Evaluating the Run-Time Performance of Synthesised Resource-Reservation Schedulers Using TAtoC, UPPAAL and Frama-C

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*—This paper evaluates the performance of four verified real-time operating-system schedulers. We used the UPPAAL and Frama-C verification tools, and our newly developed code generator TAtoC (which is one of the main contributions in this paper), to develop the verified schedulers. These schedulers isolate operating-system threads into time partitions. This technique is referred to as resource reservation and it is a commonly used scheduling technique in the aviation industry since it effectively isolates timing and memory faults within each partition (and simplifies certification). Hence, this will prevent errors from propagating to different parts of the system.**

The performance of the synthesised schedulers was evaluated in the seL4 micro-kernel. This kernel is unique since it is the only existing kernel that is 100% verified. The kernel itself uses partitioning as a means to separate critical applications from non-critical ones. The aim of this paper is to investigate and develop a performance-efficient, and verified, scheduler that is suitable for the seL4 kernel, since it currently lacks a resource-reservation based scheduler.

***Index Terms*—theorem proving, model checking, real-time systems, operating system scheduling, resource reservation**

I. Introduction

Introduction A real-time system is a hardware/software system in which every operation must be bounded in terms of time. For example, this could mean that a specific operating system (OS) thread must complete its execution within at most 32 µs counting from the point in time when it started to execute. The requirement still holds if the thread completes earlier than 32 µs. Hence, 32 µs is the bounded time that is not allowed to be exceeded.

Most real-time systems have some sort of periodic behavior. For example, an anti-lock braking system (ABS) in a car will periodically read a sensor to check if a wheel is locked. The maximum time delay between two consecutive sensor readings must be bounded, since this time will represent the maximum time delay from the time when a wheel locks until the ABS system has knowledge of it (and can react).

A function, e.g., an ABS system, in a car usually consists of more than one OS thread. Hence, it is not unusual that a group of threads collaborate in order to form a specific function. Integrating such functions onto a common hardware platform is popular today in both the avionics [1] and the automotive [2] industry. The reason for this is because of the constant increase in functionality, e.g., automatic parking assistance, advanced cruise control etc., (which is realized in software) and the limit on the number of embedded computers due to weight and space limitations. Hence, due to a larger number of software functions compared to the number of embedded computers, this will force these functions to share CPU, memory, devices etc. Going back to the ABS example which has a number of time-critical threads. These threads will be affected when they share the CPU with other time-critical threads. The risk is high that these threads will not meet their (time) deadlines since they might be delayed by other threads that are executing (with higher priorities). Even if we can manage to analyse such a system and claim that all threads will meet their deadlines (assuming that the threads will not execute longer than intended), we still might want to isolate critical threads from less critical threads just in case if some (non-critical) thread would violate its assumptions and thereby interfere with critical threads. From the certification point of view, such an isolation might even be required. Isolation will also make the analysis simpler since we can decompose the system into smaller parts which can be analysed one by one in isolation. Standards in the avionics industry, such as ARINC653 [1], define a memory and CPU partitioning scheme as part of the OS. This will isolate applications from each other both in time and space. We will go more into depth in resource-reservation (partitioning) techniques in Section II-A.

The CPU partitioning in ARINC653 is implemented as part of the OS scheduler. Such a (resource-reservation) scheduler would be useful in the seL4 [3] micro-kernel. The reason for this is because the kernel implements memory-partitioning, and even access-control to kernel system-calls and inter-process communication (IPC) which gives rise to a strong isolation between software applications. The entire seL4 kernel has been verified using the Isabelle/HOL theorem prover. Hence, there is no possibility to implement a resource-reservation scheduler inside the kernel (this would invalidate the verification property). Instead, such a scheduler would have to be implemented as an application residing in the user space. However, this imposes a performance degradation since scheduling operations issued from user-space take more time

to execute, compared to an in-kernel implementation, due to system-call delays (overhead). Performance is of course an important property for a micro-kernel. Especially if used in a real-time embedded system which has limited resources in terms of CPU, power consumption etc. and high demands on timing. Hence, efficient run-time performance (low overhead) and (verified) correctness are two important requirements on a resource-reservation scheduler (in seL4). The goal of this paper is to develop such a scheduler.

Making verified source-code performance efficient is a challenge. In this work, we focused both on theorem proving and model checking. Using model checking to verify timed automata (TA) systems works well when verifying schedulers (that are modelled with TA). However, a recent study [4] has shown that the run-time performance of resource-reservation schedulers, when synthesised from TA, is substantially worse than manually-coded (non-verified) resource-reservation schedulers. Using theorem proving to verify manually coded schedulers could potentially have better run-time performance, since there is no language-translation step involved. We used UPPAAL-TIMES [5], UPPAAL [6] and Frama-C [7] in this study to develop four different verified resource-reservation based schedulers. We evaluated the performance of these four schedulers in seL4, together with a manually coded scheduler and a previously verified scheduler [4].

Contribution The main contributions of this paper are:

1) The development of four verified resource-reservation based schedulers. This includes the development of a code generator called TAtoC. It can generate source-code from both task automata and timed automata models.
2) The evaluation of these schedulers, in terms of their run-time performance, in the seL4 micro-kernel. This evaluation includes actual (clock-cycle level) time measurements of the execution of the schedulers.

Outline The outline of this paper is as follows: Section II presents preliminary background and Section III presents related work. In Section IV we describe the verification and synthesis efforts to develop the four verified schedulers. We present the performance evaluation of the presented schedulers in Section V, and finally in Section VI, we conclude our work.

II. Preliminaries

A. Resource reservation (CPU)

Resource reservation is a general term for dividing a resource (CPU, memory etc.) into sub-parts and then allocate these to different applications (which usually consist of a set of threads, processes etc.). The applications might get different sizes of their share. This is typically common in real-time systems as opposed to general purpose systems like Linux for example. General purpose systems typically implement fair sharing which does not make sense in a real-time system where threads (or a group of threads) have different priority levels, i.e., this means that they are not equally important. Applications, in real-time systems, will get different share sizes depending on how important they are. However, most importantly, this share is a guaranteed allocation to the application. There are two positive aspects with resource reservation:

1) Each application can be developed in such a way that it is optimized for a certain resource allocation. For example, if an application development-team knows that they will get a 20% share of the CPU, then they can adapt the thread priorities, execution times etc. to this share. The application can then be carefully analysed in such a way that it guarantees that the application can execute without deadline misses (with the assumption that it will get a maximum of 20% of the CPU's processing time). This will also simplify reusability, but most importantly, it does not matter what is allocated to the remaining part (80%) of the CPU's processing time. This means that the integration phase will be simplified, i.e., the integrator will only need to analyse the CPU shares (%) when integrating applications. There is no need to inspect and analyse the internal application-threads.
2) Assume that the application from the previous example (with a 20% CPU share) is an extremely critical and important application (an ABS system for example). It would be a waste of hardware resources to let this ABS system execute alone on an embedded computer. It would be a waste of 80% of the CPU's processing time which is not acceptable since cars have very limited hardware resources. However, to let for example a Linux-based brake-diagnostics system (that requires approximately 60% of the CPU's processing time) run side-by-side with the ABS system might be hazardous. It could be a disaster if the diagnostics system would use more than 80% of the CPU's processing power at any time. A safe solution could be to put the diagnostics system in a virtual machine and allocate 60% of the CPU's processing time to it using resource reservation. Hence, the resource-reservation mechanism would prevent any unpredictable interference between these two systems. Putting a barrier like this between applications is a type of isolation that makes system integration safer. In general, isolation is the most efficient tool for avoiding hazards due to propagating software run-time errors. The reason for this is because it can guarantee error confinement and this will stop errors from propagating from one application to another. Note that we include execution-time violations, i.e., deadline misses, in our definition of a software error.

Figure 1 illustrates how the CPU resource-reservation works. A set of threads (the representation of a function/application) can be confined in a partition that we refer to as a *server*. Each server can also host a *local scheduler* that schedules threads according to, for example, internal thread priorities. No local scheduler is required when only one thread resides in a server. The CPU share that is allocated to a server

is defined in the server's *interface*. The *global scheduler* is responsible for enforcing that the CPU shares are indeed given to the servers at run-time according to the server interfaces.

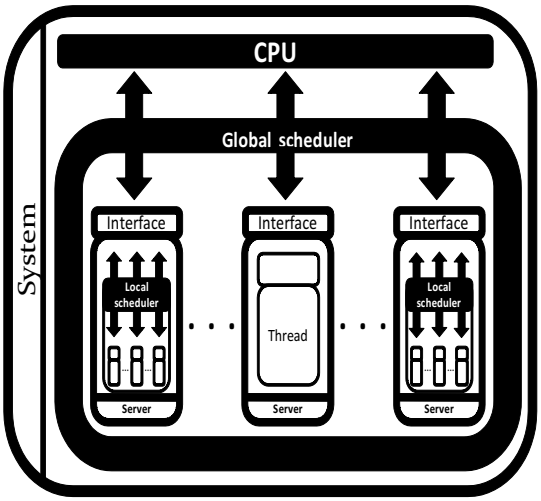

Fig. 1. Resource reservation for the CPU.

Figure 2 shows an example system with an ABS [8] and a diagnostics system integrated on the same hardware platform. The upper part shows the threads and the lower part represents the servers. The x-axis represents (absolute) time and the systems start to execute at time 0. The two systems are placed in different servers (ABS and Brake Diagnostics) and allocated 20% and 60% of the CPU's processing time respectively. An executing server is in practice a time window (called *budget*) that is used by the server's thread(s). The server's threads are allowed to execute within this time window (but never outside of it). Time windows are represented by the diagonal lines in figure 2. The ABS system consists of three threads (ABS-sensor, ABS-control and ABS-actuator) that are responsible for reading sensors, control calculations and valve actuation (if necessary). The diagnostics system consists of one thread (LinuxVM) that executes a Linux OS as a virtual machine. The diagnostics system is responsible for collecting brake-related data (from the ABS, AntiSkid, TractionControl etc.) during driving and saving it in a database. The ABS server has a higher priority than the diagnostics server (Brake Diagnostics) since it is more critical than the diagnostics system. This can be observed in figure 2 when the ABS server interrupts (preempts) the lower priority server every 50 time units. The highest priority thread/server among the ones that want/can execute will always run. The ABS server gives the ABS threads 20% CPU processing time by letting them execute in the time windows which appear for 10 time units every 50 time units (10/50=0.2=20%). The LinuxVM thread is given 240 time units every 400 time units by its server (240/400=0.6=60%). Note that the two servers can not execute simultaneously since we assume a uni-core processor platform. The scheduling algorithm used in Figure 2 for scheduling servers is fixed-priority preemptive scheduling (FPPS). The local scheduler in the ABS server schedules the three threads using FPPS as well. FPPS is one of the most common scheduling algorithms in real-time systems [1], [2].

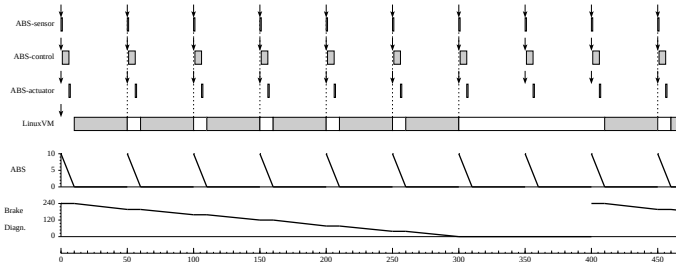

Fig. 2. Example schedule with two servers using resource reservation.

B. seL4

Software errors are difficult to avoid when the software system is complex and large. For example, a small system like the seL4 [3] (secure embedded L4) kernel consists of approximately 9000 lines of code (LOC). This is relatively small compared to the Linux kernel (version 2.6.35) which has 13.5 million LOC [9]. However, the seL4 kernel took 20 person years to verify and the verification process discovered 144 software bugs. A software system is without doubt not reliable nor safe if the underlying OS is not verified.

The concept of seL4 is straightforward. First of all, the most critical part of a software system, i.e., the OS, has control of everything. Hence, keep it small, simple and make sure that it is 100% verified. Secondly, protect applications from each other by using partitions (resource reservation). Software is more robust when it is composed into partitions compared to non-partitioned (flat) software. The reason for this is because software errors will only affect a limited portion of the software when it is partitioned [1].

As mentioned in Section I, seL4 implements partitioning to some extent, but not when it comes to thread scheduling. Threads may have unique priorities in seL4, i.e, FPPS is used, but threads are not scheduled periodically. Also, there is no support for CPU resource-reservations, i.e., the scheduler can not decide when threads should start or suspend. Threads with equal priority will be scheduled according to the round-robin scheduling algorithm. The round-robin algorithm is based on fairness, i.e, a common strategy in general purpose OSs such as Linux or Windows. However, this algorithm is not suitable for real-time systems. Figure 3 illustrates how the thread scheduling is performed in seL4. Thread A has the highest priority, while thread B and thread C have the same priority, and thread D has the lowest priority among the threads. Thread A will execute first since it has the highest priority. Once it stops to execute (due to I/O blocking for example), then threads B and C are allowed to run. Note that thread A can delay the other threads for an unbounded amount of time. This is what the resource-reservation technique is designed to avoid. Threads B and C are scheduled by the round-robin algorithm since they have the same priority level. This means that both threads will get a fair amount of CPU processing

time since they run every second time with time windows that are equally long (except for the last ones since the threads stop executing before the time windows end). We can observe that thread B and C are scheduled using enforcement since time windows are used (it resembles resource reservation). However, this enforcement is done in a fair manner (thread B and C get the same amount of CPU processing time) and this is not desirable in real-time systems. Thread D will be the last one to execute since it has the lowest priority.

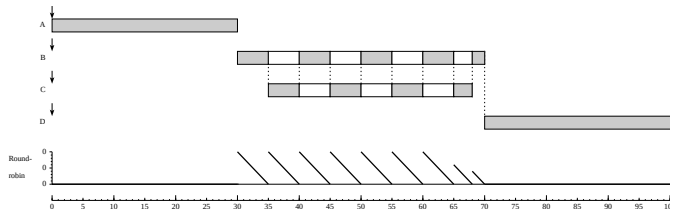

Fig. 3. Scheduling in seL4.

C. Timed and task automata

Timed automata [10] is a modelling language used for formal modelling and analysis of software and hardware. A timed automaton is a finite state-machine that is extended with real-valued clocks. The formalism of timed automata is widely used for analysing real-time systems. The UPPAAL tool (among others) supports verification of safety properties.

Timed automata have been extended with an explicit notion of real-time threads [11]. This extended model, which is referred to as *task automata*, associates asynchronous threads with the states of a timed automaton. The task-automata model assumes that threads are executed with static or dynamic priorities in a preemptive or non-preemptive manner. Task automata are supported by the UPPAAL-TIMES tool.

In this paper we use both the UPPAAL and the UPPAAL-TIMES tool.

D. Frama-C

Frama-C is a plug-in based framework that can perform verification of C programs. Deductive verification is one of the functionalities that Frama-C offers through the plug-in *Jessie* [12]. Functional properties of C programs can be expressed using the ACSL [13] formal specification language. Automated provers like *Z3*, *CVC3*, *Yices*, *Simplify*, *Alt-Ergo*, *Gappa* etc. can be used together with the *Jessie* plug-in to perform verification.

```
1: /*@ requires \valid(p);
2:      requires \valid(q);
3:      assigns *p;
4:      assigns *q;
5:      ensures *p == \old(*q);
6:      ensures *q == \old(*p); */
```

```
7: void swap(int *p, int *q) {
8:
9:     const int save = *p;
10:    *p = *q;
11:    *q = save;
12: }
```

Fig. 4. Frama-C verification example.

Figure 4 shows an example of C code that is annotated with ACSL. A **requires** clause specifies pre-conditions and an **ensures** clause specifies post-conditions. An **assigns** clause specifies the memory locations that will be modified.

III. Related work

There are two main categories when it comes to scheduler modelling and verification. In the first category, the scheduler already exists as a kernel implementation [14], [15], [16], [17], [18], [19], [20], [21]. The idea is then to model this scheduler (using code analysis or similar methods) and later verify it. In the second category, the scheduler is modelled from scratch and later verified (and perhaps also synthesised) [22], [23], [24], [25], [26], [27]. This paper presents schedulers from both categories (see Section IV).

Looking at real-time scheduler verification (from the first category), Gotsman et al. [18] used concurrent separation logic to verify a simplified version of a Linux scheduler from 2005 (2.6.11). The work from Daum et al. [15], Ferreira et al. [16] and Fidge et al. [17] used the Ergo theorem prover, Isabelle/HOL and HIP/SLEEK (respectively) to verify round-robin schedulers residing in the MIPS R3000 platform, the VAMOS micro-kernel and in the real-time operating system (RTOS) FreeRTOS (respectively). The difference to our work is that none of these schedulers are of type FPPS with resource reservations.

The following related work is focused on resource reservation in combination with verification. Åsberg et al. [4] developed a two-level resource-reservation scheduler that was verified and synthesised to the RTOS VxWorks. The main drawback with this work was the poor run-time performance of the synthesised scheduler. Another related paper, Carnevali et al. [28], was published the same year (2011). The main difference, compared to the work from Åsberg et al. [4], was that Carnevali et al. [28] used timed petri-nets instead of timed automata and the synthesised scheduler was emulated in Linux. The work from Ha et al. [29] included the verification of the resource-reservation scheduler in the DEOS kernel using theorem-proving. Singhoff et al. [27] performed schedulability analysis of two-level resource-reservation scheduling using timed automata and the simulation tool Cheddar. The Bossa framework from Luciano et al. [30] and Lawall et al. [31] used a domain specific language (DSL) to model resource-reservation based schedulers. The main difference from our work is that the authors verified that the scheduler was correct with respect to the Linux kernel interface, and not any specific scheduling policy. Zerzelidis et al. [32] used UPPAAL to model a system with several schedulers. Each resource reservation had a priority level but no release-time or budget. The verification proved that the model was free from livelock and deadlock.

Our work distinguishes itself from the listed work in that we have focused primarily on the run-time performance of the synthesised scheduler.

IV. Scheduler verification and synthesis

This section will describe how we verified and synthesised our four resource-reservation based schedulers (Section IV-B).

One of the schedulers was handwritten in C and later verified using Frama-C (Section IV-B1). Three of the schedulers (presented in Section IV-B2, IV-B3 and IV-B4) were modelled with timed/task automata and later synthesised using two different versions of our code generator (*TAtoC*) that we developed during this work (Section IV-A). The Frama-C verification source-files and the code generator *TAtoC* are available for free as open source projects[1]. The *TAtoC* code generator can hence be modified in order to fit the needs of the user, as opposed to the default (built-in) code generator in UPPAAL-TIMES which is closed source.

A. Code synthesis

We have developed a code generator called *TAtoC* (Timed Automata to C) that is compatible with both the UPPAAL and UPPAAL-TIMES tool, i.e., it can convert both task- and timed-automata to C code. *TAtoC* is an external tool (not integrated in UPPAAL) and it is written as a Bash script. Figure 5 illustrates the data flow between *TAtoC* and UPPAAL. The input to the *TAtoC* tool is a XML file representing a timed-automata system. This XML file contains timed-automaton structures that are modelled either in the UPPAAL or the UPPAAL-TIMES tool. *TAtoC* has the ability to parse timed-automaton XML-files and generate C code.

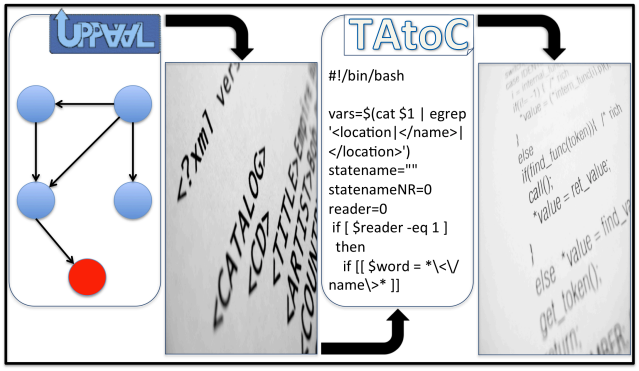

Fig. 5. The model-conversion flow from UPPAAL to *TAtoC*.

The *TAtoC* code generator is inspired by the existing code generator [33] in the UPPAAL-TIMES tool. *TAtoC* is based on the maximal-progress assumption [34]. This means that the timed-automata system should execute as many transitions as possible (during the current clock tick) until it stabilises, i.e., until no more transitions are possible to take. The clock tick will at this point be incremented by one, then follows another set of transitions until the system stabilises again (and so on). This is illustrated in Figure 6. Assume that the clock CL is initially set to one in the timed-automata system. The leftmost automata will make a transition from state Temp1State1 to state Temp1State2 (since the guard CL == 1 is true). This will in turn force one of the other two automata to make a transition since both automata have

[1]TAtoC http://www.idt.mdh.se/~tatoc/

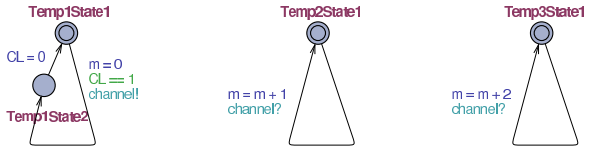

Fig. 6. Example timed-automata system with three templates.

an edge with a channel (channel?) that is connected to the leftmost automaton (channel!). The last possible transition is Temp1State2 to Temp1State1. No more transitions are possible to take after this transition (during the current clock tick) since CL is equal to zero. The system has stabilised now, so CL gets incremented by one.

TAtoC supports the majority of timed-automaton constructs. For example, it can translate task automata, inline C-code integrated in timed automata, templates, one-to-many or many-to-one synchronisation etc. The main difference between *TAtoC* and the UPPAAL-TIMES code generator is that *TAtoC* is state oriented while UPPAAL-TIMES is transition oriented. Generated source code from a code generator that is state oriented is potentially more run-time efficient.

The left part of Figure 7 shows the data structures used to represent states and transitions in a timed-automata system. A state (lines (1)-(5)) is simply represented by a number of transition objects. Each state in a timed automaton is directly mapped to one struct state. A separate data structure (line (18)) points to each active state. Each automaton (template), in a timed-automata system, will have one active state each. Hence, the number of pointers of current_active will be equal to the number of templates in a timed-automata system.

The lines (7)-(16) represent a transition (left part of Figure 7). The index variable (line (9)) is a reference to both the transition guard, and the assignment operations (if any). The guard and the assignment operations are stored in two functions, shown in the right part of Figure 7. The lines (10)-(12) (left part of Figure 7) represent the synchronisation part of a transition. It does not matter if the transition is a sender (the synchronisation initiator) or not, it may still have multiple transitions connected to it. Line (13) connects the transition to a template and the data structures on lines (14)-(15) point to the two states that the transition interconnects.

```
1: struct state {
2:
3:      int nr_of_outgoing_trans;
4:      struct trans **trans_from_state;
5: };
6:
7: struct trans {
8:
9:      int index;
10:     int nr_synced;
11:     int *synced_with;
12:     char is_send;
13:     int template_index;
14:     struct state *to_state;
15:     struct state *from_state;
16: };
17:
18: struct state **current_active;
```

```
1: int evaluate_guard(int index) {
2:
3:      switch(index) {
4:              case 1:
5:              .
6:              .
7:              .
8:      }
9: }
10: void assign_trans(int index) {
11:
12:     switch(index) {
13:             case 1:
14:             .
15:             .
16:             .
17:     }
18: }
```

Fig. 7. State and transition mapping (left sub-figure), and the guard and assign mapping of transitions (right sub-figure).

Figure 8 shows how the generated C-code from *TAtoC* and UPPAAL-TIMES differs. The leftmost pseudo code represents the main controller-loop in the generated code from *TAtoC* and the rightmost part represents the corresponding code from UPPAAL-TIMES. *TAtoC* has fewer LOC and less iteration steps than UPPAAL-TIMES. The first two for loops (line (1) and (2)) in *TAtoC* will potentially have less iteration steps than the corresponding part on line (1) in UPPAAL-TIMES. Take for example the automata system in Figure 6. *TAtoC* would have three states in Active_States and each state's Outgoing_Trans would be one. However, All_Trans in UPPAAL-TIMES would be four since its code generator would traverse all edges in a timed-automata system. *TAtoC* does not check that a transition is active, unlike UPPAAL-TIMES (line (2)), since they are organised in Active_States. Both generators will do the same check for channeled transitions (lines (4-12) on the left sub-figure and the corresponding part on lines (5-13) on the right sub-figure). When transitions are taken (with or without channels), then both generators will update variables in the same manner (Update_Variables(Trans) and Update_Variables(Ch_Trans)). An example of a variable update is shown in Figure 6 when the variable m is assigned new values (m = 0, m = m + 1 and m = m + 2). One of

```
1: for each State in Active_States do
2:  for each Trans in State.Outgoing_Trans do
3:   if Trans.Has_Channels = TRUE then
4:    Sync := FALSE
5:    for each Ch_Trans in Trans.Channeled_Trans do
6:     if Ch_Trans.Is_Active = TRUE then
7:      if Ch_Trans.Guard = TRUE then
8:       Sync := TRUE
9:       goto 13:
10:     fi
11:    fi
12:   od
13:   if Trans.Guard = TRUE and Sync = TRUE then
14:    if Trans.Is_Send = TRUE then
15:     Update_Variables(Trans)
16:     Update_Variables(Ch_Trans)
17:    else
18:     Update_Variables(Ch_Trans)
19:     Update_Variables(Trans)
20:    fi
21:    Active_States := Trans.Next_State
22:    Active_States := Ch_Trans.Next_State
23:    goto 1:
24:   fi
25:  fi
26:  else // if Trans.Has_Channels =
27:  FALSE then
28:   if Trans.Guard = TRUE then
29:    Update_Variables(Trans)
30:    Active_States := Trans.Next_State
31:    goto 1:
32:   fi
33:  fi
34: od
35:od
36:
37:
38:
39:
40:
41:
42:
43:
44:
45:
46:
47:
48:
49:
50:
```

```
1: for each Trans in All_Trans do
2:  if Trans.Is_Active = TRUE then
3:   if Trans.Guard = TRUE then
4:    if Trans.Has_Channels = TRUE then
5:     Sync := FALSE
6:     for each Ch_Trans in Trans.Channeled_Trans do
7:      if Ch_Trans.Is_Active = TRUE then
8:       if Ch_Trans.Guard = TRUE then
9:        Sync := TRUE
10:       goto 14:
11:     fi
12:    fi
13:   od
14:   if Sync = TRUE then
15:    if Trans.Is_Send = TRUE then
16:     Update_Variables(Trans)
17:     Update_Variables(Ch_Trans)
18:    else
19:     Update_Variables(Ch_Trans)
20:     Update_Variables(Trans)
21:    fi
22:    for each N_Trans in Trans.Next_State do
23:     N_Trans.Is_Active := FALSE
24:    od
25:    for each P_Trans in Trans.Prev_State do
26:     P_Trans.Is_Active := FALSE
27:    od
28:    for each N_Trans in Ch_Trans.Next_State do
29:     N_Trans.Is_Active := FALSE
30:    od
31:    for each P_Trans in Ch_Trans.Prev_State do
32:     P_Trans.Is_Active := FALSE
33:    od
34:    goto 1:
35:   fi
36:  fi
37:  else // if Trans.Has_Channels =
38:  FALSE then
39:   Update_Variables(Trans)
40:   for each N_Trans in Trans.Next_State do
41:    N_Trans.Is_Active := FALSE
42:   od
43:   for each P_Trans in Trans.Prev_State do
44:    P_Trans.Is_Active := FALSE
45:   od
46:   goto 1:
47:  fi
48: fi
49: fi
50:od
```

Fig. 8. Main control loop in the generated code from *TAtoC* (left sub-figure) and the UPPAAL-TIMES tool (right sub-figure).

the main benefits with the state oriented approach is clearly shown on the lines (21), (22) and (29) in the left sub-figure of Figure 8. The corresponding part in the right sub-figure is located on lines (22)-(33) and (39)-(45). When a transition is taken, then the generated code from UPPAAL-TIMES must deactivate all transitions in the previous active state and activate all transitions in the next active state. For *TAtoC* on the other hand, performing an assignment to a pointer (Active_States := Trans.Next_State) is all that is required when making a state transition. Observe that Active_States can contain several active states.

We will present the performance difference between the generated code from *TAtoC* and UPPAAL-TIMES in Section V.

B. Schedulers

This section will present the modelling and verification work related to the development of our four resource-reservation based schedulers. All schedulers implement FPPS of periodic servers. The scheduling hierarchy is one-level deep. Hence, our schedulers implement a global scheduler that schedules servers, and each server hosts one thread (there are no local schedulers). We believe that it makes sense to only verify the global scheduler and not the local schedulers since the local ones are application dependent. Some applications might not need verified local schedulers or even any local scheduler for that matter. Besides, a misbehaving local scheduler in one application does not affect the behavior of other applications (in the same scheduling level), whereas the global scheduler affects all applications.

We refer to [4] for more information concerning the verification of the *Times-ImprovedCgen* scheduler (Section IV-B2), the *UPPAAL* scheduler (Section IV-B3) and the *UPPAAL-ImprovedModel* scheduler (Section IV-B4). The verification of these three schedulers is not a contribution of this paper. We merely adapted these models and used them to synthesise the scheduler code, using our code generator *TAtoC*.

1) Frama-C: Figure 9 (left sub-figure) shows the pseudo-code that represents the global scheduler. The global scheduler is executed as an interrupt-handler at every OS tick. There are two main events that are handled by the global scheduler; the depletion of the budget of a server (line (2)), and the release of servers (line (13)). The variable Time represents the current clock tick. The function ContextSwitch() will stop one server and start another one. The ReadyQ_ functions are responsible for maintaining all active servers in a queue, sorted based on their priorities. The ReleaseQ_ functions are responsible for maintaining all (active and inactive) servers in a queue, sorted based on their release times (in order to achieve a periodic execution of servers). The functionality of the global scheduler and several queue-management functions were verified with Frama-C. Figure 9 (right sub-figure) shows the internal structure of one of the queue functions (lines (1) to (14)) and the corresponding Frama-C verification statements (lines (15) to (27)). These verification statements were used to verify lines (2) and (3) in the global scheduler (left sub-figure of Figure 9). We only show a subset of the verification statements due to space restrictions. The functionality of

ReadyQ_RetrieveFirstItem is to remove the server with the highest priority from the priority queue (pq *queue) and return it. Assume that line (3) in the global scheduler is replaced with the function body of ReadyQ_RetrieveFirstItem (lines (3) to (13)). The verification statements (lines (15) to (27)) could then be used to verify this part of the global scheduler. There are two behaviors defined at line (16) and (23). The correct behavior to be verified requires the pre-condition that the ready-queue has at least one item during the time when a server depletes its budget (line (17) and (18)). In this case, the post-condition requires the queue length to decrement by one (line (19)), and that the item to be returned is indeed the item in the end of the queue, before the function reorders it (line (20)). The re-ordered queue should not contain the element that we just removed (line (21) and (22)). The incorrect behavior is defined as follows; when there are no elements in the ready queue at the time of the server budget-depletion (line (24) and (25)).

The disadvantage with the Frama-C verification is that we can not verify that queue->len will never be less or equal to zero (line (25)) since it is a function parameter. We can only guarantee that the function returns NULL if queue->len is less or equal to zero.

We do not show the verification of the functions max_element (line (8)) and sort (line (10)) due to space limitations in this paper.

```
1: function global_scheduler begin
2:  if Time = BudgetDepletion then
3:   InactiveServer := ReadyQ_RetrieveFirstItem()
4:  if ReleaseQ_GetFirstItem() != Time then
5:   ActiveServer := ReadyQ_GetFirstItem()
6:   if ActiveServer = NULL then
7:    ContextSwitch(InactiveServer, IdleServer)
8:   else
9:    ContextSwitch(InactiveServer, ActiveServer)
10:  fi
11: fi
12: fi
13: if ReleaseQ_GetFirstItem() = Time then
14: while (Server:=ReleaseQ_GetFirstItem()) = Time do
15:  ReadyQ_Insert(Server)
16:  ReleaseQ_Update(Server.Period)
17: od
18: if ActiveServer = NULL then
19:  ActiveServer := ReadyQ_GetFirstItem()
20:  ContextSwitch(IdleServer, ActiveServer)
21: else if ReadyQ_GetFirstItem()!=ActiveServer then
22:  InactiveServer := ActiveServer
23:  ActiveServer := ReadyQ_GetFirstItem()
24:  ContextSwitch(InactiveServer, ActiveServer)
25: fi
26: fi
27:end
```

```
1: server_t *ReadyQ_RetrieveFirstItem(int temp,
2: server_t *SERV, pq *queue, int *first, int max) {
3:  if (queue->len <= 0) {
4:   return NULL;
5:  }
6:  else {
7:   temp = queue->index[0];
8:   max = max_element(queue->priority, queue->len);
9:   *first = queue->priority[max]+1;
10:  sort(queue->priority, queue->index, queue->len);
11:  queue->len--;
12:  return &(SERV[temp]);
13: }
14:}
15: /*@
16:behavior CorrectDepletion
17:  assumes Time == BudgetDepletion;
18:  assumes queue->len > 0;
19:  ensures (queue->len+1) == \old(queue->len);
20:  ensures \result == &(SERV[\old(queue->index[0])]);
21:  ensures \forall integer i; 0 <= i < queue->len ==>
22:          \old(queue->index[0]) != queue->index[i];
23:behavior IncorrectDepletion
24:  assumes Time == BudgetDepletion;
25:  assumes queue->len <= 0;
26:  ensures \result == NULL;
27:*/
```

Fig. 9. Pseudo-code of the resource-reservation scheduler (left sub-figure) and an example of a queue function that was verified using Frama-C (right sub-figure).

Conclusively, we could (with a lot of effort) verify the queue functionality of the global scheduler using Frama-C. As an example, we showed roughly how we verified the ReadyQ_RetrieveFirstItem function (right sub-figure in Figure 9). However, verifying the timing correctness of the scheduler was much more challenging. Line (17) in the right sub-figure of Figure 9 represents such a challenge. The variable Time represents the global (absolute) system-time, while BudgetDepletion represents the internal scheduler absolute time (a time event which in this case represents the ending of a server's budget). Line (17) alone does not verify that the server executed the amount of budget time that it was allocated. Hence, we had to implement observers that handled the bookkeeping of time spend by all servers. We then verified this time by comparing it to the system time using ensures clauses. The tricky part was to get a global view of the verification, i.e., it was challenging to verify all properties simultaneously since there were restrictions on function calls. Hence, we had to verify each function in isolation, or skip function calls and simply inline the function source-code directly. The OS tick was simulated using a for loop and the Time variable represented the OS tick.

2) Times-ImprovedCgen: The *Times-ImprovedCgen* scheduler is essentially a task-automaton model (Figure 10) that has been modelled in the UPPAAL-TIMES tool. The synthesis of the model was done using the code-generator version of TAtoC (see Section IV-A) that supports UPPAAL-TIMES. Figure 10 represents (a simplified version of) the global scheduler, i.e., it corresponds to Figure 9 (left sub-figure), but in task-automaton form. The state S_Main (in Figure 10) is the initial state and the starting point for every new scheduling event, i.e., server budget-depletion and server-release events. The transition going from S_Main to itself is taken when the system should progress one scheduling tick, i.e., the current active server is allowed to execute one tick (AllowServerToRun?). This is only possible if there are no scheduling events at the current tick (Clock < S_BudgetEvent, Clock < S_ReleaseEvent). The budget-deplete transition (going from state S_Main to S_BudgetDepletion) has a higher priority with respect to the server-release transition (S_BudgetEvent <= S_ReleaseEvent). Task events (originating from within a server) have lower priority if they occur at the same time as budget depletions (S_BudgetEvent <= NextTaskEvent). Observe that these priorities will determine the order of the scheduling events. The budget-deplete event will lead to a deactivation of the current server, followed by a queue update, and finally a context switch. The DepleteObs1! channel will synchronise with an observer automaton which is used when verifying the scheduler.

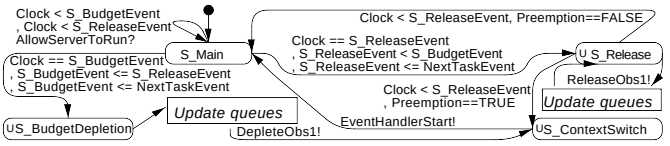

Fig. 10. Task-automaton model of the resource-reservation scheduler (simplified version).

The EventHandlerStart! channel will synchronise with an automaton which is responsible for server context-switching. A server-release event (going from state S_Main to S_Release) has a lower priority than a budget-deplete event (S_ReleaseEvent < S_BudgetEvent) but a higher priority than a task event (S_ReleaseEvent <= NextTaskEvent). The server-release event will trigger an update of the server priority-queue and it will determine the next server-release

event. Hence, the the next server-release event should lie forward in time (Clock $<$ S_ReleaseEvent) relative to the current time. Each server release will be notified (ReleaseObs1!) to the observer automaton for the purpose of verification. A server context-switch will occur if the released server has a higher priority than the current active server (Preemption == TRUE). If this is not the case (Preemption == FALSE) then no re-scheduling will occur. We direct the reader to a technical report [35] for more details regarding the scheduler automaton.

The *Times-ImprovedCgen* scheduler also contains a second task-automaton responsible for the server context-switching, and each server is modelled as a separate task-automaton.

3) UPPAAL: We refer to the third scheduler as the *UPPAAL* scheduler. It is based on a TA model similar to the one presented in Figure 10. The main differences are that we used TA (instead of task-automaton) and the queue-management functions were written in a C-like language instead of being modelled as TA. The UPPAAL tool supports C-like function-calls in the transition (edge) between states. Hence, the TA model for the *UPPAAL* scheduler becomes smaller compared to the *Times-ImprovedCgen* scheduler, i.e., it has less number of states and transitions. This improves the run-time performance of the *UPPAAL* scheduler compared to the *Times-ImprovedCgen* scheduler. The disadvantage with the TA model of the *UPPAAL* scheduler (using the C-like language) is that it becomes less readable. We used the UPPAAL version of *TAtoC* (see Section IV-A) to synthesise C code implementing this scheduler.

4) UPPAAL-ImprovedModel: The fourth scheduler (*UPPAAL-ImprovedModel*) is an improvement of the *UPPAAL* scheduler. The C code implementing the *UPPAAL-ImprovedModel* scheduler was also generated using the the UPPAAL version of *TAtoC*. The largest part of the TA model of the *UPPAAL-ImprovedModel* scheduler was replaced with C-like code. Hence, this scheduler resembles the Frama-C verified scheduler since the global scheduler and queue-management functions (Figure 9) were reused from the *Frama-C* scheduler. The *UPPAAL-ImprovedModel* scheduler can thus be seen as a fusion between the *Frama-C* and the *UPPAAL* scheduler. The advantage with the *UPPAAL-ImprovedModel* scheduler is that the model can be verified using model-checking (which is in our opinion the preferred method to verify schedulers) and the C (like) functions can be verified using Frama-C.

V. Evaluation

We ported the four schedulers (Section IV-B) to the seL4 micro-kernel platform (version 1.1). We instrumented the synthesised schedulers with time-measurement primitives so that we could measure the scheduler overhead, i.e., the run-time performance. We ran the schedulers with different number of servers (2-7), i.e., the schedulers scheduled a different number of servers in each experiment. The number of servers chosen for our experiments was inspired by a wheel-loader application from Volvo Construction Equipment [36]. The varying number of servers in our experiments shows how the performance of the schedulers varied with respect to the scheduling workload. Each server was configured with one thread. The thread itself did not do any useful work, i.e., it had an empty for loop (the computations inside the thread did not affect the measured overhead of our schedulers). The scheduler performance was only affected by the number of servers and their parameters. For example, running more servers will generate more scheduling events since each server will add more work for the scheduler in the form of server activations, budget depletions and context switches.

A. Experimental setup

We ran seL4 on an Intel 533 MHz Pentium3-Katmai processor (model 7, stepping 3) that was emulated with the Quick EMUlator (QEMU) [37] (version 1.3.0). QEMU is an open-source processor emulator that emulates hardware accurately down to CPU-cycle level. We used the ReaD Time-Stamp Counter (RDTSC) processor register (for x86 architectures) to measure the scheduler overhead. This is a reliable method for measuring time on the Pentium3 Katmai processor since Katmai does not have multi-core processors or frequency scaling (SpeedStep). We inserted a CPUID instruction-call before each call to RDTSC in order to flush the instruction pipeline. This prevents out-of-order execution of the RDTSC operation since it serializes the instruction queue.

B. Results

Figure 11 shows the number of measured clock cycles (the average out of 100 trials per server configuration) used by the schedulers to schedule 2-7 servers within a time span of 30 seconds. We included two additional schedulers (called *Non-verified* and *Times*) in the evaluation to act as reference points. The *Non-verified* scheduler is, as the name suggests, a manually coded scheduler which is not verified. The *Times* scheduler is based on the same model as the *Times-ImprovedCgen* scheduler, but we used the default code-generator in the UPPAAL-TIMES tool to generate the C code.

The difference in performance between the *Times* and the *Times-ImprovedCgen* scheduler represents the improved run-time performance of the generated code from *TAtoC* compared to the code generated from the UPPAAL-TIMES tool. We observed a run-time performance difference in most server configurations when comparing *Times* and *Times-ImprovedCgen* in the average case (the upper sub-figure in Figure 11). The run-time performance differed more when we used more servers. However, the most significant difference in performance can be found in the measured maximum values (the lower sub-figure in Figure 11). We can clearly see by these results that the state-oriented approach, implemented by *TAtoC*, has an advantage in the aspect of run-time performance compared to the default code generator in the UPPAAL-TIMES tool (which uses a transition-oriented approach).

The Frama-C verified scheduler had surprisingly better run-time performance than the non-verified scheduler. In fact, it had the lowest run-time overhead of all schedulers. We used a different queue structure/implementation (bubble-sort) in the

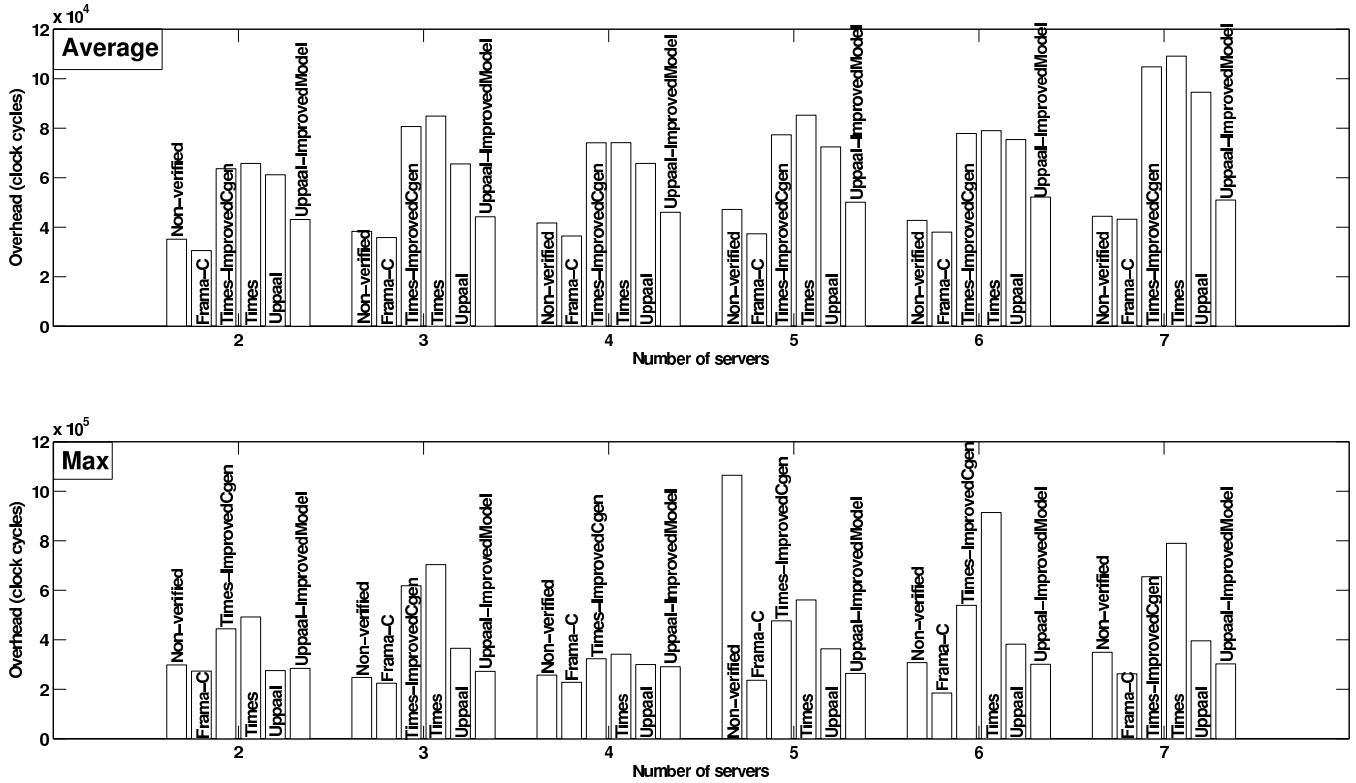

Fig. 11. Clock-cycle measurements of the schedulers in the seL4 micro-kernel.

Frama-C implementation which seemed to be beneficial in terms of performance in this setting, compared to bitmaps, which were used in the non-verified scheduler. We have, in previous studies [38], observed that schedulers with simpler queue algorithms can outperform more complex queue structures in performance when the number of queue elements (in this case servers) are low.

The biggest surprise was the efficient run-time performance of the TA-based schedulers (*UPPAAL* and *UPPAAL-ImprovedModel*). The *UPPAAL* scheduler had slightly less overhead than the *Times* and *Times-ImprovedCgen* scheduler in the average case. The measured maximum number of clock cycles of the *UPPAAL* scheduler are almost equal to the *Non-verified* and *Frama-C* scheduler. This is quite impressive considering that only the queue functions are written in C while the rest of the scheduler is modelled entirely with TA.

The *UPPAAL-ImprovedModel* scheduler had an outstanding good run-time performance. It was almost equally good in performance (both maximum and average values) compared to the *Non-verified* and *Frama-C* scheduler. This is perhaps not so surprising considering that most part of the *UPPAAL-ImprovedModel* scheduler is written in a C-like language.

To conclude, there is a clear trade off between the effort in verifying a resource-reservation scheduler, and the run-time performance (overhead) when running the scheduler. The Frama-C verified scheduler required substantially more effort in the verification process compared to TA and model checking. Although both methods achieved the same verification goals. On the other hand, our results reveal a remarkable good run-time performance of the scheduler that was verified using the Frama-C verification method.

The *UPPAAL* scheduler had a balanced trade off between TA and C (like) code. The queue functions required little verification efforts since we used Frama-C which only verifies functional properties and not timing properties. Replacing the queue TA in the *UPPAAL* scheduler with the corresponding functionality in C-like code (verified with Frama-C) greatly improved the run-time performance of the scheduler, without adding too much overhead in the verification process. Hence, the *UPPAAL* scheduler had the optimal balance between the verification effort needed (and the maintenance of it), and good run-time performance. The verification complexity is also an important factor (besides the run-time performance) since it influences how costly the maintenance of the (verified) software will become, in case it is expected to be updated frequently. Hence, we consider the *UPPAAL* scheduler as the winner in the aspect of both verification and run-time performance since it merges the verification techniques of Frama-C and model checking in a optimal way.

VI. Conclusion

We have observed that schedulers verified with theorem proving (using Frama-C) are efficient in performance. This is of course not surprising. However, the verification procedure/language (in this case ACSL) is not equally as expressive as timed-automata and model-checking. On the other hand,

we found model-checking to be more efficient in verifying schedulers than ACSL and theorem proving. The main advantage with timed-automata and model-checking is the notion of time that can be expressed between two events, such as two execution instances of a scheduler. However, synthesised code from timed automata does not have good run-time performance. Frama-C can verify the internals of C functions, but not the relation between two calls to the function, while model-checking tools such as UPPAAL do not verify C functions directly (but rather indirectly by model-checking a model which is not as expressive as Frama-C). Hence, we see a great potential in merging Frama-C and model-checking verification since the two complement each other. We found that this merge improved the overall effectiveness of the scheduler verification process, while also minimising the run-time overhead of the scheduler.

We developed a scheduler that was verified using both UPPAAL and Frama-C. We made the synthesis of the scheduler possible by developing a code generator for the UPPAAL tool. We have shown with experiments in the seL4 micro-kernel that the performance of this scheduler is comparable with a manually coded scheduler. We have also shown that our code generator *TAtoC* generates more run-time efficient C code than the UPPAAL-TIMES code generator.

To conclude, our scheduler is verified in a more expressive way by merging two verification techniques, but our main contribution is the enabling of efficient synthesis of this scheduler using our state-oriented code generator *TAtoC*.

For future work we plan to improve our code generator and also focus on other verification techniques such as Event-B.

REFERENCES

[1] ARINC, *ARINC 653: Avionics Application Software Standard Interface (Draft 15)*. Airlines Electronic Engineering Committee (AEEC), June 17th, 1996.

[2] T. Scharnhorst, H. Heinecke, K.-P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, P. Heitkämper, J. Leflour, J.-L. Mate, and K. Nishikawa, "AUTOSAR - Challenges and Achievements," in *VDI'05*.

[3] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal Verification of an OS Kernel," in *SOSP'09*.

[4] M. Åsberg, P. Pettersson, and T. Nolte, "Modelling, Verification and Synthesis of Two-Tier Hierarchical Fixed-Priority Preemptive Scheduling," in *ECRTS'11*.

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

[6] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a Nutshell," *Journal on Software Tools for Technology Transfer*, vol. 1, pp. 134–152, 1997.

[7] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C, A Software Analysis Perspective," in *SEFM'12*.

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

[9] G. Kroah-Hartman, J. Corbet, and A. McPherson, "Linux Kernel Development: How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It," in *The Linux Foundation*, 2009.

[10] R. Alur and D. L. Dill, "A Theory of Timed Automata," *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994.

[11] E. Fersman, P. Krcal, P. Pettersson, and W. Yi, "Task automata: Schedulability, decidability and undecidability," *Information and Computation*, vol. 205, pp. 1149–1172, 2007.

[12] C. Marché and Y. Moy, "The Jessie plugin for Deduction Verication in Frama-C," version 2.30. INRIA, 2012. http://krakatoa.lri.fr/jessie.pdf.

[13] P. Baudin, J.-C. Filliatre, C. Marché, B. Monate, Y. Moy, and V. Prevosto, "ACSL: ANSI/ISO C Specifcation Language," version 1.6, 2012, Frama-C Oxygen implementation.

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

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

[16] J. Ferreira, G. He, and S. Qin, "Automated Verification of the FreeRTOS Scheduler in HIP/SLEEK," in *TASE'12*.

[17] C. Fidge, P. Kearney, and M. Utting, "Interactively Verifying a Simple Real-Time Scheduler," in *CAV'95*.

[18] A. Gotsman and H. Yang, "Modular Verification of Preemptive OS Kernels," in *ICFP'11*.

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

[20] 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.

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

[22] 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.

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

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

[25] 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*.

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

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

[28] L. Carnevali, G. Lipari, A. Pinzuti, and E. Vicario, "A Formal Approach to Design and Verification of Two-Level Hierarchical Scheduling Systems," in *Ada-Europe'11*.

[29] 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*.

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

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

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

[33] T. Amnell, E. Fersman, P. Pettersson, W. Yi, and H. Sun, "Code Synthesis for Timed Automata," *Nordic Journal of Computing*, vol. 9, pp. 269–300, 2002.

[34] W. Yi, "A Calculus of Real Time Systems," Ph.D. dissertation, Department of Computer Science, 1991.

[35] M. Åsberg, "Model of Two-Tier Hierarchical Fixed-Priority Preemptive Scheduling," Mälardalen University, Technical Report, Nr. 2379, 2011.

[36] T. Nolte, I. Shin, M. Behnam, and M. Sjödin, "A Synchronization Protocol for Temporal Isolation of Software Components in Vehicular Systems," *IEEE Transactions on Industrial Informatics*, vol. 5, pp. 375–387, 2009.

[37] F. Bellard, "QEMU, A Fast and Portable Dynamic Translator," in *ATEC'05*.

[38] M. Åsberg, "Comparison of Priority Queue algorithms for Hierarchical Scheduling Framework," Mälardalen University, Technical Report, Nr. 2598, 2011.