Formal schedulability analysis based on multi-core RTOS model

Imane Haur, Jean-Luc Béchennec, Olivier Henri Roux

Formal schedulability analysis based on multi-core RTOS model

Imane HAUR
Imane.haur@ec-nantes.fr
Paris Research Lab, Huawei
École Centrale de Nantes, LS2N
Boulogne Billancourt, Nantes, France

Jean-Luc BÉCHENNEC
Jean-Luc.Bechennec@ls2n.fr
CNRS, LS2N
Nantes, France

Olivier Henri ROUX
Olivier-h.roux@ec-nantes.fr
École Centrale de Nantes, LS2N
Nantes, France

ABSTRACT

Verification of real-time application schedulability is usually performed using a very abstract representation of the system which poorly supports inter-task dependencies. This paper presents the use of model-checking techniques to check the schedulability on a detailed model of a multi-core operating system. The operating system as a whole is modeled by a High-level Petri net reproducing the control flow and using the same variables as those of the implementation. Each task of the application is represented by a Stopwatch Petri Net whose transitions carry Best-Case Execution Time and Worst-Case Execution Time $[BCET, WCET]$ firing intervals and make service calls to the OS. Preemption is supported by means of stopwatches. Verification is performed using observers and allows to determine the schedulability of the multi-core application, or, using parameters on the firing intervals, allows determining under which temporal conditions the application is schedulable.

CCS CONCEPTS

• Software and its engineering → Formal software verification; **• Computer systems organization → Real-time operating systems**; **• Theory of computation → Verification by model checking**.

KEYWORDS

High Level Stopwatch Petri Nets, RTOS, Scheduling, Model-checking

ACM Reference Format:
Imane HAUR, Jean-Luc BÉCHENNEC, and Olivier Henri ROUX. 2021. Formal schedulability analysis based on multi-core RTOS model. In *29th International Conference on Real-Time Networks and Systems (RTNS'2021), April 7–9, 2021, NANTES, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3453417.3453437

1 INTRODUCTION

Embedded systems have become ubiquitous in our daily lives in several areas. Due to their evolution, these systems have to use increasingly complex hardware architectures to achieve the required performance. Besides, multi-core chips have become more popular as applications require high-performance computing (HPC). Embedded systems must implement all the desired functionalities, execute the assigned tasks, and validate not only functional but also temporal accuracy. In [28], a definition of real-time systems (RTS) is given considering the computations' correctness plus the time-limited response. It is, therefore, necessary to perform a system verification in order to increase the level of confidence and prevent unexpected behavior.

The increase in the software-based functions has led to the adoption of Real-Time Operating Systems (RTOS), which provides an interface between hardware and software. The RTOS is a platform that implements all the required functionalities and is responsible for the management of hardware resources and the scheduling of application processes. This evolution is formalized; for example, ARINC 653 [3] and AUTOSAR [4] standards propose implementation rules for real-time operating systems intended for the avionics and automotive industries.

The structure of the RTS is thus composed of three parts: software, hardware, and a real-time operating system (RTOS). The software architecture of RTS consists of a set of tasks that interact, usually through message exchange or synchronization mechanisms. The hardware part of the RTS is composed of processors, memories, input/output devices and other components. The hardware architecture is often classified according to the number of processors and cores. The RTOS manages the hardware resources and provides a scheduler, responsible for the order of execution of tasks on the processors in order to guarantee time determinism. In the following, we present some related work on the formal verification of operating systems and real-time systems, the use of High-level Petri nets in modeling, our contribution and the paper outline.

1.1 Formal methods for operating systems verification

A variety of formal approaches exist and have been used in several research studies, especially for the verification of operating systems.

In [18] Huang et al. present work on the verification of compliance of an operating system with automotive standards using different formal approaches. An OSEK/VDX compliant operating system is modeled at code-level using CSP and then verified in the model-checker PAT. The verification does not contain all the OSEK/VDX specifications; they proved the specifications concerning only task scheduling and resource management. In addition, the absence of deadlock is guaranteed in the operating system code.

[17, 20] present verification of operating systems with theorem prover. [17] is a verification project of the L4 compatible Fiasco microkernel. The verification of the C++ sources of Fiasco is performed

RTNS'2021, April 7–9, 2021, NANTES, France

ACM ISBN 978-1-4503-9001-9/21/04…$15.00
https://doi.org/10.1145/3453417.3453437

by the general-purpose theorem prover PVS. Their approach handles type-correctness and safety proof. In [20], the functional correction of the seL4 microkernel is proved using the theorem prover Isabelle/HOL. A complete verification process is performed independently of the application, from the high-level specification of the kernel behavior to its safe execution. Their proof, however, is limited to the validation of assumptions about the proper functioning of the hardware and compiler. It does not cover application-specific properties or the interaction of applications with the operating system.

[13] shows an approach that converts the kernel source code of the RTOS Trampoline to a formal model in PROMELA, the modeling language of SPIN. The objective of this work is to verify the safety properties and the exactness of the kernel model. Using model-checking, they were able to identify some possible safety violation scenarios.

[30] proposes a complete model of the RTOS Trampoline in its mono-core version using extended and timed automata with UPPAAL tool. This model includes all the functions and services of the OS. By performing reachability analysis on the states on the OS according to an application model, infeasible paths are eliminated, and the model is pruned accordingly. From the pruned model, a source code configured for the application can be produced. [7] proposes to check the conformity of the model, whether it is pruned or not, by means of observers using the properties that need to be checked to show conformity to the OSEK/VDX standard.

1.2 Real-time systems Verification

The time verification of Real-time systems consists in proving that the system will always be able to react according to its time constraints. Timing validation is, therefore, a decision process that concerns task scheduling sequences. It can be done using simulation, model checking, and feasibility tests such as processor utilization and response time analyses [26]. However, exhaustive testing is not possible in real-time context.This motivates the use of formal verification methods using such models as timed automata (TA) [1] and timed Petri nets (TPN) [9], among others. There are thus many scheduling studies based on a representation by these formal models [8, 11, 15, 31].

Among them, [31] focuses on modeling multi-tasking applications to verify the worst time execution of the tasks using timed automata. The modelled applications considered non-preemptive tasks and interrupt service routines.G. Behrmann et al. propose in [8] a timed automata model called Priced Timed Automata (PTA). Its semantics is defined by associating to each transition and location a non-negative real-valued cost. Their analysis consists in seeking optimal offline scheduling with minimal cost.

Based on the Trampoline formal model of [30], Boukir K. et al. [11] adapt it for global scheduling. They integrate the model of the G-EDF implementation using the same formalism of extended and timed automata. The conformity of the scheduler implementation is then checked for a set of properties using synthetic application models. These models generate all possible scheduler excitation scenarios. However, the halting of the elapse of time in case of preemption is not modeled.

Indeed, for timed automata (TA) [1] and timed Petri nets (TPN), time elapses at the same speed for all components of the system. Hence they cannot abstract preemptive scheduling policies where the execution of a task can be suspended at some point and later resumed at the same point. Several extensions of these models have been proposed to express the suspension and resumption of actions by adding the stopwatch notion [12, 21]. These models belong to the class of TPN extended with stopwatches (SwPN) [10].

1.3 High-level Petri nets

For the lack of a data structure, Petri nets are not suitable for modeling systems where data have an effect on the system's behavior. High-level Petri nets [16] have been proposed for modeling scientific problems with complex structures allowing to describe both system data and control. The term High-level Petri net is then used for many Petri nets [19] such as Predicate/Transition Nets, colored Petri nets, or hierarchical Petri nets. However, the common point is that they allow to manipulate different types of expressions that use state variables. Input arcs are labeled with boolean expressions specifying conditions (guards or gates) that can also be associated with transitions. Arc annotations are expressions that can be associated with output arc. They can be viewed as computing systems that operate on shared data. Parametric High-level Petri nets is a formalism for modeling and verifying preemptive real-time systems with parameters. It benefits from the Roméomodel-checker. There is also the formalism of the parametric stopwatch automatons available in the IMITATOR model-checker. In [24], Roméo and IMITATOR provide the same conclusions on an aerial video tracking system by THALES, and Roméo had performed better than IMITATOR in terms of time and memory consumption. In [29], IMITATOR could output the exact answer to an industrial challenge by Thales with uncertain periods. A test of the practical efficiency on the IMITATOR benchmarks library is given in [2].

1.4 Contribution and outline

Our contribution consists of an approach that aims to verify the schedulability of a real-time system using model checking. We use High-Level Petri Nets with stopwatches for modeling both the multi-core RTOS and the real-time application. The RTOS is an OSEK and AUTOSAR compliant RTOS called Trampoline [6]. OSEK-OS provides a preemptive, non-preemptive and task group scheduling; a task group is a set of task that are non preemptable by each other but can be preemptable by higher priority tasks in the application. We then rely on model checking to check schedulability and provide an accurate analysis of worst-case response time computation for dependent preemptive tasks in multi-core systems. The analytical approach applied to the fixed priority algorithm is most often based on response time computation. However, these response time analysis methods considered an unrealistic critical time when tasks are dependent [23], leading to an inherent pessimism. When considering dependent tasks, it is then necessary to take the BCET and the WCET and possible interactions between the tasks. Moreover we parametrize task timing characteristics in order to synthesize the values of these characteristics that guarantee the system's schedulability. The whole process is integrated into the Roméo tool, available under a free license [25].

The rest of this paper is organized as follows. Section 2 presents the background and the work positioning. Section 3 gives the definition of High-Level Stopwatch Petri Nets that are used in Section 4 for the formal model of the RTOS and the multi-core application. Section 5 presents our approach for conducting verification of the schedulability or by using parameters to determine under which temporal conditions the application is schedulable. We apply the approach to a case study in Section 6. Finally, Section 7 concludes this paper.

2 BACKGROUND

The operating system chosen here is the RTOS Trampoline. Trampoline is a preemptive real-time operating system designed for the automotive industry, which is compliant with OSEK/VDX and AUTOSAR standards. Therefore the multi-core version of Trampoline implements fixed priority partitioned scheduling. This type of scheduling consists of statically allocating a set of tasks to a processor. This subset assigned to a particular processor is not allowed to migrate to another; it is scheduled according to mono-processor scheduling.

In its OSEK/VDX version, this type of operating system manages several types of objects: tasks, interrupt service routines (ISR), resources, that are used to implement critical sections using the IPCP protocol a variant of PCP [27], or alarms, that are used to implement periodic tasks, to name only the main ones.

In the AUTOSAR version, other objects are added such as schedule tables or OS Applications. Partitioning of tasks on the cores is done using OS Applications. An OS Application gathers several objects (tasks, ISR, alarms, resources, ...) in a logical entity and is statically assigned to a core. In addition to resources that do not work for the implementation of critical sections between code running on different cores, an AUTOSAR OS also has spinlocks.

An application running on this kind of OS is statically configured. That is to say that all the application objects handled by the OS are known at compile time and that there is no dynamic allocation of objects. Furthermore, to control the memory allocated to the stack of a task, recursion is discouraged. These features facilitate the formal modeling of such a system. The static configuration of an application is carried out by means of a description in a dedicated language. For OSEK/VDX the OIL language is used. For AUTOSAR, it has been replaced by arXML. A compiler transforms this description into C data structures and code.

Trampoline has retained, in addition to arXML, the possibility to describe the application in OIL in order to preserve readability. Since the Trampoline arXML/OIL compiler is easily extensible, the data structures manipulated by the model are also generated. This limits the risk of modeling errors.

The source code is just over 20,000 lines long and includes 180 functions for the target-independent part.

Concurrent access to data structures internal to the operating system is prevented by the use of a spinlock for most service calls and for interrupt routines built into the OS. The exception is the OS startup service, which can run in parallel on multiple cores but only changes the data structures specific to the core on which it is running.

3 HIGH LEVEL STOPWATCH PETRI NETS

Notations. The sets $\mathbb{N}$, $\mathbb{Q}_{\geq 0}$ and $\mathbb{R}_{\geq 0}$ are respectively the sets of natural, non-negative rational and non-negative real numbers. An interval I of $\mathbb{R}_{\geq 0}$ is a $\mathbb{Q}$-interval iff its left endpoint $^{\uparrow}I$ belongs to $\mathbb{Q}_{\geq 0}$ and its right endpoint $I^{\downarrow}$ belongs to $\mathbb{Q}_{\geq 0} \cup \{\infty\}$. We denote by $\mathcal{I}(\mathbb{Q}_{\geq 0})$ the set of $\mathbb{Q}$-intervals of $\mathbb{R}_{\geq 0}$.

B^A stands for the set of mappings from A to B. If A is finite and $|A| = n$, an element of B^A is also a vector in B^n. The usual operators $+, -, <$ and $=$ are used on vectors of A^n with $A = \mathbb{N}, \mathbb{Q}, \mathbb{R}$ and are the point-wise extensions of their counterparts in A.

3.1 Informal presentation

High-level Petri nets. A Petri net, also known as a place/transition (PT) net, is one of several mathematical modeling languages for the description of distributed concurrent systems. A place can contain any number of tokens. A marking M of a Petri Net is a vector representing the number of tokens of each place. A transition is enabled (it may fire) in M if there are enough tokens in its input places for the consumptions to be possible. Firing a transition t in a marking M consumes one token from each of its input places s, and produces one token in each of its output places s.

High-level Petri nets have been proposed for modeling scientific problems with complex structures and manipulate different types of expressions made up by variables and written in terms of a predefined syntax.

In this paper we consider that precondition (guard) and postcondition (update) over a set of variables (X) are associated with transitions. A transition is enabled (it may fire) if there are enough tokens in its input places and if the guard is true. When the transition fires the corresponding updates are executed modifying the values of the variables. The variables take their values in a finite state (such as bounded integer or enumerate type...), guards are boolean expressions over X and updates can be described as a sequence of imperative code expressed in a programming language but whose execution is atomic from the transition firing point of view.

Stopwatch Petri Nets. Time Petri nets (TPN) extend Petri nets with temporal intervals associated with transitions, specifying firing delay ranges for the transitions. Assuming transition t became last enabled at time d and the end points of its firing interval are α and β, then t cannot fire earlier than $d + \alpha$ and must fire no later than $d + \beta$ unless disabled by firing of another transition. Firing a transition takes no time.

Stopwatch Petri nets (SwPN), extend TPN by adding the notion of stopwatch: a stopwatch is associated with each transition. The time derivative of the stopwatch of a transition is in the set of rate $\{0, 1\}$ and is given by a function from Markings.

Hence the time associated with a transition can be suspended and later resumed at the same point. Moreover, transition with a 0 time derivative can not fired.

3.2 Definition and semantics of High Level Stopwatch Petri Nets

The semantics of High Level Stopwatch Petri Nets is formally defined in RTNS The semantics of Colored Time Petri Nets is formally defined in ICESS

We consider a Petri Nets model which encompasses both stopwatches and High-level and colored functionalities. We now give the informal definition by example.

Definition 3.1 (High level Stopwatch Petri Net (HSwPN)). A *High level Stopwatch Petri Net* (HSwPN) is a tuple $\mathcal{N} = (P, T, {}^\bullet., .^\bullet, m_0, guard, update, I, \dot{v})$ where

- P is a finite non-empty set of *places*,
- T is a finite set of *transitions* such that $T \cap P = \emptyset$,
- X is a finite set of *variables* taking their value in the finite set $\mathbb{X}$ (such as bounded integer),
- ${}^\bullet. : T \to \mathbb{N}^P$ is the backward incidence mapping,
- $.^\bullet : T \to \mathbb{N}^P$ is the forward incidence mapping,
- $guard : T \times \mathbb{X}^X \to \{true, false\}$ is the guard function,
- $update : T \times \mathbb{X}^X \to \mathbb{X}^X$ is the update function,
- $(m_0, x_0) \in \mathbb{N}^P \times \mathbb{X}^X \to$ is the initial values m_0 of the marking and x_0 of the variables,
- $I : T \to \mathcal{I}(\mathbb{Q}_{\geq 0})$ is the *static firing interval* function,
- $\dot{v} : T \times \mathbb{N}^P \times \mathbb{X}^X \to \{0, 1\}$ is the *time derivative* function

Definition 3.2 (High level Colored Time Petri Net). A *High level Colored Time Petri Net* (HCTPN) is a tuple $\mathcal{N} = (P, T, X, C, , , (m_0, x_0), guard, update, I)$ where

- P is a finite non-empty set of *places*,
- T is a finite set of *transitions* such that $T \cap P = \emptyset$,
- X is a finite set of *variables* taking their value in the finite set $\mathbb{X}$ (such as bounded integer),
- C is a finite set of *colors* and $C_{any} = C \cup \{any\}$,
- $: P \times T \to \mathbb{N}^{C_{any}}$ is the backward incidence mapping,
- $: P \times T \to \mathbb{N}^{C_{any}}$ is the forward incidence mapping,
- $guard : T \times \mathbb{X}^X \to \{true, false\}$ is the guard function,
- $update : T \times \mathbb{X}^X \to \mathbb{X}^X$ is the update function,
- $(m_0, x_0) \in \mathbb{N}^P \times \mathbb{X}^X \to$ is the initial values m_0 of the marking and x_0 of the variables,
- $I : T \to \mathcal{I}(\mathbb{Q}_{\geq 0})$ is the *static firing interval* function,

Discrete behaviour. For a marking $m \in \mathbb{N}^P$, $m(p)$ represents a number of *tokens* in place p. A valuation of the set of variables X is noted $x \in \mathbb{X}^X$. (m, x) is a discrete state of HSwPN.

A transition $t \in T$ is said to be *enabled* by a given marking $m \in \mathbb{N}^P$ and a valuation $x \in \mathbb{X}^X$ if $m \geq {}^\bullet t$ and $guard(t, x) = true$. We denote by $\text{en}(m, x)$ the set of transitions that are enabled by (m, x): $\text{en}(m, x) = \{t \in T \mid m \geq {}^\bullet t \text{ and } guard(t, x) = true\}$.

Firing an enabled transition t from (m, x) leads to a new marking $m' = m - {}^\bullet t + t^\bullet$ and a new valuation $x' = update(t, x)$. We denote by $\text{newen}((m, x), t)$ the set of transitions that are newly enabled by the firing of t from (m, t): $\text{newen}((m, x), t) = \{t' \in \text{en}(m - {}^\bullet t + t^\bullet, update(t, x)) \mid t' \notin \text{en}(m - {}^\bullet t, x) \text{ or } t = t'\}$

Time behaviour. For any $t \in T$, $v(t)$ is the valuation of the stopwatch associated with t. i.e. it is the time elapsed since the transition t has been newly enabled while the rate of t is equal to 1. $\bar{0}$ is the initial valuation with $\forall t \in T, \bar{0}(t) = 0$.

```
typedef enum {task1,task2} id;
int cpt = 0;

int isRunning(id task)  {
   if (task==task1) {
       if ((m(Ready2)==1) && (cpt>2)) return 0; else return 1;
   } else if (task==task2) {
       if ((m(Ready1)==1) && (cpt<3)) return 0; else return 1;
   }
}
```

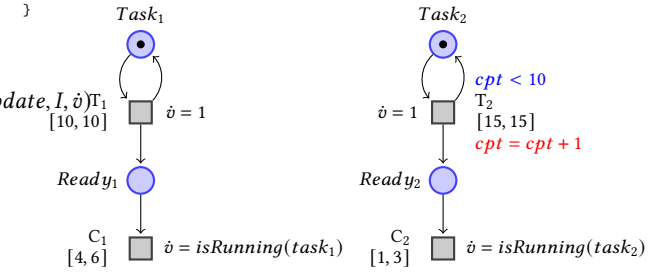

Figure 1: HSwPN model of two tasks scheduling with a counter

For a discrete state (m, x), $\dot{v}(t, (m, x) \in \{0, 1\}$ is the time derivative function of the stopwatch associated with t for this discrete state.

A *state* of the net $\mathcal{N}$ is a tuple $((m, x), v)$ in $\mathbb{N}^P \times \mathbb{X}^X \times {\mathbb{R}_{\geq 0}}^T$, where m is a marking, x is a variable valuation and v is a valuation of the stopwatches.

3.3 Example of HSwPN

This example is the modeling of the preemptive scheduling of two tasks. The first task $task_1$ is a periodic task. The second task $task_2$ is also periodic but is executed only 10 times. For the first two executions of the task $task_2$, the priority of $task_1$ is higher than that of $task_2$ after which it becomes the opposite.

The model given in Figure 1 is a HSwPN with a single variable cpt. The initial value of cpt is zero.

Only the transition T_2 has a guard and an update that manipulate the variable cpt. Hence the transition T_2 is enabled if there is a token in its input place $task_2$ and if $cpt < 10$ modeling the fact that the task $task_2$ is executed only 10 times. The update that increment the value of cpt is executed each time the transition T_2 is fired.

The scheduling is captured by the derivative function of the stopwatches associated with C_1 and C_2 whose values are given by a function called *isRunning* shown in Figure 1.

In the sequel a marking is written by the vector $m = (task_1, task_2, Ready_1, Ready_2)$. The initial marking $(1, 1, 0, 0)$ enables the transitions T_1 and T_2. The valuations of the stopwatches are given by the vector $v = (T_1, T_2, C_1, C_1)$. The initial valuation is $(0, 0, 0, 0)$.

We note a state $\begin{bmatrix} m \\ cpt \\ v \end{bmatrix}$ and the initial state is $q_0 = \begin{bmatrix} (1,1,0,0) \\ 0 \\ (0,0,0,0) \end{bmatrix}$

Assume that the execution times of the two tasks $task_1$ and $task_2$ are respectively 5.3 and 2.4. It means that the transitions C_1 and C_2 fire when their stopwatches reach these values. Let us develop the corresponding run:

$$q_0 = \begin{bmatrix} (1,1,0,0) \\ 0 \\ (0,0,0,0) \end{bmatrix} \xrightarrow{10} \begin{bmatrix} (1,1,0,0) \\ 0 \\ (10,10,0,0) \end{bmatrix} \xrightarrow{T_1} q_1 = \begin{bmatrix} (1,1,1,0) \\ 0 \\ (0,10,0,0) \end{bmatrix}$$

In q_1, we have $\dot{v}(C_1) = 1$ then

$q_1 \xrightarrow{5} q_2 = \begin{bmatrix} (1,1,1,0) \\ 0 \\ (5,15,5,0) \end{bmatrix} \xrightarrow{T_2} q_3 = \begin{bmatrix} (1,1,1,1) \\ 1 \\ (5,0,5,0) \end{bmatrix}$

In q_3, we have $\dot{v}(C_1) = 0$ and $\dot{v}(C_2) = 1$ meaning that the task 1 is preempted by the task 2. Then $v(C_1)$ will keep its value 5 until the firing of C_2 that will change $\dot{v}(C_1)$.

$q_3 \xrightarrow{2.4} q_4 = \begin{bmatrix} (1,1,1,1) \\ 1 \\ (7.4,2.4,5,2.4) \end{bmatrix} \xrightarrow{C_2} q_5 = \begin{bmatrix} (1,1,1,0) \\ 1 \\ (7.4,2.4,5,0) \end{bmatrix}$

In q_5, we have $\dot{v}(C_1) = 1$ hence

$q_5 \xrightarrow{0.3} q_6 = \begin{bmatrix} (1,1,1,0) \\ 1 \\ (7.7,2.7,5.3,0) \end{bmatrix} \xrightarrow{C_1} q_7 = \begin{bmatrix} (1,1,0,0) \\ 1 \\ (7.7,2.7,0,0) \end{bmatrix}$

For the sake of conciseness, we do not detail the following run from q_7

$q_7 \xrightarrow{2.3} \xrightarrow{T_1} \xrightarrow{4} \xrightarrow{C_1} \xrightarrow{6} \xrightarrow{T_1} \xrightarrow{T_2} \xrightarrow{1} \xrightarrow{C_2} \xrightarrow{6} \xrightarrow{C_1} \xrightarrow{3} \xrightarrow{T_1} \xrightarrow{5} \xrightarrow{T_2} q_{13}$

It leads to a state q_{13} that have exactly the same marking and the same value of stopwatches than q_3 but with $cpt = 3$.

$q_{13} = \begin{bmatrix} (1,1,1,1) \\ 3 \\ (5,0,5,0) \end{bmatrix}$ then we have $\dot{v}(C_1) = 1$ and $\dot{v}(C_2) = 0$ meaning that the task $task_1$ is not preempted by the task $task_2$. Hence we have : $q_{13} \xrightarrow{0.3} q_{14} = \begin{bmatrix} (1,1,1,1) \\ 3 \\ (5.3,0.3,5.3,0) \end{bmatrix} \xrightarrow{C_1} q_{15} = \begin{bmatrix} (1,1,0,1) \\ 3 \\ (5.3,0.3,0,0) \end{bmatrix}$

Atomicity. An update can be described as a sequence of imperative code expressed in a programming language such as C. This code is evaluated sequentially w.r.t. the semantics of the C language however its execution is considered as atomic from the HSwPN point of view.

Hence, if x and x' are respectively the values of the variables before and after the execution of the code of an update of a transition t from x, the firing of t leads atomically to $x' = update(t, x)$.

4 SYSTEM MODELING

Previous work has focused on synthesizing a specialized OS for a given application [30] and on the verification of the specialized OS [7]. These works, where the modeling had been done using a network of timed automaton, served as a basis for the multi-core model. Here, a modeling using HSwPN was preferred because it is better adapted to the parallelism found in a multi-core system.

We assume that the operating system functions run in zero time. Therefore, only the operating system's functional properties and the application's functional and non-functional properties are verified. The choice of HSwPN is justified because, in the model of the application, the execution time of the tasks is taken into account in order to be able to check the temporal properties of the application as well as the scheduling of the tasks.

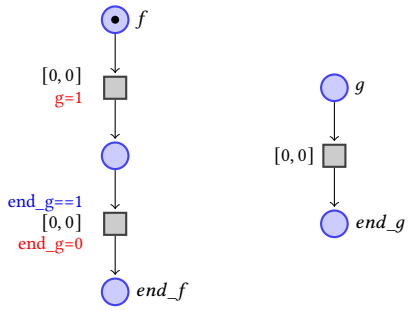

Figure 2: Function call mechanism.

HSwPN allows to associate to each transition a set of imperative expressions whose syntax is close to that of the C language. So, all Trampoline operating system functions are modeled by a combination of a Petri net and of imperative expressions. Therefore, the Petri net and imperative expressions are an integral part of the complete model. The foundation of the modeling is built on the following bases:

- The structure of the Petri net describes the control flow of the operating system.
- The variables used in the model are the operating system control variables.
- The code of the imperative expressions associated with the Petri net transitions faithfully reflects the control flow of the operating system. control variables.
- The actions and conditions on these variables associated with each Petri net transition are the same actions and conditions as those of the operating system program.

In order to ensure that the operating system code respects the atomicity of the model, we impose the following rule: *The code associated with a transition corresponds to uninterruptible operating system code*, which guarantees atomicity.

Thus, variables and operating system code are embedded in the formal model and all variables are bounded because they are either integers or enumerated types. As in [30], we thus obtain the following property:

Property 5.1: Our complete model (OS+application) contains all (and only all) the paths that could be traversed by the system composed of the operating system program and of the application program during its execution.

4.1 RTOS model

The entire RTOS code is modeled by a HSwPN whose transitions carry guards corresponding to conditional expressions and which update the RTOS variables. The firing time intervals are all set to $[0, 0]$ and, therefore, time does not elapse within the OS model. Without changing the semantics of HSwPN, it is possible to update the number of tokens in a place without explicitly drawing an arc between a transition and a place. This feature is used to lighten the design of the model. The model is thus drawn in the form of Petri subnets which appear independent but which, in reality, form only one. Each C function in the RTOS code is modeled as a Petri subnet. A function call is made by dropping a token in the initial place of the Petri subnet modeling the function and then waiting via a guard for the final place to receive a token. This indicates that the function has completed its execution. When this happens, the final place of the called function is emptied. This mechanism is shown in Figure 2.

The passing of arguments and returning of results are done using arrays of global variables indexed by the number of the core on which the function call is made. Conditional instructions are modeled by guarded transitions whose guards are complementary. Figure 3 shows an example from the modeling of the function `tpl_terminate` with two successive conditional instructions.

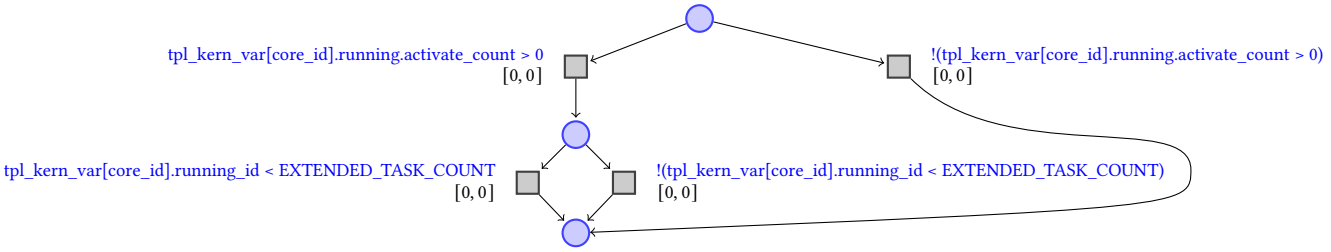

Figure 3: Conditional instruction modeling. Updates have been omitted so as to avoid burdening the figure.

The multi-core implementation of Trampoline uses a global lock that prevents concurrent execution of OS code by the cores. This lock is acquired when calling a service of the OS and is usually implemented by a spinlock. In the model, a boolean variable is used to represent this lock. This variable serves as a guard on the transitions modeling the service call. The transition is fireable if it is false, and it is set to true when the transition is fired.

4.2 Application model

An application model in the form of HSwPNs is built manually from its source code and its OIL description. From the OIL description the attributes of the objects (tasks, resources, alarms, spinlocks, ...) are extracted: priorities, periods, core on which a task runs, ... and constitute part of the model variables. The source code of each task gives the control flow and service calls of the OS, such as `ActivateTask(...)` or `TerminateTask()`, interspersed with execution times. The latter are given in the form of an interval $[BCET, WCET]$. As presented in section 3, the fact that a task runs in the model is controlled by the OS model by means of the derivative function of the stopwatches associated with the transitions representing the execution of the tasks between services calls. When the task $task_i$ is scheduled, and the OS is not running, then the function $isRunning(task_i)$ returns 1, allowing the time to elapse; otherwise, it returns 0 and blocks the elapse of time.
Figure 4 shows a modeling example of a multi-core application. This simple application consists of 3 tasks whose priorities are such that $Prio(task_1) < Prio(task_3) < Prio(task_2)$. $task_1$ runs on core 0 while the other two tasks run on core 1. The IsReady($task_i$) guards on Act_{task_i} transitions are controlled by the OS model and allow the task's model to become ready for execution. The upstream places of the Act_{task_i} transitions are not strictly speaking part of the corresponding task model. After a task is completed (the $TerminateTask()$ update is performed on the transition), the token returns to the initial place, waiting for a new activation.
In this application, $task_1$ and $task_3$ are activated at operating system startup. After a time in the range $[BCET11, WCET11]$, $task_1$ activates $task_2$. The latter having a priority higher than $task_3$ and if the firing date is such that the Run_{31} transition has not yet been fired, a preemption occurs on core 1, $IsRunning(task_3)$ becomes 0, and the elapse of time for the Run_{31} transition is blocked.

5 FORMAL VERIFICATION

According to the modeling rules, we obtained an entire HSwPN model containing the model of the RTOS and the application. We form a verification chain that includes the steps shown in the Figure 5. We use the Roméo model checker.

5.1 Model checking of HSwPN with Roméo

Reachability and most other properties of interest are undecidable for HSwPN, even when bounded [10]. However, efficient semi-algorithms allow state-space explorations that terminate in most of the practical purposes. The tool Roméo [22] implements these semi-algorithms by encoding firing domains with the Parma Polyhedra Library [5]. In particular Roméo allows the model checking of non nested TCTL properties over HSwPN. Moreover Roméo allows to use parameters in the time interval. In this case the values of the parameters that guarantee the property to be true are synthesized.

Schedulability observer. A classical method for schedulability analysis is to rely on the use of observers [14], allowing to reduce the verification problem to a simpler model-checking problem such as a simple reachability property. It is then necessary that every trace that contradicts the schedulability property can be detected by the observer but also that the observer is innocuous, meaning that it cannot interfere with the system under observation.
To analyze the schedulability of tasks, we use the classical observer represented in yellow in Figure 6 linked to each task model. The delay D_i represents the deadline of the task. The firing of transition ok_i means that the task terminates before its deadline. The firing of transition $deadline_i$ means $task_i$ does not respect its deadline. Hence the task meets its deadline iff for all state of the state space, there is no token in place Obs_i. The place Obs_i is emptied by transition $empty_i$ to avoid the accumulation of tokens. The schedulability property is then written for this observer by the in CTL logic formula $AG(Obs_i < 1)$.

Parameters synthesis. We can replace any interval bound (such as an offset, the BCET or the WCET of a task) of the HSwPN model by parameters. Given a property φ, checking φ with Roméo will synthesize the set of values of the parameters such that φ is true. For example, by parametrizing the WCET of a task $task_i$ and by using the previous observer, checking $AG(Obs_i < 1)$ will synthesize all the values of the WCET such that $task_i$ respects its deadline.

Response time of a task. To automatically compute the response time of a task $task_i$, we just replace D_i with a parameter d_i in its observer. Then the verification of the property $AG(Obs_i < 1)$ will synthesize all the values of d_i such that the task terminates before a time units i.e. the time between the job activation and the end of its execution. The smaller value of d_i is the response time of $task_i$.

```
TASK(t1) {
    ...// [BCET11,WCET11]
    ActivateTask(t2);
    ...// [BCET12,WCET12]
    TerminateTask(t1);
}

TASK(t2) {
    ...// [BCET2,WCET2]
    TerminateTask(t2);
}

TASK(t3) {
    ...// [BCET3,WCET3]
    TerminateTask(t3);
}
```

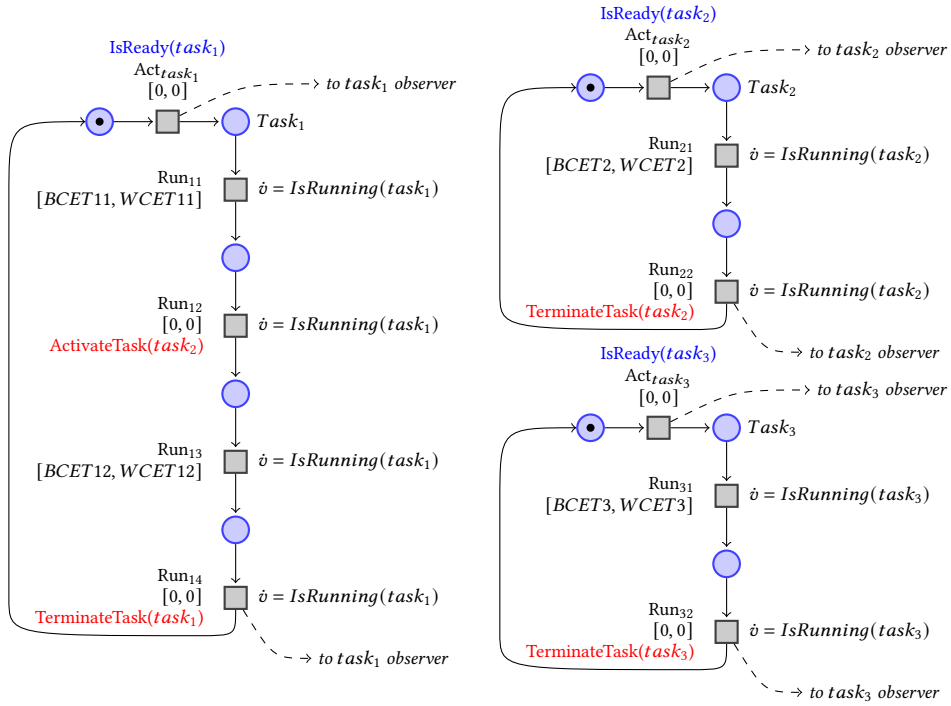

Figure 4: Application model: activation of a higher priority task $Prio(task_1) < Prio(task_3) < Prio(task_2)$; $task_1$ running on core 0; $task_2$, $task_3$ running on core 1.

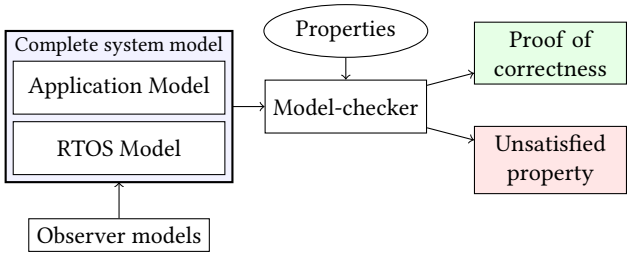

Figure 5: Verification approach.

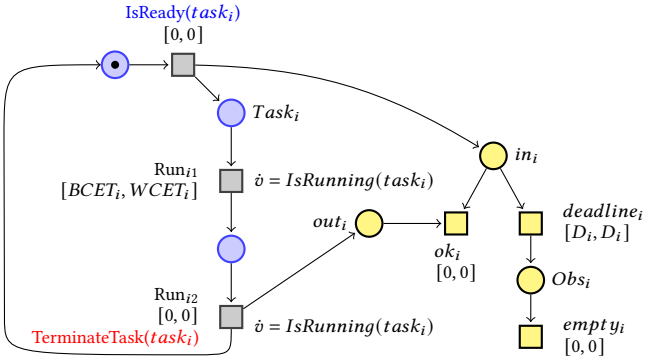

Figure 6: Observer model (in yellow) linked to a task model.

5.2 Scheduling analysis of tasks using Roméo

Let us consider the system of the figure 4 with the characteristics of Table 1. The tasks meet their deadlines if the execution time of the first part of $task_1$ is its WCET (i.e. $WCET_{11} = 11$) as we can see in Figure 7. In this case, $task_2$ starts its execution whereas the first job of $task_3$ is terminated. But we now ask for the whole execution time interval of $task_1$.

We first construct the entire HSwPN model containing the model of the RTOS (section 4) and the application model given Figure 4 with the timing values of Table 1. We add one observer per task, as shown in the figure 6. Then we use Roméo to check that the place Obs_i are never marked by a token. The property: $AG(Obs_1 < 1 \; and \; Obs_2 < 1 \; and \; Obs_3 < 1)$ is not satisfied, and a counter-example execution trace is generated, proving that $task_3$ can miss its deadline. Indeed, if the execution time of $task_1$ is its BCET, $task_3$, running on core 1 does not have the time to finish before its deadline as shown in Figure 8. The task $task_1$ activates $task_2$ at time 8, since $task_2$ has a higher priority, $task_3$ running on the same core as $task_2$ is preempted. Then $task_2$ terminates its execution at time 16, the deadline of $task_3$. The tasks set is, therefore, not schedulable under the partitioned fixed priority scheduling policy and the worst temporal behavior of the system happens with the BCET of $task_1$.

To synthesize the $task_1$ execution time interval that allows the tasks system to meet their deadlines and then to be schedulable, we set the first execution part (Run_{11}) in the parametric interval $[a,b]$ and we bound b by 11. The result of Roméo synthesis is $(10 < a \leq 11) \wedge (10 < b \leq 11) \wedge (a \leq b)$ then the execution time of $task_1$ must be in the interval $]10, 11]$.

We have run a full analysis of the application with the RTOS, performed in the first time using no parameters to verify schedulability and in the second time with parameter synthesis to found the execution time interval of $task_1$. The computing time and used memory for this analysis are shown in the table 2.

Table 1: Tasks set characteristics.

	A_i	D_i	T_i	C_i: [BCET,WCET]	HSwPN Transition
$task_1$	0	32	32	[8,11] + [2,2]	Run_{11} Run_{12}
$task_2$	0	32	32	[8,8]	Run_{21}
$task_3$	0	16	16	[10,10]	Run_{31}

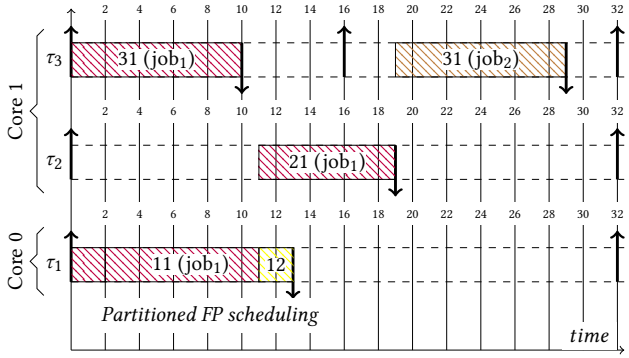

Figure 7: Schedule of tasks set with the WCET$_1$. The symbols ↑ and ↓ indicate activation and completion of tasks, respectively.

Table 2: Case-study: computing time and memory used.

	$AG(Obs_1 < 1$ *and* $Obs_2 < 1$ *and* $Obs_3 < 1)$	
Parameters	no	yes
Model checker result	false	$(10 < a) \wedge (b \leq 11) \wedge (a \leq b)$
Memory used	55.1MB	99.5MB
Computing time	4.8s	17.3s

6 AD-HOC SCHEDULING SYSTEM

The multi-core version of Trampoline implements fixed priority partitioned scheduling. However, new scheduling policies can be implemented within the RTOS, and their behavior can be formally verified using model checking. In this section, we present an implementation of an ad-hoc scheduler within the multi-core version of Trampoline. First, we construct the application model, and then we provide the scheduling specifications to be ensured. The application consists of 4 tasks: $task_1$, $task_2$, $task_3$ and $task_4$.

Ad-hoc scheduler implementation . The scheduling policy is as follow:

- $task_1$ and $task_4$ are assigned to run on core 0;
- $task_2$ and $task_3$ are assigned to run on core 1;

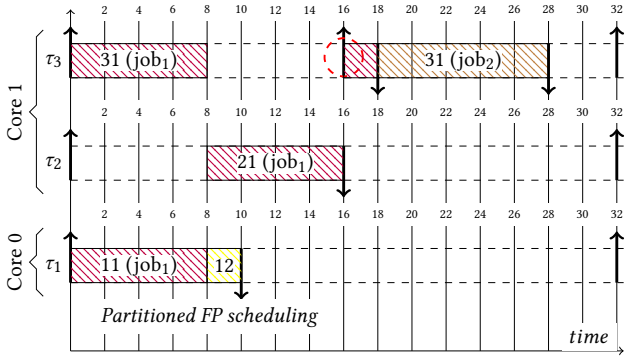

Figure 8: Schedule of tasks set with the BCET$_1$. The symbols ↑ and ↓ indicate activation and completion of tasks, respectively. Here job 1 of τ_3 misses its deadline as indicated by the dashed red circle.

- $Prio(task_1) < Prio(task_4)$;
- When $task_4$ runs on core 0, $Prio(task_2) < Prio(task_3)$ otherwise $Prio(task_3) < Prio(task_2)$.

The activation and termination of $task_1$ and $task_4$ on core 0 lead to rescheduling $task_2$ and $task_3$ on core 1.
When the scheduler is called for core 0, it is based on the job currently running on that core, $task_1$ or $task_4$, it calculates the priority of jobs, $task_2$ or $task_3$ for core 1. It then decides whether a context switch is required on a core and have to be achieved. For example, $task_2$ continues to run on core 1 until it is preempted by $task_3$ because $task_4$ is activated on core 0. The priority of $task_2$ and $task_3$ are recalculated such that: $Prio(task_2) < Prio(task_3)$. $task_3$ starts thus running on core 1 until it is preempted when $task_4$ terminates on core 0.
We now apply the modeling rules of section 4.2, the verification chain shown in the figure 5 of section 5. We use the Roméo model checker to analyze the system.

Response time analysis. Let us consider this ad-hoc scheduling system with the characteristics of Table 3. The application model is constructed according to the modeling rules (section 4.2). Our task model is the one shown in Figure 6 linked with the observer. To analyze the response time of $task_i$, we just replace D_i with a parameter d_i in its observer (section 5.1). Roméo synthesizes the values of the parameter d_i and the response time of tasks R_i is the smallest value of d_i. Table 4 provides the results obtained by Roméo model checker with time computing and memory use. We instantiate the D_i with their values (see the deadline values in Table 3) in the observers and we verify the property $(AGObs_i < 1)$. Roméo replies that the property is not satisfied and automatically generates a timed trace as counter example represented by the chronogram in Figure 10. In this case, $R_3 > D_3$, and $task_3$ is preempted twice by $task_2$. The $task_3$ misses its deadline when its execution time is the WCET (i.e. $WCET_3 = 6$).

Task parameters synthesis. We replace the interval bound [BCET, WCET] of the model $task_3$, respectively $task_4$, by the parametric interval [3,b], respectively [a,a] (Table 4). We bound the parameters a and b in the interval [3, 6]. We use the non-parameterized observer

and we instantiate the D_i. Checking the property $AG(Obs_3 < 1$ *and* $Obs_4 < 1)$ with Roméo will synthesize the tasks execution time interval such that the tasks meet their deadlines. We obtain two results with Roméo synthesis:

- $(3 \leq a \leq 6) \wedge (3 \leq b < 5)$;
- $(3 \leq a \leq 6) \wedge (3 \leq b \leq 6) \wedge a - b > -1$.

To verify these results, we use the simulator of Roméo tool that allows to run timed traces (i.e. chronograms). The first result is verified with the chronogram of Figure 9 with the execution time of the $task_3$ and $task_4$ in the interval [3, 3] (i.e. $BCET_3 = 3$). In this case, $task_2$ terminates its execution, whereas the first job of $task_3$ is terminated. Assuming that a and b are bounded in the parameter constraints in the interval [3,6]. The second result provides a relationship between a and b in this interval such that $a > b - 1$.This property is satisfied with $a = b = 5.5$. We thus set the intervals of $task_3$ and $task_4$ models in [5.5,5.5]; the extracted timed trace is presented in the Figure 11. The tasks set is, therefore, schedulable.

Table 3: Tasks set characteristics.

	A_i	D_i	T_i	C_i: [BCET,WCET]		HSwPN Transition
$task_1$	0	10	10	[4,4]		Run_{11}
$task_2$	1	20	20	[5,5]		Run_{21}
$task_3$	0	10	10	[3,6]	[3,b]	Run_{31}
$task_4$	2	20	20	[3,3]	[a,a]	Run_{41}

Table 4: Response time computation using the parametric observer.

Response time	$AG(Obs_i < 1)$	Memory	Computing time
$task_1$	$R_1 = 7, (d_1 > 7)$	273.4MB	81.7s
$task_2$	$R_2 = 8, (d_2 > 8)$	254.8MB	72.2s
$task_3$	$R_3 = 11, (d_3 > 11)$	332.1MB	88.0s
$task_4$	$R_4 = 3, (d_4 > 3)$	260.6MB	70.6s

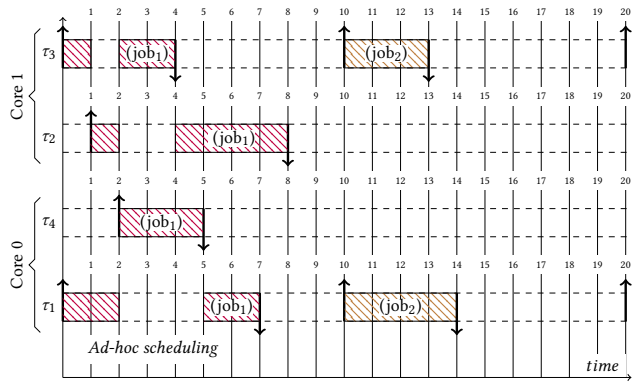

Figure 9: Schedule of tasks set with the $BCET_3$. The symbols ↑ and ↓ indicate activation and completion of tasks, respectively.

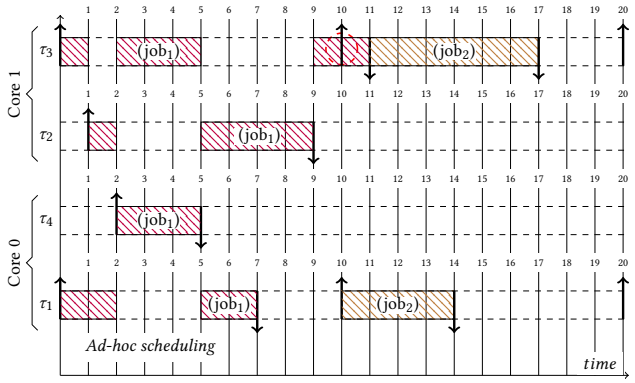

Figure 10: Schedule of tasks set with the $WCET_3$. The symbols ↑ and ↓ indicate activation and completion of tasks, respectively. Here job 1 of τ_3 misses its deadline as indicated by the dashed red circle.

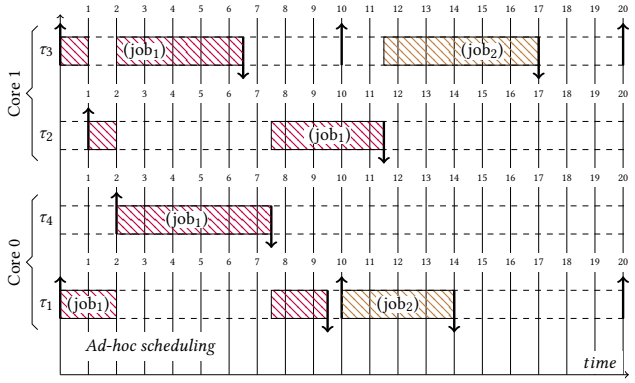

Figure 11: Schedule of tasks set with Roméo. The symbols ↑ and ↓ indicate activation and completion of tasks, respectively.

7 CONCLUSION AND FUTURE WORKS

In this paper we have briefly presented the model of Trampoline operating system which is OSEK/VDX and AUTOSAR compatible. This model is done by using High level Stopwatch Petri Net formalism and is very detailed and very close to the implementation. We also presented in more detail the principle of modeling an application which, thanks to the capabilities of stopwatches, includes the possibility to block the elapse of time on the transitions of the application model and, thus, to model the preemption. A complete model can be used to check the schedulability of an application and task response times. It also allows, in its parameterized version, to calculate the execution times required to guarantee schedulability or response times. The support of this type of model is implemented in the Roméo tool and two case studies were presented as examples. The current approach considers a global lock that prevents this concurrent execution. Modeling the parallel execution of the kernel by the different cores can be done using colors (one color corresponding to one core) and the formalism of High level Stopwatch Petri Net already includes colors. However, Roméo does not support colors yet and it will be necessary to extend it to do this work. In

the next step, we aim to model the multi-core version that allows parallel execution of the kernel code on different cores and verify its alignment to the AUTOSAR standard. We will also evaluate the approach's scalability and how effectively it can be used for a bigger and more complicated system.

REFERENCES

[1] R. Alur and D. Dill. 1994. A Theory of Timed Automata. *Theoretical Computer Science* 126, 2 (1994), 183–235.

[2] Étienne André. 2018. A Benchmark Library for Parametric Timed Model Checking. In *Formal Techniques for Safety-Critical Systems - 6th International Workshop, FTSCS 2018, Gold Coast, Australia, November 16, 2018, Revised Selected Papers (Communications in Computer and Information Science, Vol. 1008)*, Cyrille Artho and Peter Csaba Ölveczky (Eds.). Springer, 75–83. https://doi.org/10.1007/978-3-030-12988-0_5

[3] ARINC Group. 2019. 653P1-5 Avionics Application Software Standard Interface, Part 1, Required Services. https://www.aviation-ia.com/products/653p1-5-avionics-application-software-standard-interface-part-1-required-services

[4] AUTOSAR GbR. 2009. Specification of operating system.

[5] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. 2008. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.* 72, 1-2 (2008), 3–21.

[6] Jean-Luc Béchennec, Mikaël Briday, Sébastien Faucou, and Yvon Trinquet. 2006. Trampoline an OpenSource Implementation of the OSEK/VDX RTOS Specification. In *IEEE Conference on Emerging Technologies and Factory Automation. ETFA'06.* 62–69.

[7] Jean-Luc Béchennec, Olivier H. Roux, and Toussaint Tigori. 2018. Formal model-based conformance verification of an OSEK/VDX compliant RTOS. In *International Conference on Control, Decision and Information Technologies (CODIT 2018).*

[8] Gerd Behrmann, Kim Larsen, and Jacob Rasmussen. 2005. Optimal scheduling using priced timed automata. *SIGMETRICS Performance Evaluation Review* 32 (03 2005), 34–40. https://doi.org/10.1145/1059816.1059823

[9] B. Berthomieu and M. Diaz. 1991. Modeling and Verification of Time Dependent Systems Using Time Petri Nets. *IEEE Trans. on Soft. Eng.* 17, 3 (1991), 259–273.

[10] Bernard Berthomieu, Didier Lime, Olivier H. Roux, and Francois Vernadat. 2007. Reachability Problems and Abstract State Spaces for Time Petri Nets with Stopwatches. *Journal of Discrete Event Dynamic Systems - Theory and Applications (DEDS)* 17, 2 (2007), 133–158.

[11] Khaoula Boukir, Jean-Luc Béchennec, and Anne-Marie Déplanche. 2020. Requirement Specification and Model-Checking of a Real-Time Scheduler Implementation. In *Proceedings of the 28th International Conference on Real-Time Networks and Systems* (Paris, France) *(RTNS 2020)*. Association for Computing Machinery, New York, NY, USA, 89–99. https://doi.org/10.1145/3394810.3394817

[12] G. Bucci, A. Fedeli, L. Sassoli, and E. Vicario. 2004. Time state space analysis of real-time preemptive systems. *IEEE transactions on software engineering* 30, 2 (February 2004), 97–111.

[13] Y. Choi. 2011. Safety Analysis of Trampoline OS Using Model Checking: An Experience Report. In *2011 IEEE 22nd International Symposium on Software Reliability Engineering*. 200–209. https://doi.org/10.1109/ISSRE.2011.22

[14] Silvano Dal-Zilio and Bernard Berthomieu. 2015. Automating the Verification of Realtime Observers Using Probes and the Modal mu-calculus. In *Topics in Theoretical Computer Science - The First IFIP WG 1.8 International Conference, TTCS 2015, Tehran, Iran, August 26-28, 2015, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 9541)*. Springer, 90–104.

[15] Emmanuel Grolleau and Annie Choquet-Geniet. 2000. Scheduling Real-Time Systems by Means of Petri Nets. *IFAC Proceedings Volumes* 33, 7 (2000), 89 – 94. https://doi.org/10.1016/S1474-6670(17)39938-X 25th IFAC Workshop on Real-Time Programming (WRTP'2000), Palma, Spain, 17-19 May 2000.

[16] Lom Hillah, F. Kordon, Laure Petrucci, and Nicolas Trèves. 2006. PN Standardisation: A Survey. In *Formal Techniques for Networked and Distributed Systems - FORTE 2006 (Lecture Notes in Computer Science, Vol. 4229)*. Springer Berlin Heidelberg, 307–322.

[17] Michael Hohmuth and H. Tews. 2005. The VFiasco approach for a verified operating system.

[18] Y. Huang, Y. Zhao, L. Zhu, Q. Li, H. Zhu, and J. Shi. 2011. Modeling and Verifying the Code-Level OSEK/VDX Operating System with CSP. In *2011 Fifth International Conference on Theoretical Aspects of Software Engineering*. 142–149. https://doi.org/10.1109/TASE.2011.11

[19] Ekkart Kindler and Laure Petrucci. 2009. A framework for the definition of variants of high-level Petri nets. In *Proceedings of the Tenth Workshop and Tutorial on Practical Use of Coloured Petri Nets and CPN Tools (CPN '09)*. 121–137.

[20] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. SeL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[21] Didier Lime and Olivier Roux. 2009. Formal Verification of Real-time Systems with Preemptive Scheduling. *Real-Time Systems* 41 (02 2009), 118–151. https://doi.org/10.1007/s11241-008-9059-0

[22] Didier Lime, Olivier H. Roux, Charlotte Seidner, and Louis-Marie Traonouez. 2009. Romeo: A Parametric Model-Checker for Petri Nets with Stopwatches. In *15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009) (Lecture Notes in Computer Science, Vol. 5505)*. Springer, 54–57.

[23] Thanh Dat Nguyen. 2020. *Aide à la validation temporelle et au dimensionnement de systèmes temps réels dans une démarche dirigée par modèles.* Theses. ISAE-ENSMA Ecole Nationale Supérieure de Mécanique et d'Aérotechique - Poitiers. https://tel.archives-ouvertes.fr/tel-03079085

[24] Baptiste Parquier, Laurent Rioux, Rafik Henia, Romain Soulat, Olivier H. Roux, Didier Lime, and Étienne André. 2016. Applying Parametric Model-Checking Techniques for Reusing Real-Time Critical Systems. In *Formal Techniques for Safety-Critical Systems - 5th International Workshop, FTSCS 2016, Tokyo, Japan, November 14, 2016, Revised Selected Papers (Communications in Computer and Information Science, Vol. 694)*, Cyrille Artho and Peter Csaba Ölveczky (Eds.). 129–144. https://doi.org/10.1007/978-3-319-53946-1_8

[25] Olivier H. Roux and Didier Lime. [n.d.]. Roméo: formal verification and synthesis for timed systems. http://romeo.rts-software.org.

[26] Lui Sha, Tarek Abdelzaher, Karl-Erik årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K. Mok. 2004. Real Time Scheduling Theory: A Historical Perspective. *Real-Time Systems* 28, 2 (Nov. 2004), 101–155. https://doi.org/10.1023/B:TIME.0000045315.61234.1e

[27] L. Sha, R. Rajkumar, and J. P. Lehoczky. 1990. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Trans. Comput.* 39, 9 (1990), 1175–1185. https://doi.org/10.1109/12.57058

[28] J. A. Stankovic. 1988. Misconceptions about real-time computing: a serious problem for next-generation systems. *Computer* 21, 10 (1988), 10–19. https://doi.org/10.1109/2.7053

[29] Youcheng Sun, Giuseppe Lipari, and Étienne André. 2015. Verification of Two Real-Time Systems Using Parametric Timed Automata. In *WATERS - International Workshop on Analysis Tools and Methodologies for Embedded Real-Time Systems (WATERS - International Workshop on Analysis Tools and Methodologies for Embedded Real-Time Systems)*. Lund, Sweden. https://hal.archives-ouvertes.fr/hal-01240583

[30] 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 (Sept. 2017), 1 – 25. https://doi.org/10.1145/3015777

[31] Libor Waszniowski and Zdenundefinedk Hanzálek. 2008. Formal Verification of Multitasking Applications Based on Timed Automata Model. *Real-Time Syst.* 38, 1 (Jan. 2008), 39–65. https://doi.org/10.1007/s11241-007-9036-z