# Modelling, Verification and Synthesis of Two-Tier Hierarchical Fixed-Priority Preemptive Scheduling

Mikael Åsberg, Paul Pettersson and Thomas Nolte
MRTC/Mälardalen University
P.O. Box 883, SE-721 23 Västerås, Sweden
{mikael.asberg, paul.pettersson, thomas.nolte}@mdh.se

***Abstract*—Hierarchical scheduling has major benefits when it comes to integrating hard real-time applications. One of those benefits is that it gives a clear runtime separation of applications in the time domain. This in turn gives a protection against timing error propagation in between applications. However, these benefits rely on the assumption that the scheduler itself schedules applications correctly according to the scheduling parameters and the chosen scheduling policy. A faulty scheduler can affect all applications in a negative way. Hence, being able to guarantee that the scheduler is correct is of great importance. Therefore, in this paper, we study how properties of hierarchical scheduling can be verified. We model a hierarchically scheduled system using task automata, and we conduct verification with model checking using the Times tool. Further, we generate C-code from the model and we execute the hierarchical scheduler in the VxWorks kernel. The CPU and memory overhead of the modelled scheduler is compared against an equivalent manually coded two-level hierarchical scheduler. We show that the worst-case memory consumption is similar and that there is a considerable difference in CPU overhead.**

***Index Terms*—real-time systems, hierarchical scheduling, modelling, formal verification, code-synthesis**

## I. Introduction

Hierarchical scheduling [1], [2], [3] has been introduced as a means to simplify parallel development of embedded systems. It facilitates the integration of such systems by providing mechanisms for temporal isolation between software parts, called subsystems. The schedulable entity manifested by a subsystem is referred to as a *Server*. A system (a product, a large piece of software etc.) can be composed of a number of subsystems, where each of these typically implement a particular function or feature of the whole system. For example, a car has a number of features/subsystems, and two examples of these are the engine control system and the anti-lock braking system. These features/subsystems should ideally be developed in parallel and integrated smoothly [4]. Integration related problems include having to cope with different scheduling policies among subsystems, sharing the CPU resource among subsystems according to their need (and keeping that share during runtime), and ensuring that timing faults do not propagate from one subsystem to another. An example of such a fault is a piece of software that requires more time to execute than originally intended (exceeding its analysed worst-case execution time), and thereby causing unforseen interference with the rest of the system. Yet another integration problem is the introduction of new software functions, not apparent at early design.

Hierarchical scheduling allows for timing analysis of an entire system, as well as for subsystems in isolation, before they are integrated. It supports multiple scheduling policies and it has a runtime mechanism that multiplexes the CPU resource among subsystems, hence, making sure that no unpredictable interference between subsystems will occur in the time domain. Also, the size of the CPU share can easily be re-configured, allowing for ”last minute” changes when introducing new software late in the development process.

One important property of hierarchical scheduling, when it comes to hard real-time applications, is the safe execution environment for a subsystem. The scheduling entity of a subsystem, i.e., a server, should ensure (together with the scheduler) that the subsystem will get the exact CPU share that it was promised. Even though a subsystem is executed together with other (potentially faulty) subsystems, it should still get the CPU share that it is entitled to. In practice, hierarchical scheduling can prevent faulty subsystems from propagating timing faults to other subsystems. However, hierarchical scheduling cannot deal with timing faults propagating from itself, i.e., a faulty scheduler causing incorrect scheduling events, and thereby violating the contracted CPU shares that belong to the subsystems. This is of course not acceptable in applications with hard real-time constraints.

We have experience in the implementation of two-level hierarchical scheduling frameworks in operating systems such as VxWorks [5] and Linux [6]. Our implemented frameworks operate in two levels using periodic/polling servers (PS) [7] and, inside these, fixed priority preemptive scheduling (FPPS) of periodic tasks. Even though the setup of these frameworks are quite simple (two-level, PS and FPPS), it gives rise to a large implementation complexity, since we are dealing with multiple schedulers (multiple scheduling-related timing events). From our experience, debugging/tracing of this kind of scheduling [6] is very time consuming. Also, debugging/tracing does not guarantee 100% correctness, since it can be difficult to determine wheather the schedule is correct or not. Due to this, in this paper we look at modelling, formal verification and code-synthesis of hierarchical scheduling with FPPS.

The motivation for modelling hierarchical FPPS is inherent in its wide support for schedulability analysis [8], [9], [10], as well as the evolving research in synchronisation protocols [11], [12], which need hierarchical scheduling implementations/models for its development and evaluation.

Recently, automata based approaches have been proposed to describe/analyse a broad set of real-time scheduling policies. One of the advantages of these approaches is the ability to generate generic task release patterns. In task automata models [13], task release patterns are modelled using timed automata [14]. It has been shown that the schedulability analysis problem is resolvable for both FPPS and dynamic scheduling policies such as earliest deadline first (EDF). Other benefits of such approaches are that simulation, formal verification of timing/functional safety properties, as well as code-synthesis [15] is possible. The Times tool [16] supports modelling with the task automata model, and it can perform simulation, verification, code-synthesis etc. However, hierarchically scheduled systems cannot be verified using existing solutions.

In this paper our overall goal is to model, verify and synthesise a two-level hierarchical scheduling framework. The main contributions of this paper are:

1) We have modelled two-level hierarchical scheduling, with FPPS and PS at the global level with support for an arbitrary number of servers with FPPS and periodic tasks at the local level. We have used the modelling language task automata and implemented the model using the Times tool. To the best of our knowledge, this is the first task-automata model of two-tier FPPS with PS.
2) We have extended the model with support for verification (using what we call *observers*), allowing us to verify that the model matches the scheduler behavior (properties) that we have specified. Note that we are NOT verifying schedulability analysis, but the scheduler itself (two-level FPPS with periodic tasks/servers). The contribution to the state-of-the-art is the verification of the schedulers (scheduling policies) in a hierarchically scheduled system.
3) We have used the built-in code generator in Times to synthesise our model. However, the manual work needed includes adapting the code for our large model (which has 370 edges and 155 locations), since the Times code-generator currently supports a limited size. This work also includes removing platform (Linux simulator) dependent code, and inserting VxWorks related code. This gives us the possibility to get real overhead estimates of the modelled scheduler when executing it. The results presented are the actual execution traces of the scheduler executed in the VxWorks kernel, as well as a comparison of CPU- and memory-overhead against an equivalent manually-coded hierarchical scheduler. To the best of our knowledge, there is no prior work on synthesis (from model) for this type of scheduling.

The outline of this paper is as follows: in Section II we outline preliminaries on hierarchical scheduling, task automata and Times. In Section III we present the model of two-level hierarchical scheduling, in Section IV we show how we have verified the behavior of the modelled scheduler, and finally in Section V, we show the result of the synthesis. Section VI presents related work, and finally, Section VII concludes.

## II. Preliminaries

### A. Hierarchical scheduling

Hierarchical scheduling has been introduced to support CPU multiplexing in combination with different scheduling policies. It can generally be represented as a tree of nodes with arbitrary size, where each node represents a subsystem with its own local scheduler for scheduling internal workloads (tasks). Looking at the tree-structure representation, the CPU resource is allocated from a parent node to its children nodes. One of the main advantages of hierarchical scheduling is that it provides means for decomposing a complex system into well-defined parts (subsystems). In essence, hierarchical scheduling gives rise to time-predictable *composition* of coarse-grained subsystems. This means that subsystems can be developed and tested independently, and at a later stage assembled without introducing unwanted temporal behavior. Hierarchical scheduling also facilitates *reusability* of subsystems, since their computational requirements are characterised by well defined *interfaces*.

Figure 1 illustrates two-level hierarchical scheduling. The left side illustrates the structure: the top node is defined as the *Global scheduler* and it is responsible for distributing the *CPU* capacity to the servers (the schedulable entity of a subsystem). Servers are allocated a defined time (*budget*) every predefined *period* [17] and they are executed based on their *priority*. They are scheduled according to the scheduling policy of the global scheduler (for example FPPS or EDF) and the parameters just mentioned, hence, they can be viewed as ”virtual tasks”. Each server can comprise a *Local scheduler* which schedules the workload inside it, i.e. its tasks, when its server is selected for execution by the global scheduler. Note that the local scheduling policy may differ from the global policy. The interfaces (T,C,Pr) for tasks and servers shows the allocated CPU capacity. It includes the release period, execution time (or budget in the case for a server) and priority (lower value corresponds to higher priority). The right side of the figure corresponds to the runtime behavior of the structure.

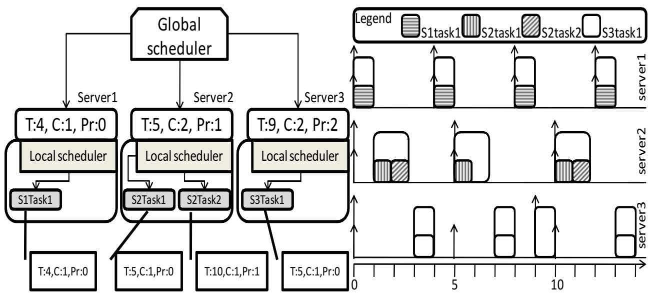


Fig. 1. Example hierarchical FPPS

### B. Task automata and Times

*Timed automata* [14] is a widely used modelling language for formal modelling and analysis of real-time systems. A timed automaton is essentially a finite state automaton extended with real-valued clocks that can be tested and reset. The formalism has shown to be suitable for a wide range of real-time systems.

The timed automata model has been extended with an explicit notion of tasks, with parameters such as periods, priorities, execution times etc. The model, referred to as *task automata* (of *timed automata with tasks*), associates asynchronous tasks with the locations (states) of a timed automaton, and assumes that the tasks are executed using static/dynamic priorities with a preemptive or non-preemptive scheduling policy. This model is supported by the Times tool. One of the main benefits of using this tool (in the context of this paper) is that it supports task automata, which is suitable for modelling schedulers. Secondly, it can verify properties of a modelled system. Last but not least, the tool has a code-generator which gives the possibility for synthesis.

In case that tasks are released periodically (with or without offsets), or aperiodically, the input to the Times tool is merely a task table in which the following parameters are defined for a task: name, execution time, (relative) deadline, priority (in case of static priority scheduling), offset and period (if applicable), interface, semaphore usage, and its C-code. Alternatively, a task can be of type *controlled*, meaning that the release pattern of a task is defined by a given task automata. All tasks in our modelled hierarchical scheduler are of type controlled.

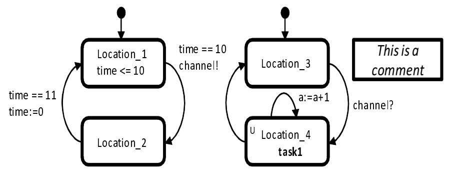


Fig. 2. Example task automata

Figure 2 shows an example of a task automata that releases a (*controlled*) task for execution, at minimum, every 10 time units. The arrows (with a dot) to state **Location_1** and **Location_3** defines that they are the start locations. The invariant **time <= 10** defines that control can only be at this state up until time 10, then a transition has to be made. The condition **time==10** defines that a transition may take place if this holds. The channel **channel!** defines that when this transition is made, the corresponding channel **channel?** must be activated, i.e., there has to be a transition between state **Location_3** and **Location_4**. The latter location has a task release statement (**task1**), and this means that upon arrival at this state, task *task1* is released for execution. State **Location_4** is flagged as *urgent* (**U**), which defines that no time will pass when computing **a:=a+1** or before the transition to state **Location_3**. A transition from state **Location_2** to **Location_1** may take place when **time==11**, if so, the clock **time** will be reset to zero.

## III. Model

This section will describe the hierarchical scheduler, modelled in Times. The modelling language of task automata is used for modelling the framework. This language allows task releasing, and transitions/actions can be controlled with clock constraints (as shown in Figure 2). However in general, in order to implement hierarchical scheduling, one either need to be able to release tasks and suspend them, **or**, release tasks and change task priorities dynamically during runtime (in order to perform a server context switch). Unfortunately, task suspension and dynamic priority (of controlled tasks) is not supported by the Times tool. In order to solve this issue, we model an executing task as a series of task releases, where each task release will execute the task 1 time unit. Hence, the minimum task execution time is 1 time unit, and the execution time is discrete, i.e., it has to be divisible by 1 (without generating a remainder). What this means in practice, is that when there is a task executing within a server and its budget depletes, then we simply stop releasing the task (and take a note of the amount of time executed so far). This is illustrated in Figure 3 where a task is supposed to execute 5 time units, within 2 budget instances of its server. This results in 3 task releases at the first server instance and 2 releases in the second instance. This fragmentation does not affect the task model, schedulability analysis or verification, it just makes the task automata model more complicated to implement. A more practical approach is to only model task releases and no actual task execution (hence there will be no task suspension in the model). The downside of such a non-fragmented approach is restricted verification capabilities as well as no possibilities of graphical representation during simulation (Figure 8). We will show verification using the fragmented task model, and we will show code-synthesis for both the fragmented and the non-fragmented model (Section V).

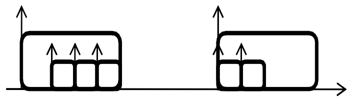


Fig. 3. Discrete task execution

The model structure is illustrated in Figure 4. The global scheduler activates the servers with channels, through the *EventHandler* automata. The global scheduler is unchanged when adding/deleting servers, only the *EventHandler* is affected. Servers are activated periodically and they run according to their budget and priority, i.e., PS with FPPS. In our model, Server 3 has a local scheduler, scheduling periodic tasks with FPPS. Server 1 has no scheduler, i.e., it just releases a task upon activation and lets it run until budget depletion.

Each scheduler (global or local) has a ready- and a release-queue. The ready-queue contains the servers/tasks, ordered by priority. The release-queue stores the release times (in absolute time) of the servers/tasks, ordered with the earliest time first. The queues are implemented as arrays and insertion is based on a binary search algorithm.

As mentioned previously, a server is activated/deactivated through channels (where the global scheduler is the initiator). This means that a server must always be prepared to be activated/deactivated, i.e., all of its states which are not marked as *urgent* must have an activation/deactivation channel. If this is fulfilled, then the server will be in total control by the global scheduler, hence, scheduling errors will not propagate from local to global level. Also, if the global scheduler is verified, then the local scheduler can assume that it is getting its correct timeslots (according to its interface), making verification at the local level easier (the power of compositional verification). The local scheduler releases its tasks according to the model illustrated in Figure 3, which will prevent the tasks from executing outside of its servers budget (the Times simulation in Figure 8 illustrates this).

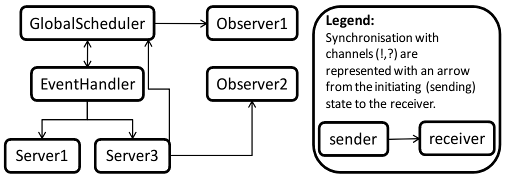

GlobalScheduler
Observer1
EventHandler
Observer2
Server1
Server3
Legend:
Synchronisation with channels (!,?) are represented with an arrow from the initiating (sending) state to the receiver.
sender
receiver

Fig. 4. Structure of the model

*Observer1* and *Observer2* (Figure 4) will get notifications of scheduling events through channels. We define scheduling events as being task/server releases, server budget depletion and task suspension (due to the task finishing its current execution). The observers themselves do not initiate these synchronisations and they do not affect the clocks, hence, they do not affect the behavior of the model. The observers are mainly used for the purpose of verifying the schedulers [18], this will be elaborated in more detail in Section IV.

## A. Global scheduler

Figure 5 illustrates a simplified version of the global scheduler. The excluded parts include initialisation, queue management etc. Basically, whenever there are no scheduling events, the automata waits in the main state, i.e., the one without the *urgent* symbol (**U**). This is the only state where time is allowed to pass. From the main state, there are in total three transitions possible: server budget deplete, server release and allowing for a task-event (i.e., task release etc.) that belongs to the current active server. As can be seen, the depletion transition has highest priority, followed by the release and task-event transitions. The latter is necessary since the global scheduler needs precedence over local scheduling events when they occur at the same time. As with the priority of the other two, it is simply more convenient to handle a budget-deplete event before a release event (when they occur at the same time).
As can be seen by the model, we model that scheduling events do not consume any time (hence the *urgent* symbols). The reason for this is to reduce the complexity of the model. This means that during simulation, the scheduler produces no overhead. However, running experiments would of course yield some scheduler overhead, these details will be shown in Section V.

$Observer1$ is notified about server budget-deplete (**DepleteObs1!**) and server release events (**ReleaseObs1!**), this is shown in Figure 5.

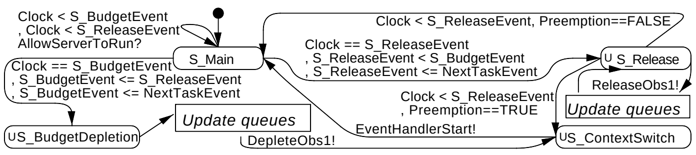


Fig. 5. Model of the global scheduler (simplified)

## B. Event handler

Figure 6 shows the model of the event handler. The motivation for its existence is that it abstracts the number of servers from the global scheduler, i.e., adding/removing servers only affects the number of states in the event handler and not in the global scheduler. Since *channels* cannot be declared as arrays, every server requires 2 states (activation and deactivation) in this model. As can be seen in this model, the global scheduler observer (*Observer1*) is notified if there is a server scheduling event, and which servers that are activated/deactivated.

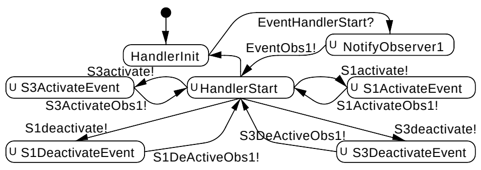


Fig. 6. Model of the event handler (simplified)

## C. Local scheduler

The local scheduler model (Figure 7) is similar to the global scheduler. Discretising the time is important for keeping track of events, hence the added time pass state that increments time (clocks are not allowed be read in timed automata). The time-pass state is crucial since the local scheduler has more scheduling events to keep track of, compared to the global scheduler.
Whenever the server is deactivated, it stays in the sleep state. In active mode, the server can release, stop and increment a tasks execution. The latter goes back to the statement that a tasks execution is discrete with sections of 1 time unit of execution.
*Observer2* is notified of events by getting triggered by the local scheduler through a number of channels.
Each upcoming task scheduling-event must be passed to the global scheduler so that it does not schedule a server event (such as deactivating the server) without letting the local scheduler handle task scheduling events that are earlier in time. The upcoming task scheduling event is calculated in the **CalcNextEvent** state and stored in the **NextTaskEvent** variable, which is visible in the global scheduler.

All models (including schedulers, observers etc.) can be viewed in our technical report [19].

## IV. VERIFICATION

We have specified 5 respectively 4 properties for each scheduler level (global/local) that should be satisfied by our modelled schedulers. We use two so called $observer$ automata that will implement the behavior (properties) that we have

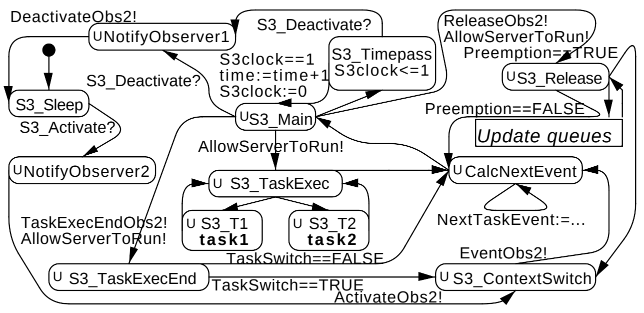


Fig. 7. Model of the local scheduler (simplified)

specified. The next step is to use the built in verifier in Times, and simply construct logic statements (TCTL) that checks if certain states are reached in the observers. The observers will reach these states if they detect a scheduling fault that contradicts our proposed properties. $Observer1$ is used to verify the global scheduler, and $Observer2$ is used for the verification of the local scheduler. The reason for using observers, instead of only using logic statements in Times, is that the verifier cannot determine the amount of time elapsed from one location to another, which we need in order to conduct our verification. Naturally, all automata have been checked for the absence of deadlock before proceeding with the verification.

### A. *Task/server systems used in the verification*

It is well known that model checking requires a finite model, and thus, it might cause problems when verifying schedulers [20], [21] since the tasks give rise to unknown factors such as number of tasks, task parameters etc. In essence, different task sets will give rise to different automata transitions (behavior), so the scheduler will behave different depending on task sets. Due to this, we explore the fact that the modelled scheduler has a small set of scheduling events (task/server release, task/server suspension, context switch etc.), even when including the combinations of these events (as we will see). We identify all of these events, which represents the entire behavior of the scheduler. Then we run the scheduler together with selected task/server sets that will generate all of these (combinations of) scheduling events, during the verification. Alternatively (just to be safe), since the process from modelling/verification down to synthesis is short, once the model is finished (the verification of models in this size takes just a few minutes on a standard PC), a system can be verified with scheduler and load (task/server) together before deployment.

| **Name** | $T$ | $Budget$ | $D$ | $Prio$ | **Tasks** |
|---|---|---|---|---|---|
| Server1 | 19 | 2 | 19 | Low | {server1} |
| Server3 | 5 | 3 | 5 | High | {s3task1,s3task2} |

TABLE I
SERVER SET (USED IN SYSTEM 1 AND 2)

We ran three different task/server systems (system 1, 2 and 3) during the verification of our scheduler properties.

| **Name** | $T$ | $Budget$ | $D$ | $Prio$ | **Tasks** |
|---|---|---|---|---|---|
| Server1 | 19 | 2 | 19 | Low | {server1} |
| Server3 | 10 | 6 | 10 | High | {s3task1,s3task2} |

TABLE II
SERVER SET (USED IN SYSTEM 3)

The three systems are presented in Table III, IV and V. The corresponding execution traces can be found in Figure 9, 10 and 11. The server parameters used for systems 1 and 2 (Figure 9 and 10) are listed Table I, and the server parameters for system 3 (Figure 11) is shown in Table II. Figure 8 shows a simulation trace (in Times) of system 1, i.e., Figure 9.

| **Name** | $T$ | $C$ | $D$ | $Prio$ |
|---|---|---|---|---|
| server1 | - | - | - | - |
| s3task1 | 10 | 3 | 10 | Low |
| s3task2 | 11 | 1 | 11 | High |

TABLE III
TASK SET OF SYSTEM 1

| **Name** | $T$ | $C$ | $D$ | $Prio$ |
|---|---|---|---|---|
| server1 | - | - | - | - |
| s3task1 | 16 | 4 | 16 | Low |
| s3task2 | 11 | 2 | 11 | High |

TABLE IV
TASK SET OF SYSTEM 2

| **Name** | $T$ | $C$ | $D$ | $Prio$ |
|---|---|---|---|---|
| server1 | - | - | - | - |
| s3task1 | 10 | 3 | 10 | High |
| s3task2 | 11 | 1 | 11 | Low |

TABLE V
TASK SET OF SYSTEM 3

Table VI list all possible scheduling events at the global level. A release or suspension of a task/server can lead to a context switch (c.s.). If not (in case of suspension), then there will be a switch to an idle task/server, which is not part of our model, hence we define a context switch only when the model switches between tasks/servers that are defined in the model. A simultaneous suspension/release will always lead to a context switch. We do not differentiate if the task/server that is released is to be switched in, or, if there is another higher priority task/server ready to be switched in. We differentiate in that local scheduling events can occur when its server is active, the time when its server activates and the time when its server deactivates. Local scheduling events happen only during the time when its server is active (according to the model). Related to the undefined events in Table VII, a task suspension cannot happen during a server release since it cannot finish its execution at the same time as its server activates. The local scheduler does not differentiate the cause of its servers activation/deactivation, e.g., there is no differentiation if the server activation is due to a release, or suspension of a higher priority server. Hence, we do not need to consider all possible

| **Server event** | Example |
|---|---|
| Release (c.s.) | Fig. 9, time=20 |
| Release (no c.s.) | Fig. 9, time=57 |
| Suspend (c.s.) | Fig. 9, time=03 |
| Suspend (no c.s.) | Fig. 9, time=08 |
| Suspend/Release (c.s.) | Fig. 9, time=38 |

TABLE VI
SERVER SCHEDULING EVENTS

cases/combinations of local and global scheduling events. All scheduling events in Table VI and VII are referred to the execution traces presented in systems 1, 2 and 3. These scheduling events will occur during the verification of the global (section IV-B) and local scheduler (section IV-C).

| **Task event** | **Server event** | | |
|---|---|---|---|
| | Active | Activate | Deactivate |
| Release (c.s.) | Fig. 9, t=11 | Fig. 10, t=55 | Fig. 10, t=33 |
| Release (no c.s.) | Fig. 11, t=22 | Fig. 10, t=00 | Fig. 11, t=66 |
| Suspend (c.s.) | Fig. 11, t=13 | - | Fig. 9, t=23 |
| Suspend (no c.s.) | Fig. 9, t=06 | - | Fig. 10, t=08 |
| Suspend/Release (c.s.) | Fig. 11, t=33 | - | Fig. 9, t=33 |

TABLE VII
TASK SCHEDULING EVENTS

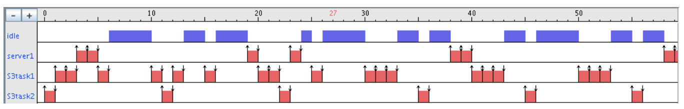


Fig. 8. TIMES simulator (simulating system 1)

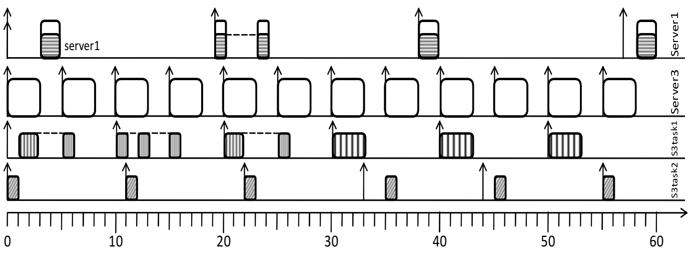


Fig. 9. System 1

## B. Global level verification

In the verification of the global scheduler, we use the server parameters shown in Table I, which will generate all server scheduling events (shown in Table VI). The following properties are defined (and later verified):

$Property1$ : A server $S_i$ (with index $i$) should never get more than $C_i$ budget at any discrete interval (non sliding) of length $P_i$, where the first interval starts at time 0.

$Property2$ : A server $S_i$ (with index $i$) should never get less than $C_i$ budget at any discrete interval (non sliding) of length $P_i$, where the first interval starts at time 0, if there is unused time within this interval.

$Property3$ : A server $S_i$ (with index $i$) should always be released (inserted in the server ready-queue) according to its specified period $P_i$.

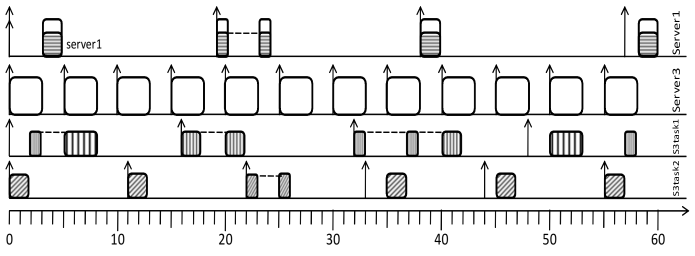


Fig. 10. System 2

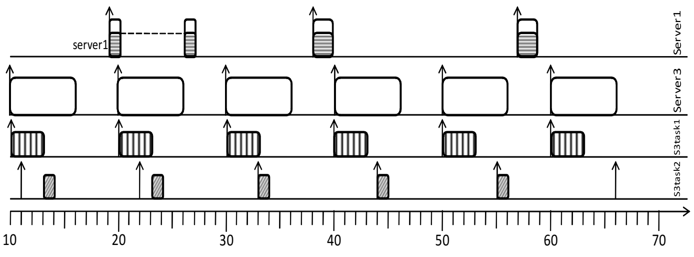


Fig. 11. System 3

$Property4$ : A server should always be removed from the server ready-queue upon server budget depletion.

$Property5$ : The highest priority server in the server ready-queue should always be the current running server in the system.

We have modelled a task automata called $Observer1$ (Figure 13 and 14) that will check that each of the 5 properties are fulfilled.

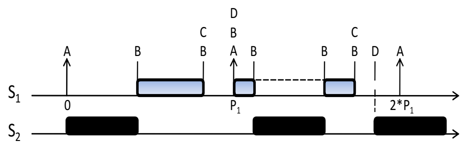
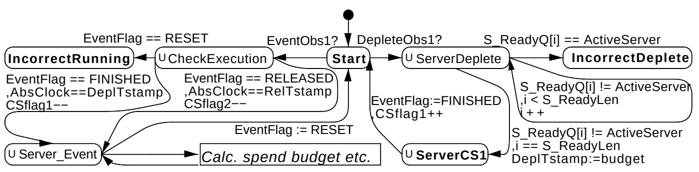


Fig. 12. $Observer1$ events

Figure 12 shows at which server scheduling events the observer executes, the following list explains each event:

- Event **A** represents a release event.
- Event **B** represents the start/stop of a budget (not necessarily the beginning and end of a budget).
- Event **C** represents the end of a budget.
- Event **D** represents the the beginning of a budget in case it was idle previously.


EventFlag == RESET
EventObs1?
DepleteObs1?
S_ReadyQ[i] == ActiveServer
IncorrectRunning
U CheckExecution
Start
U ServerDeplete
IncorrectDeplete
EventFlag == FINISHED ,AbsClock==DeplTstamp CSflag1−−
EventFlag == RELEASED ,AbsClock==RelTstamp CSflag2−−
EventFlag:=FINISHED ,CSflag1++
S_ReadyQ[i] != ActiveServer ,i < S_ReadyLen ++
EventFlag := RESET
U Server_Event
Calc. spend budget etc.
U ServerCS1
S_ReadyQ[i] != ActiveServer ,i == S_ReadyLen DeplTstamp:=budget
Fig. 13. $Observer1$: Server context-switch and depletion

$Property1$ and $Property2$ are checked by the observer by measuring the server budget, event **B** (Figure 12) illustrates

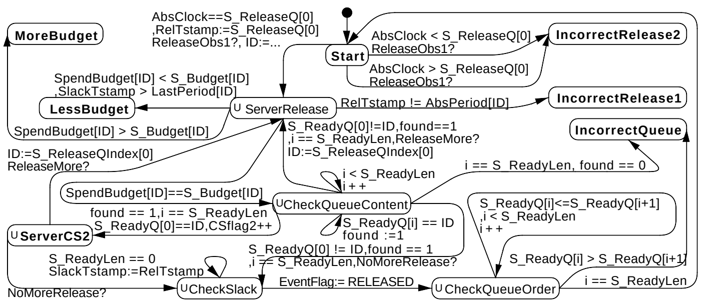


Fig. 14. $Observer1$: Server release

these events. $Property2$ is not valid if there is no unused budget within the period, since that indicates a schedulability problem. At event **D**, a server is activated, and the observer timestamps this point if no previous server was running. This timestamp value is checked at event **A** together with the measured budget. If the timestamp is within the period, then there was unused time. At each event **A**, $Property1$ and $Property2$ are checked. In Figure 14, either a transition to state **LessBudget** or **MoreBudget** is made if the budget has been underused or exceeded. Event **D** corresponds to **CheckSlack** (Figure 14). The logical expressions (1) and (2) in Figure 15 checks that there is no path leading to the error states, i.e., for all paths ($\forall$), on every state along the path ($\Box$), a state is never ($\neg$) visited. The transition to these error states contradicts the requirements of $Property1$ and $Property2$. For more details on the modelling of the error states, we direct the reader to our technical report [19].

$$\begin{array}{ll}
\forall\,\Box\,\neg LessBudget & (1)\\
\forall\,\Box\,\neg MoreBudget & (2)\\
\forall\,\Box\,\neg IncorrectRelease1 & (3)\\
\forall\,\Box\,\neg IncorrectRelease2 & (4)\\
\forall\,\Box\,\neg IncorrectDeplete & (5)\\
\forall\,\Box\,\neg IncorrectQueue & (6)\\
\forall\,\Box\,\neg IncorrectRunning & (7)\\
\forall\,\Box\,(ServerCS2 \implies & \\
(\forall\,\Diamond\,Server\_Event \wedge CSflag2 = 0)) & (8)\\
\forall\,\Box\,(ServerCS1 \implies & \\
(\forall\,\Diamond\,Server\_Event \wedge CSflag1 = 0)) & (9)
\end{array}$$

Fig. 15. TCTL expressions

$Property3$ is checked at event **A** (Figure 12). State **IncorrectRelease2** (Figure 14) is active if the global scheduler tries to release a server at an incorrect time. A transition to state **IncorrectRelease1** (Figure 14) is done if there should be an incorrect value in the server release queue, which does not match the calculated release time of the observer. We have used the logical expressions (3) and (4) in Figure 15 to check $Property3$ in Times.

$Property4$ is checked at event **C**, Figure 12. Whenever there is a server deplete event, the observer checks that the server is no longer in the server ready-queue (**IncorrectDeplete**, Figure 13). The logical expression (5) (Figure 15) verifies this property.

$Property5$ is checked at event **A** by checking the server ready-queue content and order, the logical expression used is (6) (Figure 15). We check that a server is in the ready queue after its release (**CheckQueueContent**) and that the queue is ordered correctly (**CheckQueueOrder**), both states are found in Figure 14. **Server_Event** is entered whenever there is a server context switch (Figure 13). It is not possible to enter this automata part if no budget depletion or server release has occurred (**CheckExecution**, Figure 13), this refers to expression (7) (Figure 15). Yet two more expressions are important to check, (8) and (9) (Figure 15), in order to verify $Property5$. Whenever there is a server release that affects the server ready-queue in such a way that it ends up as the head node (**ServerCS2**, Figure 14), then it implies that a server context switch should occur (**Server_Event**, Figure 13). This is checked in expression (8) (Figure 15), for all paths and states ($\forall\,\Box$), whenever state **ServerCS2** is reached, it implies ($\implies$) that at some state in all the upcoming paths ($\forall\,\Diamond$), state **Server_Event** is reached and ($\wedge$), at the same time the condition $CSflag2 = 0$ holds. The condition is that it should happen directly, i.e., no time should pass. This is a condition in the model where the transitions between **CheckExecution** and **Server_Event** checks the elapsed time (Figure 13). Also, there should not be any nesting, i.e., two server releases (where both imply server context switch) followed by one context switch (hence the check $CSflag2 = 0$ in the expression). The same check is made for budget depletion, expression (9) (Figure 15).

## C. Local level verification

During the verification of the local level we use all three task systems presented in section IV-A, and we use another observer called $Observer2$ (due to space restrictions we direct the reader to the technical report [19] for this figure) to verify the following 4 properties.

$Property6$ : A task $t_i$ (with index $i$) should always be released (inserted in the task ready-queue) according to its specified period $P_i$, OR if later, directly when its server is activated.

$Property7$ : A task should always be removed from the task ready-queue upon finishing its execution.

$Property8$ : The highest priority task in the task ready-queue (in each server) should always be the current running task in the server, when it is active.

$Property9$ : All tasks should run within their respective server.

The only properties that are different in the local level compared to the global level are $Property6$ and $Property9$, we will explain these two briefly.

$Property6$ is checked with the same expressions as in the global level, but the local level observer will also allow task releases that coincide with its server releases.

Regarding $Property9$, $Observer2$ assumes that all context switches that happen during its observation are within the server under observation. Hence, it is only required to check that no task context switch (where the next running task belongs to the observed server) will occur during server deactivation. The property is checked by timestamping all task context switches. A transition is made to an error state if a task context switch occur at the same time as a server deactivation.

## V. Code synthesis

We have synthesised the model into two different kernel-level implementations; the original model which has fragmented task executions and that is fully verified (Section IV), and the more simple model (without fragmentation) where only the global level is fully verified, and the local level is partially verified (only $Property6$ is fulfilled). In the simple model we don't use any internal task ready-queue (tasks are just released according to the release-queue), hence, the local level cannot be fully verified. We synthesised these two models for the sake of comparing the CPU overhead, further, we also included our previously manually coded hierarchical scheduler HSF [5] (as a reference point) in the comparison. The fragmented model is of course not practical, in terms of synthesis (real applications cannot have this kind of fragmentation), but still we show that it is possible to synthesise a fully verified hierarchical scheduler. Removing the fragmentation (and keeping the full verification) is just a matter of adding dynamic priority support (or the ability to suspend tasks) in the Times tool.

We measured the CPU overhead of all 3 schedulers as well as the memory consumption. The platform used for the experiments is VxWorks 6.6, running on an Intel Pentium4 (1,66 GHz, uni-core) desktop machine. The CPU overhead was measured with the sysTimestamp facility and the dynamic memory consumption was analysed with the Wind River Workbench Memory Analyzer. The tasks used in the experiments were executing empty for-loops and the execution times were estimated using the VxWorks Timex facility. During the experiments, the tick resolution was set to 1000 Hz. We let 1 time unit in the system represent 1 scheduler tick.

| **Scheduler** | CPU (%) |
|---|---|
| Times (fragmented) | 1.78 |
| Times (non-fragmented) | 1.36 |
| HSF | 0.08 |

TABLE VIII
CPU OVERHEAD

| **Scheduler** | Dynamic memory | | Static memory |
|---|---|---|---|
| | Max | Average | |
| Times (fragmented) | 1646 | 1646 | 10874 |
| Times (non-fragmented) | 1646 | 1646 | 10874 |
| HSF | 11456 | 1692 | 24 |

TABLE IX
MEMORY OVERHEAD (BYTES)

Table VIII shows the measured CPU overhead of the schedulers. The measurements were done in the first 2090 scheduler ticks, i.e., the least common multiple (LCM) of all task and server periods of system 1. The CPU overhead (%) represents the LCM of all task and server periods divided by the measured execution time of each scheduler. As can be observed, the non-fragmented version has less overhead than the fragmented, which is due to less automaton transitions and task releases. Both generated schedulers has substantial more overhead than the manually coded scheduler, i.e., 17 respectively 22 times more CPU overhead. We experimented on the generated code with an optimisation which reduced the amount of scheduler invocations by 50% (1045 instead of 2090 scheduler invocations), however, the total CPU overhead was reduced by only 5%. We have identified more ways to optimise the code, but we defer this to future work.

Table IX shows the amount of dynamic/static memory used by the schedulers. During the actual scheduling (after initialisation), the memory allocation of HSF drops down to 1692 bytes. The total memory used (during the scheduling) by HSF is 1716 bytes, and for the generated schedulers it counts up to 12520 bytes in total. The conclusion is that there is a similar worst-case memory usage (11480 vs. 12520 bytes), but less CPU overhead by HSF (0.08% vs. 1.36%).

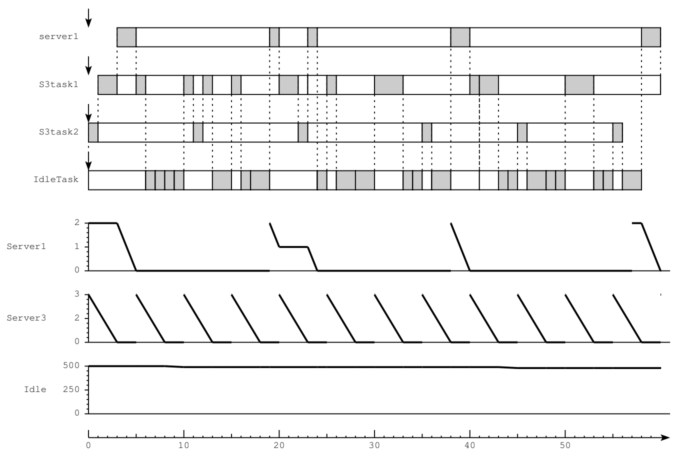


Fig. 16. Execution trace of HSF, running system1

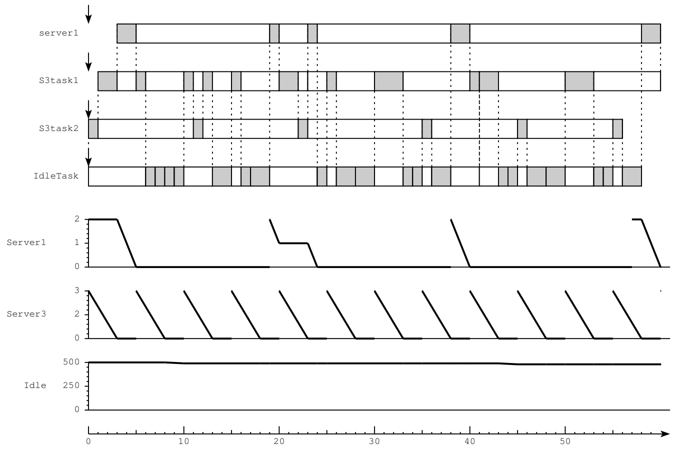


Fig. 17. Execution trace of TIMES scheduler (non-fragmented), running system1

Figure 16, 17 and 18 shows the actual runtime execution recording of the tasks and servers. As can be seen, our generated schedulers (Figure 17 and 18) gives the same trace as the manually coded scheduler (Figure 16). What can also be

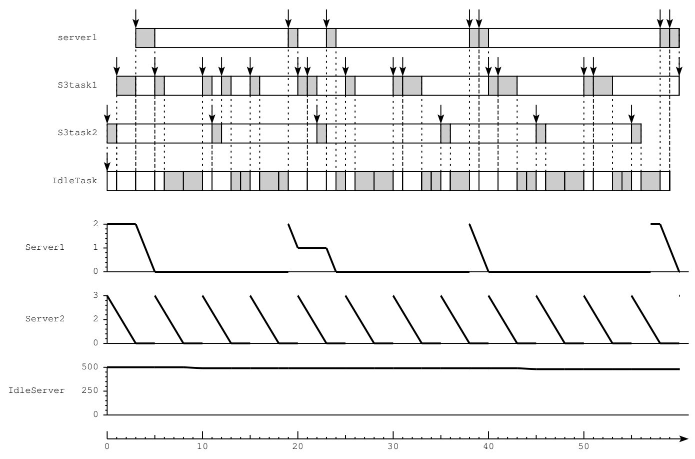


Fig. 18. Execution trace of TIMES scheduler (fragmented), running system1

noted is how the fragmented model gives a slightly different execution trace than the non-fragmented since there are more task releases due to the fact that the task execution is divided into several one-time-unit sections of execution.

Our execution recorder uses the VxWorks taskHookLib and we use the visualisation tool Grasp [22] to display the recordings.

## VI. Related work

*a) Hierarchical scheduling theory:* There is a growing attention in that little prior work has been done on verification of hierarchical scheduling implementations [23], as compared to the great amount of work on schedulability analysis [8], [9], [10], [24], [25], [26], [27], [28] (where there is an assumption that the scheduling policy is correctly implemented), which has originated from open systems [2] in the late 1990's.

*b) Hierarchical scheduling implementation:* Among the implementation work, Kim *et al.* [29] propose the SPIRIT uK-ernel that is based on a two-level FPPS hierarchical scheduling framework, simplifying integration of real-time applications. A mix of theory and practice is presented in [3] where the authors reason about general scheduling trees with arbitrary scheduling policies and scheduling depths. They also present an implementation in Windows 2000. More recently, [5] and [30] implemented a two-level FPPS HSF in the commercial real-time operating systems VxWorks and $\mu$C/OS-II.

*c) Scheduler modelling:* There are two main categories of scheduler modelling, either the scheduler already exists as an implementation and it is modelled (and verified) after code analysis or other techniques [31], [32], [33], [34], [35], [36], or (as in our paper) the scheduler is modelled and later verified (and perhaps also synthesised) [37], [38], [39], [40], [41].

In the area of modelling hierarchical scheduling, the authors in [42] show how modelling and schedulability analysis of two-level hierarchical scheduling, with timed automata, can be accomplished in the simulation tool Cheddar. Ha *et al.* [43] describes the verification, using theorem-proving, to verify the IMA scheduler DEOS, used for safety critical domains such as aerospace and space. The scheduler assigns a period and budget to each thread, the scheduling policy used is RMA. The work of Muller *et al.* [44], [45] is most similar to our work. They use a domain specific language (DSL) to model schedulers (including hierarchical schedulers). The difference is that they verify that the scheduler is correct with respect to the kernel interface, and not the actual scheduling policy. Their framework support synthesis for early Linux kernel versions. Zerzelidis *et al.* [46] model a system with multiple schedulers, including resource sharing with SRP. The modelling tool UPPAAL is used, and the model is compatible with RTSJ. Each partition (local scheduler) has a priority level, but no release time or budget. The verification shows the absence of livelock/deadlock and the correctness of SRP.

Few papers touch upon the area of code-synthesis in the context of scheduler modelling. Hsiung *et al.* [47] presents a framework (VERTAF) for developing real-time embedded software. The application, as well as the scheduler is specified as UML diagrams. The framework does a transformation to extended timed automata (ETA) and model checking is used to verify properties such as livelock and deadlock. The framework supports code-synthesis for the OS's MontaVista Linux, $\mu$C/OS, Embedded Linux, and eCOS. Li *et al.* [48] introduce a meta-scheduler framework, compliant with POSIX-supported OS's. Basically, the framework is a middleware layer which uses OS primitives, and it exports an interface to schedulers, which in turn are implemented by the users. The correctness of the framework is verified using UPPAAL. They implement several flat-schedulers in various platforms (VxWorks for example), and they measure the overhead of the schedulers.

To sum up, modelling of hierarchical scheduling has been done, but not specifically for two-tier FPPS with PS. To the best of our knowledge, there is no prior work on verification of hierarchical scheduling policies, nor code-synthesis (from model) for this type of scheduling.

## VII. Conclusion

In this paper we deal with modelling, verification and synthesis of hierarchically scheduled real-time systems. We have looked at two-level hierarchical scheduling, with fixed priority preemptive scheduling of periodic tasks/servers. The scheduler has been modelled using the task-automata language and the model was implemented in the Times tool. However, the Times tool does not support dynamic change of priorities, nor task suspension, which are two fundamental properties required when implementing hierarchical scheduling. In the paper we show how to get around this problem through an innovative approach for how the system is modelled.

In addition we modelled *observers* which monitored the behavior of the schedulers. We implemented rules for the observers, based on the criteria that we have specified as properties. These properties are appropriate behaviors that comply with hierarchical fixed priority preemptive scheduling of periodic tasks and servers. The observers are then modelled to enter error states if they detect a contradiction to any of our properties. We check that the observers do not enter these error states through the use of model checking. We use task/server systems that stress the schedulers to generate all combinations of scheduling events, so that we can verify the entire behavior of the hierarchical scheduler.

The code synthesis results showed a considerable difference in CPU between the generated schedulers and an equivalent manually coded scheduler. However, the worst-case memory consumption showed to be similar to each other.

To sum up, this paper presents a proof of concept, showing that we can model, verify, and generate source-code that executes a hierarchical scheduler on an industrial platform.

As future work, we plan to optimise the synthesis of the model by implementing a new (optimised) code generator. This will make the synthesis fully automated, which will open up the possibility to generate systems in a larger scale.

## REFERENCES

[1] P. Goyal, X. Guo, and H. M. Vin, "A Hierarchical CPU Scheduler for Multimedia Operating Systems," in *OSDI'96*.

[2] Z. Deng and J. W.-S. Liu, "Scheduling Real-time Applications in an Open Environment," in *RTSS'97*.

[3] J. Regehr and J. A. Stankovic, "HLS: A Framework for Composing Soft Real-Time Schedulers," in *RTSS'01*.

[4] M. Åsberg, M. Behnam, F. Nemati, and T. Nolte, "Towards Hierarchical Scheduling in AUTOSAR," in *ETFA'09*.

[5] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. J. Bril, "Towards Hierarchical Scheduling on top of VxWorks," in *OSPERT'08*.

[6] M. Åsberg, T. Nolte, and S. Kato, "A Loadable Task Execution Recorder for Hierarchical Scheduling in Linux," Mälardalen University, Technical Report 2377, 2010. [Online]. Available: http://www.mrtc.mdh.se/publications/2377.pdf, [Accessed:2011-01-05]

[7] L. Sha, J. P. Lehoczky, and R. Rajkumar, "Solutions for some Practical Problems in Prioritized Preemptive Scheduling," in *RTSS'86*.

[8] R. I. Davis and A. Burns, "Hierarchical Fixed Priority Pre-emptive Scheduling," in *RTSS'05*.

[9] T.-W. Kuo and C.-H. Li, "A Fixed-Priority-Driven Open Environment for Real-Time Applications," in *RTSS'99*.

[10] I. Shin and I. Lee, "Periodic Resource Model for Compositional Real-Time Guarantees," in *RTSS'03*.

[11] M. Behnam, I. Shin, T. Nolte, and M. Nolin, "SIRAP: A Synchronization Protocol for Hierarchical Resource Sharing in Real-Time Open Systems," in *EMSOFT'07*.

[12] R. I. Davis and A. Burns, "Resource Sharing in Hierarchical Fixed Priority Pre-emptive Systems," in *RTSS'06*.

[13] E. Fersman, P. Krcal, P. Pettersson, and W. Yi, "Task Automata: Schedulability, Decidability and Undecidability," *International Journal of Information and Computation*, vol. 205, no. 8, pp. 1149–1172, 2007.

[14] R. Alur and D. L. Dill, "A Theory of Timed Automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[15] T. Amnell, E. Fersman, P. Pettersson, W. Yi, and H. Sun, "Code Synthesis for Timed Automata," *Nordic Journal of Computing*, vol. 9, no. 4, pp. 269–300, 2002.

[16] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, "TIMES: A Tool for Modelling and Implementation of Embedded Systems," in *TACAS'02*.

[17] C. Liu and J. Layland, "Scheduling Algorithms for Multi-programming in a Hard Real-Time Environment," *ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[18] L. Andriantsiferana, J.-P. Courtiat, R. C. d. Oliveira, and L. Picci, "An Experiment in using RT-LOTOS for the Formal Specification and Verification of a Distributed Scheduling Algorithm in a Nuclear Power Plant Monitoring System," in *FORTE'97*.

[19] M. Åsberg, "Model of Two-Tier Hierarchical Fixed-Priority Preemptive Scheduling," Mälardalen University, Technical Report 2379, 2011. [Online]. Available: http://www.mrtc.mdh.se/publications/2379.pdf, [Accessed:2011-01-05]

[20] L. Lensink, S. Smetsers, and M. Van Eekelen, "Machine Checked Formal Proof of a Scheduling Protocol for Smartcard Personalization," in *FMICS'07*.

[21] G. Grov, G. Michaelson, and A. Ireland, "Formal Verification of Concurrent Scheduling Strategies using TLA," in *ICPADS'07*.

[22] M. Holenderski, M. M. H. P. van den Heuvel, R. J. Bril, and J. J. Lukkien, "Grasp: Tracing, Visualizing and Measuring the Behavior of Real-Time Systems," in *WATERS'10*.

[23] R. Glaubius, T. Tidwell, W. D. Smart, and C. Gill, "Scheduling Design and Verification for Open Soft Real-Time Systems," in *RTSS'08*.

[24] X. Feng and A. Mok, "A Model of Hierarchical Real-Time Virtual Resources," in *RTSS'02*.

[25] G. Lipari and S. K. Baruah, "Efficient Scheduling of Real-Time Multi-Task Applications in Dynamic Systems," in *RTAS'00*.

[26] G. Lipari and E. Bini, "Resource Partitioning Among Real-Time Applications," in *ECRTS'03*.

[27] J. Regehr, A. Reid, K. Webb, M. Parker, and J. Lepreau, "Evolving Real-time Systems Using Hierarchical Scheduling and Concurrency Analysis," in *RTSS'03*.

[28] S. Matic and T. A. Henzinger, "Trading End-to-End Latency for Composability," in *RTSS'05*.

[29] D. Kim, Y. Lee, and M. Younis, "SPIRIT-uKernel for Strongly Partitioned Real-Time Systems," in *RTCSA'00*.

[30] M. M. H. P. van den Heuvel, M. Holenderski, W. Cools, R. J. Bril, and J. J. Lukkien, "Virtual Timers in Hierarchical Real-time Systems," *Proc. WiP session of the RTSS'09*.

[31] J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger, "Verification of Time Partitioning in the DEOS Scheduler Kernel," in *ICSE'00*.

[32] D. Cofer, E. Engstrom, and N. Weininger, "Using Model Checking for Verification of Partitioning Properties in Integrated Modular Avionics," in *DASC'00*.

[33] T. K. Iversen, K. J. Kristoffersen, K. G. Larsen, M. Laursen, R. G. Madsen, S. K. Mortensen, P. Pettersson, and C. B. Thomasen, "Model-Checking Real-Time Control Programs - Verifying LEGO MINDSTORMS Systems Using UPPAAL," in *ECRTS'00*.

[34] M. Daum, J. Drrenbcher, and B. Wolff, "Proving Fairness and Implementation Correctness of a Microkernel Scheduler," *Journal of Automated Reasoning*, vol. 42, pp. 349–388, 2009.

[35] N. Narasimhan, E. Teica, R. Radhakrishnan, S. Govindarajan, and R. Vemuri, "Theorem Proving Guided Development of Formal Assertions in a Resource-Constrained Scheduler for High-Level Synthesis," *Formal Methods in System Design*, vol. 19, pp. 237–273, 2001.

[36] M. Kleine, B. Bartels, T. Gothel, and S. Glesner, "Verifying the Implementation of an Operating System Scheduler," in *TASE'09*.

[37] L. Didier and O. H. Roux, "Formal Verification of Real-Time Systems with Preemptive Scheduling," *The International Journal of Time-Critical Computing Systems*, vol. 41, pp. 118–151, 2009.

[38] P.-A. Hsiung and S.-W. Lin, "Model Checking Timed Systems with Priorities," in *RTCSA'05*.

[39] C. Shu and W.-G. Qing, "Modeling and Formal Analysis of Real-Time System via CCS," *ISCSCT'08*.

[40] L. Durante, R. Sisto, and A. Valenzano, "Formal Specification and Verification of the Real-Time Scheduler In FIP," in *WFCS'95*.

[41] O. Nasr, J.-P. Bodeveix, M. Filali, and M. R. Irit, "Verification of a Scheduler in B Through a Timed Automata Specification," in *SAC'06*.

[42] F. Singhoff and A. Plantec, "AADL Modeling and Analysis of Hierarchical Schedulers," in *SIGAda'07*.

[43] V. Ha, M. Rangarajan, D. Cofer, H. Rues, and B. Dutertre, "Feature-Based Decomposition of Inductive Proofs Applied to Real-Time Avionics Software: An Experience Report," in *ICSE'04*.

[44] L. P. Barreto and G. Muller, "Bossa: A Language-Based Approach to the Design of Real-Time Schedulers," in *RTS'02*.

[45] J. L. Lawall, G. Muller, and H. Duchesne, "Invited Application Paper: Language Design For Implementing Process Scheduling Hierarchies," in *PEPM'04*.

[46] A. Zerzelidis and A. Wellings, "Model-based Verification of a Framework for Flexible Scheduling in the Real-Time Specification for Java," in *JTRES'06*.

[47] P.-A. Hsiung, S.-W. Lin, and C.-S. Lin, "Real-Time Embedded Software Design for Mobile and Ubiquitous Systems," *Journal of Signal Processing Systems*, vol. 59, pp. 13–32, 2010.

[48] P. Li, B. Ravindran, S. Suhaib, and S. Feizabadi, "A Formally Verified Application-Level Framework for Real-Time Scheduling on POSIX Real-Time Operating Systems," *IEEE Transactions on Software Engineering*, vol. 30, pp. 613–629, 2004.