Formal verification process of the compliance of a multicore AUTOSAR OS

Imane Haur, Jean-Luc Béchennec, Olivier H. Roux

Formal verification process of the compliance of a multicore AUTOSAR OS.

Imane HAUR[1,2*†], Jean-Luc BÉCHENNEC[1†] and Olivier H. ROUX[1†]

[1] Laboratoire des sciences du numérique de Nantes (LS2N), 1 Rue de la Noë Nantes, 44300, France.
[2]Huawei Paris Research center, 20 Quai du Point du Jour Boulogne Billancourt, 92100, France.

*Corresponding author(s). E-mail(s): imane.haur@ec-nantes.fr;
Contributing authors: Jean-Luc.Bechennec@ls2n.fr;
olivier-h.roux@ec-nantes.fr;
†These authors contributed equally to this work.

Abstract

AUTOSAR conformance testing is based on requirements verification. This work focuses on multicore operating system (OS) requirements, of which there are eighty. We present a semi-automated formal process to check multicore OS compliance using High-Level Colored Time Petri Net and model-checking methods. To apply our approach, we use the Roméo tool to build an operating system model called Trampoline that conforms to the AUTOSAR OS specification. Each requirement of the multicore OS is formalized by an observer modeled by a Petri net to evaluate compliance. The observers evolve according to the operating system evolution without altering its behavior to check whether the specification is true or false. The approach ensures that the operating system model respects the multicore specification of AUTOSAR OS.

Keywords: High-Level Colored Time Petri Nets, Model-checking, Real-Time Operating System (RTOS), AUTOSAR OS verification

1 Introduction

The development of technologies has led to the increased use of embedded software in all areas of our daily life. In the automotive field, embedded electronic control units have multiplied, performing different functions in the vehicle over the last few years. Therefore, the complexity of the embedded software evolves and requires high safety standards to ensure proper behavior. That complexity is handled by the operating system, which serves as the hardware and software interface. The operating system implants the functionalities, manages the hardware and software resources, and schedules the application processes.

In addition, several standard software architectures have been implemented for the automotive industry. We can cite OSEK/VDX (**?**) and AUTOSAR (AUTomotive Open System ARchitecture) (**?**) standards that aim to facilitate software development by providing the necessary mechanisms for software and hardware independence. These standards provide a specification for developing real-time operating systems (RTOS) with numerous features such as scheduling policy, timing, and memory protection. Thus, the RTOS must comply with the OS specification to guarantee security and functional safety.

Compliance verification of an OSEK/VDX and AUTOSAR OS is generally performed using a test suite that includes several sets of applications or test sequences (**??**). The standard certification is thus obtained by executing the conformance test suite on the RTOS. Software testing has been the standard technique, but it cannot perform exhaustive tests to show bugs and error absence. Verification approaches based on formal methods have shown high efficiency. They permit proving mathematically that a system satisfies its specification. Two categories exist, deductive methods based on theorem proving (**?**) and automatic methods based on model-checking (**?**).

In theorem proving, we examine infinite systems specified in an appropriate mathematical logic to verify the properties and provide proof. On the other hand, model-checking is an automated approach to verify that a model of a system conforms to a specification expressed as a property. This specification defines the requirements for the expected behavior of the system. The verification is performed by exploring the model's states with the help of algorithms and allows to guarantee the properties. Achieving the system abstraction and specification is a crucial step that may require system mastery and expertise in the methods used. The model must also be accurate and as close as possible to the system from a behavioral point of view. Therefore, the property verification must be the same for the system and its model. In this work, we rely on model-checking to verify the RTOS's conformity to the AUTOSAR multi-core standard.

1.1 Contribution and outline

In this paper, we present a formal verification approach to verify the compliance of a multicore RTOS with the AUTOSAR OS specification[1]. We use model-checking technique, that exhaustively checks the state-space if the model respects specific specification and automatically generates a counter-example. This approach has been applied to the Trampoline operating system, a multicore RTOS compliant with OSEK/VDX and AUTOSAR standards used in automotive embedded systems, and written in C language.

A C program model-checker may seem usable for RTOS requirements verification, such as (**?**). However, RTOS service calls are performed through assembly code that would require a complete hardware architecture model for formal analysis, making it non-portable. Furthermore, the aim is not only to check the properties of a C program but also to handle simultaneous execution and concurrency in a multicore context and to deal with interleaving and the resulting interruptions. Therefore, we need a model that considers all these aspects to be accurate and close to the multicore RTOS behavior.

Petri nets formalism is one of the many mathematical modeling languages used to describe distributed systems whose vertices are places and transitions. However, it does not directly capture the concurrency of multicore real-time systems and is unsuitable for modeling systems where data affects the system's behavior. We will thus use the formalism of time Petri nets, extended with colors, i.e., each color modeling the core on which the code is executed, and high-level functionality, i.e., a predefined syntax manipulating different types of expressions made up of variables. The detailed presentation of this extended formalism has been proposed in our previous contribution (**?**).

This work presents a systematic approach to verifying the AUTOSAR compliance of multi-core RTOS using High-Level Colored Time Petri Nets (HCTPN). The steps of the approach are illustrated in Figure 1. It shows the two main stages of the process. We rely on the ROMÉO model-checker tool, available under a free license (**?**) and recently improved to support HCTPN.

- The first phase involves developing a complete model that includes the multi-core RTOS model and is complemented by application models. We construct the models manually based on the HCTPN translation rules presented in Section 5.2. The RTOS model is described by HCTPNs and ROMÉO functions written in a syntax similar to the C language called C-like ROMÉO syntax. These functions can be called on the transitions of the model, as shown in Figure 6. The model contains 115 Petri sub-networks that form a single one and describe the RTOS behavior. The application model is a HCTPN which describes all the system calls performed based on the application's source code. We developed a module in the compilation phase of an application on the RTOS to automatically extract the data structures and constants and generate a C-like configuration file to be included in the ROMÉO tool. Once the configuration file

[1] https://www.autosar.org/fileadmin/user_upload/standards/classic/4-2/AUTOSAR_SWS_OS.pdf

is defined, the application model can be simulated and formally verified within the RTOS model.

- The verification phase is based on two possible formalizations of properties with model-checking: (i) expressing them formally with temporal logic to be verified by the model-checker and (ii) adding Petri nets manually as external observers able to evaluate whether requirements are respected. In the first case, the TCTL (Timed Computation Tree Logic) (**?**) allows expressing requirements as properties in Roméo. However, these requirements can easily become difficult to express by involving several parameters, leading to nested properties where one property is defined inside another. The Roméo tool does not support this kind of formula, which motivates our choice to use observers. Indeed, the expression of the requirements is systematically performed through an observer. The verification is therefore achieved automatically by a reachability test on a given observer state. Observers are read-only processes that keep track of some invariants in the execution of the Petri Net and do not modify the system state. The multi-core AUTOSAR requirements are thus translated into observers describing the expected behavior. Then, with the help of properties written in TCTL logic, we can verify by the model-checker their satisfaction or generate, on the contrary case, a counter-example trace.

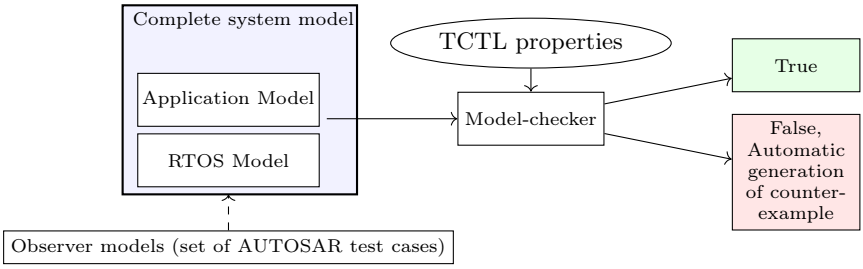


Fig. 1 Formal verification approach

The remains of this paper are as follows. Section 2 presents some related works, and Section 3 the Trampoline RTOS. Section 4 defines the HCTPN formalism used for modeling. Section 5 explains the formal model of the RTOS. Section 6 describes our formal verification approach and observer models. Section 7 presents some results of verification of compliance with the AUTOSAR standard. Section 8 concludes the paper.

2 Related works

These works are selected because the verification conducted on operating systems is based on formal methods.

Many studies have been done on AUTOSAR OS verification. Peng et al. in (**?**) use timed CSP to model AUTOSAR OS and the engine management

system (EMS) application. They verify some safety properties through Process Analysis Toolkit (PAT). The authors in (**?**) propose a formalization of the AUTOSAR OS memory protection specification. They use the Event-B specification language and verify the consistency. Yan et al. in (**?**) focus on the AUTOSAR schedule table mechanism. They formally model a schedule table using a transition system and analyze the schedulability. Many studies have been done on the verification of the AUTOSAR operating system. They focus on scheduling, timing, and memory protection aspects. However, research on the verification of the AUTOSAR multicore RTOS has been limited.

Some studies (**??**) propose a test program generator for OSEK/VDX and multicore AUTOSAR standards. Chen et al. (**?**) present an exhaustive test generation method to ensure compliance with the OSEK/VDX standard using the model-checking tool SPIN. The authors built a test model based on the formal test specification and the formal model of the OSEK operating system in PROMELA. Using model-checking techniques on these models, they developed their test case generation tool (TGT). Fang et al. in (**?**) show a formal model-based approach to improve the test coverage for AUTOSAR multicore RTOS. They first defined the concrete formal model conforming to the requirement of AUTOSAR RTOS in Promela. Then, with the model, they proposed a test program generator. Finally, they calculated the optimal test sequence for every test case and translated it into an execution program. This approach can complement ours to generate test cases and ensure good coverage.

There is some research (**??**) that present a compliance verification to OSEK/VDX standard. Huang et al. in (**?**) concentrate on the compliance verification of an OSEK/VDX operating system. They model the code-level with the process algebra CSP and verify some properties with the PAT model-checker. The approach does not contain the complete OSEK/VDX specifications. Therefore, it does not guarantee that the operating system conforms to the OSEK/VDX specification. Tigori T. et al. in (**?**) propose to check the conformity of the RTOS model to the OSEK/VDX standard through observers and the UPPAAL model-checker. They first model the complete mono-core version of the Trampoline RTOS with extended and timed automata in the UPPAAL tool. Then, they translate the OSEK/VDX conformance test cases into observers that allow checking whether the RTOS model meets the OSEK/VDX specification.

Several works are done on the Trampoline RTOS. In (**?**), the authors convert the Trampoline kernel source code into a formal model specified in PROMELA. Using model-checking, they check the exactness of the kernel model and identify some possible safety violation scenarios. The authors in (**?**) propose a complete mono-core model of the Trampoline RTOS with extended and timed automata in the UPPAAL tool. They perform a reachability analysis on the application and OS model states to eliminate infeasible paths, and prune the model appropriately. From the pruned model, they generate the configured application source code. Based on the Trampoline formal model done by Tigori T. et al. in (**?**), Boukir K. et al. (**?**) integrate the model of

the global EDF scheduling and verify the scheduler implementation. However, the Tigori T. model does not represent the accurate time aspect. Indeed, the execution time is discrete, expressed with invariant and clock variables. In a previous work (**?**), we propose a formal approach that verifies the schedulability of a real-time system using model-checking. We use Parametric High-Level Stopwatch Petri Nets in the ROMÉO tool to model the application and the multicore kernel with Biglock. We use the Petri net formalism for modeling preemption and temporal precision.

3 Trampoline RTOS

Trampoline is a real-time operating system[2] developed by the STR group of the LS2N laboratory in Nantes, France. This operating system is OSEK/VDX 2.2.3 and AUTOSAR 4.0 compliant. It is used by many European car manufacturers and in several French universities.Trampoline is mostly written in C language with some parts, like context switching, written in assembly language because they depend on the Instruction Set Architecture (ISA) of the microcontroller.

This type of operating system occupies few resources, both memory and CPU, and is suitable for both 8-bit and 32-bit targets. It offers the classic services:

- Management and scheduling of tasks according to a fixed priority scheduling policy;
- Synchronization between tasks via signaling (events) and mutual exclusion (resources) mechanisms;
- Periodic execution of tasks or setting of events (alarms and schedule tables);
- Communication between tasks on the same Electronics Control Unit (ECU) or running on different ECUs.
- Interrupt Service Routines (ISR) management.

Additionally, AUTOSAR is an evolution of OSEK and specifies a multicore design that implements a partitioned scheduling policy with fixed priority. In this type of scheduling policy, the OS manages a list of ready tasks by computing core. Partitioning is obtained by assigning the objects managed by the OS (tasks, ISR, alarms, schedule tables, events, resources, ...) to an entity named OS Application. Then, each OS Application is assigned to a computing core. An additional mutual exclusion mechanism, dedicated to multicore and using spinlocks is also present.

In the multicore version of Trampoline, following a service call, rescheduling is performed on the core where the service call occurred. Thus, when a task running on core 0 activates a task assigned to core 1, the task activation service is performed on core 0 and modifies the list of ready tasks of core 1. Additionally, when this activation requires a context switch on core 1, it must necessarily be performed on core 1 as well. To trigger this context switch, core 0 therefore sends an inter-core interrupt request to core 1.

[2] https://github.com/TrampolineRTOS/trampoline

The OSEK and AUTOSAR OS are configured according to the application. The objects necessary for the application, tasks, ISRs, alarms, ... are described, as well as their relations, in a dedicated language called OIL[3] for OSEK and in XML for AUTOSAR. A dedicated compiler called Goil reads this description and, using templates described in a Goil template language (GTL), produces C data structures (task descriptors, alarms, etc.) and code that is then compiled and linked with the OS and application code, as shown in Figure 8, page 21.

While in the past the OSEK/VDX consortium has published several documents specifying how the conformance of an OS to the OSEK/VDX standard should be tested (**?**), this is not the case for the AUTOSAR consortium. However, the Trampoline project has developed a test suite based on the requirements listed in (**?**). Among more than 200 requirements related to the implementation of AUTOSAR OS, 80 concern multicore implementation. In this work, we have focused on the multicore requirements. Of course, we will not list all of them here and we will only point out the ones that are necessary to understand the examples used in Section 6. Table 4, given in the appendix, lists the requirements related to the multicore implementation of an AUTOSAR OS.

4 Background

Petri nets (**?**) are a mathematical formalism and a bipartite graph whose vertices are places and transitions. Places are drawn as circles and transitions as squares. 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 consumption to be possible. Firing a transition from a marking M consumes tokens from each input place and produces tokens in each output place.

Petri nets are unsuitable for modeling systems where data affects the system's behavior because of the lack of a data structure. High-level Petri nets (**?**) have been proposed to model scientific problems with complex structures that describe both system data and control. The term High-level Petri net is then used for many Petri nets (**?**) such as Predicate/Transition Nets, colored Petri nets, or hierarchical Petri nets. However, the common point is that they allow the manipulation of 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.

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

[3] OSEK Implementation Language.

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.

4.1 Informal presentation

High-level Petri nets

Among the mathematical modeling languages, Petri nets are well suited to describe distributed concurrent systems. A place can contain any number of tokens. 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 input place s and produces one token in each of its outputs places s.

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

In this paper, we consider that preconditions (guard) and postconditions (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, and producing tokens in its output places. The variables take their values in a finite state, such as bounded integers or enumerate types. Guards are Boolean expressions over X, and updates can be described as a sequence of imperative code expressed in a programming language whose execution is atomic from the transition firing point of view. This class is illustrated in Figure 3.

Colored Petri nets. Colored Petri nets extend marked Petri nets to allow the distinction between tokens.

Although the set X of High-level Petri nets presented in the previous paragraph can be of arbitrarily complex type, places in colored Petri nets contain tokens of one type. This type noted C is called the color set of the place.

An arc from a place to a transition (PT) specifies the color(s) that enabled the transition and will be consumed by its firing. An arc from a transition to a place (TP) specifies the token's color produced in that place by the firing of the transition. A particular color called *any* indicates in a PT arc that any color enabled the transition and implies in a TP arc that the color consumed in the input place will be the same produced in the output place.

A marking M of a colored Petri Net represents the number of tokens in each place and their respective colors. That is represented either by a multiset or by a matrix.

Colored Time 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 endpoints 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 the firing of another transition. Firing a transition takes no time.

The introduction of color in time Petri nets leads to multiple enableness of transitions and then allows the modeling of multiple servers and multiple instances of codes (**?**). For Colored Time Petri Nets, multiple enableness occurs when several combinations of colors enable a transition simultaneously, requiring a dynamic number of timers. This class is illustrated in Figure 2.

4.2 High-level Colored Time Petri Net

Colored Petri nets allow tokens to have a data value called the token color. In the applications we are considering, the color of a token actually represents the processor on which the code is executed. We, therefore, consider a token of integer type that designates the processor number. Moreover, we add a special color called *any* to specify that any color can be used for enabling and firing a transition.

Definition

Definition 1 (High-level Colored Time Petri Net) A *High-level Colored Time Petri Net* (HCTPN) is a tuple $\mathcal{N} = (P, T, X, C, \mathsf{pre}, \mathsf{post}, (m_0, x_0), guard, update, I)$ 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 $C_{any} = C \cup \{any\}$ where *any* is a variable that can be instantiated to any value of C,
- $\mathsf{pre} : P \times T \to \mathbb{N}^{C_{any}}$ is the backward incidence mapping,
- $\mathsf{post} : P \times T \to \mathbb{N}^{C_{any}}$ is the forward incidence mapping,
- $guard : T \times X \times P \times C_\bullet \to \{true, false\}$ is the guard function with $C_\bullet = C \cup \{\bullet\}$ where $\bullet$ denotes the fact that no color is specified,
- $update : T \times X \times P \times C_\bullet \to \mathbb{X}^X \times \mathbb{N}^{P \times C}$ is the update function,
- $(m_0, x_0) \in \mathbb{N}^{P \times C} \times \mathbb{X}^X \to$ 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 behavior

For a marking $m \in \mathbb{N}^{P \times C}$, $m(p)$ is a vector in $\mathbb{N}^C$, and $m(p)[c]$ represents the number of *tokens* of color $c \in C$ in place $p \in P$. A valuation of the set of variables X is noted $x \in \mathbb{X}^X$. (m, x) is a discrete state of HCTPN.

Enabling a transition. Informally, an arc is associated either with a color $c \in C$ or with a particular color called *any*. To enable transition t, a place p with an arc from p to t must have enough tokens with the arc's color. Moreover, all the arcs of t associated with *any* must agree on the color given to *any*. Therefore, we forbid an arc to be associated with both *any* and a color $c \in C$.

An arc $\mathsf{pre}(p,t) \in \mathbb{N}^{C_{any}}$ is a vector such that $\mathsf{pre}(p,t)[c]$ is the number of tokens of color $c \in C$ in place p needed to enable the transition t and $\mathsf{pre}(p,t)[any] > 0$ represents the fact that any color can enable the transition. Let $T_{any} \in T$ the set of transitions that can be enabled by *any* color: i.e. $T_{any} = \{t \in T, \exists p \in P, \text{ s.t. } \mathsf{pre}(p,t)[any] > 0\ \}$. Moreover, we define the set $T_{\overline{any}} = T \setminus T_{any}$.

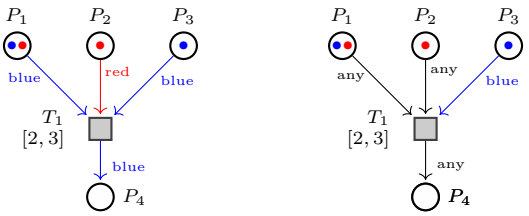
A transition $t \in T$ is said to be *enabled* by a given marking $m \in \mathbb{N}^{P \times C}$ in two cases depending on whether $t \in T_{any}$ or not:

- if $t \in T_{\overline{any}}$, and $\forall p \in P$ and $\forall c \in C$, $m(p)[c] \geq \mathsf{pre}(p,t)[c]$. We denote $\mathsf{en}(m,t) \in \{true, false\}$, the true value of this condition.
- if $t \in T_{any}$, and $\exists c_a \in C$ such that $\forall p \in P$, $m(p)[c_a] \geq \mathsf{pre}(p,t)[any]$ and $\forall c \in C \setminus \{c_a\}$, $m(p)[c] \geq \mathsf{pre}(p,t)[c]$. The corresponding set of color c_a is noted $\mathsf{colorSet}_{\mathsf{any}}(m,t) \subseteq C$

Finally, a transition $t \in T$ is said to be *enabled* by a given marking $m \in \mathbb{N}^{P \times C}$ and a valuation $x \in \mathbb{X}^X$ if $\mathsf{en}(m,t) = true$ and either $\mathsf{colorSet}_{\mathsf{any}}(m,t) = \emptyset$ and $guard(m,t,x,\bullet) = true$ or $\exists c_a \in \mathsf{colorSet}_{\mathsf{any}}(m,t) \neq \emptyset$ and $guard(m,t,x,c_a) = true$.

We illustrate the enabling condition with two examples with two colors $C = \{blue, red\}$ For the HCTPN given in Figure 2.a, the transition $T_1 \in T_{\overline{any}}$.

We have $\mathsf{pre}(T_1) = \begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \\ P_4 \end{array} \begin{array}{c} \begin{array}{ccc} red & blue & any \end{array} \\ \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{array}$. The initial marking is $m_0 = \begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \\ P_4 \end{array} \begin{array}{c} \begin{array}{cc} red & blue \end{array} \\ \begin{pmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \end{array}$ that enables the transition T_1 and $\mathsf{en}(m_0, T_1) = true$.



2.a: $T_1 \in T_{\overline{any}}$ 2.b: $T_1 \in T_{any}$

Fig. 2 Enabling transition

Now we consider the HCTPN given in Figure 2.b with the same initial marking m_0 but where the transition $T_1 \in T_{any}$ since at least one arc (here two) is associated with the color *any*.

We have $\mathsf{pre}(T_1) = \begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \\ P_4 \end{array} \begin{array}{c} \begin{array}{ccc} red & blue & any \end{array} \\ \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{array}$. The transition is enabled only if *any* takes the *red* value then $\mathsf{colorSet}_{\mathsf{any}}(m_0, T_1) = \{red\}$. If place $P2$ had two tokens with one token per color, then the transition would be multi-enabled by the two colors leading to $\mathsf{colorSet}_{\mathsf{any}}(m_0, T_1) = \{blue, red\}$.

The firing of a transition. An arc $\mathsf{post}(p,t) \in \mathbb{N}^{C_{any}}$ is a vector such that $\mathsf{post}(p,t)[c]$ is the number of tokens of color $c \in C$ produced in place p by the firing of the transition t, and $\mathsf{post}(p,t)[any]$ gives the number of tokens produced in p with the color $c \in \mathsf{colorSet}_{\mathsf{any}}(m,t)$ used for the enabling and then for the firing of t.

Firing an enabled transition $t \in T_{\overline{any}}$ from (m,x) such that $\mathsf{en}(m,t) = true$ and $guard(m,t,x,\bullet) = true$ leads to a new marking m' defined by $\forall c \in C, \forall p \in P$, $m'(p)[c] = m(p)[c] - \mathsf{pre}(p,t)[c] + \mathsf{post}(p,t)[c]$ and a new valuation $x' = update(m,t,x,\bullet)$. This new marking is denoted $m' = \mathsf{firing}(m,t,\bullet)$ where $\bullet$ denotes the fact that no *any* color has to be instantiated for this firing.

Firing an enabled transition $t \in T_{any}$ from (m,x) with the *any* color $c_a \in \mathsf{colorSet}_{\mathsf{any}}(m,t)$ leads to a new marking defined by $\forall c \in C \setminus \{c_a\}, \forall p \in P$, $m'(p)[c] = m(p)[c] - \mathsf{pre}(p,t)[c] + \mathsf{post}(p,t)[c]$ and $\forall p \in P$, $m'(p)[c_a] = m(p)[c_a] - \mathsf{pre}(p,t)[c_a] - \mathsf{pre}(p,t)[any] + \mathsf{post}(p,t)[c_a] + \mathsf{post}(p,t)[any]$ and a new valuation $x' = update(m,t,x,c_a)$. This new marking is denoted $m' = \mathsf{firing}(m,t,c_a)$.

We denote by $\mathsf{newen}((m,x),t,c)$ the set of transitions that are newly enabled by the firing of t from (m,x) with the color c ($c = \bullet$ if $t \in T_{\overline{any}}$).

Let us go back to the HCTPN of Figure 2.a, the firing of $T_1 \in T_{\overline{any}}$ from m_0 leads to the marking $m_1 = \begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \\ P_4 \end{array} \begin{array}{c} \begin{array}{cc} red & blue \end{array} \\ \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix} \end{array}$. It is noted $m_0 \xrightarrow{(T_1,\bullet)} m_1$.

Let us now consider the HCTPN of Figure 2.b, the firing of $T_1 \in T_{any}$ is possible only for $any = red$ and leads to the marking $m_2 = \begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \\ P_4 \end{array} \begin{array}{c} \begin{array}{cc} red & blue \end{array} \\ \begin{pmatrix} 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \end{pmatrix} \end{array}$.

It is noted $m_0 \xrightarrow{(T_1,red)} m_2$.

If place P_2 had two tokens with one blue and one red color, T_1 is multi-enabled, and the firing of $T_1 \in T_{any}$ is possible for $any = red$ or $any = blue$. For $any = blue$, it leads to the following marking m_3 from this new initial marking m'_0. $m'_0 = \begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \\ P_4 \end{array} \begin{array}{c} \begin{array}{cc} red & blue \end{array} \\ \begin{pmatrix} 1 & 1 \\ 1 & 1 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \end{array} \xrightarrow{(T_1,blue)} m_3 = \begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \\ P_4 \end{array} \begin{array}{c} \begin{array}{cc} red & blue \end{array} \\ \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix} \end{array}$.

High-level functionalities

We now illustrate the high-level functionalities. In the figures, the guards are in brown, and the updates are in purple.

The model in Figure 3 is an HCTPN with a set of three colors $C = \{blue, red, black\}$. Several combinations of color usage, on guards and in updates, via the $\$any$ variable are presented[4].

Transition $T_1 \in T_{any}$ since at least one arc is associated with the color any. A firing of this transition produces a blue token in P_3 and produces a token in P_2 with the color ($\$any$) used for the firing. Moreover, the value of $\$any$ is used in the precondition (guard) and the postcondition (update). Hence transition T_1 is not enabled by the blue token because of the guard $\$any \geq 1$. Moreover, the firing of T_1 leads to the execution of the update cpt[$any]=f($any,cpt). Then the transition T_1 will be fired twice, respectively, with red and black tokens leading to a marking with red and black tokens in P_2 and 2 blue tokens in P_3. A blue token remains in P_1, and the final value of cpt is {2,4,4}.

```
typedef color {blue = 0, red = 1, black = 2};
int[3] cpt = {2,2,5};

int f(int firedColor, int[3] c) {
  if (firedColor == red) {
    return c[firedColor]*2;
  }
  else if  {
    return c[firedColor] -1 ;
  }
}
```

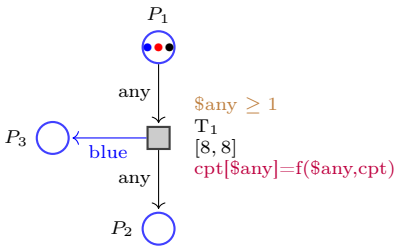


Fig. 3 HCTPN illustrating high-level manipulation of variables

Time behavior

For any $t \in T_{any}$, $v(t, c)$ is the valuation of the clock associated with t and the color $c \in C$. i.e., it is the time elapsed since the transition t has been newly enabled by m with $c \in \mathsf{colorSet}_{\mathsf{any}}(m, t)$. For other transitions $t \in T_{\overline{any}}$, $v(t, \bullet)$ is the valuation of the clock associated with t.

$\bar{0}$ is the initial valuation with $\forall t \in T$, $\forall c \in C \cup \{\bullet\}$, $\bar{0}(t, c) = 0$.

[4]In the example in this section and in the examples that follow we present models designed with the tool Roméo. In this tool, $\$any$ is used instead of any in guards and updates for syntactic reasons but both have the same meaning.

As an example, if we keep only the useful clocks, the initial valuation of the HCTPN of Figure 2, is $v_0 = T_1 \begin{pmatrix} \bullet & red & blue \\ 0 & & \end{pmatrix}$ for Figure 2.a, and $v_0 = T_1 \begin{pmatrix} \bullet & red & blue \\ & 0 & 0 \end{pmatrix}$ for Figure 2.b,

We now go back to the HCTPN given in Figure 3

The initial marking $m_0 = \begin{matrix} \\ P_1 \\ P_2 \\ P_3 \end{matrix} \begin{pmatrix} blue & red & black \\ 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$ enables the transition T_1. The valuations of the clocks are given by the matrix such that the initial valuation is $v_0 = T_1 \begin{pmatrix} \bullet & blue & red & black \\ & 0 & 0 & \end{pmatrix}$. Since the set of variables is $X = \{cpt\}$, we note a state $s = (m, cpt, v)$. The initial state is $q_0 = (m_0, \{2, 2, 5\}, v_0)$. The transition T_1 is enabled twice and can fire after elapsing 8 time units for both enabling. After 8 time units, T_1 fires with either the red or the black colors and then can fire again with the other one. Assume that we first fire with the red color, the corresponding run is as follows: $\left(m_0, \{2, 2, 5\}, \begin{pmatrix} \bullet & blue & red & black \\ & 0 & 0 & \end{pmatrix}\right) \xrightarrow{8}$

$\left(m_0, \{2, 2, 5\}, \begin{pmatrix} \bullet & blue & red & black \\ & 8 & 8 & \end{pmatrix}\right) \xrightarrow{(T_1, red)}$

$\left(\begin{matrix} \\ P_1 \\ P_2 \\ P_3 \end{matrix} \begin{pmatrix} blue & red & black \\ 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}, \{2, 4, 5\}, \begin{pmatrix} \bullet & blue & red & black \\ & & 8 & \end{pmatrix}\right) \xrightarrow{(T_1, black)}$

$\left(\begin{matrix} P_1 \\ P_2 \\ P_3 \end{matrix} \begin{pmatrix} blue & red & black \\ 1 & 0 & 0 \\ 0 & 1 & 1 \\ 2 & 0 & 0 \end{pmatrix}, \{2, 4, 4\}, \begin{pmatrix} \bullet & blue & red & black \\ & & & \end{pmatrix}\right)$

Decidability and complexity

The formal semantics of HCTPN is given in (**?**), in which the complexity and decidability of verification problems are studied. We give here the result that will interest us in the following sections: Reachability problem and TCTL (**?**) model-checking for bounded High-level Colored Time Petri Net are decidable and PSPACE-complete.

5 Modeling with HCTPN

This section presents our modeling approach applied to the Trampoline RTOS. In general, there are three challenges in modeling an RTOS with its application:

- Find a model class that is sufficiently expressive and on which model-checking algorithms exist, as proposed in the previous Section 4;
- Choose the level of abstraction regarding the desired verification, which will be specified in Section 5.1;
- Establish modeling rules that allow the same approach on a different OS and automate the process, as will be presented in Section 5.2.

The Trampoline source code is over 20,000 lines and is mainly written in the C language. It includes 180 functions modeled manually by a systematic approach based on HCTPN translation rules in Section 5.2, using the Roméo

tool (**?**), extended to support the HCTPN formalism. We describe 55 functions that do not require fine-grained, as imperatives code with C-like Roméo syntax. The model contains 115 Petri subnets that form a unique one. It contains in total 600 transitions, 550 places, and 250 variables, and other data structures and variables generated automatically in the compilation phase of the application, using a developed Goil Template Language (GTL) module.

5.1 Level of abstraction

The code associated with a transition in a HCTPN is executed sequentially and considered atomic in the state space, i.e., if several variables are updated on a transition, the intermediate state(s) are not present. This code can be one or a sequence of instructions, and it can also be a function call written in the C-like Roméo language. In the modeling step, the association of an instruction sequence or a C-like function call to a transition reduces the state space. The execution of the function call associated with a transition is also considered atomic in the modeling.

The level of abstraction can be defined with atomicity. We can consider the maximum level of abstraction with one OS service call per transition or adopt a more detailed level with one instruction per transition. The maximum abstraction level leads to a much less fine-grained state observation and makes checking AUTOSAR requirements impossible. With a detailed abstraction level, where instruction is assigned to a transition, we consider all possible interleavings, and the number of states will increase considerably. So we decided to define an intermediate level of abstraction where we have one function call per transition, and all non-interruptible code is put in a transition update in null time. An update can read and/or write variables as the kernel access is sequential thanks to a global lock. When it is a shared variable of the modeled system, one must be careful to reproduce the race condition of the real system. Therefore the modification of a shared variable accessible in concurrency situations must be cut in two: the reading on a transition and the writing on the following transition.

5.2 Modeling rules

We consider the following modeling bases:

- The Petri net faithfully describes the operating system control flow.
- The variables and structures used in the model are the same as those of the operating system.
- Actions and conditions on variables in the model are those of the operating system control.
- Pointers are represented by indexes in arrays in the model.
- The number of cores is represented by the number of colors in the model.
- All transitions are fired in a time interval $[0, 0]$ because (i) the time is not necessary for the modeling of the kernel (ii) the knowledge of the time would apply only to a precise hardware target, and the genericity of

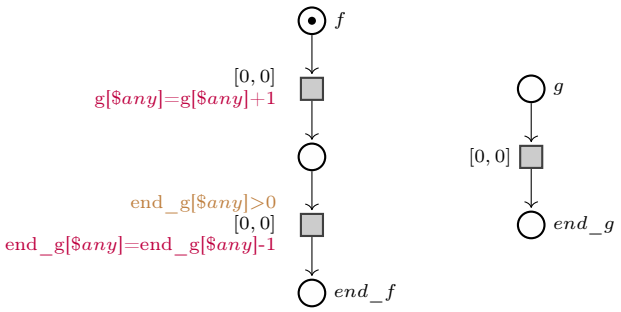


Fig. 4 Function call mechanism.

the model would be lost, (iii) it is necessary to capture all the possible interleavings.

- Functions that do not require fine-grained model checking instruction by instruction (e.g., functions that initialize or increment variables or results and error comparison functions) are written in the C-like ROMÉO language and are associated with single transitions in the model;
- The application model is described by service calls.
- The conditional expressions, as found in the if-else or loops statement, are represented by the transition guards corresponding to the conditions. For example, an if-else statement is modeled by a place and two outgoing transitions where the condition and complementary condition are assigned to the transition guards.
- The updates of the transitions correspond to the atomic block of instructions to be executed.

Function call

The function call synchronization is done by tokens deposited in places, indexed by the variable *any* representing the core identifier on which the function call is made. The calling function drops a token in the initial place of the Petri subnet modeling the called function. A guard on the token blocks the execution of the calling Petri subnet. Once the called Petri subnet completes its execution, the calling Petri subnet is released. The token is finally consumed to avoid accumulation in the last place, causing an unbounded Petri net. Figure 4 presents the mechanism.

5.3 Multi-core RTOS modeling

The source code of Trampoline includes both the single-core and the multi-core versions. To unify the code of the two versions, a set of macros allows us to generate adequate code according to whether we compile for multi-core or single-core. The RTOS model is composed of the Application Programming Interface (API) services and the kernel. Each modeled Trampoline source code function is described by a Petri subnet and, if needed, by a ROMÉO function defined in a C-like syntax. The ROMÉO tool allows using a variable *any*, which

gives the value of the color used for the transition firing. A global lock prevents concurrent execution of the kernel by the cores in the multi-core implementation of Trampoline. This lock is acquired when calling an OS service by an application task. We represented it by a Boolean variable serving as a guard on the transitions modeling the service call. The transition is fireable if the variable is false; the variable is set to true when the transition is fired.

5.3.1 Services modeling

The API contains the various services available to the application. The API function calls allow the application's tasks to access the requested service in user mode. The kernel is locked during its execution on a core by entering kernel mode → the global lock is then taken to prevent competitive situations between the cores in the kernel.

All the services of the API layer are modeled with HCTPNs in the same manner. The first transition of the model describes that when an API function is called to execute the requested service, the core switches from user mode to kernel mode using the $kernel\_mode$ array. This passage is local to this core, hence the array $kernel\_mode$ is indexed by $\$any$. The variable $\$any$ gives the value of the color used for the transition firing. Thus, the transition firing can be performed simultaneously for different cores. The global lock variable $lock\_kernel$ is a shared variable that prevents simultaneous service calls by different cores. When the API function completes its execution, it unlocks the kernel ($lock\_kernel = 0$), and another service can then be called → it finally leaves the kernel mode ($kernel\_mode[\$any] = 0$).

Let us consider the API GetAlarmBase service that allows to obtain the requested information on the alarm base and to store it in a global variable. An error is returned if the alarm identifier is invalid. This service call model is shown in Figure 5. GetAlarmBase calls the kernel function tpl_get_alarm_base_service model[5], represented in details in Figure 6.

5.3.2 Kernel functions modeling

The Kernel contains all the low-level functions on which the Trampoline services are based. It ensures the start and shutdown of the OS and allows the activation of tasks, their scheduling, and their synchronization.The kernel model consists mainly of three components that contain the functions required by the Trampoline services. We explain each module in the following.

Task manager

The task manager contains the function models that manage the application tasks' activation, synchronization, and termination. They also perform scheduling and context switch if necessary.All the functions contained in the

[5]The two double dots (::) are equivalent to an arc in the model. This syntax proposed by Roméo allows a clear and better organization of the Petri subnet in different XML files, which form only one Petri net. Thus a function call is ensured by the following syntax: the XML file name of the Petri subnet:: the place name to which we want to send a token.

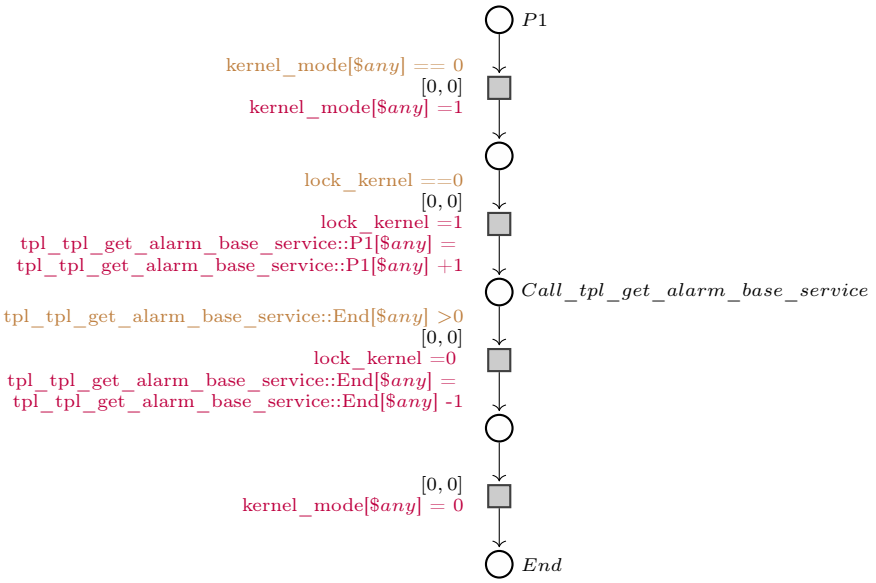


Fig. 5 GetAlarmBase service model

task manager are modeled. It includes the function models tpl_activate_-task_service and tpl_terminate_task_service responsible for activating and terminating a task and setting its state, respectively.

Scheduler

The scheduler model is the core module of the kernel; it is based on the one proposed by the OSEK/VDX and AUTOSAR standards. The multicore version of Trampoline implements a fixed priority partitioned scheduler. The scheduler manages a data structure, tpl_kern, to store information about the running process and uses functions to handle the list of ready tasks and ISRs of category 2. Among them, we can highlight the following modeled functions:

- tpl_put_new_proc adds a newly activated process to the list of ready processes;
- tpl_put_preempted_proc adds a preempted process to the list of ready processes;
- tpl_remove_front_proc removes the highest priority process from the list of ready processes.

Counter manager

The counter manager model handles any interruptions coming from the timer. When an interrupt occurs, the action related to the set of expiring alarms is executed. The action of the alarm can correspond to the activation of a task, an event, or a call-back function. The interrupt can also cause a rescheduling. Alarms and counters are defined statically according to the OSEK/VDX and AUTOSAR standards. In the model, tpl_call_counter_tick increments the

counter tick and checks the next alarm date. tpl_raise_alarm model describe when an alarm time object is raised.

Alarm base information can be obtained through the tpl_get_alarm_base_service kernel function called by GetAlarmBase API service (Figure 5). Figure 6 present this HCTPN model that provides the information on the alarm base. The Petri subnet also checks if the interrupts are not disabled by the user when calling an API service and that the alarm_id is a valid alarm identifier using function calls written in Romeo language.

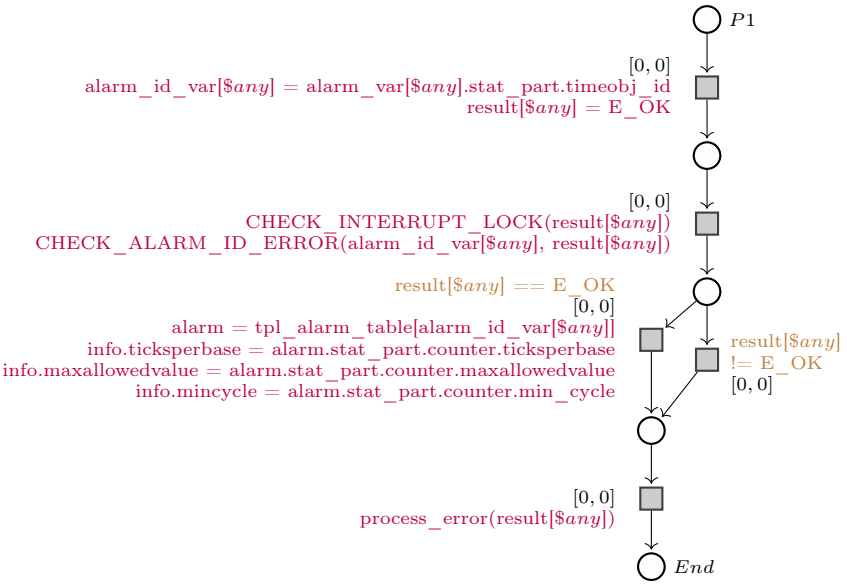


Fig. 6 tpl_get_alarm_base_service model

5.3.3 Properties of the model

In the absence of an application, the model of the OS kernel remains in its initial state. In this section, we study the properties of OS kernel state space when it is called upon by any application.

The variables and the code of the kernel are included in the model.

Let $\mathcal{N} = (P, T, X, C, \mathsf{pre}, \mathsf{post}, (m_0, x_0), guard, update, I)$ the HCTPN model of the OS. The set X is the set of variables of the OS. The state of program pointers is given by the marking. An observable state $s = (M, x)$ of the model is a marking M and a valuation x of X.

By modeling the OS kernel with an assembler instruction per Petri net transition, all the states would be observable and we would get a perfectly equivalent net to the kernel but at the cost of a state space explosion. As explained in Section 5.1, this is not the level of abstraction chosen in the modeling phase. Atomicity avoids this explosion and allows conciseness of the

model, but this means that all the states of the kernel are not observable and are not in the state space of the model.

Recall that all instructions associated with a transition $t \in T$ are executed sequentially (like the actual code of the kernel on a given core) and considered atomic in the state space. To observe a particular state enclosed in a sequence of instructions associated with a transition t, we only need to add a place in the Petri net at the point we want to observe.

The use of colors allows the simultaneous enabling of transitions for different cores. However, the kernel access is sequenced thanks to a global lock. In addition, atomicity is applied on uninterruptible code executed in kernel mode in null time. Hence, the state space of the complete model abstracts the state space of the kernel. We then have the following proposition:

Proposition 1 *Modulo atomicity, the formal model $\mathcal{N}$ and the RTOS kernel have the same state space over the RTOS variables.*

It means that as in (**?**), for any application, $\mathcal{N}$ contains all the paths that might be traversed during the execution of the operating system program.

It is important to note that for another version of the OS kernel without global lock, it would be necessary to ensure that any access to a global OS variable is in a separate transition.

Property 1 *The model $\mathcal{N}$ of the kernel is bounded.*

Proof The variables manipulated by the kernel (and then by the model $\mathcal{N}$) take their values in a finite set i.e. either bounded integers (such as the value of task priority) or enumerated types (such as the state of a task). Moreover the program pointers have a finite number of values hence the markings are bounded and then also the model $\mathcal{N}$. □

Since reachability and TCTL model-checking are decidable for bounded HCTPN, as discussed in Section 4.2, this property is essential to guarantee verification termination. When the HCTPN is not bounded, the number of markings is unlimited, preventing the state calculation from being completed.

5.4 Application modeling

An application contains a concurrent set of tasks that interact with the operating system through system calls such as ActivateTask() or TerminateTask(). An application model consists of two parts: (i) the modeling of tasks and their interaction with the RTOS model and (ii) the automatic generation of data structures for the Roméo tool from the OIL application description.

5.4.1 RTOS and application models interaction

The first part is a set of Petri subnets, one per application task. The network reproduces the control flow graph of the task that accesses the operating system services through system calls. The RTOS service calls are made as explained in Section 5.2. The OS model controls the task running in the model through the function $isRunning(task_i)$ associated with the transition guards. When $task_i$ is scheduled, and the OS is not running, then the function $isRunning(task_i)$ returns 1, allowing the firing of the transition. The $IsReady(task_i)$ guards on Act_{task_i} means that the task model is ready for execution. The transitions contain intervals of the form $[BCET, WCET]$[6] and allow checking the application schedulability and temporal properties as we did in our previous work (**?**). Since our objective is not to study the application's schedulability, time is neglected at the task level, setting the intervals to $[0, 0]$, and all interleaving API system calls are considered.

Figure 7 shows an example of modeling an application task that calls the GetAlarmBase service, which provides information on the alarm base.

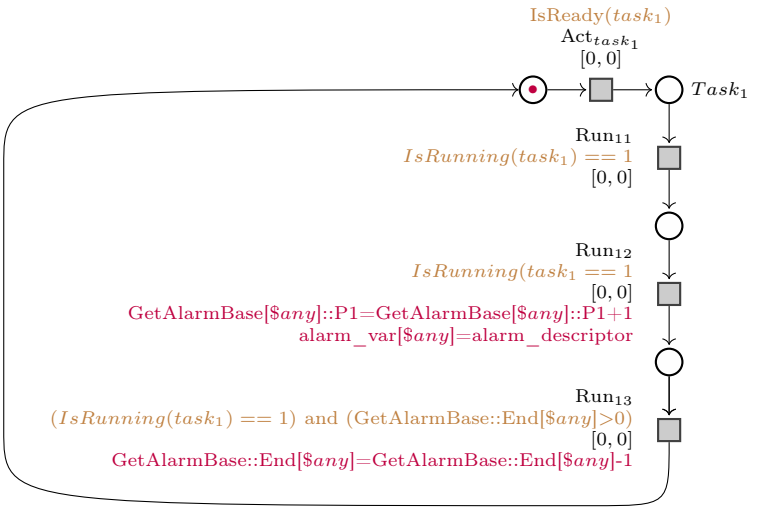


Fig. 7 Application task model. Time is not considered at the task level, setting the intervals to $[0, 0]$.

5.4.2 The GTL module

The second part is the set of data structures corresponding to the descriptors of the different objects in the application: tasks, alarms, spinlocks, etc. These data structures are automatically transposed from the OIL description to the Roméo C-like language.

[6] $WCET$ corresponds to the worst execution time of the code between two service calls, and $BCET$ to the best case execution time of the code between two service calls.

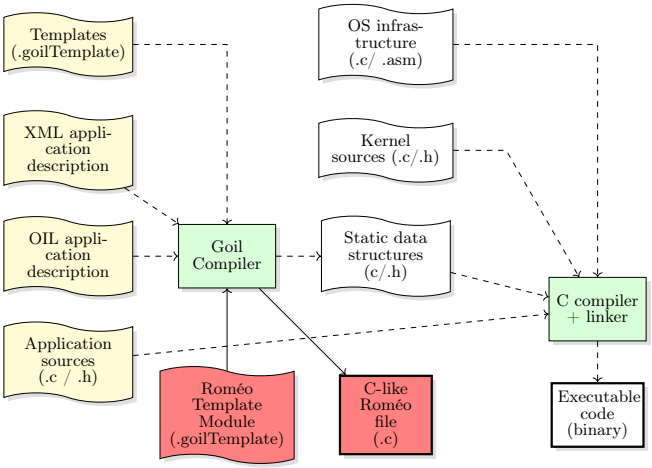


Fig. 8 Trampoline application configuration with the added GTL module.

In practice, we added a module to the Trampoline OIL compiler to automatically generate the structures used by ROMÉO from the OIL description of the application, as shown in Figure 8 in red. Goil compiler includes a template interpreter for file generation with the extension *.goilTemplate*. These template files are created in the Goil Template Language (GTL), which allows the application's configuration data to be combined with text to generate files. The syntax of this language is detailed in the Trampoline OS documentation in the git[7]. The added GTL module is a set of template files that produces the ROMÉO file.

6 Formal verification of Autosar compliance

AUTOSAR conformance testing is based on requirements verification by executing a test suite. In our work, we propose a verification chain that includes the steps presented in Figure 1, page 4. The requirement expression can be simple through an observer modeled by an additional HCTPN associated in a non-intrusive way with the original model without altering its behavior. The satisfaction of the requirement is thus verified by a reachability property of a particular state. This section presents the observer model used to verify multicore OS compliance with the AUTOSAR standard using model-checking. The advantage of this technique is to be able to verify the AUTOSAR requirements on any application and not only on the Autosar test suite by using observer and RTOS models.

[7] https://github.com/TrampolineRTOS/GTL/blob/master/documentation/GTL.pdf

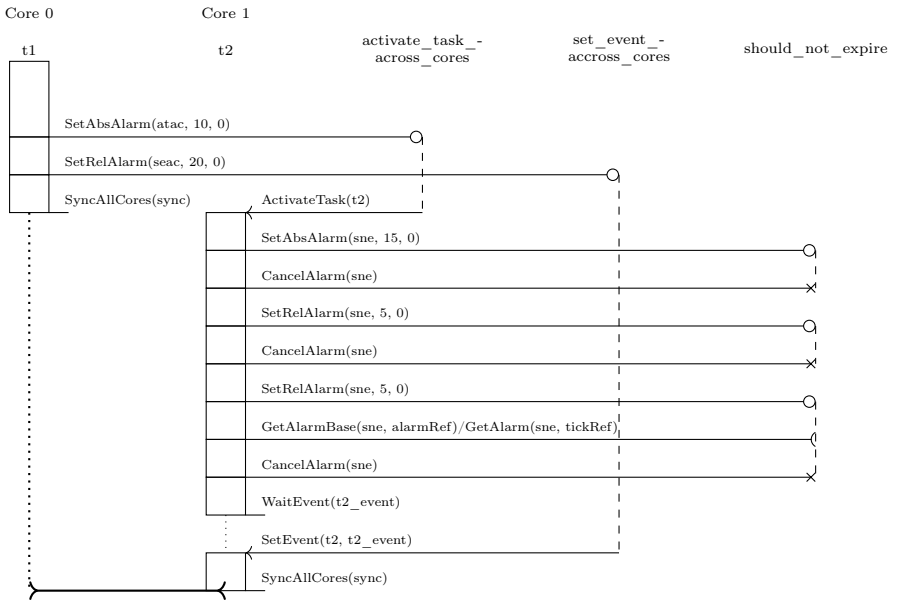


Fig. 9 mc_alarm_s1 test sequence

6.1 AUTOSAR OS tests

The operating system compliance with the AUTOSAR standard is determined at the end of the test suite that comprises a set of applications. The application is a test sequence containing a set of service calls, and each service call represents a test case. When all test cases succeed, the test sequence is verified. Similarly, all test sequences that are completed correctly lead to the success of the test suite, thus verifying the conformance.

We rely on the set of multicore test cases developed by the Trampoline project to verify the OS compliance with the AUTOSAR standard. The project is available in the Trampoline repository[8], and it contains 75 AUTOSAR OS-specific tests, of which 18 are dedicated to multicore. These tests implement a series of test cases that are derived from the requirements listed in (**?**) with good coverage.

We illustrate the first AUTOSAR test sequence of the Trampoline repository, mc_alarm_s1, in Figure 9. This example contains a set of three tasks $\tau = \{t1, t2, should\_not\_run\}$ to be executed on two cores (Core 0 and Core 1), and three alarms assigned to Core 0, $\Lambda = \{activate\_task\_accross\_cores, set\_event\_accross\_cores, should\_not\_expire\}$. Since the $should\_not\_run$ task does not run, it is not shown on Figure 9. This sequence was developed to verify the AUTOSAR requirements from SWS_Os_00632 to SWS_Os_00640 in Table 4. We detail our verification approach on this application in Section 7.1.

[8]They are available in the Trampoline repository: https://github.com/TrampolineRTOS/trampoline/tree/master/tests/functional

6.2 AUTOSAR requirements observers

According to our approach, each AUTOSAR requirement is formalized by an observer able to assess its compliance. Therefore, the AUTOSAR specifications are individually verified by model-checking on an application. The observer is modeled by a Petri net that evolves according to the operating system evolution without altering its behavior. The reachability of the observer's states is examined to verify the satisfaction of the requirement.

How an observer works in the model?

The observer relies on a function written in ROMÉO language, returning a boolean according to the satisfaction of the conditions forming the requirement. The observers are built in a general way. Each requirement is formalized by an observer and translated by a C-like function that returns *true* for each satisfied condition. These functions are called in the RTOS model at locations updating the data structures involved in verifying the requirement. The observer uses a ROMÉO feature called committed transitions, i.e., transitions with a higher priority, to guarantee they are fired before all the other system transitions. Thus, if there are several firable transitions at a given state of execution, the committed ones are fired first before all the others.

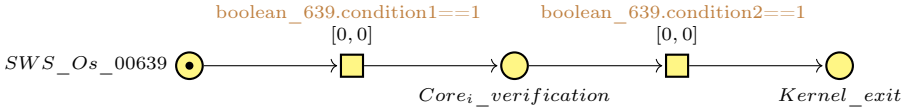
The requirement verification is satisfied if the observer reaches his final place. First, the observer waits in its initial state containing a token until the first condition of its transition guard becomes true to evolve and fire its committed transition. It then moves to the next state to check the second condition until reaching the final state and does not modify, in any case, the RTOS model's behavior or the application. Proper verification of future observer states is ensured by resetting the other conditions of the requirement once the first one is true.

Requirement observer model

Let's consider the observer model of the SWS_Os_00639 requirement (Figure 10), which consists in verifying that the $GetAlarmBase$ service shall also work on an alarm that is bound to another core. This requirement is checked using two conditions during the service call. First, we check whether the core to which the alarm is statically assigned differs from the core identifier on which the service is executed. Then, we verify that the service call finalized its execution and exited the kernel mode. Thus, the test function is called at the beginning and end of the service call using *any* that represents the $core\_id$. The final state of the observer is reached only if both conditions are satisfied. All the observers used are based on the same structure, contain only committed transitions, and do not compromise the evolution of the RTOS model.

6.3 Model-checking with Roméo

Model-checking allows the exploration of the system's state space from its initial state, taking as input a logic formula (such as TCTL temporal logics




```
typedef struct{
        bool condition1;
        bool condition2;
} Requirement_conditions

Requirement_conditions  boolean_639;

Requirement_conditions SWS_639_Observer(int core_id,
Requirement_conditions boolean_639, tpl_time_obj alarm_desc)
{
        // core_id comparison
        if (alarm_desc.core_id != core_id)
        {
                boolean_639.condition1=1;
                // Erasing the past
                boolean_639.condition2=0;
        }
        // Kernel exit if everything was OK
        if((alarm_desc.core_id!=core_id)&&(kerne_mode[core_id]==0)
        &&(result_var[core_id]==E_OK))
        {
                boolean_639.condition2=1;
        }
        return boolean_639;
}
```

Fig. 10 SWS_Os_00639 Observer model

(**?**)) to be verified. Thanks to property in Section 5.3.3, the system model is bounded and reachability problem and TCTL model-checking are decidable with PSPACE-complete complexity (**??**). Requirement in Roméo are expressed in a subclass of TCTL and verified with an on-the-fly efficient algorithm (**?**).

The requirement verification is performed using the logical formula AG((p) implies AF(q)), expressed by the syntax `(p)->(q)`. The formula `(p)->(q)` holds if and only if whenever `p` holds, eventually `q` will hold. Verifying a requirement is written in the form of this response property (AG((p) implies AF(q))), which can be considered a liveness property and be reduced by duality to a safety property. This response property of the generic observer reduces the problem to the verification of two simple atomic properties, p, and q. Some requirements can be checked by $AF(g)$, i.e., A: Forall and F: Eventually, such as requirements SWS_Os_00668 or SWS_Os_00669. However, one can reduce all requirements to the $AG((p) \implies AF(q))$ response property by setting p to true and thus always keeping the same observer.

Based on the observer model of the SWS_Os_00639 requirement (Figure 10) for the first AUTOSAR application test, the corresponding verification formula is as follows: `(SWS_Os_0063[0])->(Kernel_exit[0])`. The token in the initial place of the model triggers the observer once the guard condition is satisfied.

7 Compliance of the AUTOSAR Trampoline OS

The set of multicore test sequences proposed by the Trampoline project is modeled following the procedure detailed in the previous section to conduct a formal verification with the ROMÉO model-checker. This section illustrates the application of our formal verification approach on the two applications *mc_alarm_s*1 and *mc_spinlock_s*1 with the verification results obtained. These examples include several test cases that verify the satisfaction of a set of requirements related to alarms and spinlocks.

7.1 Alarm application

This part focuses on the first multicore test sequence of the Trampoline repository, *mc_alarm_s*1, represented in Figure 9. Tasks are partitioned such that $t2$ runs on Core 1, while $t1$ and *should_not_run* run on Core 0 and task $t1$ has a lower priority than task *should_not_run*.

Initially, $t1$ is an autostart task that runs on Core 0 in the RTOS startup phase. This task calls the API service *SetAbsAlarm* and *SetRelAlarm*. $SetAbsAlarm(AlarmID, start, cycle)$ activates the task $t2$ assigned to alarm *activate_task_across_cores* when its absolute value in *start* ticks is reached. If the alarm is single, *cycle* is equal to zero, otherwise the *cycle* value is greater than 0 in the case of a cyclic alarm. Core 0 must then acquire the kernel lock and set alarm *activate_task_across_cores*. When it expires, the rescheduling is done for Core 1, and as a result, a context switch notification is sent with an inter-core interrupt to execute task $t2$. The *SetAbsAlarm* service call will verify the *SWS_Os_*00632 requirement, checking if an alarm can activate a task on a different kernel. The *set_event_across_cores* alarm activates the assigned event for task $t2$ considered as an extended task with the $SetRelAlarm(AlarmID, increment, cycle)$ service call, after *increment* ticks have elapsed. Once the interrupt sent by Core 0 is considered, task $t2$ starts executing and calls the following services for the *should_not_expire* alarm: *SetAbsAlarm*, *CancelAlarm*, *SetRelAlarm*, and *GetAlarmBase* (Figure 5), ending with the event waiting. Task *should_not_run* assigned to alarm *should_not_expire* will never be executed on Core 1 as the alarm is canceled at the end of the test sequence. This service calls set ensures that they work when an alarm occurs on a different core.

7.1.1 Application model

The developed application model precisely describes the life cycle of each task through the performed system calls. Figure 11 shows the test sequence of task $t1$. This task allows activating $t2$ through the expiration of alarm *activate_task_across_cores* and setting its event by alarm *set_event_across_cores*. Alarm *activate_task_across_cores* is enabled by the *SetAbsAlarm* system call, taking as parameters the required alarm and the expected absolute value to reach for expiry through the alarm_var and start_var variables, respectively. The $t2$ test sequence is modeled similarly

based on the called services. The application model is thus constructed for each test sequence to verify the whole multicore requirements.

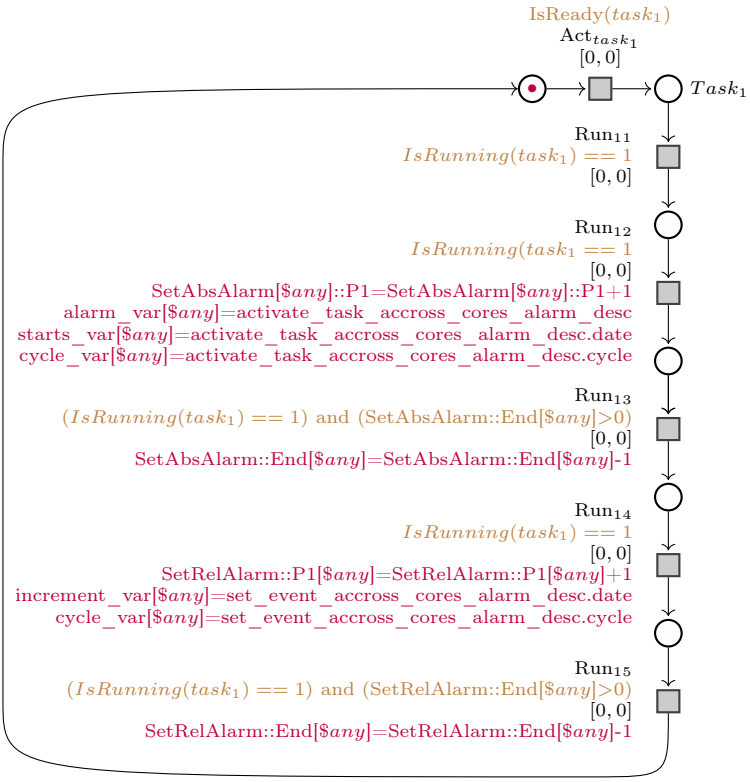
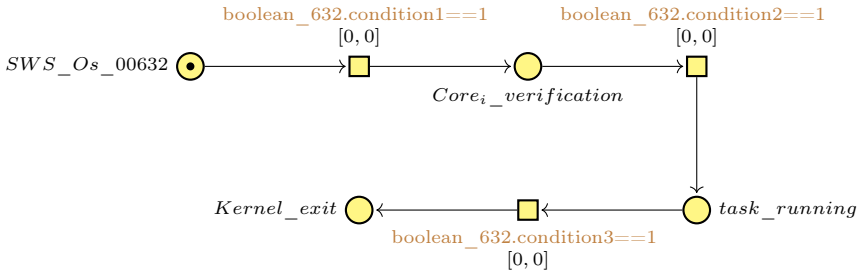


Fig. 11 $Task_1$ model of the mc_alarm_s1 test sequence

7.1.2 Verification results

We apply the verification approach presented in this section to check the requirements covered by this example. We formalize each requirement by an observer model as presented in Section 6.2. The first requirement SWS_Os_00632, for example, is represented by the observer model in Figure 12. It verifies that an alarm can activate a task on a different core. Thus, we must check that alarm $activate\_task\_across\_cores$ assigned to core 0 can activate task $t2$ on Core 1. This alarm is set by the service $SetAbsAlarm$ that task $t1$ calls, as shown in the model in Figure 11. We first verify that the alarm core ID and the task core ID are distinct, then we ensure that the task is correctly activated on Core 1. Two states are observed by the function: the ready state when the task is elected and the running state when it is in execution. Finally, the kernel-mode exit condition is verified after the activation of the task.




```
Requirement_conditions  boolean_632;

Requirement_conditions SWS_632_Observer(int core_id,
tpl_time_obj alarm_desc, Requirement_conditions boolean_632,
tpl_proc task)
{
        // core_id comparison + task is ready
        if ((alarm_desc.core_id != task.core_id) &&
        (tpl_dyn_proc_table[task.id].state_d==READY_AND_NEW)
        or (tpl_dyn_proc_table[task.id].state_d==READY))
        {
                boolean_632.condition1=1;
                // Erasing the past
                boolean_632.condition2=0;
                boolean_632.condition3=0;
        }

        // task is running
        if((alarm_desc.core_id!=task.core_id)&&
        (tpl_dyn_proc_table[task.id].state_d==RUNNING))
        {
                boolean_632.condition2=1;
        }

        // Kernel exit if everything was OK
        if((alarm_desc.core_id!= task.core_id)&&(kerne_mode[core_id]==0)
        &&(result_var[core_id]==E_OK))
        {
                boolean_632.condition3=1;
        }
        return boolean_632;
}
```

Fig. 12 SWS_Os_00632 Observer model

Table 1 shows the verification results of the requirements covered by this application, listed in Table 4. The column *time* (s) refers to the time needed to obtain the model-checker's response. The *memory* (MB) column is the memory consumed when checking the property `(p)->(q)` of the observer corresponding to a requirement. The result column shows that the property is satisfied by the model. The requirements verification is performed in a similar time and memory.

7.2 Spinlock application

The AUTOSAR standard defines the spinlock mechanism for tasks and ISR2s with several locking methods. For example, with the method *LOCK_WITH_RES_SCHEDULER*, the specific pre-declared resource

Table 1 Computing time and memory used for observer verification - mc_alarm_s1. For all verifications, the result is *true*.

$(p) \rightarrow (q)$		
	Memory used (MB)	Computing time (s)
SWS_Os_00632	662.0	12.4
SWS_Os_00633	647.7	12.1
SWS_Os_00636	666.5	12.1
SWS_Os_00637	663.7	12.0
SWS_Os_00638	656.0	12.1
SWS_Os_00639	662.5	12.1
SWS_Os_00640	672.7	12.1

$RES\_SCHEDULER$ is got, and all other processes will be prevented from preempting for the time that the resource is held. Following the methods $LOCK\_ALL\_INTERRUPTS$ or $LOCK\_CAT2\_INTERRUPTS$, all interrupts or OS interrupts are suspended, respectively. Tasks and ISR2s can simultaneously access the kernel by calling spinlock services on different cores. Only one core can acquire a specific spinlock with the *GetSpinlock* or *TryToGetSpinlock* API services. *GetSpinlock*(*SpinlockId*) allows the spinlock to be occupied by the calling core. If another core already takes the spinlock, the tasks or ISR2s wait in a loop, repeatedly checking for the shared lock to become free. *TryToGetSpinlock*(*SpinlockId*, *success*) is similar to *GetSPinlock*, except the busy-waiting if a different core acquires the spinlock. Thus, TryToGetSpinlock returns without waiting for the spinlock release, setting its return variable *success* to $TRYTOGETSPINLOCK\_NOSUCCESS$. The spinlock previously taken by the *GetSpinlock* and *TryToGetSpinlock* services is released using the *ReleaseSpinlock*(*SpinlockId*) service.

We present in Figure 13 the *mc_spinlock_s*1 multicore test sequence of the Trampoline repository. The application contains four spinlocks handled by the services mentioned above, *already_taken*, *not_successor*, *lock_isr*, and *lock_task*, and defines the correct nesting of spinlocks to avoid deadlocks. Task $t1$ runs on Core 0, task $t2$ and the Cat2 Interrupt Service Routine $ISR2$ run on Core 1 such that task $t2$ has a lower priority than $ISR2$. Cat2 ISRs are supported by OSEK and can make OS calls that may cause a rescheduling. Task $t1$ is an autostart task that begins automatically at system start-up. It calls a set of spinlock API services as shown in Figure 13. First, it gets two times the same spinlock *already_taken* with the *GetSpinlock* service and ends with the activation of task $t2$ on core 1. Task $t2$ tries to get the spinlock *lock_task* that Core 0 has. It has an execution budget with protection time enabled. Once its execution budget is consumed, the operating system module calls the *ProtectionHook*() function that sends a software interrupt to the interrupt handler for enabling the core1's $ISR2$. This application verifies the requirements from SWS_Os_00649 to SWS_Os_00661 listed in Table 4.

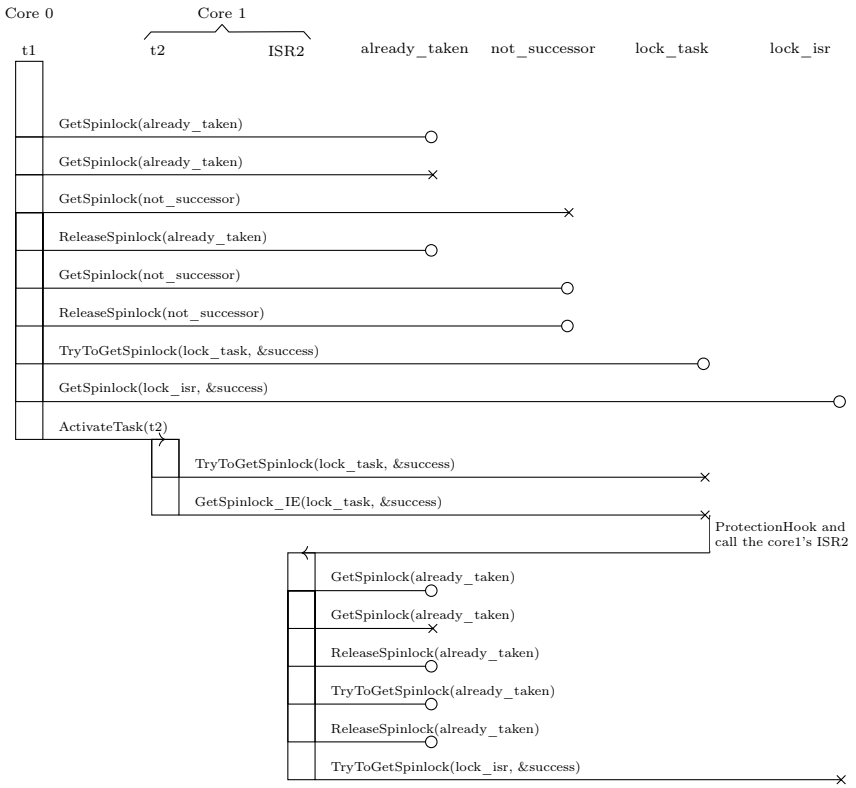


Fig. 13 mc_spinlock_s1 test sequence

7.2.1 Application model

We build the application by HCTPN and a declaration written in Roméo language as presented in Section 5.4. The declaration gathers constants and data structures with their initialization, handled by the application. These pieces of information are extracted from the files generated during the compilation phase. The application model gathers the models of the two tasks and $ISR2$, which constitute it. $ISR2$ is considered a process activated by the software interrupt sent at the end of task t2 execution. Its detailed model is represented in Figure 14. It starts with calling the *GetSpinlock* service twice to acquire the same spinlock *already_taken* and then follows up with a set of service calls. Each service call represents a requirement check scenario. For example, the error $E\_OS\_INTERFERENCE\_DEADLOCK$ is expected on the second attempt to get the spinlock because it already belongs to the calling core.

7.2.2 Verification results

We conduct the verification on the elaborated application model by adding the observers. All the requirements tested by this application are

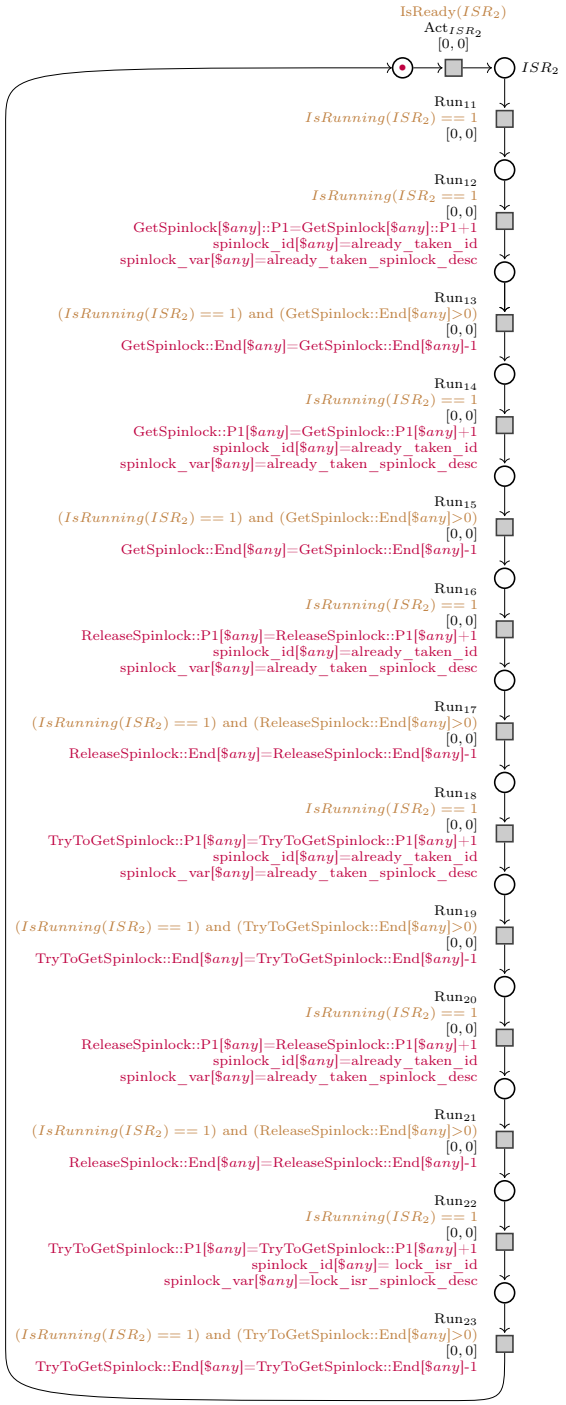


Fig. 14 ISR_2 model of the *mc_spinlock_s*1 test sequence

formalized. Figure 15 shows the observer models verifying the requirements SWS_Os_00650 and SWS_Os_00651. Both requirements concern the ability to call *GetSpinlock* from tasks and ISR2s and are checked through the same function $SWS\_650\_651\_Observer$. The first field of the $boolean\_650\_651$ data structure, $condition1$, triggers the observer of requirement SWS_Os_00650 when the task is running. Similarly, the second field, $condition2$, triggers the observer of requirement SWS_Os_00651 when the $ISR2$ is executed. The observers end their verification with the kernel-mode exit condition via the third field $condition3$, reset with the check of each trigger condition.

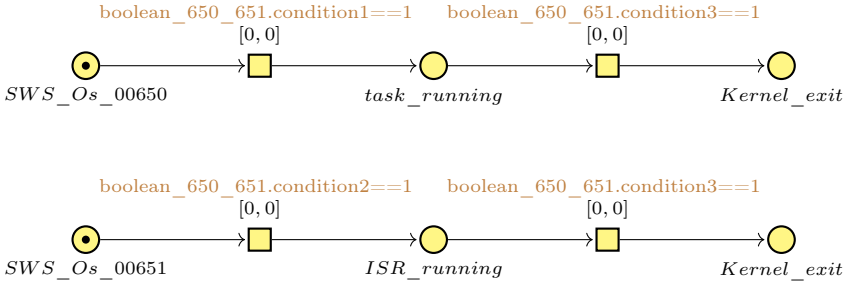
The description of the requirements from SWS_Os_00649 to SWS_Os_00661, verified by this test sequence, are found in Table 4. The property `(p)->(q)` is satisfied by the ROMÉO model-checker for each requirement observer, such that p represents the first place and q the last place of the observer. The verification time in seconds and the memory consumed in MB for each observer verification are included in Table 2.

Table 2 Computing time and memory used for observer verification - mc_spinlock_s1. For all verifications, the result is *true*. The state space size is around 2000 symbolic states with a Lower Bound of 1897 states for the verification of the requirement SWS_Os_00650 and an Upper Bound of 2101 for the verification of the requirement SWS_Os_00654.

$(p) \rightarrow (q)$		
Observer	Memory used (MB)	Computing time (s)
SWS_Os_00649	108.2	2.7
SWS_Os_00650	107.7	2.7
SWS_Os_00651	111.6	2.8
SWS_Os_00652	121.7	2.7
SWS_Os_00653	120.9	2.7
SWS_Os_00654	130.2	2.8
SWS_Os_00655	114.4	2.7
SWS_Os_00656	113.5	2.7
SWS_Os_00657	126.6	2.7
SWS_Os_00658	108.5	2.7
SWS_Os_00659	121.6	2.7
SWS_Os_00661	109.7	2.7

7.3 Discussion

In our verification process, the computer on which the verification is conducted has a quad-core Intel Core i5 processor running at 2.4 GHz and a RAM of 16 GB. We were not confronted with the combinatorial explosion problem of the state spaces. The combinatorial explosion can be induced by the interleaving of multicore scenarios such that all concurrent events are enumerated in the state space, which makes its size vary exponentially. Interleaving is induced by the time added at the application level. We consider all possible interleaving by setting all application model transitions to $[0, 0]$. The AUTOSAR test sequences are sequential and do not involve concurrency, therefore, we did not




```
Requirement_conditions  boolean_650_651;


Requirement_conditions SWS_650_651_Observer(int core_id,
tpl_proc task,tpl_proc isr,Requirement_conditions boolean_650_651)
{
        // Task is running
        if (tpl_dyn_proc_table[task.id].state_d==RUNNING)
        {
                boolean_650_651.condition1=1;
                // Erasing the past
                boolean_650_651.condition3=0;
        }

        // ISR2 is running
        if (tpl_dyn_proc_table[isr.id].state_d==RUNNING)
        {
                boolean_650_651.condition2=1;
                // Erasing the past
                boolean_650_651.condition3=0;
        }

        // Kernel exit
        if(kerne_mode[core_id]==0)
        {
                boolean_650_651.condition3=1;
        }
        return boolean_650_651;
}
```

Fig. 15 SWS_Os_00650 and SWS_Os_00651 Observer models

have interleaving in our verification. However, the model allows checking all possible execution paths and interleaving of service calls (**?**).

Thanks to the expressiveness of the chosen HCTPN model class, all test applications[9] can be easily modeled with the stated observer technique. All AUTOSAR multi-core operating system specifications were met for the fifteen modeled test applications, and the verification time is between 2.7 and 11 seconds consuming between 100 and 600 MB of memory. The response time of the model-checker represents the time needed to explore the set of state spaces and check the property.

[9] *mc_alarms_s1* to *mc_taskTermination_s2* are modeled and verified except for *mc_taskTermination_s1*, *mc_schedtables_s1*, and *mc_autostart_s3* that are in progress: https://github.com/TrampolineRTOS/trampoline/tree/master/tests/functional

The difference between the computation time and memory size obtained when checking the properties is explained by the fact that the state space calculation is distinct for applications. For example, the alarm application leads to the rescheduling of core 1 on core 0. In contrast, there is never any rescheduling in the spinlock application, and a core cannot progress anymore when another core holds the spinlock. That implies a different calculation of the state space.

The time and memory size computed when checking the requirements of an application are almost the same because the response property is true and its verification involves exploring the whole state space. In contrast, if the property is false, the state space computation will stop as soon as possible, and the measures will differ. However, this was not the case in our verification, where the properties were always satisfied.

Several factors helped prevent the exponential computation time or memory size explosion. Among them are the model atomicity, the sequential access to the kernel through a global lock, except for the spinlock services where the cores can access simultaneously, the complexity of the application, and its small number of cores. The approach is efficient for AUTOSAR compliance testing.

8 Conclusion

In this work, we have presented an approach that determines the compliance of the AUTOSAR multicore real-time operating system (RTOS) from its formal model. The RTOS formal model built with a High-level Colored Time Petri Net (HCTPN) embeds the operating system's control flow, variables, and data structures. The application models constructed represent the AUTOSAR multicore test sequences. The RTOS and application models form a complete model that allows performing verification. It describes the deployment of the application on the operating system through service calls. For each application, the conformity of the operating system is verified according to the AUTOSAR specifications. Each specification is formalized with an observer connected to the model that verifies the satisfaction of its conditions. When its final state is reached, the specification they translate is well respected by the operating system during its execution. Reachability verification is thus performed through model-checking by exhaustively exploring the system's state space from its initial state.

We applied our approach to Trampoline's RTOS, an embedded operating system aligned to the OSEK/VDX and AUTOSAR standard. We modeled the set of multicore test cases developed by the Trampoline project and verified the compliance of the RTOS with the AUTOSAR standard.

In future work, we propose to design a Domain Specific Language (DSL) that would be dedicated to the specification of small OS such as an OSEK/VDX or AUTOSAR compatible OS. From this description, the OS source code and the HCTPN model would be generated. In this way, the guarantee that the implemented OS strictly matches the model and that the

checked properties are also respected by the target implementation would be strengthened by removing manual translations between the OS source code and its model.

This research work has been partly funded by ANRT and Huawei Technologies France under doctoral contract CIFRE2019-0798.

Declarations

The authors declare that they have no conflict of interest. The authors also declare that the data used in this work is available on the Trampoline repository under a free license: https://github.com/TrampolineRTOS/trampoline/tree/master/tests/functional.

A Requirements corresponding to multicore OS

SWS_OS_00568	Execute a TASK on each core
SWS_OS_00569	Scheduling on each core
SWS_OS_00602	Possible to set an Event of another core if application has access
SWS_OS_00604	SetEvent's call is synchronous
SWS_OS_00605	SetEvent's error are handled in the calling core
SWS_Os_00622	WaitEvent returns E_OS_SPINLOCK if the calling core has spinlocks
SWS_Os_00624	Schedule returns E_OS_SPINLOCK if the calling core has spinlocks
SWS_Os_00625	GetCoreId callable before StartOs
SWS_Os_00626	GetNumberOfActivatedCores returns the number of activated cores
SWS_Os_00627	Macros OS_CORE_ID_0, OS_CORE_ID_1
SWS_Os_00628	Macros OS_CORE_ID_MASTER
SWS_Os_00632	An Alarm can activate a task on a different core
SWS_Os_00633	An Alarm can set an event on a different core
SWS_Os_00634	An Alarm is processed on the alarm's core
SWS_Os_00635	An Alarm callback is executed on the alarm core (SC1 only)
SWS_Os_00636	SetRelAlarm work on an alarm on a different core
SWS_Os_00637	SetAbsAlarm work on an alarm on a different core
SWS_Os_00638	CancelAlarm work on an alarm on a different core
SWS_Os_00639	GetAlarmBase work on an alarm on a different core
SWS_Os_00640	GetAlarm work on an alarm on a different core
SWS_Os_00641	A schedtable can activate tasks bound on another core
SWS_Os_00642	A schedtable can set an event bound on another core
SWS_Os_00643	Schedtable be processed on its own core
SWS_Os_00644	StartScheduleTableAbs can start schedtable on another core
SWS_Os_00645	StartScheduleTableRel can start schedtable on another core
SWS_Os_00646	StopScheduleTable can stop schedtable on another core
SWS_Os_00647	GetScheduleTableStatus can get the status of a schedtable on another core
SWS_Os_00648	OS Provides a Spinlock mechanism
SWS_Os_00649	GetSpinlock service
SWS_Os_00650	GetSpinlock callable from Tasks
SWS_Os_00651	GetSpinlock callable from ISRS2
SWS_Os_00652	TryToGetSpinlock service
SWS_Os_00653	TryToGetSpinlock callable from Tasks
SWS_Os_00654	TryToGetSpinlock callable from ISRS2
SWS_Os_00655	ReleaseSpinlock service
SWS_Os_00656	ReleaseSpinlock callable from Tasks
SWS_Os_00657	ReleaseSpinlock callable from ISRS2
SWS_Os_00658	Error if trying to get a spinlock that already belongs to the calling core from a task
SWS_Os_00659	Error if trying to get a spinlock that already belongs to the calling core from an ISRS2
SWS_Os_00661	Error if trying to get a spinlock that is not the successor of a spinlock the core already occupies
SWS_Os_00668	All Autostart Tasks are activated
SWS_Os_00669	All Autostart Alarms are activated
SWS_Os_00670	All Autostart Schedule Tables are activated

Table 4 Subset of AUTOSAR OS requirements related to multicore.