Requirement specification and model-checking of a real-time scheduler implementation

Khaoula BOUKIR
khaoula.boukir@ls2n.fr
University of Nantes
Nantes, France

Jean-Luc BÉCHENNEC
jean-luc.bechennec@ls2n.fr
CNRS
Nantes, France

Anne-Marie DÉPLANCHE
anne-marie.deplanche@ls2n.fr
University of Nantes
Nantes, France

ABSTRACT

Implementing a new scheduler within a real-time operating system is challenging. The transition from a theoretical scheduling policy specification to a real platform implementation requires several constraints to be taken into account. Therefore, a verification process must support the implementation work to give it a level of confidence and validate its correctness.

In this paper, we present such a verification approach which is based on model-checking. It aims to identify subtle issues in our implementation of scheduling policies within an OSEK/VDX real-time operating system called Trampoline. As an example, the approach is conducted on elaborated models of an implemented G-EDF scheduler along with other OS components that contribute to the scheduling decision. Then, the verification is carried out by checking a set of relevant requirements which are identified based on the expected behavior of the scheduler as specified in the literature. This approach demonstrated its feasibility since potential issues are detected in our implementation.

CCS CONCEPTS

• **Computer systems organization** → **Real-time operating systems**; • **Theory of computation** → *Verification by model checking*.

KEYWORDS

Model-checking, RTOS, Implementation, scheduler

ACM Reference Format:
Khaoula BOUKIR, Jean-Luc BÉCHENNEC, and Anne-Marie DÉPLANCHE. 2020. Requirement specification and model-checking of a real-time scheduler implementation. In *28th International Conference on Real-Time Networks and Systems (RTNS 2020), June 9–10, 2020, Paris, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3394810.3394817

1 INTRODUCTION

Adopting multicore architectures for executing real-time applications has given rise to multiple research projects in order to propose efficient scheduling policies [3]. This has motivated some new works of implementing dynamic scheduling policies such as EDF [11, 15, 22]. However, to adopt such policies, it is essential to first study the feasibility of their implementation within a real platform and to evaluate their true performances. In this context, our goal is to extend our RTOS Trampoline [5] so that it can support global multiprocessor scheduling policies.

In the current design of Trampoline, the integration of new schedulers is *kernel-based*. However, scheduler implementation at the kernel level is not an easy task: it requires a complete understanding of the OS architecture and code. Multiple implementation constraints which are completely abstracted in literature must be considered and the gap between theoretical scheduling specification and concrete RTOS considerations makes the scheduler programming task complex.

This gives rise to many issues: how to ensure that the implementation of a new scheduling policy within an RTOS is correct? How to verify that the implemented scheduler always produces the expected behavior in accordance with the specifications given in theory? What are the effective means to verify this correctness and how to establish an efficient approach for this verification? In fact, if we want theoretical results in real-time scheduling to emerge in practice, implementations should be supported with verification methods that can provide confidence on their functional correctness. Manual verification cannot be an option considering the implementation complexity and constraints. A possible approach to detect implementation errors is simulation under significant scenarios. However, one cannot identify all possible scenarios that the system must deal with.

In order to address this problem, we aim to use formal approaches to formally verify the implementation of global scheduler within a real-time operating system. To do so, we focus on verifying requirements of the implemented scheduler. Our study is not intended to verify the schedulability of an application but rather to verify the behavior of the scheduler implementation. Our approach is based on a previously elaborated model of an implementation of G-EDF in Trampoline [10]. This model describes carefully the control flow of the implementation since it abstracts every instruction of the source code into a transition in the model using the same variables and actions of the implementation.

Contribution and outline: The work described in this article presents a formal approach that addresses the issue of verifying the correctness of a global scheduler implementation within the purpose-built embedded *Trampoline* RTOS. The main idea is to provide evidence that the implemented scheduler behavior matches its specifications.

This study is the follow-up of a prior work where an implementation of G-EDF scheduler within Trampoline, that only supported static partitioned scheduling, was proposed. This previous work was also associated with initial ideas to verify the implementation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
RTNS 2020, June 9–10, 2020, Paris, France
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7593-1/20/06…$15.00
https://doi.org/10.1145/3394810.3394817

[10]. However, the verification was partial and performed for one application which does not guarantee an exhaustive verification of the scheduler. Our rational in this paper is to rather use the previously achieved implementation and conduct a modular verification approach on it based on specific requirements. These requirements are identified in accordance with the expected behavior of the implementation and demonstrated to be significant for it. They are verified using model-checking to ensure that the implementation is compliant with the scheduler's specifications. To do so, verification engines are elaborated to generate all possible scenarios for activating and executing a given task set.

The rest of this paper is organized as follows. Section 2 discusses related works. Section 3 provides a review of the initial work of this study. Section 4 explains the choice of requirements to be verified for the implementation. Section 5 presents our approach for conducting verification of these requirements and Section 6 illustrates some of its results.

2 RELATED WORKS

There have been a number of works using formal methods to verify real-time operating systems regarding various problems. Fundamentally two main approaches are being used:

Theorem proving: is a formal method characterized by describing the specification of a system in a mathematical reasoning system, which consists of a set of axioms and a set of inference rules that can be used to derive new theorems. In the field of OS kernel verification, this approach has been used to verify SeL4 [19] and CertikOS [18]. Both are certified OSs using interactive theorem proving although they don't support real-time applications. SeL4 is a micro-kernel that uses the theorem prover Isabelle/HOL [21]. CertikOS uses the Coq proof assistant [14] for its certification.

The theorem proving was also used in numerous works that are interested in studying the schedulability within a real-time operating system. The work around Prosa for instance [13] addresses the response time analysis for global fixed priority and EDF schedulers and provides a flexible foundation for formally proven schedulability analysis based on mechanized proofs. This work does not match with our objective since we are interested in the verification of implementations and not in the analytical aspects of real-time systems.

The aforementioned verification approaches require user guidance and the specification of many lemmas and a large number of commands. This demands a significant expertise in the field and a large human investment to prove small theorems, which gets worse in the event of proving sophisticated properties.

Model-checking: is an automated formal verification approach that allows the exploration of all possible states that the verified system can go through and checking desirable properties over its state-space. Thanks to its advantages, model-checking has been successfully applied in numerous research works related to the field of real-time systems. Those related to scheduling have as their objective either to verify the schedulability of a system [16], or to build a scheduler tailored to a particular system using the paradigm of controller synthesis [1]. For the most part, they are based on the theory of timed models such as timed automata or time Petri nets, and on state space analysis methods. Another study [24] of modeling an OSEK/VDX application and the core part on its kernel have been proposed, it uses UPPAAL[7] to perform rigorous timing analysis. Bodeveix et al. [8] used also model-checking to verify some scheduler properties within BOSSA framework [4] as well as several safety properties of the OS. This formal method has been also applied in hardware and software verification. For instance, Intel has been using it since 1990 to build industrial verified systems [2, 17]. It has also reported the success of this technique to detect programming errors that would have been escaped all the verification tools.

Our rationale for employing formal verification is quite different from the works presented above: given a scheduler implemented inside an RTOS kernel, we aim at checking the correctness of its implementation formally based on a model. By correctness, we mean not only the absence of programming errors that are likely to crash the OS, but also the accuracy of the scheduling operations and decisions. It is a similar objective that can be found in [20]. In the context of a *meta scheduler* for implementing real-time scheduling algorithms on POSIX compliant RTOS, model-checking is used to ensure that the outputs from a scheduling algorithm using the meta scheduler framework conform to the scheduling decisions of its native version implemented inside the kernel. This study however, does not consider the multicore/multiprocessor constraints, and the verification is conducted on one application. Our verification work on the other hand, aims to verify the implementation of a global multiprocessor scheduling policy regardless of any specific task. It is rather in line with the work of Tigori et al [23] where the whole platform independent source code of Trampoline is modeled as a network of extended finite automata. By adding an application model and doing a reachability analysis, the operating system can be configured so that all the dead code is removed. In [6] the model is extended with observers to verify the OSEK/VDX conformance of the configured operating system.

3 BACKGROUND

This section provides a summary of the work previously carried out [9, 10] to implement and model a G-EDF scheduler within Trampoline RTOS [5].

3.1 Implementation of G-EDF

Trampoline[1] [5] is an open source real-time operating system that implements the OSEK/VDX and AUTOSAR standards. It was developed in the Real-Time Systems group at LS2N in Nantes (France). Initially, it supported partitioned scheduling in multicore. Trampoline's scheduler is implemented at the kernel level. This means that multiple components and functions of low-level are involved in calculating the scheduling decision and they define the *scheduling perimeter*. These components are detailed below (cf. Fig. 1).

Task Manager: it gathers low-level functions that are responsible for activating and terminating jobs.

Time Manager: it handles the calculation, the representation and the comparison of absolute deadlines.

Task List Manager: it handles all the operations related to the insertion and extraction of ready jobs into task lists on the request

[1] http://trampoline.rts-software.org

of *Task Manager* or the *Scheduler*. It also takes in charge of sorting these lists.

Scheduler: it is responsible for determining the scheduling sequence according to G-EDF policy. It is called by *Task Manager* whenever a job is activated or terminated.

Context Switch Manager: performs the context saving of the job that is preempted and restores the context of the one that gains the CPU following the `Scheduler`'s decisions.

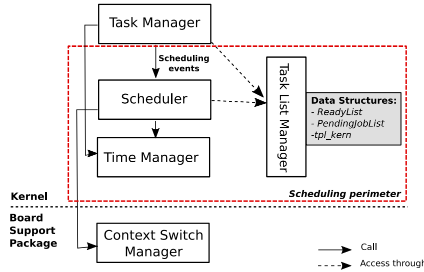

Figure 1: Interactions of G-EDF scheduler with other Trampoline's components.
The scheduling decision is calculated by the Scheduler using the informations provided by the Task List Manager and the Time Manager

3.1.1 *Some Trampoline features.*

In Trampoline's G-EDF scheduler, an absolute deadline is computed whenever a job is activated by adding its corresponding relative deadline to the current time. The time in Trampoline is represented on a circular time model with 32-bit variable and $1\mu s$ resolution. The comparison of deadlines is performed using the ICTOH algorithm (for *Circular Timer's Overflow Handler*) [12].

Trampoline OS allows successive activations of the same task up to a statically fixed limit that is called the maximum activation count. This means that the release of a new job of a task may occur even if the previous one did not terminate. The number of jobs activated for a task and not yet completed is called the activate count. However, two jobs of the same task should not be executed at the same time and the priority of execution shall be attributed to the oldest released job. For this purpose, two types of ready job lists are considered: 1) the `ReadyList:` to store the oldest released and non-terminated ready jobs of each task that we call *active jobs*. It is implemented as a *min heap* in which absolute deadlines of jobs are used as keys (values of the nodes). If several jobs have the same deadline, they are attached as a linked list to the same heap node. 2) the second type is used to store new released jobs of a task while the previous one is not terminated. They are called *pending jobs*. Each task τ_i has its own list $\texttt{PendingJobList}_i$. It is implemented as a FIFO according to the task activation order and it stores its pending job absolute deadlines.

In Trampoline, the scheduling operations always take place on the core where the scheduling triggering event occurs. However, rescheduling may lead to a preemption and a context switching that have to be achieved on another core. So as to separate jobs that are considered by the scheduler from the actual ones that are running on processors, the `tpl_kern` structures are available, one by core. Each one contains the `running`, `elected` and `need_switch` fields: - `running` gives the identifier of the running task, - `elected` contains the identifier of the task that is selected by the *Scheduler* and, - `need_switch` indicates if a context switching must take place.

Note that in the multi-core version of Trampoline, there is only one instance of the OS that runs sequentially in the cores in which the OS should be executed. Thus, the access to the kernel by a core is sequentialized due to a global lock that prevents competition between cores. It is important to note that the main purpose of this paper is not to propose an implementation of G-EDF scheduler. Thus, for more details about this implementation please refer to [9, 10].

3.1.2 *Scheduling procedure.*

In our implementation of G-EDF, only basic and independent tasks are considered. Meaning that we do not focus on tasks that can wait for events or share resources. Thus, the scheduler is triggered only for job activation or completion. The steps for making the scheduling decision within Trampoline are summarized below:

1. When a scheduling event occurs and it is a new job activation of a task τ_i, the absolute deadline of the job is computed by the *Time Manager*. If the activation count indicates that there is already an active job for that task, the *Task Manager* **calls** the *Task List Manager* in order to store the new activated job in the $\texttt{PendingJobList}_i$ along with its absolute deadline. Otherwise, the job is stored in the `ReadyList` according to its absolute deadline and the *Task Manager* puts the task τ_i in the `READY` state. If the scheduling event is a termination, the $\texttt{PendingJobList}_i$ of the terminated task τ_i is checked. If there is a pending job, it is removed from the $\texttt{PendingJobList}_i$ and stored in the `ReadyList` according to its absolute deadline. In that case the state of the task is changed to `READY`, otherwise it becomes `SUSPENDED`. In the end, the *Task Manager* **calls** the *Scheduler*.

2. When the *Scheduler* is called, it selects the highest priority jobs by **consulting** the `ReadyList` and the absolute deadlines of jobs calculated by the *Time Manager*. In the implementation, the `ReadyList` is sorted in an increasing absolute deadlines order. Therefore, the *Scheduler* must only select the m jobs from the head of the `ReadyList` to be executed on the m free processors. To do so, two tests are iteratively performed: 1) is there a free processor while the `ReadyList` is not empty? 2) does the job at the head of the `ReadyList` has a higher priority than the running one? In the first case, the job at the head of the list is extracted to be executed on the free processor. In the second case, the running job is preempted, put back in the `ReadyList` and its state is changed to `READY`. Then, the higher priority task is extracted and scheduled for execution.

3. When the scheduler decides that a context switch shall take place on a core, it updates `elected` and sets `need_switch` in the corresponding `tpl_kern`. Then, the *Context Switch Manager* saves the context of the task identified in `running`, loads the context of the one given by `elected`, copies the value of `elected` into `running` and the state of the task becomes `RUNNING`.

3.2 Implementation modeling

3.2.1 ***General modeling techniques.*** Our work concerns only the implementation of the scheduler and does not focus on modeling the entire OS. As a result, a pre-existing OS model proposed by Tigori and al. [23] is used as a basis for building models concerning our implementation. The OS model is elaborated using UPPAAL model-checker [7] which accounts for the use of it in our work. In the models, each Trampoline's function is abstracted by an Extended Finite Automaton (EFA) or by a function written in UPPAAL language (a syntax similar to C language) using the following rules:

- the structure of the automaton describes the control flow of the modeled system.
- the variables used in the model are the control variables of the system.
- a sequence of instructions is abstracted by a single action with a set of updates and/or guards on the set of variables. In such a case, their execution is considered "*atomic*".
- actions and conditions on variables attached to each transition are the same ones that correspond to the source code of the system.
- the imperative code associated with each transition of the automaton is similar to the source code of the system.

In every automaton of the OS model, a transition corresponds either to a test or an action (an assignment or a function call). The transition can be firable only if the test is satisfied, and as soon as it is fired, the associated action is performed.

Tests are translated to a guard and assignments by an update over the control variables. Function calls are handled by a synchronization mechanism between the two automata. First, the caller automaton releases the execution of the callee automaton with a synchronization over a channel (with "!" to emit and "?" to receive) and sets a shared variable to 0. A guard using this variable blocks the caller while it is equal to 0. Second, the callee automaton completes its instructions and sets the shared variable to 1 to release the caller. This ensures the sequential execution conforming to the real execution. The function parameters are passed from the caller automaton to the callee automaton by using shared variables.

In the model, the kernel is considered *untimed*. Each state, except the initial state, of an automaton is urgent. Accordingly, from the point of view of the other components of the system, the execution of a sequence of kernel code is performed in zero time. In fact, taking into account the OS execution overheads can be useful in the case of checking whether or not the deadlines are met as in schedulability analysis. This does not interfere with our objectives since we aim to verify the functional aspect of the G-EDF implementation. Thus, our model does not consider overheads and only elapses time while tasks are being executed but not at the kernel level.

3.2.2 ***G-EDF model.*** Using the rules described above, the implemented functions of the components involved in the scheduling are modeled using EFAs. Due to space limits, we only present EFAs abstracting the *Scheduler* and the *Timer* functioning.

In the model, the scheduling function `tpl_schedule()` is executed whenever the EFA that models the *Task Manager* is synchronized with the *Scheduler*'s EFA. Then, the scheduling is performed in two loops. The first one is used to start the highest priority ready jobs

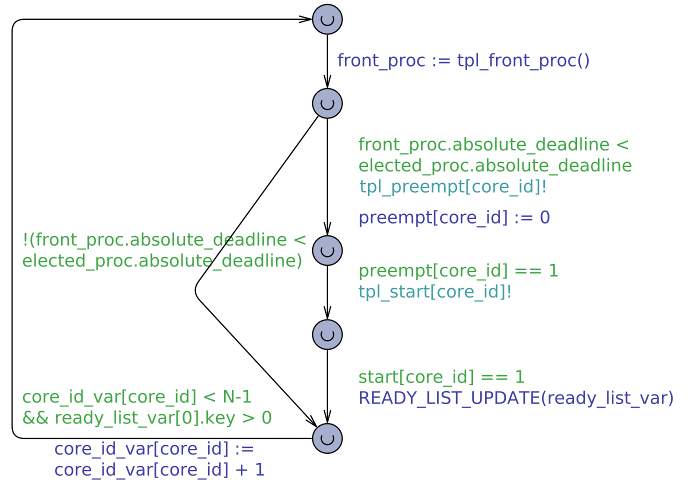

Figure 2: part of the *Scheduler* automaton: Double circle represents initial location, the ∪ inside a location denotes an urgent location and each location describes the function execution state. Guards are depicted in green, synchronization in light blue and actions in dark blue.

on the free cores available. The second one is used to check if there is a job stored in the `ReadyList` which has a higher priority than a running job. A part of the scheduling function model is illustrated in Fig 2. This part first describes the process of comparing the deadlines of the running task (the one in the head of the `ReadyList` (`front_task`)) and the preemption of the running task if it has lower priority followed by the execution of the `front_task`.

3.2.3 ***Timer model:*** The G-EDF scheduler is based on deadlines to perform the scheduling. Thus, it is necessary to model a mechanism that allows retrieving the current time in order to calculate the absolute deadlines. To do so, a timed automaton that models a *Timer* is considered. It uses a clock variable to represent the progression of time. The automaton handles also a shared variable that represents the time in microseconds (as in the real implementation in Trampoline). This variable is incremented whenever the *Timer* automaton is synchronized with other automata in which passing time is needed as shown in 3. It is necessary to distinguish this *Timer* which is used only in the model for time progression, from the *Time Manager* which is an OS component responsible for the calculation and comparison of absolute deadlines.

4 REQUIREMENTS SPECIFICATION

G-EDF is a work-conserving and fixed job priority policy. The priority assignment is based on the absolute deadlines of jobs. Therefore, the implemented scheduler must ensure that at any time t, it is the m jobs (at most) with the closest absolute deadlines that are running on the m processors. It is expressed by the two following requirements:

- at any time t, all the tasks that are in the `RUNNING` state have smaller absolute deadlines compared to any other task in the `READY` state.

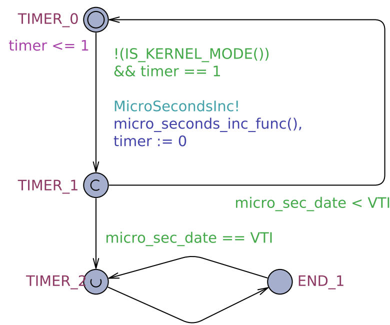

Figure 3: the *Timer* automaton: The timer emits a call through a synchronization on a broadcast channel `MicroSecondesInc!`. *IS_KERNEL_MODE*() is a function used as a transition guard to make sure that the execution is not at the kernel level since the kernel is *untimed*. When it is fired, it calls a function `micro_secondes_inc_func()` to increment the time variable in μs.

- a processor cannot be free while there is at least one task in the `READY` state.

4.1 Taking implementation specification into account

When moving to the actual implementation of G-EDF policy in Trampoline, other requirements have to be taken into account. As presented in 3.1, the scheduling decision within Trampoline involves other components of the OS other than the *Scheduler*. Therefore, checking the correct functioning of these components cannot be ignored. Even if the *Scheduler* operates correctly according to the requirements expressed above, the produced scheduling result might be false if the behavior of the other components is wrong. In our implementation, we can highlight some situations that may occur:

situation 1: the *Task Manager* does not call the *Scheduler* after a job activation or termination. Thus, the scheduling is not performed.

situation 2: the *Task Manager* does not put a new activated job of task τ_i in the `ReadyList` or the $\texttt{PendingJobList}_i$. In that case, the *Scheduler*'s decision does not take into account this new activated job.

situation 3: the *Task Manager* miscalculates the activation count of a task. Then, the jobs can be lost during the process of scheduling.

situation 4: the *Context Switch Manager* does not apply the scheduling decision computed by the *Scheduler*. In that case, the following rescheduling will be based on false results.

In order to take into account such situations, behavioral requirements to analyze the proper functioning of the system components must be expressed.

4.2 OS components requirements

Our approach consists of carrying out the verification in a modular way by verifying separately every component of the scheduling perimeter. It leads to clearly delimit the verification perimeter of each component and to better locate implementation errors. For every component, a set of requirements is specified according to its expected behavior. We distinguish intrinsic requirements that depend on the G-EDF policy and could be reused for another implementation of the policy in another operating system. On the other hand, we also consider requirements that depend on how Trampoline is designed and that must be re-adapted according to the implementation platform.

***Task Manager**:* ensures that the necessary functions for activations or terminations of jobs are performed correctly. The requirements that must be verified are:

- *when a job of a task* i *is activated:*
  - *if the activation count of task* i *is zero, the new activated job shall be put in the* `ReadyList`.
  - *if the activation count of task* i *is zero, the task's state shall become* `READY_AND_NEW`.
  - *if the activation count of task* i *is greater than 0 and less than its maximum activation count, the new activated job shall be put in the* $\texttt{PendingJobList}_i$.
  - *if the activation count of task* i *is equal to its maximum activation count, the new activated job shall be ignored.*
  - *if the activation count of task* i *is less than its maximum activation count, it shall be incremented.*
- *when a job of a task i is terminated:*
  - *the activation count of task* i *shall be decremented.*
  - *if the activation count of task* i *is equal to 1, the task's state shall become* `SUSPENDED`.
  - *if the activation count of task* i *is greater than 1, the task's state shall become* `READY`.
  - *if the activation count of task* i *is greater than 1, the oldest pending job shall be removed from the* $\texttt{PendingJobList}_i$ *and put in the* `ReadyList`.
- *for every job activation or termination of task* i*:*
  - *if the activation count of task* i *is greater than 0, the size of the* $\texttt{PendingJobList}_i$ *shall be greater than 0.*
  - *if the* `ReadyList` *does not contain a job of task* i *and no job of task* i *is running, the* $\texttt{PendingJobList}_i$ *shall be empty.*
  - *the Scheduler shall be called.*

Time Manager: these functions handle the computation and the comparison of absolute deadlines which are used to store ready jobs in the `ReadyList` and `PendingJobLists`. To ensure that the `ReadyList` is sorted according to the correct deadline, the following requirements must be checked:

- *absolute deadlines are computed according to the ICTOH algorithm.*
- *absolute deadlines are compared correctly according to the ICTOH algorithm.*

Task List Manager: As the scheduler is implemented to select the m jobs at the top of the ready list to execute them, it is necessary to ensure that these jobs are precisely the highest priority ones ie. the `ReadyList` and the `PendingJobLists` must be managed as expected. To this end, the following requirements are considered:

- *when the Scheduler is called, the* `ReadyList` *and all of the* `PendingJobLists` *shall be sorted in an increasing absolute deadline order.*
- *2 nodes of the heap implementing the* `ReadyList` *shall have distinct keys.*

Scheduler: these requirements are used to check whether the final results of the scheduling are consistent with what is expected or not:

- *during the execution the* m *jobs in the* `RUNNING` *state have always a lower absolute deadline than any other job in the* `ReadyList` *or any of the* `PendingJobLists`.
- *a processor shall never be idle while the* `ReadyList` *or the* `PendingJobList` *are not empty.*

Context Switch Manager: to check that the context switching is always performed based on the *Scheduler*'s command. Then, we can simply verify that:

- *the context switching shall be performed according to the Scheduler decisions.*

5 VERIFICATION APPROACH

Our approach consists of checking the implementation correctness during the execution of the OS model over a time interval driven by the *Timer Model*. The complete model (OS + Timer) is stimulated using task activation and termination scenarios generated by what we call *verification engines* (cf. 5.3). The verification of the complete model is carried out by checking the requirements introduced in section 4.2. They are checked using observers that run in parallel with the complete model to observe its behavior. Since our goal is to carry out the verification in a modular way, an observer per OS component is elaborated (cf. 5.1). Then, reachability properties are expressed in CTL (*Computation Temporal Logic*) and checked on the observer states to verify that the system is behaving as required. The properties are all expressed in a identical way, they are stated in Section 5.2. A combination of observer models and the complete system model is formed and inserted as input to the UPPAAL model-checker (VerifyTA) along with expressed properties. The model-checker verifies if the given properties describing the expected behavior hold, or else it generates a counterexample scenario with the corresponding execution trace (cf. Fig. 4).

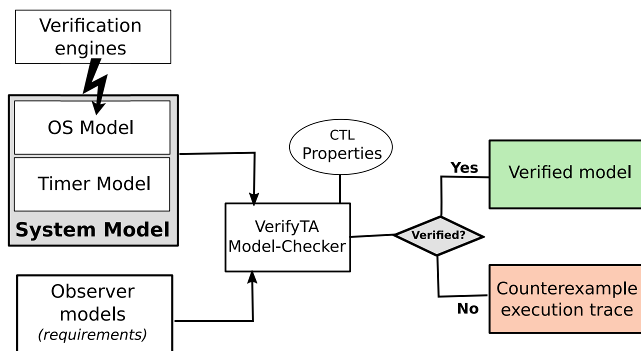

Figure 4: Formal verification process

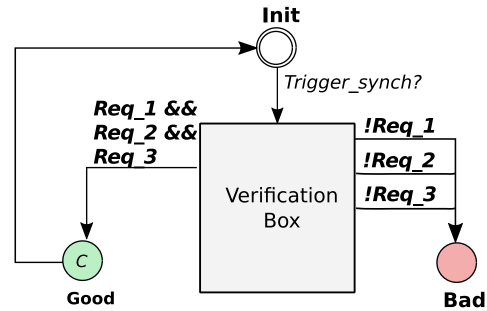

Figure 5: Structure of an observer

5.1 Observer models

The previously presented requirements are complex and some of them depend on the implementation choices. The expression of a CTL formula verifying one of them quickly becomes complicated and requires the nesting of several sub-formulas into a single one with an exponential size. Especially when dealing with a `ReadyList` of *n* tasks and *n* `PendingJobLists` at most. Therefor, we choose to check them using observer models and express properties on their states.

The observer models are EFAs with committed states. These states are priority states, which means that if there are several fireable transitions, the transition related to the committed states is fired before the others. This makes it possible to check the system without the risk of altering its evolution.

Each observer describes a set of requirements used to check an OS component. All the observers inserted in our model have the same structure shown in Fig. 5. The execution of an observer is launched when receiving a synchronization `Trigger_synch?` from the system notifying that the corresponding component ended its execution. The *Verification Box* is the part of the observer that helps checking the desired requirements using a set of test functions. Each requirement is translated to a test function that returns true or false depending on the result of meeting the requirement. When a requirement does not hold, the corresponding output transition leads to a `Bad` state of the system. On the other hand, if all the requirements are satisfied, output transition leads to a `Good` state and the observer goes back to its initial state to wait for the next trigger.

Figure 6 shows the observer used to verify the *Scheduler*. It is called to check if the two intrinsic requirements of G-EDF scheduler hold: the priority and the idleness check.

For the verification of the first requirement concerning the priority, the observer uses the synchronization `end_run_elected[core_id]?` to wait for the function `tpl_run_elected()` to be executed. This synchronization is to ensure that the observer does not permanently check the associated requirements since there are system states that are not significant for it. Note that the `tpl_run_elected()` function changes the state of the task elected by the scheduler to `RUNNING` state and load its context for execution. Thus, at the end of the execution of this function, the observer checks whether the task that is being set for execution has a shorter deadline than any other task in the `ReadyList` or in a `PendingJobList`. This function is

Requirements of the scheduler component	Test function	Formal property
"during the execution the m *jobs in the* RUNNING *state have always a lower absolute deadline than any other job in the* ReadyList *or any of the* PendingJobLists*"*	check_edf_prio()	E <> Scheduler_Observer.Bad
a processor shall never be idle while the ReadyList *or the* PendingJobList *are not empty.*	check_idle_cores()	

Table 1: Scheduler requirements formal specification

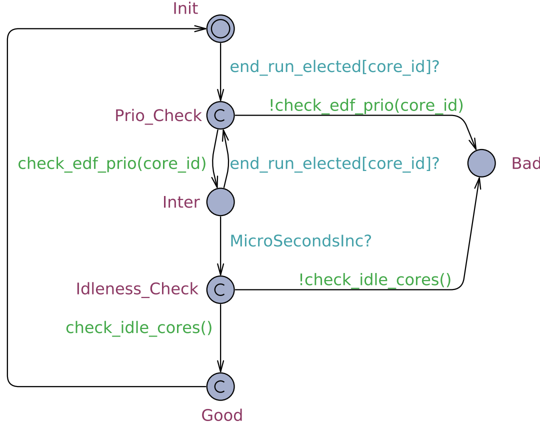

Figure 6: The Scheduler's observer

always executed on the core where the elected task must be running. Therefore, the observer is parameterized by the identifier of the core on which the function is executed. This is specific for a multi-core system verification. It prevents being in the situation of checking the requirement for a core that has not yet finished loading the context of its elected task. Once the observer is synchronized with the `tpl_run_elected()` function, the priority check of the elected task is performed at the `Prio_Check` state. If the test is satisfied, the observer goes to the `Inter` state. It is possible that after the execution of `tpl_run_elected()` another scheduling event occurs before the elected task starts its execution. For this purpose, a transition synchronized with `tpl_run_elected()` is used to return to the `Prio_Check` state in order to do the verification again after the re-scheduling.

The second requirement, concerning the idleness, it is checked when all the elected tasks start their execution. It is performed by using the synchronization channels `end_run_elected[core_id]` and `MicroSecondesInc`. This is performed by synchronizing the observer with the Timer automaton through the `MicroSecondesInc?` channel which indicates the beginning task execution. This way, when it is synchronized, the observer checks its requirement based on a test related to the `Idleness_Check` state.

5.2 Requirements formal specification

Each requirement is formally specified using *reachability* test on the `Good` and `Bad` states of the corresponding observer. This property is defined below.

Definition 5.1 (Reachability). This property checks for a given state s of the model if there exists a path starting at the initial state, such that s is eventually reached along that path.

The property is expressed in CTL language using two types of logic operators:

- *quantifiers over all paths*: like $A(\phi)$ to express that ϕ holds along all execution paths (*inevitably*); $E(\phi)$ for ϕ holds along at least one path (*possibly*).
- *quantifiers over a specific path*: like $G(\phi)$ to say that ϕ holds on the entire subsequent path (*globally*); $F(\phi)$ to say that ϕ eventually holds somewhere on the subsequent path (*finally*). In UPPAAL, these quantifiers are expressed using the syntax `[]` and <> respectively.

For every observer, the logic operators are used to express two types of *reachability* tests: 1) we check if all paths lead finally to the `Good` state (`A <> Observer.Good`), 2) we verify if there exists a path leading to a `Bad` observer state (`E <> Observer.Bad`). If a property is not verified, the error can be detected by following the transition that led to the `Bad` state and a simulation trace generated by UPPAAL model-checker. Table 1 illustrates the formal specification of the scheduler requirements.

5.3 Verification engines

In pursuance of verifying the requirements over the elaborated models, the *Scheduler* must be triggered and forced to operate in different situations and scheduling events. For this purpose, an element which is responsible for generating different scenarios to call and operate our *Scheduler* is built. This element is called the *verification engine*. The idea is to generate for a given number n of tasks, all possible sequences of scheduler calls and task execution over a given Verification Time Interval (VTI). This requires using models in which time can evolve in order to generate events triggering the *Scheduler*. Since UPPAAL offers the possibility of using timed automata, we can use them to model the *verification engine*. Thus, two main parts are considered:

activation engine: given a number n of tasks, this engine allows to generate all possible scenarios of job activation within the interval $[1, VTI]$ which is specified. All possible tasksets that contains at most n tasks are verified and activations can occur randomly at anytime and in any order between 1 and VTI time units. For the sake of simplicity, two timed automata are used to describe this behavior:

- `activate_once`: this template is used to activate at most n jobs only one time between 1 and VTI. Figure 7 shows an example of activation engine that can activate up to 6 tasks ($n = 6$). In the initial state, all tasks are in the `SUSPENDED` state. The time progresses continuously using a clock variable `timer_c`. When the time starts to run, the automaton is synchronized with the timer automaton using the channel `MicroSecondsInc?`. This synchronization is used to prevent the activation of jobs before that the OS starts. Once the

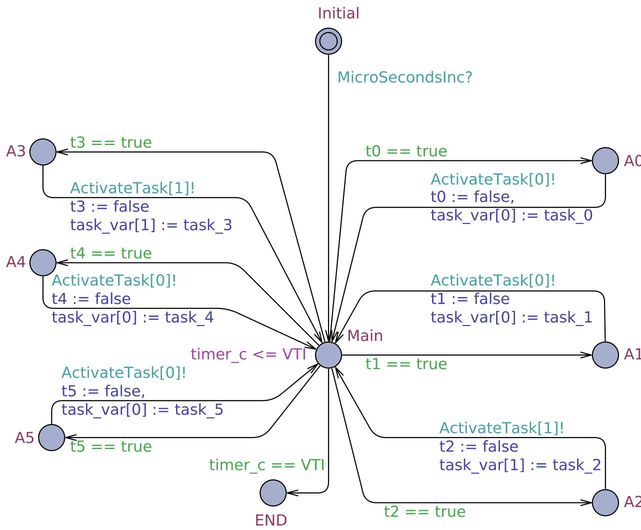

Figure 7: `activate_once` **template**

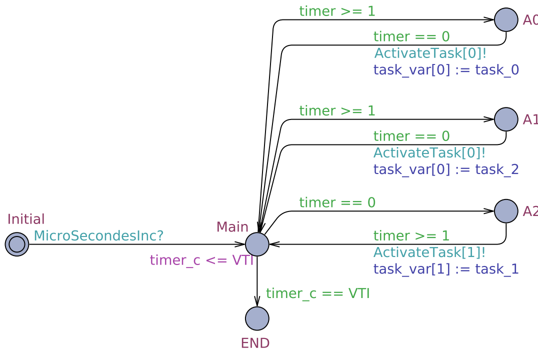

Figure 8: `activate_several_jobs` **template**

time starts flowing, the automaton moves to the `Main` state and can choose or not to take any transition to activate any task at any time as long as the VTI date is not reached. This is carried out using the invariant `timer_c <= VTI` that also allows the automaton to end the scenario when the verification time is expired. We present some possible scenarios that are covered by this automaton:

scenario 1: n tasks where $n <= 6$ (since the automaton can activate at most 6 tasks): $\tau_i / \ i \in \{0, 1, ..., n-1\}$ are activated simultaneously at $t = 1$.

scenario 2: n tasks where $n <= 6$: τ_i are activated respectively at different times $\{t_1, t_2, ..., t_n\}$ such that $\{1 \leq t_1 < t_2 < ... < t_n \leq VTI\}$.

- `activate_several_jobs`: This template allows the activation of several jobs of the same task while previous jobs are not yet terminated. It follows the same principle as the previous template. Activations can occur at anytime and in any order as long as the *VTI* is not reached. This is mainly used to check the properties of the `PendingJobList` since it allows the activation of multiple jobs for the same task even if the oldest ones are not yet terminated.
Figure 8 presents an example an automaton that can activate several jobs for 3 tasks at anytime and in random order. Some possible activating scenarios using this template are presented in the following:
  scenario 1: only jobs of τ_0 are activated in different dates $\{t_1, t_2, ..., t_{max}\}$ such that $\{1 \leq t_1 < t_1 < ... < t_{max} \leq VTI\}$.
  scenario 2: no job is activated which can be a possible scenario as well.

execution engine: it's a task model which is introduced to describe an indeterministic task behavior. Our purpose is not to provide a model that accurately describes how a task runs during its execution, but to develop a model that covers as many cases of execution scenarios as possible. To this end, each task is abstracted by a timed automaton that gathers all the states in which a basic task can be (cf. Fig. 9). In the `Main` state of the automaton, a task can either be `SUSPENDED`, `READY` or `RUNNING`. When it is activated by one of the previous activation engines (Fig. 7, 8), the task is considered `READY` but stays in the `Main` state. When it is selected for execution by the *Scheduler*, the task becomes `RUNNING` and the function `IS_RUNNING()` returns `true`. The execution time of the task is indeterministic, the model handles all the possible cases where a task can be executed between 1 and VTI time units as long as it is in the `Main` state. For termination, the task can finish its execution at any time before the expiration of VTI. To do so, it goes to the `Termination` state using a transition which is guarded by the `IS_RUNNING()` function. This guard is used to avoid calling the `TerminateTask` automaton while the task is not running or preempted. The execution can be performed in a best case scenario in which a task runs for zero time units and the worst case scenario in which it runs for VTI time units. All scenarios in between are treated as well. This widens the scope of verification and allows multiple task configurations to be verified.

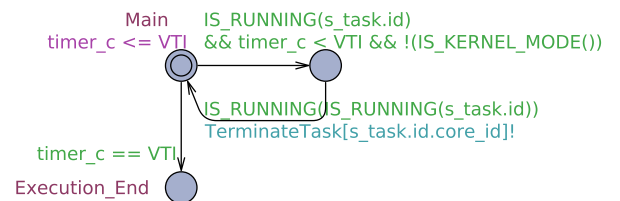

Figure 9: Task model template

5.4 Verification scenarios

Since our goal is to be able to verify the functional correctness of the implementation, task configurations are chosen in such a way that they provide scenarios with the respect to the requirements presented in 4.2. In this part, we present examples of activation and execution scenarios used for verification. We consider that the tasks have to compete for execution on 2 cores. Note that the value of *VTI* chosen for each scenario corresponds to the value

required to conclude that the verified component is behaving correctly with the respect to its requirements. For example, to verify that the scheduler is called when an activation occurs, we just need a task to be activated and check the corresponding property in the component observer in question. Such a task can be activated using the `activate_once` engine and a *VTI* of one time unit is sufficient to observe the activation. The same logic is used for every requirement to chose the values of *VTI* in the following.

(1) for the *Task Manager*: checking its requirements can be sufficient by specifying an application with one task that can run in an interval allowing multiple activations of it using the `activate_several_jobs` template (Fig. 8). Activating several jobs of a task before terminating the old one allows the checking of requirements related to the activation count incrementation/decrementation, the switching between the `ReadyList` and the `PendingJobList` and also if the *Scheduler* is called for every scheduling event (activation/termination). Since the automaton allows job activations that are separated by one time unit minimum, it is necessary to have $VTI > max_activation_count + 1$ in order to consider the case of activating all the jobs of the task.
(2) for the *Time Manager*: the purpose is to calculate and compare successfully deadlines of different tasks. Thus, the *VTI* to verify the requirements of the *Time Manager* must be higher than the maximum value that a deadline can take in a cycle. If an absolute deadline is represented on n bits, the maximum value that it can have is $P = 2^n - 1$. Thus, it is possible to consider a *VTI* of 8 time units for deadlines that are represented on 3 bits in order to exceed the timer's period and cover the case of two deadlines represented on two different cycles. An example of a task configuration can be: 2 tasks to be executed on 2 processors with activations generated by the `activate_once` template. A possible scenario of this task configuration can be the activation of the two tasks at the same time. This necessarily leads to a deadline calculation first and comparison since the 2 tasks are compared before being stored in the `ReadyList`.
(3) for the *Task List Manager*: to check the heap property for the `ReadyList`, 2 tasks are needed. To cover the case of having two jobs linked in the same heap node, we consider that 2 tasks have the same relative deadline. In order to check the `PendingJobLists` we consider that the tasks have a maximum activation count of 3 which covers the case of having one active job per task in the `ReadyList` and 2 pending jobs per task in their `PendingJobLists`.
(4) for the *Scheduler*: considering that the execution is carried out on two cores, at least two running tasks are needed. In order to check the priority requirement, a third task with different deadline must be considered to compare with the other two tasks. The requirement asserts that a running job shall have a lower deadline that any other job in the `ReadyList` or any of the `PendingJobList`. Thus, pending jobs must be considered for the three tasks. In that case, an application composed of 3 tasks with a maximum activation count of two each is proposed.
(5) for the *Context Switch Manager*: the consideration of a single task in a VTI greater than its maximum activation count can be relevant to check the context switching requirement after the activation and the termination of the task.

6 DETECTED SCHEDULING ERRORS

To apply our verification approach, the scenarios described above were tested on the complete OS model. Reachability tests on the observers bad states were carried out to check the requirements introduced in Section 4.2 (`E <> Observer.Bad`). If this property holds, it means that the expected behavior of the corresponding component is not achieved. The counterexample sequence execution is generated along with the sequence variable to indicate the branch that led the system to the `Bad` state. This check allowed us to find out some implementation errors in the scheduling perimeter source code which are mainly the consequence of Trampoline's transition from partitioned to global scheduling. These errors were detected by analyzing the generated counterexample scenarios.

1. In the initial version of Trampoline, in which partitioned scheduling was supported, each task is statically assigned to a single core for its execution. Thus, if a task τ_i is preempted, there is no possibility that it will be selected to be executed on another core. Therefore, in the initial implementation of the Trampoline's scheduler, a preempted task is only put back into the `ReadyList` at the end of context switching.

When moving to global scheduling, Some call sequences of OS functions has been preserved which conducted to the following error. A task τ_i which is preempted on a core C_i, might have a higher priority than a task τ_j running on a core C_j. However, if τ_i is only put back in the `ReadyList` at the context switching level (after the scheduling), the *Scheduler* can not take it into account while taking its decision. As a result, the running task τ_j will have a lower priority than the ready task τ_i.

Counterexample scenario. The aforementioned error was detected under the scenario intended to verify the *Scheduler*'s requirements (cf. 5.4): 3 tasks: τ_0, τ_1, and τ_2 to be executed on 2 cores that are denoted C_0 and C_1. Each task τ_i has relative deadline denoted by D_i. The activation is performed using the `activate_several_jobs` template since each task τ_i have a maximum activation count equal to 2. The considered *VTI* is 7 time units. The counterexample scenario that was generated by UPPAAL model-checker is described below:

- at t_1, J_1^1 and J_2^1 are released. Their absolute deadlines are calculated: $d_1^1 = D_1 + t_1$ and $d_2^1 = D_2 + t_1$ such that $d_1^1 < d_2^1$. Thus, J_1^1 is scheduled to be executed on core C_0 and J_2^1 on core C_1.
- at $t_2 > t_1$, J_0^1 is released, its absolute deadline is calculated: $d_0^1 = D_0 + t_2$ such that $d_0^1 < d_1^1 < d_2^1$. J_0^1 is put in the `ReadyList` and the *Scheduler* is called. Since $d_0^1 < d_1^1$, the *Scheduler* preempts J_1^1 and extracts J_0^1 from the `ReadyList` to be executed on core C_0.
- after the context switching, J_1^1 is put back in the `ReadyList`.

This scenario violates the requirement of the priority according to G-EDF policy. At t_2 in the execution, job J_1^1 is stored in the

Component	Scenario	Number of explored states	Runtime (ms)	Memory (Kibytes)
Task Manager	- one task - max activation count = 2 - VTI = 3	10 965	1 640	63 780
Time Manager	- 2 tasks - max activation count = 1 - VTI = 8	178 691 012	3.0733e+07	33 650 960
Task List Manager	- 2 tasks - max activation count = 3 - VTI = 7	13 451 183	2.42331e+06	1 537 936
Scheduler	- 3 tasks - max activation count = 2 - VTI = 7	1 362 177 190	1.78361e+08	157 922 424
Context Switch Manager	- one task - max activation count = 1 - VTI = 2	6 285	900	63 368

Table 2: Verification performances

ReadyList while J_i^2 which has a lower priority is running on core C_1.

2. As mentioned before, in the partitioned scheduling version of Trampoline, a task is statically assigned to a core. The identifier of this core is initialized in an API layer function that does not belong to the scheduling perimeter. In some specific cases, such as the case of AUTOSTART tasks [2], a macro of the *Context Switch Manager* is called through the API aforementioned function taking the wrong identifier of the core and ignoring the global scheduling decision. This leads to two possible situations where the G-EDF decision is not respected: (i) the first one involves executing a task on a core other than the one indicated by the G-EDF scheduler; (ii) The second, which is the most serious, is to execute a non-priority task according to EDF on the core saved by the API function.

Counterexample scenario. This error was detected for the previously presented scenario of the *Scheduler* verification, but also under a scenario intended to verify the *Context Switch Manager*.

3. Other minor errors were also detected using our scenarios: (i) saving the context of a terminating task by calling the wrong *Context Switch Manager* function; (ii) request loading the context of a newly activated task which is a consequence of the previous error ; (iii) not updating the size of the PendingJobList after removing a pending job.

This approach allowed us to fix the found :errors in the implemented scheduler and ensure its correctness. Due to verification engines, several non-deterministic scenarios of task activation and termination were tested in the model. Throughout the development of this experiment some limitations were encountered. One notable issue is the exponential growth of state space during the execution which limits the interval VTI during which the verification can be conducted. Still, thanks to the scenarios discussed in Section 5.4, the requirements were successfully verified. Table 2 shows the verification performances of the approach using a machine of 128 Mbytes of memory and 282428 MHz of CPU.

[2]Tasks that should start automatically and synchronously with the OS start.

Note that in addition to the requirements presented in Section 4.2, the complete model was also previously checked using the property "AG not deadlock" in order to conduct our approach on a deadlock free system. This property means that deadlock does not occur for any path of the model, globally.

7 CONCLUSION

We proposed a formal method to verify the implementation of scheduling policies based on model-checking. The approach was tested on an implementation of G-EDF scheduler within Trampoline. To do so, a first step of modeling Trampoline's components that are involved in the scheduling decision was conducted. Models abstracting the functioning of these components were elaborated using Extended Finite Automata and integrated to an existing model of the OS. The variables, the actions and the conditions manipulated in the models are those present in the source code of Trampoline's components. Thus, these models are very close to the actual source code due to the similarity between the semantics of C language and UPPAAL language which makes the verification on those proposed models reliable. The elaborated models were combined with a timer models that abstracts the time progress in the OS to form a complete model (OS model + timer). The second step was to establish requirements that are sufficient to verify on the elaborated models. These requirements were defined according to the expected behavior of the scheduler. They are used in observers that are inserted in parallel with the complete model to observe its behavior during the execution. The execution of the model was stimulated using verification engines which provide non-deterministic scenarios of activation and termination of jobs. Therefore, the scenarios that are able to trigger the scheduler in accordance with the requirement to be checked were proposed. This process allowed us to identify some implementation errors and fix them. So far, the proposed approach showed promising results for the given scenarios. Some constraints were noted, namely the combinatorial explosion of the state space beyond a certain verification time since most formal methods have this property. In order to remedy this problem, our model is abstracted as much as possible. The model has only 2 clock variables for time progression, the path interleaving is avoided

using synchronization channels in all automata and observers contain only *Committed* states. A next step of this work will be to demonstrate that the conducted tests are sufficient to validate the functional correctness of the G-EDF implementation. Furthermore, other techniques for model abstraction to limit the explosion of the state space may also be studied.

This work proposed an analysis of feasibility of this formal verification approach for schedulers implementation. Since it shows promising results, model-checking can be engineered and applied for the implementation of other sophisticated scheduling policies. Accordingly, we intend to pursue this work and extend it to some of these policies.

REFERENCES

[1] Karine Altisen, Gregor Gößler, and Joseph Sifakis. 2002. Scheduler modeling based on the controller synthesis paradigm. *Real-Time Systems* 23, 1-2 (2002), 55–84.

[2] Tamarah Arons, Elad Elster, Limor Fix, Sela Mador-Haim, Michael Mishaeli, Jonathan Shalev, Eli Singerman, Andreas Tiemeyer, Moshe Y Vardi, and Lenore D Zuck. 2005. Formal verification of backward compatibility of microcode. In *International Conference on Computer Aided Verification*. Springer, 185–198.

[3] Theodore P Baker. 2010. What to make of multicore processors for reliable real-time systems?. In *International Conference on Reliable Software Technologies*. Springer, 1–18.

[4] Luciano Porto Barreto and Gilles Muller. 2001. *Bossa: a DSL framework for application-specific scheduling policies.* Ph.D. Dissertation. INRIA.

[5] Jean-Luc Béchennec, Mikael Briday, Sébastien Faucou, and Yvon Trinquet. 2006. Trampoline an open source implementation of the OSEK/VDX RTOS specification. In *Emerging Technologies and Factory Automation, 2006. ETFA'06. IEEE Conference on.* IEEE, 62–69.

[6] Jean-Luc Béchennec, Olivier H. Roux, and Toussaint Tigori. 2018. Formal Model-Based Conformance Verification of an OSEK/VDX Compliant RTOS. In *CODIt 2018-5th International Conference on Control, Decision and Information Technologies.*

[7] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. 1996. UPPAAL—a tool suite for automatic verification of real-time systems. *Hybrid Systems III* (1996), 232–243.

[8] Jean-Paul Bodeveix, Mamoun Filali, Julia L Lawall, and Gilles Muller. 2007. Automatic verification of bossa scheduler properties. *Electronic Notes in Theoretical Computer Science* 185 (2007), 17–32.

[9] Khaoula Boukir, Jean-Luc Béchennec, and Anne-Marie Déplanche. 2017. Reducing the gap between theory and practice: towards A Proven Implementation of Global EDF in Trampoline. *JRWRTC 2017* (2017), 9.

[10] Khaoula Boukir, Jean-Luc Béchennec, and Anne-Marie Déplanche. 2018. Formal approach for a verified implementation of Global EDF in Trampoline. In *Proceedings of the 26th International Conference on Real-Time Networks and Systems.* ACM, 83–92.

[11] John M Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C Devi, and James H Anderson. 2006. LITMUS^{RT}: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International.* IEEE, 111–126.

[12] Alessio Carlini and Giorgio C Buttazzo. 2003. An efficient time representation for real-time embedded systems. In *Proceedings of the 2003 ACM symposium on Applied computing.* ACM, 705–712.

[13] Felipe Cerqueira, Felix Stutz, and Björn B Brandenburg. 2016. PROSA: A case for readable mechanized schedulability analysis. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS).* IEEE, 273–284.

[14] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Benjamin Werner, and Christine Paulin-Mohring. 1991. *The COQ proof assistant user's guide: version 5.6.* Ph.D. Dissertation. INRIA.

[15] Dario Faggioli, Michael Trimarchi, Fabio Checconi, Marko Bertogna, and Antonio Mancina. 2009. An implementation of the earliest deadline first algorithm in Linux. In *Proceedings of the 2009 ACM symposium on Applied Computing.* ACM, 1984–1989.

[16] Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. 2006. Schedulability analysis of fixed-priority systems using timed automata. *Theoretical Computer Science* 354, 2 (2006), 301–317.

[17] Limor Fix. 2008. Fifteen years of formal property verification in Intel. In *25 Years of Model Checking.* Springer, 139–144.

[18] Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao, and David Costanzo. 2011. CertiKOS: a certified kernel for secure cloud computing. In *Proceedings of the Second Asia-Pacific Workshop on Systems.* ACM, 3.

[19] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles.* ACM, 207–220.

[20] Peng Li, Binoy Ravindran, Syed Suhaib, and Shahrooz Feizabadi. 2004. A formally verified application-level framework for real-time scheduling on posix real-time operating systems. *IEEE Transactions on Software Engineering* 30, 9 (2004), 613–629.

[21] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: a proof assistant for higher-order logic.* Vol. 2283. Springer Science & Business Media.

[22] Evidence Srl. [n. d.]. Erika enterprise rtos. *URL: http://www.evidence.eu.com* ([n. d.]).

[23] Kabland Toussaint Gautier Tigori, Jean-Luc Béchennec, Sébastien Faucou, and Olivier Henri Roux. 2017. Formal Model-Based Synthesis of Application-Specific Static RTOS. *ACM Transactions on Embedded Computing Systems (TECS)* 16, 4 (2017), 97.

[24] Libor Waszniowski, Jan Krákora, and Zdeněk Hanzálek. 2009. Case study on distributed and fault tolerant system modeling based on timed automata. *Journal of Systems and Software* 82, 10 (2009), 1678–1694.