. V. Bertin, J.-M. Daveau, P. Guillaume, T. Lepley, D. Pilat, C. Richard, M. Santana, and T. Thery. FlexCC2: An optimizing retargetable C compiler for DSP processors. In *EMSOFT'02*, pages 382–398, 2002.
8. Matteo Bordin, Marco Panunzio, and Tullio Vardanega. Fitting schedulability analysis theory into model-driven engineering. In *ECRTS'08*, pages 135–144, July 2008.
9. Eric Bruno. Java RTS real-time enables financial applications. (Online) `www.devx.com/Java/Article/35246/`, August 23 2007.
10. Nicholas Carr. Avatars consume as much electricity as Brazilians. (Online) `http://www.roughtype.com/archives/2006/12/avatars_consume.php`, December 05 2006.
11. F. Cassez and K. G. Larsen. The impressive power of stopwatches. In *CONCUR'00*, volume 1877 of *LNCS*, pages 138–152, 2000.
12. J. Combaz, J.-C. Fernandez, J. Sifakis, and L. Strus. Symbolic quality control for multimedia applications. *R-T Sys.*, 40(1):1–43, October 2008.
13. James Coplien, Daniel Hoffman, and David Weiss. Commonality and variability in software engineering. *IEEE Software*, 15(6):37–45, November/December 1998.
14. R. Dobrin and G. Fohler. Reducing the number of preemptions in fixed priority scheduling. In *ECRTS'04*, pages 144–152, June 2004.
15. G. Fohler and G. C. Buttazzo. Introduction to the special issue on flexible scheduling. *R-T Sys.*, 22(1/2):5–7, January 2002.
16. L. Gauthier and M. Richard-Foy. Expresso RNTL project - High Integrity Profile. `www.irisa.fr/rntl-expresso/docs/hip-api.pdf`, 2002.
17. M. G. Harbour, M. H. Klein, and J. P. Lehoczky. Timing analysis for fixed-priority scheduling of hard real-time systems. *IEEE TSE*, 20(1):13–28, 1994.
18. T. Henzinger, C. Kirsch, and S. Matic. Schedule carrying code. In *EMSOFT'03*, volume 2855 of *LNCS*, October 2003.
19. T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? *J. of Comp. and Sys. Sci.*, 57(1):94–124, August 1998.
20. T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Inf. and Comp.*, 111(2):193–244, 1994.
21. T. A. Henzinger and J. Sifakis. The discipline of embedded systems design. *Computer*, 40(10):32–40, 2007.

22. G. Hoffmann and H. Wong Toi. The input-output control of real-time discrete event systems. In *CDC'91*, 1991.
23. IEEE. *POSIX.1. IEEE Std 1003.1:2001. Standard for Information Technology - Portable Operating System Interface (POSIX)*. IEEE, 2001.
24. D. Isović and G. Fohler. Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints. In *RTSS'00*, November 2000.
25. B. Joy, G. Steele, J. Gosling, and G. Bracha. *The Java Language Specification (2nd edition)*. Addison-Wesley, 2000.
26. Y. Kesten, A. Pnueli, J. Sifakis, and S. Yovine. Decidable integration graphs. *Inf. and Comp.*, 150:209–243, 1999.
27. C. Kloukinas. Data-mining synthesised schedulers for hard real-time systems. In *ASE'04*, pages 14–23, September 2004.
28. C. Kloukinas, C. Nakhli, and S. Yovine. A methodology and tool support for generating scheduled native code for real-time Java applications. In *EMSOFT'03*, volume 2855 of *LNCS*, October 2003.
29. C. Kloukinas and S. Yovine. Synthesis of safe, QoS extendible, application specific schedulers for heterogeneous real-time systems. In *ECRTS'03*, pages 287–294, July 2003.
30. H. Kwak, I. Lee, A. Philippou, J. Choi, and O. Sokolsky. Symbolic schedulability analysis of real-time systems. In *RTSS'98*, December 1998.
31. C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *JACM*, 20(1):46–61, January 1973.
32. J. McManis and P. Varaiya. Suspension automata: A decidable class of hybrid automata. In *CAV'94*, volume 818 of *LNCS*, pages 105–117, 1994.
33. A. K. Mok, D. C. Tsou, and R. C. M. Rooij. The MSP.RTL real-time scheduler synthesis tool. In *RTSS'96*, December 1996.
34. Real-Time for Java Expert Group. The real-time specification for Java. Tech. report, RTJ.org, December 2001.
35. Glenn E. Reeves. What really happened on mars? (Online) `http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/Authoritative_Account.html`, December 1997.
36. Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, February 2006.
37. L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE TC*, C-39(9):1175–1185, September 1990.
38. S. Tripakis. Description and schedulability analysis of the software architecture of an automated vehicle control system. In *EMSOFT'02*, October 2002.
39. S. Tripakis and K. Altisen. On-the-fly controller synthesis for discrete and dense-time systems. In *FM'99*, volume 1708 of *LNCS*, September 1999.
40. S. Tripakis and S. Yovine. Timing analysis and code generation of vehicle control software using Taxys. In *RV'01*, volume 55(2) of *El. Notes in Th. Comp. Sci.*, pages 277–286, July 2001.
41. R. J. van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics. *JACM*, 43(3):555–600, 1996.
42. Enrico Vicario. Static analysis and dynamic steering of time-dependent systems. *IEEE TSE*, 27(8):728–748, August 2001.
43. Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. A. Mahlke. The theory of deadlock avoidance via discrete control. In *POPL'09*, pages 252–263, January 2009.
44. M. Weiss, F. de Ferrière, B. Delsart, C. Fabre, F. Hirsch, E. A. Johnson, V. Joloboff, F. Roy, F. Siebert, and X. Spengler. TurboJ, a Java bytecode-to-native compiler. In *LCTES'98*, volume 1474 of *LNCS*, pages 119–130, June 1998.
45. H. Wong Toi. The synthesis of controllers for linear hybrid automata. In *CDC'97*, pages 4607–4612, 1997.
46. W. M. Wonham and P. J. Ramadge. On the supremal controllable sublanguage of a given language. *SIAM J. of Control and Optimization*, 25(3):637–659, May 1987.