AUTOSAR compliant multi-core RTOS formal modeling and verification

Imane Haur

CENTRALE NANTES

THÈSE DE DOCTORAT DE

L'ÉCOLE CENTRALE DE NANTES

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Automatique, productique et robotique*

Par
Imane HAUR

AUTOSAR compliant multi-core RTOS formal modeling and verification

Thèse présentée et soutenue à l'École Centrale de Nantes, le 30/11/2022
Unité de recherche : UMR 6004, Laboratoire des Sciences du Numérique de Nantes (LS2N)

Rapporteurs avant soutenance :

Emmanuel GROLLEAU	Professeur des universités, ISAE-ENSMA, Chasseneuil-du-Poitou
Patrick MARTINEAU	Professeur des universités, Polytech Tours

Composition du Jury :

Président :	Emmanuel GROLLEAU	Professeur des universités, ISAE-ENSMA, Chasseneuil-du-Poitou
Examinateurs :	Isabelle PUAUT	Professeure des universités, Université de Rennes
	Gaétan HAINS	Professeur des universités, Université Paris-Est Créteil
Dir. de thèse :	Olivier Henri ROUX	Professeur des universités, École Centrale de Nantes
Co-encadrant de thèse :	Jean-Luc BÉCHENNEC	Chargé de recherche CNRS, École Centrale de Nantes

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

DEDICATIONS

To my parents, the love you have given me is priceless. All the words in the world cannot express my gratitude. I place in your hands the fruit of long years of study, long months of your love and tenderness, and long days of learning. Your support and encouragement have always given me the strength to persevere in prospering in life.

To Sofia, I know I am not the best sister in the world, but I adore you. You will find a modest testimony of my most sincere affection and deepest attachment in this work.

To my grandmother, you have always treated me as your daughter. I can never thank you enough for that. May God bless you with health, long life, and prosperity.

To my aunts and uncles, I would have liked to thank you all one by one as you deserve, but if I do, my dedications will fill more pages than the whole report.

To Imad, you have always supported me, and no words can describe my gratitude for your encouragement.

To my friends and all the people who have, in one way or another, helped me to get here. May you all be proud of me.

ACKNOWLEDGEMENT

This experience spent within Huawei Paris Research Lab and within the Real-Time Systems team of the LS2N (Laboratoire des Sciences du Numérique de Nantes) would never have been so fruitful without the contribution of some people that I would like to thank particularly:

First, I would like to thank my three supervisors: Olivier H. ROUX, Jean-Luc BÉCHENNEC, and Gaétan HAINS, who knew how to play their roles perfectly. They ensured this project's good progress and managed to provide rigorous and professional seriousness in a friendly and relaxed climate with a disconcerting virtuosity. I am very grateful for their trust and the wise advice they gave me. I am also very honored by the friendship they have shown me, and it is a great opportunity and pleasure for me to work with them during this period.

I want to take this opportunity to thank all the members of the RSD Grenoble team and the Real-Time Systems team of the LS2N for their kindness and the technical and financial resources they have given me to carry out this project in good condition. I cannot forget to thank the members of the CSI Paris team for their welcome, help, and sympathy.

I want to thank the jury members, Mr. Emmanuel GROLLEAU and Mr. Patrick MARTINEAU, who honored me by being the reviewers of this manuscript and for having accepted to evaluate my work. I also thank Mrs. Isabelle PUAUT and Gaétan HAINS, who agreed to examine this project and be a jury members.

Finally, I thank my family, friends, and colleagues who supported me during my thesis years.

Chapter 1

Introduction

Embedded systems are now present in various contexts and applications in our everyday lives. These systems have evolved and need to use increasingly complex hardware and software architectures to achieve the required level of performance. In addition, the demand for high-performance computing among applications has led to a rise in the usage of multi-core chips. Embedded systems are required to implement all of the desired functionalities and validate both the functional and temporal accuracy.

Considering the time, the embedded system is designed as a real-time system for safety-critical reasons. A real-time system interacts with a complex external environment and should meet deadlines, guarantee timing constraints, and consider the correctness of the computation. The design of such a system must thus be predictable, i.e., its behavior must be expected concerning the time requirements. The real-time system has three classifications:

- **Hard real-time system:** Failure to meet the deadline and time constraints in this system has disastrous consequences;
- **Soft real-time system:** This system can occasionally miss its deadline without severe consequences;
- **Firm real-time system:** The system allows infrequently missed deadlines with a specification of which deadlines can be lost. If the system misses more deadlines than defined in the specification, the system fails and can cause serious consequences.

As software-based functions increase, Real-Time Operating Systems, also known as RTOS, become extremely important. This system serves as an interface between software and hardware and provides all necessary functionalities. The RTOS is thus responsible for managing hardware resources and scheduling application processes.

In order to increase confidence and prevent unexpected behavior, system verification is essential. It is crucial to perform time verification and validation to demonstrate that the system will always be able to react according to its time constraints and that the task scheduling sequences will always be appropriate.

The real-time system (RTS) architecture, as represented in Figure 1.1, is composed of a software component, i. e. applications that interact with a Real-Time Operating System (RTOS), and a hardware part that allows its execution.

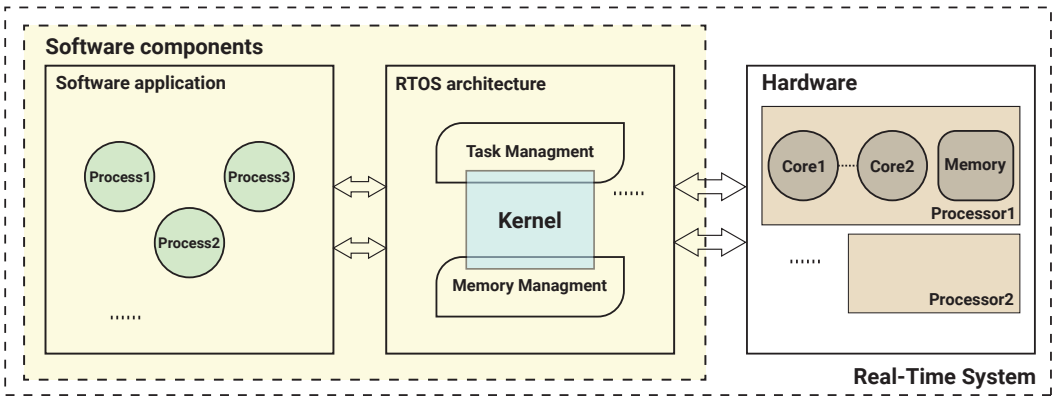


Figure 1.1: Real-Time System (RTS)

Software components In the RTS software application, tasks communicate with each other by exchanging messages or through synchronization mechanisms. The real-time operating system manages the hardware resources and provides a scheduler. The scheduler is in charge of the order in which tasks are executed to ensure the deadlines are respected. The RTOS evolution is formalized with standards; In the avionics and automotive industries, ARINC 653 [1], OSEK/VDX, and AUTOSAR [2, 3] standards propose implementation rules for software and hardware independence and requirements that an operating system must ensure. Thus, standards provide specifications for developing an RTOS with various features, including scheduling policy, memory, and timing protection.

Hardware component The RTS hardware comprises processors, memories, input/output devices, and additional components. The hardware architecture is often categorized according to the number of processors and cores.

Serious incidents can occur if errors are present in the real-time software system. Therefore, developing a computer system that is free of errors is imperative. Testing is a standard method for identifying software program errors. Because it is not exhaustive, it does not guarantee the elimination of all errors. Furthermore, it is insufficient for real-time critical systems where a failure could have disastrous consequences. Formal verification is a solution to increase the implementation reliability of the system. It is a technique that has been recommended as a safety verification method in the ISO-26262 standard for the automotive industry [4].

Verifying that the system's specification properties are satisfied is required to establish that the system is accurate. Safety and liveness are the two most significant types of these properties [5]. While the safety property indicates that an unexpected event will not occur, the liveness property typically implies that a program's finite parts will complete their execution if their input is correct. In addition, it is critical to ensure that the operating system behaves as expected. In fact, the entire system's reliability depends on the operating system that handles its complexity. Therefore, RTOS must adhere to the OS specification to ensure security and functional safety. That emphasizes the need to use a formal approach to provide system verification.

Compliance verification of an OSEK/VDX and AUTOSAR operating system is typically carried out with a test suite consisting of multiple applications. The conformance test suite is thus executed on the RTOS to obtain the standard certification. Contrary to software testing methods that analyze software behavior to reveal defects, formal methods examine the behavior of the software to prove the presence or absence of defects. Formal verification approaches have proven to be highly effective. They allow mathematically proving a system's specification.

1.1 Motivations

We have a double goal: i) Verifying the RTOS behavior by stimulating it with application models such as those proposed for compliance with the OSEK/VDX [6] and AUTOSAR [3] standards. Thus, determining whether or not the RTOS meets the requirements of the standard ii) Verifying the application issues using this operating system, such as real-time multi-core schedulability.

We have mainly three challenges in reasoning about multi-core RTOS behavior: i)The operating system is susceptible to a variety of stimuli, and in particular, real-

time stimuli, such as periodic interruptions generated by timers ii) Applications and code blocks of the OS are executed concurrently on several cores iii) Some parts of the OS code can be executed simultaneously by several cores (StartOS service, Spinlock services).

Choice of the model For this purpose, we use formal methods, particularly the model-checking technique. Model-checking [7] effectively deals with concurrency and interaction between parallel processes, which are the significant sources of error in the systems. These concurrency errors are subtle and complex to reproduce or find in tests because there are many possible interleavings in the parallel processes execution. However, model-checking is ideally suited to identify concurrency bugs and demonstrate their absence in a system. It relies on the algorithmic exploration of the system model's whole state space to verify the correctness of properties on the entire execution path. We apply our approach to the real-time operating system Trampoline [8], a multi-core RTOS compliant with OSEK/VDX and AUTOSAR standards.

As the RTOS we are dealing with is written in C, a model-checker running on concurrent C programs, such as [9], may seem usable, but the call to services goes through assembly code whose formal analysis would require a complete hardware architecture model, and which would not be portable. Furthermore, the goal is not to check only the properties of a C program but to check behavioral properties against real-time stimuli and manage the resulting interruptions. Therefore, we need a model-checker on timed models as we consider the execution times of the application tasks. The execution times of the OS instruction blocks are neglected for the genericity of the approach. The knowledge of the execution time would apply only to a precise hardware target.

A product of timed automata as used in [10] can simulate concurrency and its interleaving. However, it will not be able to model the simultaneous execution of a code sequence on several cores unless the model of this code sequence is artificially duplicated. We will therefore use time Petri nets for concurrency and time modeling. We will extend them with a particular notion of colors so that the same sequence of transitions can be traversed by several tokens, each with a different color modeling the core on which the code is executed.

1.2 Scientific contribution

Our thesis work leads us to three main contributions to achieving the objectives presented in the previous section. The first contribution concerns the chosen formalism for the modeling. Since the control of multi-core real-time systems often requires simultaneous access in true parallelism to shared resources and time Petri nets do not capture these features directly, we propose extending the formalism with colored transitions and high-level functionality, i.e., a predefined syntax manipulating different types of expressions made up of variables. The High-Level Colored Time Petri Nets encompasses both timed multi-enableness of transitions and sequential pseudo code, and the reachability problem is decidable for this model. We then use this extended formalism to model the real-time application as a sequence of RTOS system calls in addition to the multi-core RTOS that reproduces the control flow and uses the same variables as those of the implementation.

The second contribution is composed of two parts. First, the formal verification of the RTOS conformity to the AUTOSAR standard; we model the test suite that includes several applications with High-Level Colored Time Petri Nets (HCTPN). Second, we check the inter-core synchronization mechanism involved in concurrent OS service execution. AUTOSAR conformance testing is based on requirements verification. We focus on multi-core operating system (OS) requirements, for which there are eighty. Each specification is formalized by an observer that evaluates compliance. The observer models are innocuous i.e. they do not interfere with system behavior. Cores are associated with Petri net transition colors. Using the model-checking technique, we verify the AUTOSAR specifications. The approach results conclude that the operating system model respects the AUTOSAR requirements. As part of the AUTOSAR compliance verification of the multi-core RTOS and since the AUTOSAR test cases are synchronous; they do not include concurrent situations, we are interested in verifying simultaneous service calls execution on cores for the safety analysis. Specifically, we rely on the model-checking technique to formally verify multi-core RTOS synchronization mechanisms: concurrent access to OS data structures, multi-core scheduling, and inter-core interrupt handling. That automatically identified two possible errors in the simultaneous execution, proving insufficient data protection and faulty synchronization. Both errors have been corrected and the updated model verified to satisfy AUTOSAR compliance.

Finally, we provide a verification approach to determine the schedulability of real-

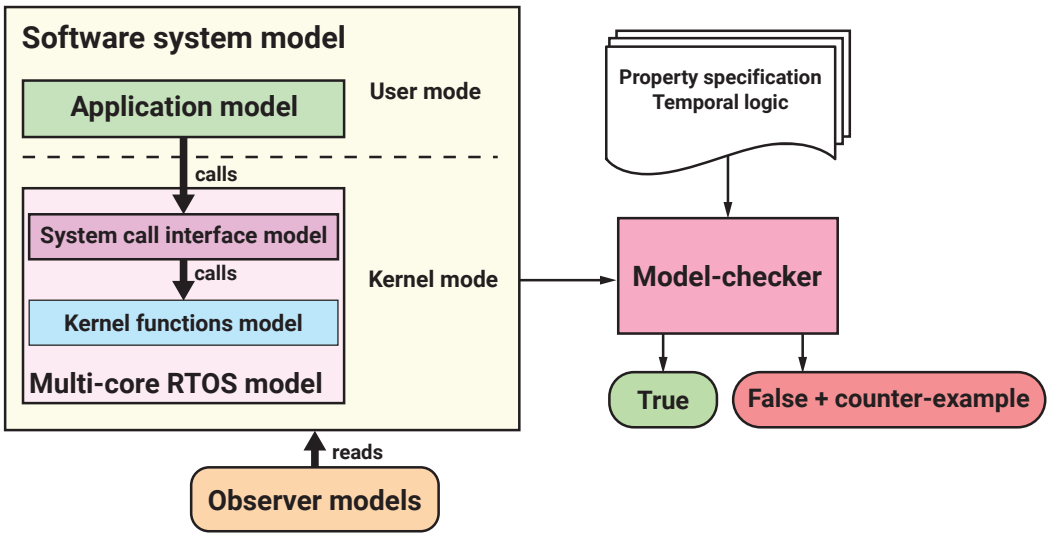


Figure 1.2: Formal verification approach

time applications with dependent preemptive tasks on the detailed multi-core RTOS model. It also allows determining under which temporal conditions the application is schedulable using parameters on the firing intervals. Verification of real-time application schedulability is usually performed using a very abstract system representation, which poorly supports inter-task dependencies. We represent each application task by a 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. We accurately analyze worst-case response time computation for dependent preemptive tasks in multi-core systems.

Thus, our contribution is a complete approach to verifying the schedulability of a real-time system, the AUTOSAR compliance of multi-core RTOS, and the inter-core synchronization mechanism involved in concurrent OS service execution using High-Level Colored Time Petri Nets (HCTPN). The approach steps are illustrated in Figure 1.2. We rely on the Roméo model-checker tool for the verification, available under a free license [11].

1.3 Manuscript outline

The manuscript is structured in two parts:

- **Part I** introduces the general context of the thesis. It starts with a state-of-the-

art in chapter 2 concerning the different formal methods and their application to verifying real-time systems with an RTOS. Chapter 3 presents the Trampoline operating system on which we apply our formal approach. It shows its mono-core and multi-core architectures;

- **Part II** concerns our contribution and is organized into four chapters: (i) Chapter 4 presents the High-Level Colored Time Petri Nets (HCTPN), the extended formalism used for modeling. (ii) Chapter 5 describes the formal model of the RTOS and the application built with the HCTPN formalism using the Roméo tool. (iii) Chapter 6 is dedicated to the verification approach of the operating system's compliance with the AUTOSAR specification based on its formal model. It also includes verifying concurrent situations and the errors proving faulty synchronization with the model-checker. (iv) Chapter 7 presents the schedulability verification of real-time application with dependent preemptive tasks.

The last part of this manuscript is a conclusion and some perspectives on our work.

Part I

General context

Chapter 2

Formal methods for real-time systems

2.1 Introduction

Formal approaches ensure system confidence, and the emergence of new software tools has led to their usability. We examine in this chapter the existing formal verification methods in the literature, focusing on their use for verifying the application's schedulability and real-time operating systems. We rely on the two most popular families of formal methods: theorem proving and 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, in model-checking, we examine whether the desired property is satisfied by exploring the entire state space of the finite constructed model. It is an automatic and efficient technique that can also cope with the problem of state space explosion when the number of states grows to infinity with increasing variables and their distinct values as well as components.

2.2 Formal verification methods

Testing is widely used in practice, although it is clearly impossible to use it in highly critical systems where test data could cause damage if errors are made before actual deployment. Another solution is to simulate the behavior of the system on a computer. The simulation does not work directly on the real system but on a model. A model represents an abstract representation of the real system, usually written using mathematics or logic. Both testing and simulation are widely used in industrial applications, and their use has proven to be very useful. A drawback, however, is that it is not usually possible to simulate or test all possible scenarios or behaviors of a given system.

Here are some examples where testing and simulation failed. The Air France Flight 447 crash in June 2009 caused the death of the people on board. When the plane was flying from Rio de Janeiro to Paris, the storm caused the airspeed sensors to freeze, leading the autopilot to disconnect. The pilots misinterpreted the noise, leading the plane to ram into the sea. The worst is knowing that the crash could have been prevented. In August 2005, The Boeing 777-200 of Malaysia Airlines Flight 124 suddenly and without warning climbed higher than expected. The crew faced a supposedly impossible situation where the stall and overspeed indicators turned on simultaneously. The aircraft landed about 18 minutes into the flight, and the failure occurred in its air data inertial reference unit. One of the two accelerators controlling the airspeed failed, and due to a software anomaly, the second one used incorrect data from the first accelerator [12]. Another example is the failure of the Computer Aided Dispatch (CAD) in the London ambulance service. The inquiry team's investigations show that the system and the resilience of the hardware were not fully tested before implementation [13].

Formal verification, in contrast to testing and simulation, permits the exhaustive investigation using static analysis based on mathematical models to verify the accuracy of hardware or software behavior. Accidents can then be avoided if the systems are verified and analyzed mathematically. Two categories exist, deductive methods based on theorem proving [14] and automatic methods based on model-checking [15].

2.2.1 Theorem-based methods

Formal theorem proving is one of the fastest developing areas in recent years, that verifies the correctness of the system's properties through mathematical reasoning. With the new powerful tools of theorem provers, unsolvable problems several decades ago are being treated today, and new challenges are emerging. Many fields, including computer science [16], biomedical [17], economics [18], machine learning [19], and artificial intelligence [20], have successfully used theorem provers. It provides a statement from a logical set of axioms or hypotheses to check a system's properties defined with mathematical logic. Theorem provers can be divided into two categories:

- **Interactive Theorem Provers** (**ITPs**), known as proof-assistants [21]: this approach allows proofs to be constructed with a reliance on user guidance. It involves human interaction with the tool in the formal proof development process. Coq [22] and Isabelle/HOL [23] are some of the most well-known existing tools.

Their performance is outstanding, and they formalized and proved many theorems in the first hundred theorems list [1].

- **Automated Theorem Provers (ATPs)** [24]: This type consists in building the proofs automatically by the tool without user intervention based on a description of the system to be verified, a set of axioms, and inference rules. Current ATP systems can solve non-trivial problems, such as the Robbins problem [25] solved by the EQP automated theorem proving program for first-order equational logic [26]. In practice, the complexity of most problems is enormous and cannot be solved within resource limits. Thus a significant concern of ATP research is developing more powerful systems that can solve problems within the same resource constraints.

Theorem-proving techniques have limitations, such as the slow process of building proof, even with automatic provers. In addition, most theorems do not support graphical and visualization tools, and logic is not practical as a language. The process requires a high-level of expertise on the user's part, especially for ITPs that require heavy interaction and a lot of energy. These drawbacks are thus an obstacle to adopting theorem proving when dealing with complex systems. However, future works in this direction continue to improve faster and more efficient provers and make them more suitable for the industrial sector by attempting to combine different techniques.

2.2.2 Model-based verification methods

Among the formal methods, 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 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.

The system is described by a model that abstracts the system, most often using state machines such as automata, Petri nets, and process algebras. The choice of model de-

1. The first hundred theorems list is available at: `http://www.cs.ru.nl/~freek/100/`

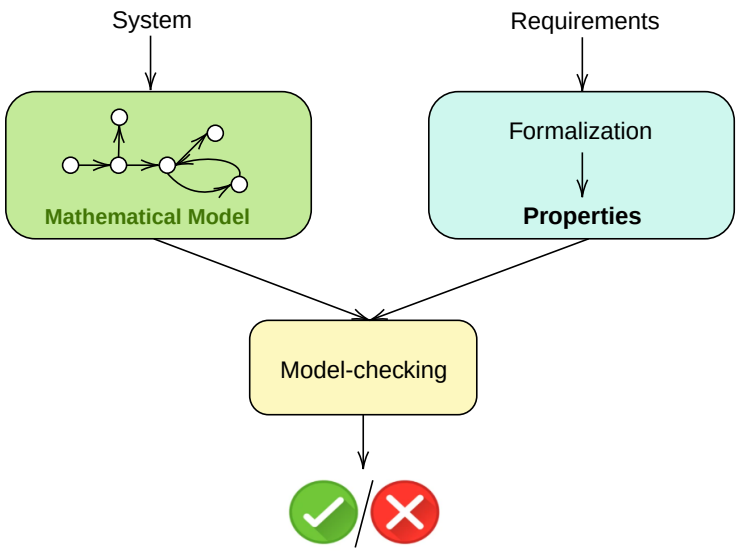


Figure 2.1: Model-checking approach.

pends on its expressiveness, i.e., its capacity to represent many system characteristics. In general, the expressiveness of a model can be opposed to its simplicity of verification. As an expressive formalism can be very helpful in modeling, it can also be blocking in the verification phase. The specification of the system is described by properties that can be expressed in the form of observers of the model or using a particular logic such as Linear Temporal Logic (LTL) [27] and Computation Tree Logic (CTL) or the temporal extension of the latter: TCTL [28]. In [29], L. Lamport decomposes the correctness properties of a system into two categories: safety properties which express that an undesired situation will never happen, and liveliness properties which ensure that under certain conditions, the desired situation will eventually occur. These two categories of properties can also be reduced to a reachability verification that looks for a path where the desired situation is met.

Once the model and its specification are built, an analysis of whether the model satisfies the specification is performed. This analysis explores all possible executions of the system from its initial state. The generation of a counter-example is automatic when the property is false in the form of an execution trace starting from the initial state to the state violating the property. Thus, the model-checking approach is performed on two main phases (modeling and specification verification), as shown in Figure 2.1.

With model-checking, the user does not intervene in the verification process and

easily identifies the states of the system causing its violation through counter-examples generation. The main advantage is, therefore, its automatic character. However, the approach is limited by computing capacities. The problem of the combinatorial explosion is due to the exhaustive exploration of the system's state space. Several reduction studies are proposed to cope with this limitation of exhaustive approaches [30, 31].

2.3 Timed models

Time-based models allow the modeling and verification of real-time applications by considering task execution times and synchronization mechanisms. Adding temporal parameters to the application can restrict its behaviors, limiting the number of states of its model. Moreover, it is necessary to check the quantitative temporal properties to identify specific reasons for failure. The main families of models are extended with time, such as timed automata [32] and time Petri nets [33].

2.3.1 Timed automata

A timed automaton [32] is an extended finite automaton with clocks to consider time. A finite automaton is an abstract machine with a limited number of states that accepts an input alphabet to evolve its state. The values of the clocks increase during the execution of the timed automaton and can be associated with constraints called invariants [34]. The invariants of the system control the duration for which the system can remain in a given location and will leave it once the invariant is no longer satisfied. The clocks are then reset to zero when the transition is fired, and the associated action is completed.

The formalism is supported by several models checking tools [35–37]. Among them, UPPAAL [35] is one of the best-known and most efficient tools. The tool is conceived for the modeling and formal verification of real-time systems using a network of a timed extended finite automaton with useful functions written in the UPPAAL language [38]. Kronos, proposed by S. Yovine [36], is a software tool that allows users to verify the specifications of a real-time system during its design phase. T.Amnell et al. [37] propose the TIMES tool as a scheduling analyzer based on timed automata and their extensions. It supports simulation, formal verification and code generation of the model. TIMES provides a graphical editor that allows the user to specify the parameters of a set of tasks such as priority, deadline and execution time. Nevertheless, the tool does not allow the

analysis of task sets with shared resources.

2.3.2 Petri nets and time

Petri nets have two main temporal extensions: Time Petri nets [39] and the timed Petri nets [40]. Time Petri nets are an extension of the classical Petri net known as a place-transition net, where each transition is associated with a time interval. This interval specifies the possible firing dates. The second temporal extension of Petri Nets is Timed Petri Nets[40] where transitions are fired as soon as possible while a transition can be fired within a given interval for Time Petri Nets. Time is thus represented by minimum (or exact) durations for Timed Petri nets. Time can also be associated to transitions (T-time [41]), places (P-time [42, 43]) and arcs (A-time [44]). T-time Petri nets are the most widely used in real-time systems and those used in our modeling in this thesis project, and they have the same expressiveness as Turing machines [45], contrary to Timed automata. We present the formalism with its color extension in detail in Chapter 4.

Several software environments for analyzing Petri nets with temporal extensions are developed[2], allowing the users to edit the system graphically [46–48]. TiPNet [46] is a tool supporting the analysis of Timed Petri Nets to simulate discrete systems. TINA (TIme Petri Net Analyzer)[3] [47] allows editing, simulation, and building state space abstractions for time Petri nets. Roméo is the tool used in our research work [48], and it allows the analysis of time Petri Nets with different extensions. Its representation is given in Section 4.5.

2.3.3 Timed model with stopwatches

Timed models can be extended with stopwatches instead of clocks to model the temporal interruption of actions and subsequent resumption. Indeed, for timed automata (TA) and time Petri nets (TPN), time elapses at the same speed for all system components. Hence they cannot abstract preemptive scheduling policies where the execution of a task can be suspended and later resumed at the same point.

Several extensions of these models have been proposed to express the suspension

2. The database on `http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html` provides an overview of existing tools for Petri Nets.

3. `http://www.laas.fr/tina`

and resumption of actions by adding the stopwatch notion [49–53]. Stopwatch automata [49] is an extension that allows modeling preemptive real-time tasks. For TPNs, several extensions are proposed: Scheduling-TPN [50, 51], Preemptive-TPN [52], and Time Petri nets with inhibitor hyperarcs (IHTPN) [53]. The first extension [50] is based on adding two new attributes associated with places: the allocation of the processor or resource and the priority of the modeled task. The scheduling policy considered is preemptive with fixed priorities. Preemptive time Petri nets [52] rely on a resource assignment mechanism determining timer progress. IHTPN proposed in [53] controls stopwatches associated with transitions using classical arcs and branch inhibitor hyperarcs. However, modeling a real-time system with preemptive scheduling using timed automata, especially in a multi-core context, is not always simple. It requires an automaton per core, and the application is described by an automata product. Extended Petri nets with stopwatches include a semantic of the behavior of real-time schedulers and can represent parallel or concurrent systems.

2.3.4 Scheduling studies based on timed models

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. Many scheduling studies are based on a representation by timed automata [54–57] or TPNs [58–60].

Timed automata and scheduling Several studies use timed automata for the real-time system's verification context and consider scheduling analysis. G. Behrmann et al. propose in [54] a model of timed automata 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 using UPPAAL's model-checker. The authors in [56] focuse on modeling multitasking applications to verify the worst time execution of the tasks using timed automata. The modeled applications considered non-preemptive tasks and routine service interrupts. The temporal and logical properties of these applications are verified in the UPPAAL model-checker. T. Zaharia and P. Haller [55] present a framework for modeling and verification of embedded microsystems. The mini real-time applications running under a multitasking kernel are described through networks of timed automata, and the properties are specified in UPPAAL's CTL subset. They focused on preemptive

and non-preemptive scheduling tasks with different priorities. Besides, source code is automatically generated. The study presented by the authors in [57] proposes an approach to simulate preemptive scheduling using UPPAAL. They associate temporal diagrams with timed automata by mapping rules to check the time constraints and the deadlock.

Time Petri nets and scheduling Several studies proposed modeling with time Petri to verify complex real-time systems and analyze schedulability. E. Grolleau and A. Choquet-Geniet present in [58] the modeling of complex systems with concurrent actions using colored PNs [61]. Their works, however, do not consider online schedulers, and the PN generates an offline sequence to execute. The work proposed by the authors in [60] shows a formal verification approach for real-time systems with a preemptive scheduling policy, including Fixed Priority and Earliest Deadline First, with the possibility to use Round-Robin for tasks with the same priority. The modeling is done with scheduling time Petri nets and also allows the verification of temporal properties for other scheduling policies. Dianxiang et al. analyze in [59] the scheduling of real-time systems using time Petri nets. Behavioral properties are separated from temporal properties during verification. Behavioral specifications are verified by reachability properties, and temporal analysis is conducted based on absolute and relative trigger domains.

2.4 Formal methods for operating systems verification

Formal verification of real-time operating systems is helpful to guarantee the correctness of the system and to provide proof that the system is well implemented. This is possible nowadays thanks to several tools that have been developed in recent years. Several works have been performed in this context that we mention in the following. The list is not exhaustive, and other studies not discussed in our work may exist. Some are based on formal methods to verify the same objectives as those we have for verifying operating systems compliant with OSEK/VDX and AUTOSAR standards. Among the studies, some do not focus on temporal verification when checking the OS and consider other aspects of correction, such as the absence of deadlock or compliance with standards. Other research works are more interested in verifying temporal properties and schedulability analysis considering the RTOS.

Deductive methods for operating systems verification Existing formal techniques have been utilized in a number of research studies for operating system software using deductive methods. The authors in [62–66] use proof assistants to verify formally a real-time operating system. M. Hohmuth and H. Tews in [62] propose a verification project of the L4 compatible Fiasco microkernel. The verification of the C++ sources of Fiasco is performed by the general-purpose theorem prover PVS. Their approach handles type-correctness and safety proof. The verification of seL4 microkernel in [63] is done inside 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. In [64], the Coq proof assistant is applied to the formalized specification description of FreeRTOS - to verify the correctness of significant properties expressed in Separation Logic. In [67], authors implemented an Earliest Deadline First (EDF) real-time scheduling policy in seL4 microkernel and provided the time management and periodic task model. Fengwei et al. [65] propose a verification framework for preemptive operating system kernels. The framework allows the definition of the model by a specification language and its verification by program logic. All proofs are in Coq, and the verification of the functional correctness of the kernel is performed. Gu et al. [66] develop a certified concurrent OS kernel mC2 using CertiKOS and Coq proof assistant. They verify its correctness and system-call specification.

OSEK/VDX operating systems verification Several studies are conducted on operating systems compliant with OSEK/VDX standard [68–71] . Huang et al. [68] present work on the verification of an operating system's conformance with OSEK/VDX automotive standard. They used process algebra CSP to describe and reason about the code-level of the operating system. The model is implemented and validated in the model-checker PAT. It is a CSP-based tool that supports arrays, variables, and other code-level constructs. The specification conformity of the OSEK/VDX code-level operating system is confirmed. However, not all OSEK/VDX specifications are verified; only those related to task scheduling and resource management are proved. Using the model-checking tool SPIN, Chen et al. [69] propose an exhaustive test generation technique to guarantee conformance with the OSEK/VDX standard. Based on the OSEK/VDX OS design model and the formal test specification, the authors constructed the test models in PROMELA and built their test case generation tool through model-checking. Their approach was

applied only to OSEK/VDX OS's task and resource management functions. ORIENTAIS [70] is an OSEK-compliant real-time operating system certified by the OSEK Certification Group and installed in more than 1.38 million cars in China. The authors focus on the correctness of the OS API and the behaviors of the entire system with CSP models. Tigori et al. in [71] propose to check the conformity of the Trampoline 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.

AUTOSAR operating systems verification AUTOSAR OS verification was the subject of numerous studies [72–75]. The authors in [72] show a formal model-based approach to improve the test coverage for AUTOSAR multi-core RTOS. They first defined the concrete formal model conforming 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. Peng et al. in [73] 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 [74] 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 [75] focus on the AUTOSAR schedule table mechanism. They formally model a schedule table using a transition system and analyze the schedulability.

Trampoline operating system verification Several works are done on the Trampoline RTOS [10, 76–78]. In [76], the authors show 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. The performance of this study is enhanced by Yunja in [77] using embeddedC constructs in PROMELA. The number of states and transitions is reduced, which leads to an improvement in verification costs. The authors in [10] propose a complete model of the RTOS Trampoline in its mono-core version us-

ing extended and timed automata with the UPPAAL tool. This model includes all the functions and services of the OS. 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, a source code configured for the application can be produced. Based on the Trampoline formal model done by Tigori et al. in [10], Boukir et al. [78] integrate the model of the global EDF scheduling and verify the scheduler implementation. 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 Tigori model does not represent the accurate time aspect. Indeed, the execution time is discrete, expressed with invariant and clock variables.

2.5 Conclusion

Many studies have been done on the formal verification of operating systems. The works cited in this chapter come close to our verification approach. Our objective remains mainly different since we seek to propose a verification approach that aims to verify the OS's conformity to the AUTOSAR multi-core standard, the multi-core synchronization mechanisms, and the application schedulability, considering their interaction with the multi-core RTOS.

Among the studies, some do not focus on the temporal aspects of OS verification. It is because some operating systems do not satisfy user requirements in real time. For example, the SeL4 microkernel is based on a round-robin with 256 priority levels, and the threads have no time attributes such as budget, period, or deadline. For some works, the proofs cover neither application-specific properties nor the interaction of applications with the operating system. Other studies focus on verifying the correctness properties and compliance verification. However, research works on the formal verification of multi-core AUTOSAR RTOS are limited.

Chapter 3

Trampoline real-time operating system

3.1 Introduction

In this chapter, we first introduce the automotive standards OSEK/VDX and AUTOSAR for implementing real-time operating systems. Then, we present the Trampoline real-time executive on which our thesis work is based. This RTOS respects the OSEK/VDX and AUTOSAR standards. We describe its mono-core and multi-core architecture and present examples of operating system calls in both versions.

3.2 The OSEK/VDX standard

OSEK (Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug) stands for "Open Systems and their corresponding Interfaces for the Electronics in Motor Vehicles". The standard was created in 1993 by a German automotive company consortium (BMW, Bosch, DaimlerChrysler, Opel, Siemens, and Volkswagen Group) and a department at the University of Karlsruhe. In 1994, the French car manufacturers Renault and PSA joined the consortium, developing a similar project called VDX (Vehicle Distributed eXecutive). OSEK/VDX is widely recognized in the automotive industry, and several parts are proposed for the standard. The architecture shown in Figure 3.1, page 36, shows the three elements in the list related to the code that runs on targets:

- Network Management (NM) for automotive embedded systems. The network layer provides reliable communication between vehicle networks and services for the segmented transfer of application messages. It has two types of management mechanisms: direct NM, and indirect NM;

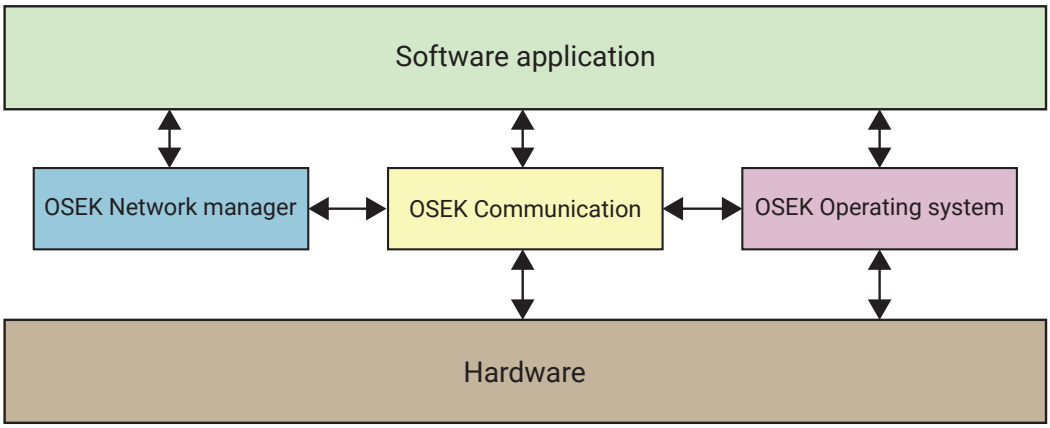


Figure 3.1: The three modules of the OSEK/VDX standard.

- Communication stack and data exchange between control units. This Interaction layer provides communication services for the transfer of application messages;
- Real-time operating system.

We will present the mono-core OSEK/VDX real-time operating system in the following.

3.3 The OSEK/VDX OS

The OSEK real-time operating system specification is a support that helps real-time applications and the embedded software used by electronic control units meet their execution requirements. It is a mono-processor operating system for distributed automotive control units. Its specification [2] describes a uniform environment for the electronic control unit software application to efficiently use resources. The OSEK operating system offers various services, processing levels, and conformance classes for the portability of software applications and the reduction of development costs. For the same purpose, OSEK defines an OSEK Implementation Language (OIL) for standardized configuration information. This language allows a portable description of all OSEK-specific objects such as tasks, alarms, events, etc.

3.3.1 Operating system services

OSEK-OS offers several services that are statically defined at the compilation phase. The task management and interrupt handling services are detailed below.

3.3.1.1 Task management

The task manager handles the activation, scheduling, and synchronization in addition to the termination of the application's tasks. In OSEK, two types of tasks can be defined: basic and extended.

Basic tasks The basic tasks release the processor in two cases: When they complete their execution or if they are pre-empted by other higher priority tasks. Thus, basic tasks have three states: suspended, ready, and running, as shown in the state model in Figure 3.2, page 38:

- Suspended: This is the task's passive state where it can be activated;
- Ready: Activated or preempted tasks are ready to run and waiting for processor allocation. The scheduler chooses which task to execute;
- Running: Only one task can be in this state, executing its instructions.

Extended tasks The execution of an extended task can be waiting for an event when using the operating system call *WaitEvent*. An *event* is an object defined by the application and assigned to the extended tasks to communicate binary information or synchronize tasks. The event mechanism induces the states of tasks to and from the *waiting* state, as represented in Figure 3.3, page 38. That distinguishes extended from basic tasks. Thus, when the state of the calling task is set to *waiting*, the task releases the CPU and awaits an event. It can also wait for a resource or use a blocking instruction.

Task priority The priority of a task is defined by a numeric value, with 0 representing the lowest priority and greater values representing higher priorities. Different tasks can share the same priority level; in this case, they can start considering their activation FIFO order, and the oldest task on the same priority list is the first to be processed. Figure 3.4, page 39, shows several tasks of different priorities in the ready state. Depending

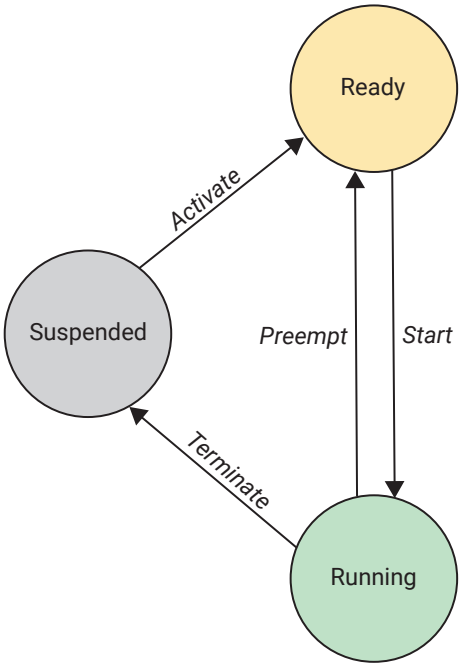


Figure 3.2: State model of a basic task.

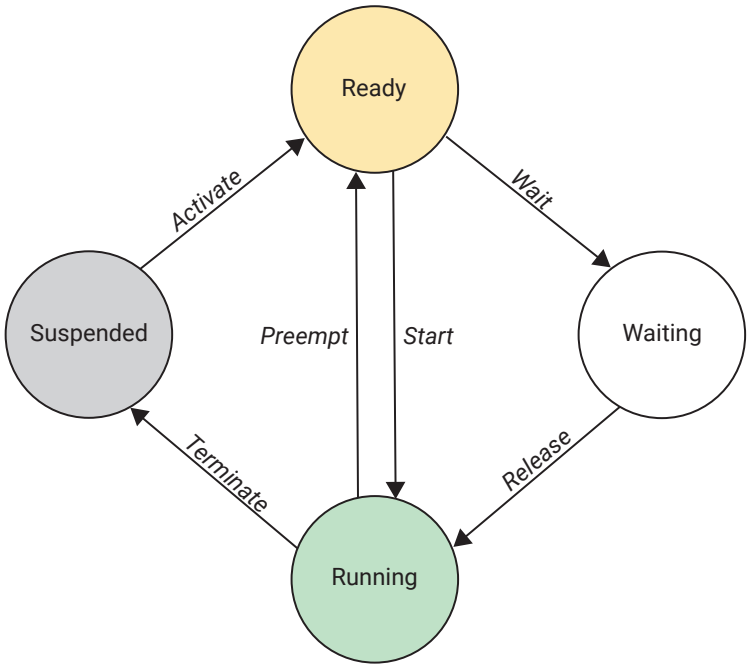


Figure 3.3: State model of an extended task.

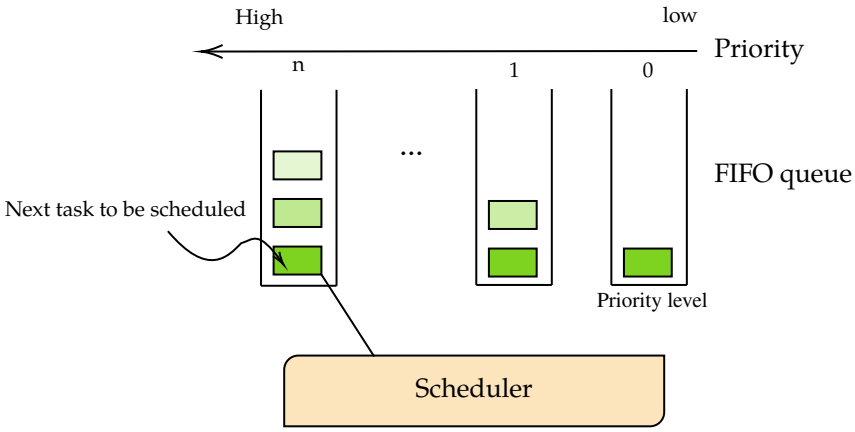


Figure 3.4: Tasks priority.

on the task's priority, the scheduler selects the next task to be processed and decides which ready task will become running.

Scheduling policy The OSEK standard defines a fixed priority scheduling policy. The tasks can be preemptive, non-preemptive, or belong to a task group; a task group is a set of tasks that are non-preemptable by each other but can be preemptable by higher priority tasks in the application. Tasks in the group can share a common internal resource. An internal resource is a resource that is automatically taken by a task when it enters the running state, except if it has already taken the resource, and is released when the task terminates. Priorities and preemption of tasks are statically assigned as attributes of tasks. Therefore, the type of scheduling depends on the preemption attribute, regardless of the task type (basic or extended). Three possible scheduling types are defined in OSEK.

- **Full preemptive scheduling**: The scheduler can preempt a running task to allocate the CPU to a higher priority task;
- **Non preemptive scheduling**: The running task must complete its execution or be waiting for an event to allow a higher priority ready task to start its execution;
- **Mixed preemptive scheduling**: Scheduling can be both preemptive and non-preemptive. It depends on whether the task is defined as preemptable or non-preemptable.

API services The OSEK operating system provides several services; here are some services grouped by category:

Tasks ActivateTask, TerminateTask, ChainTask, GetTaskState, GetTaskId;

Events synchronization between tasks. WaitEvent, SetEvent, ClearEvent, GetEvent;

Resources for single-core critical sections. GetResource and ReleaseResource;

Schedule table Similar to alarms but can only be bound to a counter assigned to the same core. StartScheduleTableAbs, StartScheduleTableRel, GetScheduleTableStatus, StopScheduleTable;

Application GetApplicationID, TerminateApplication, GetApplicationState.

Some of these OSEK operating system services ensure task management, such as:

- *ActivateTask* is used to activate a suspended or new task, setting its state to READY;
- *TerminateTask* allows terminating the running task, and its state is changed from *Running* to *Suspended*;
- *ChainTask* activates a task after the termination of the calling task;
- *Schedule* permits the execution of the ready task with the highest priority. If the calling task uses an internal resource, a rescheduling [1], occurs.

3.3.1.2 Interrupt handling

Interrupt processing is performed by an Interrupt Service Routine (ISR), which is scheduled by the hardware and classified into two categories:

- **ISR category 1**: These interrupt service routines do not use any operating system services, except services that enable or disable interrupts. The interrupt does not influence task management. The program processing resumes at the instruction where the interrupt occurred once the ISR is completed;
- **ISR category 2**: This ISR category can call operating system services with restrictions on their use. After the ISR category 2 has been terminated, rescheduling is performed if a preemptable task has been interrupted and no other interrupt is active.

1. Calling the scheduler another time.

3.3.2 Processing levels

OSEK-OS defines three levels of processing: (i) interrupts, (ii) scheduler, and (iii) tasks. The processing of interrupts has the highest priority, and the scheduler, defined as a logical level, has a higher priority than the tasks. At the task level, tasks are scheduled according to the user's statically assigned priority and the interrupt processing can consist of several interrupt priority levels.

3.3.3 Conformance classes

The Conformance Classes (CC) define multiple real-time executive versions to meet application requirements. The OSEK operating system has four conformance classes that allow it to be compatible with diverse applications and hardware. They depend on the multiple requests of task activation, task types (basic or extended), and the number of tasks per priority. The following are the four distinct levels:

- **BCC1**: This class contains only basic tasks with one task per priority and one activation request per task;
- **BCC2**: Like BCC1, it allows multiple task activations and the possibility of more than one task per priority level;
- **ECC1**: Like BCC1, it also supports extended tasks;
- **ECC2**: Like ECC1, it is possible to have multiple activations of a task and multiple tasks per priority level.

3.4 The AUTOSAR standard

AUTOSAR(AUtomotive Open System ARchitecture) is an industrial standard for automotive software architecture, created in 2003. The objectives include scalability to different vehicles, software portability, and the product's lifecycle maintainability. AUTOSAR is an evolution of OSEK/VDX standard for the specification of operating systems [3]. Unlike OSEK/VDX, it defines a multi-core design that implements a partitioned scheduling policy with fixed priority. Partitioning is obtained by assigning the objects managed by

the OS (tasks, ISR, alarms, schedule tables, events, resources, ...) to an OS-Application[2], and the operating system module schedules processor resources for OS-Applications bounded to cores. Partitioned scheduling involves statically allocating tasks to a ready list per core before being scheduled and cannot migrate.

AUTOSAR architecture

The architecture proposed by AUTOSAR consists of three software layers that are executed on a microcontroller, as shown in Figure 3.5, page 43:

- **Application Layer**: This layer includes the various software components of the application that interact with the runtime environment and contain the functions to be executed;
- **Runtime Environment**: It is a middleware that supports abstract communication between the different application software components and between the Basic Software and the applications;
- **Basic Software Layer**: It provides the necessary services to the upper software layer and consists of several sublayers. The service layer is the highest sublayer and contains various functions such as the Autosar OS, vehicle network communications, and management memory service. The Microcontroller Abstraction Layer (MCAL) accesses the peripheral modules of the hardware. The ECU Abstraction Layer is an interface to the MCAL and provides access to peripherals and devices. The complex Drivers layer can access the microcontroller directly, and are used mainly for complex functions not found on other layers.

3.5 Trampoline RTOS

Trampoline is a Real-Time Operating System (RTOS) developed by the STR (Real Time Systems) group of the LS2N laboratory in Nantes, France. This operating system is OSEK/VDX 2.2.3 and AUTOSAR 4.0 compliant [8]. Trampoline is mostly written in C

2. An OS-Application is an object allowing to gather tasks, ISR, ... to assign them to a computing core but also to restrict interactions between objects belonging to different OS-Applications in order to improve security and safety.

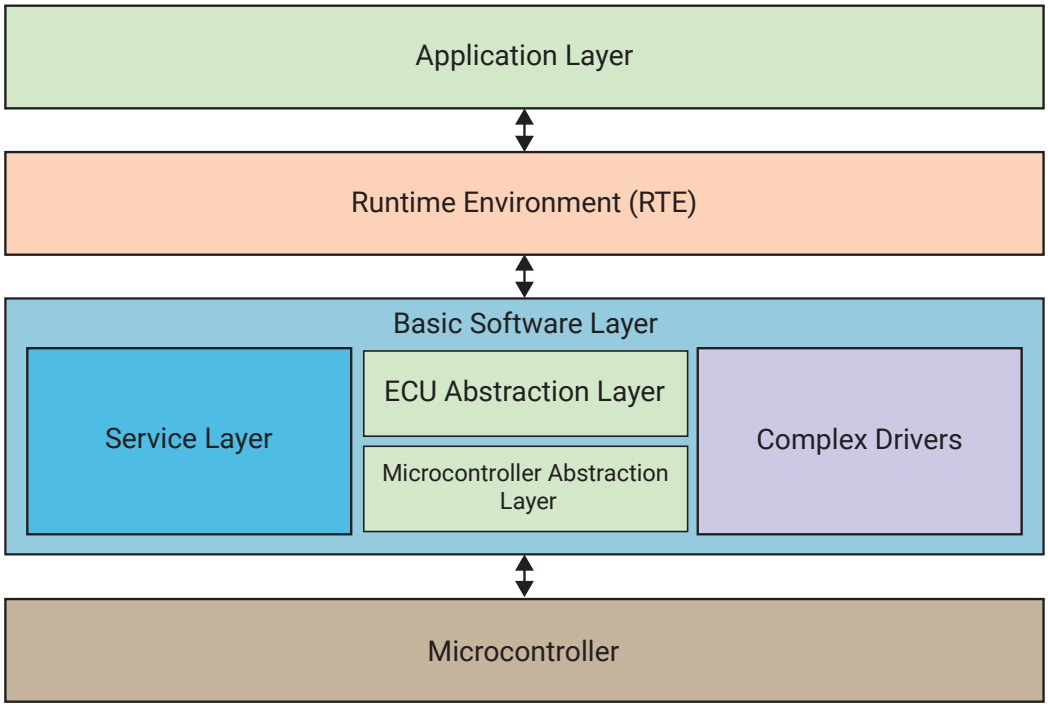


Figure 3.5: AUTOSAR architecture.

language with some parts, like context switching, written in assembly language because they depend on the Instruction Set Architecture (ISA) of the microcontroller. The source code is over 20,000 lines long and includes 180 functions for the target-independent part. The 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.

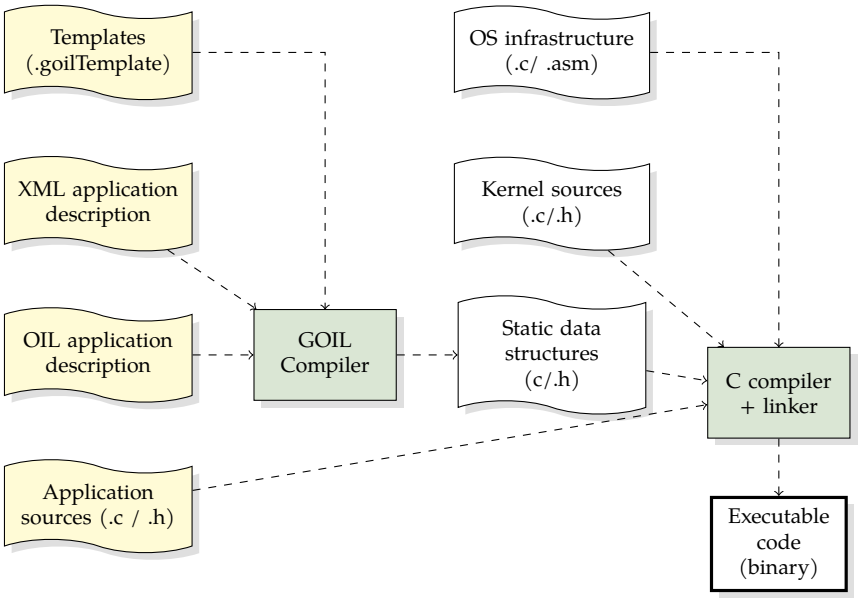


Figure 3.6: Trampoline application configuration.

The OSEK/VDX and AUTOSAR OS are configured according to the application. The objects necessary for the application, tasks, ISRs, alarms, ... are described with their relations in the dedicated language OIL for OSEK/VDX 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 3.6, page 44.

3.5.1 Mono-core Trampoline architecture

The mono-core architecture of Trampoline consists of three components, as shown in Figure 3.7, page 45, which are as follows:

1. **The API (Application Programming Interface)** includes the services defined by OSEK/VDX and AUTOSAR standards. AUTOSAR OS adds services to OSEK/VDX, limiting some configurations. The 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 [79], or alarms that are used to implement periodic tasks or to set an event;

2. The **Kernel** contains all the low-level functions on which the API services are

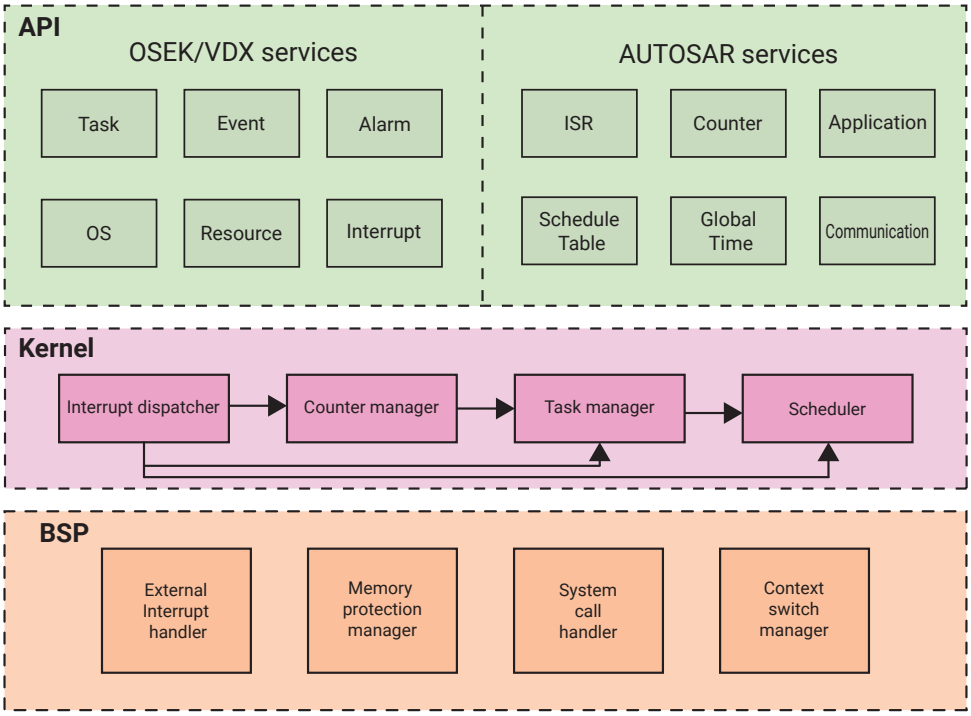


Figure 3.7: Mono-core Trampoline architecture.

based. These C-language-implemented functions allow the management of tasks and their scheduling and ensure the handling of counters and treatment of interrupts;

3. The **BSP** (**Board Support Package**) is the low-level function part of the Trampoline that depends on the target machine. It contains modules that handle calls to API services and context switch of tasks, implemented in assembler. In addition, there is a component to handle interrupts from external sources like a timer and a memory protection module to manage access rights to the operating system memory.

Start-up The OS starts with the `StartOS` service call in the AppModeID application mode. The operating system first performs some hardware-specific and application initializations. Some tasks can be defined as AUTOSTART; if they exist, they are activated. Autostart tasks are ready to be executed after the operating system's start-up. Then Trampoline calls the scheduler, and the CPU will be allocated to the autostart task with the highest priority. If no task is available, a particular idle task runs. Its priority is set

to 0 (i.e. the lowest priority in the system), while the application tasks have a priority greater than or equal to 1.

Scheduling In OSEK/VDX, the scheduling policy uses a static priority assigned to each task which can be in 4 states: suspended, waiting (if extended), ready, and running. In Trampoline, two extra states are used for internal management: Autostart and ready_and_new. An autostart task is automatically activated when `StartOS` is called. The `ready_and_new` state is used for a ready task, but its context is uninitialized. That occurs when the task has just been activated for the first time.

The scheduler of Trampoline manages a ready task queue dynamically during the execution of the application according to their priority and whether they are preemptable or not. This queue is a FIFO list table, as shown in Figure 3.4, page 39, where the tasks are ordered by priority or activation order if their priorities are equal. The priority and the preemptability of a task are set in the OIL task description using the `PRIORITY` and the `SCHEDULE` attributes, respectively. The scheduler uses functions to handle the ready list of tasks, such as `tpl_put_new_proc`, which puts a new process in a ready list, `tpl_put_preempted_proc` to put a preempted process in a ready list, and `tpl_remove_proc` for removing all the process instances in the ready queue.

The scheduler also manages a data structure called `tpl_kern`. This implemented structure contains all the data about a task during its execution. It points to the static and dynamic task descriptors represented in the C data structure file (Listing 3.1, page 46), storing all the information describing the task in memory (identifier, priority, type, state, etc.). The static descriptor contains the data that is not susceptible to vary and stored in the ROM (Read-Only Memory). The dynamic descriptor contains information that is updated during execution and which can be stored in RAM (Random Access Memory). `tpl_kern` thus gathers several pieces of information: the currently running task, the task that has been selected to replace the currently running task (if any) after a rescheduling, a flag indicating if a rescheduling shall be done (`need_schedule`), a flag indicating if a context switch shall be done (`need_switch`) and, finally, a flag indicating if a context save shall be done (`need_save`). The stored information is detailed below:

Listing 3.1: tpl_kern data structure

```
typedef struct
{
```

```
/* Pointer to the static descriptor of the running/elected task */
        P2CONST(tpl_proc_static, TYPEDEF, OS_CONST) s_running;
        P2CONST(tpl_proc_static, TYPEDEF, OS_CONST) s_elected;

/* Pointer to the dynamic descriptor of the running/elected task */
        P2VAR(tpl_proc, TYPEDEF, OS_VAR) running;
        P2VAR(tpl_proc, TYPEDEF, OS_VAR) elected;

/* The identifier of the task being executed/elected */
        VAR(sint32, TYPEDEF) running_id;
        VAR(sint32, TYPEDEF) elected_id;

/* This field indicates a context switch is needed after calling a service */
        VAR(uint8, TYPEDEF) need_switch;

/* Boolean used to notify a rescheduling should be done */
        VAR(tpl_bool, TYPEDEF) need_schedule;

/* Boolean used to indicate a context save of the process that loses the
* CPU should be done*/
        VAR(tpl_bool, TYPEDEF) need_save;

} tpl_kern_state;

tpl_kern_state tpl_kern;
```

The scheduler allows the manipulation of state transitions (Figure 3.3, page 38) via the following functions:

- *tpl_start*(): Start the highest priority ready job at the top of the ready list. Its information is copied into the elected attributes of the *tpl_kern* structure.
- *tpl_preempt*(): Preempt the running process, and its state is switched from the running state to the ready state.
- *tpl_run_elected*(): The elected process becomes the running process, and the running process is preempted. This function copies the elected attributes of *tpl_kern* into the running attributes.

All the above functions that manipulate the task states are called by the *tpl_schedule_service* function, which calls the principle scheduling function *tpl_schedule_from_running*[3]

3. *tpl_schedule_service* calls *tpl_start*() and *tpl_preempt*()

and then performs a context switch to run the elected process.

Context switch The context switch is performed when the scheduler indicates that it is necessary through the `need_switch` flag of the `tpl_kern` data structure. For example, the context switch is performed when a task is preempted, its context is saved, and the elected task's context is restored. The scheduler updates the elected task in the `tpl_kern` structure. In task termination, it is necessary to perform a context switch without saving its context. Thus, the `need_switch` field takes the value `NO_NEED_SWITCH` when no context switch is required and the `NEED_SWITCH` value when it is necessary to perform a context switch. `NEED_SAVE` indicates the need to save the context of the process that loses the CPU and restore the elected one. The context switch is done between the running and the elected task when they are different. The elected attributes of the `tpl_kern` structure are copied to the running ones in the `tpl_run_elected` function called at the context switch.

Calling operating system services In Trampoline, the call to an OS service is made in a wrapping function through a dedicated instruction of the microcontroller[4] which, acting as a software interrupt, transfers the execution to a system call handler in supervisor mode. The latter is in charge of calling the corresponding kernel function. In the process, the interrupts are masked. That prevents an interrupt routine calling the OS from running simultaneously and the internal data structures of the operating system from being corrupted.

The execution of service without a context switch in the mono-core implementation is shown in Figure 3.8, page 49. The execution of service when there is a context switch to a newly activated task in the mono-core implementation is given in Figure 3.9, page 49. `sc` is the service call in the PowerPC instruction set and `rfe` is the return.

Two data structures are used to manage rescheduling and context switching: the list of ready tasks and the `tpl_kern` structure. `tpl_kern` gathers several pieces of information represented in Listing 3.1, and its flags are reset in the system call handler before calling the kernel function in the call sequence shown in Figure 3.9, page 49. If the kernel function performs an operation that adds a task to the list of ready tasks or if the task being executed ends (TerminateTask or ChainTask services), `need_schedule` is set, and the kernel function ends by performing a rescheduling. If the rescheduling leads to a context switch, the `need_switch` flag is set, and the context switch is performed at

4. at least for the microcontrollers that have it

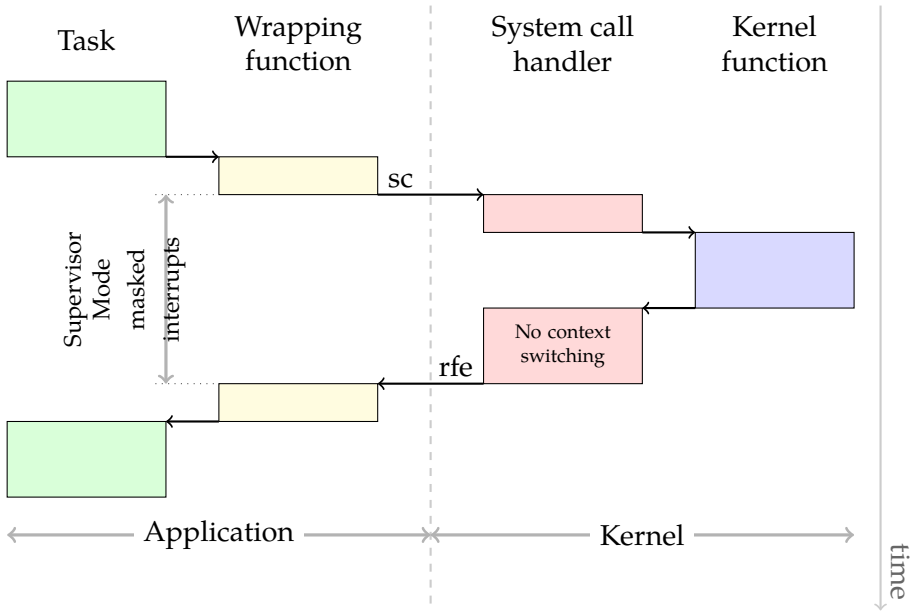


Figure 3.8: Flow of a service execution without context switching.

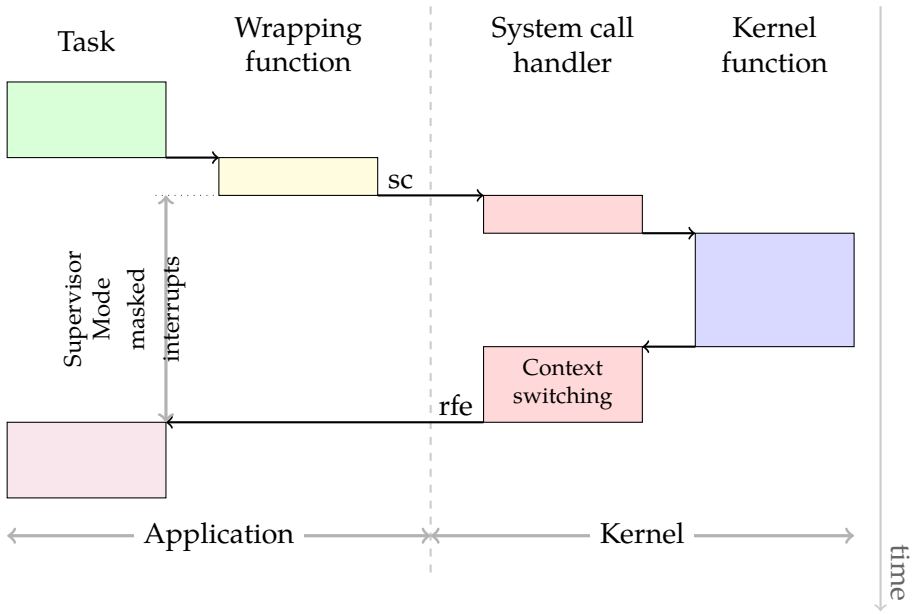


Figure 3.9: Flow of a service execution with context switching.

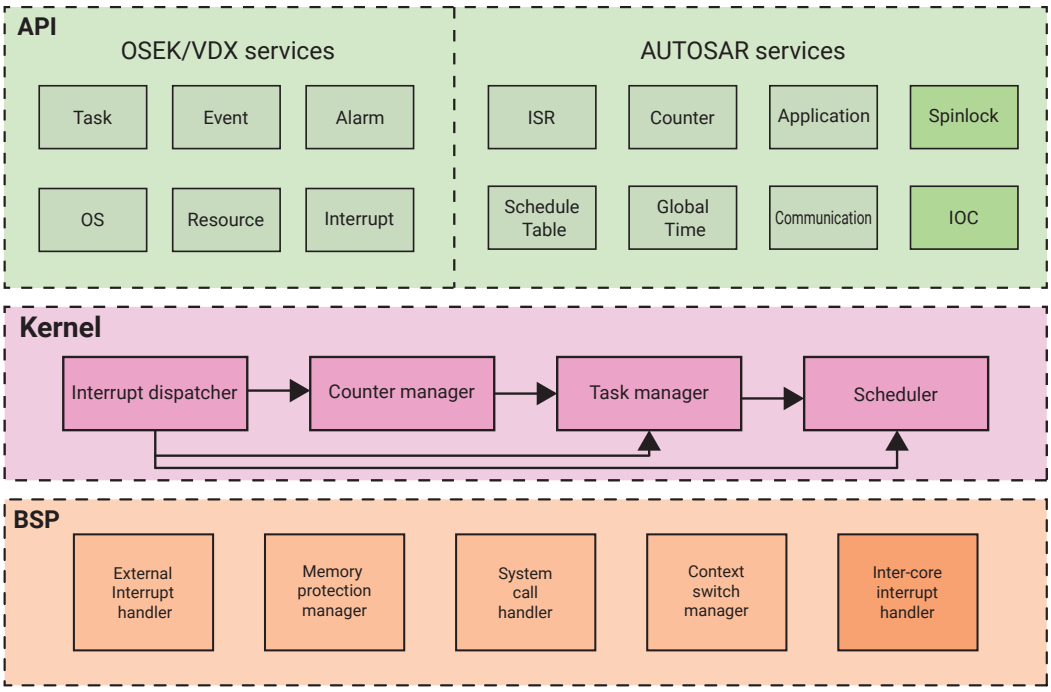


Figure 3.10: Multi-core Trampoline architecture.

the end of the system call handler just before returning to the newly scheduled task.

3.5.2 Multi-core Trampoline architecture

The multi-core version of the Trampoline presents an extension of its mono-core architecture shown in Figure 3.10, page 50. Lock services and an inter-core interrupt handler have been added to deal with the simultaneous execution of the kernel and services on two or more different cores, as well as inter-core interactions and communication. The IOC module provides communication between OS-Applications and transmits data based on the RunTime Environment (RTE) configuration information.

Multi-Core start-up In the multi-core configuration of Trampoline, the cores are distinguished by their identifier and start with the master-slave system. The master core is activated first, and then the slave cores are started. All cores are synchronized through two barriers with the StartOS service call. The first synchronization point occurs before the operating system is initialized, and the OS-application-specific StartupHooks are executed on the core to which they are bound. The second barrier is after the end of

StartupHooks and before the scheduler starts.

The global lock In the multi-core implementation, interrupt masking is not enough. Indeed, it is necessary to prevent the OS kernel from running simultaneously on two or more cores if the called service can access data structures of the OS that are common to the cores. For this purpose, the kernel can be executed on two different cores with sequential access through a global lock. This lock protocol is called the *Biglock* in Trampoline. The kernel is locked during a system call, preventing more than one core from entering the kernel mode or handling interrupts simultaneously. When a core calls a service, it acquires the *Biglock* and only returns it at the end of its execution before leaving kernel mode. Thus, the other cores wait for the lock to be released. Nevertheless, spinlocks and startup services can run in parallel and simultaneously on several cores with no *Biglock*.

Scheduling Trampoline implements a partitioned scheduling policy with fixed priorities for its multi-core version. The tasks are statically allocated to each core that has its ready list. This ready list is represented by a binary heap where each entry has a key for sorting and the process ID. The key is the concatenation of the job [5] priority and the job rank. The heap size is computed with the sum of the process activations. The head of the queue contains the highest priority job. Thus the `tpl_put_new_proc` function inserts a newly activated job into the heap according to its calculated priority (i.e., static priority and activation rank), unlike the mono-core version where the job is placed at the end of the FIFO list corresponding to its priority. Scheduling is performed on the core where it is triggered and can lead to preemption and context switching on another core. To better separate the information for each core, the data structure `tpl_kern` (Listing 3.1), managed by the scheduler, is duplicated according to the core number.

Executing a multi-core service call The entry in the multi-core critical section protected by the *Biglock* is made at the beginning, and then the exit at the end of the system call handler.

When a service is called and results in a rescheduling, for example, if one task activates another, the rescheduling is performed on the core where the service call occurs. Thus, when a task running on core 0 activates a task assigned to core 1, the task ac-

5. Each activation of a process is a job.

tivation service is performed on core 0 and modifies the list of ready tasks of core 1. Additionnally, 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, and the interrupt routine which will execute on core 1 will perform the context switch. This sequence is illustrated in Figure 3.11, page 52:

1. In Ⓐ, a rescheduling is performed by core 0 for core 1;
2. In Ⓑ, an inter-core interrupt is sent to notify core 1 that it must make a context switch;
3. The interrupt is handled by core 1, which blocks on an active wait for the release of the *Biglock* in Ⓒ;
4. Core 0 releases the *Biglock* in Ⓓ. The release of the *Biglock* allows core 1 to take it and to enter the critical multi-core section;
5. Finally core 1 performs the context switch from task τ_1 to task τ_2.

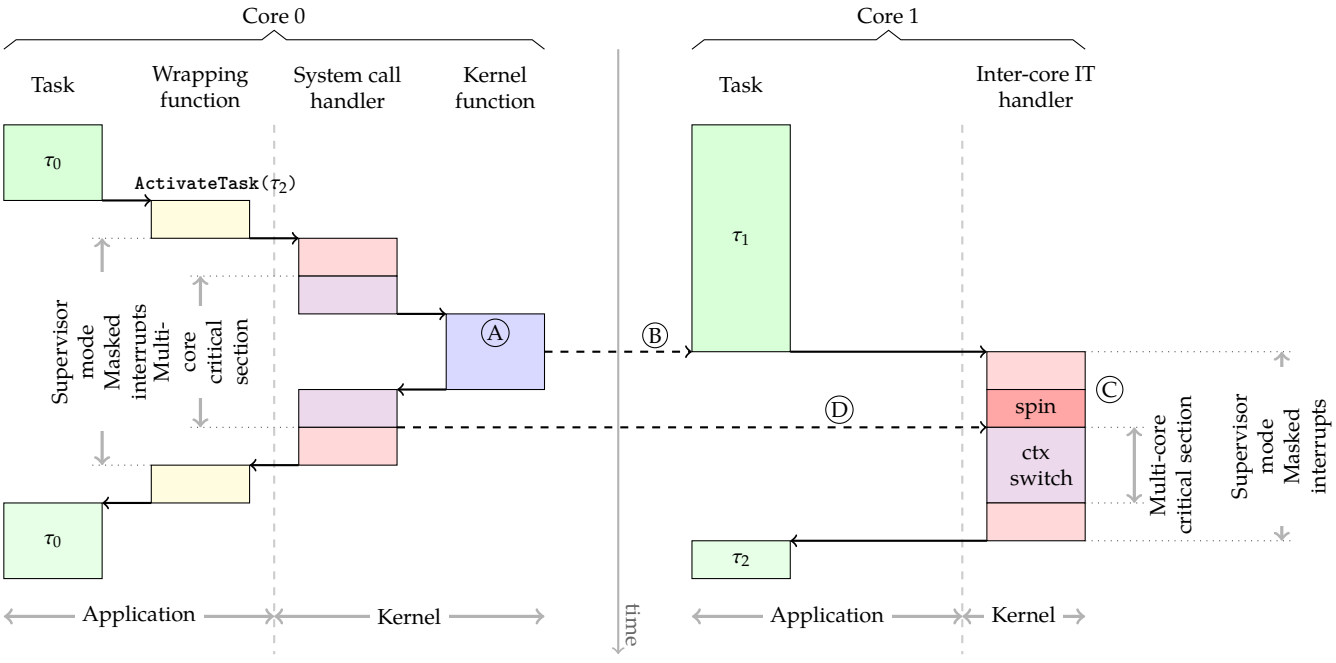


Figure 3.11: The activation of a task τ_2 on core 1 by a task τ_0 running on core 0.

Having simultaneous service calls in parallel on several cores leads to a more complex scheme, especially if a core is the target of an inter-core interrupt while executing a service call that leads to a rescheduling. We can thus extend the example presented in Figure 3.11, page 52, with task τ_1 making a service call on core 1, e.g. `TerminateTask`, just after `ActivateTask` has been called by task τ_0 on core 0. This scenario is presented in Figure 3.12, page 54:

1. In Ⓐ, a rescheduling is performed by core 0 for core 1 and in parallel, core 1 waits for the *Biglock* in Ⓑ;
2. An inter-core interrupt is sent in Ⓒ to notify core 1 that it must make a context switch, but since the interrupts are masked on core 1, it remains pending;
3. Core 0 releases the *Biglock* by Ⓓ. The release of the *Biglock* allows core 1 to take it and to execute the `TerminateTask()` service;
4. The context switch to the τ_2 task is carried out, the *Biglock* is released, and core 1 returns to user mode to immediately take into account the inter-core interrupt by Ⓔ;
5. The execution of this inter-core interrupt consists essentially in acknowledging the interrupt and does not lead to a context switch because τ_2 is the highest priority task on core 1;
6. We finally return to the execution of τ_2.

3.6 Conclusion

In this chapter, we first presented the two automotive standards, OSEK/VDX and AUTOSAR, on which the Trampoline real-time operating system is based. Then, we presented the mono-core and multi-core architectures of Trampoline, explaining the execution of the operating system service calls in both versions. In the next chapter, we will present the modeling formalism used to describe the Trampoline operating system in order to apply our formal verification approach.

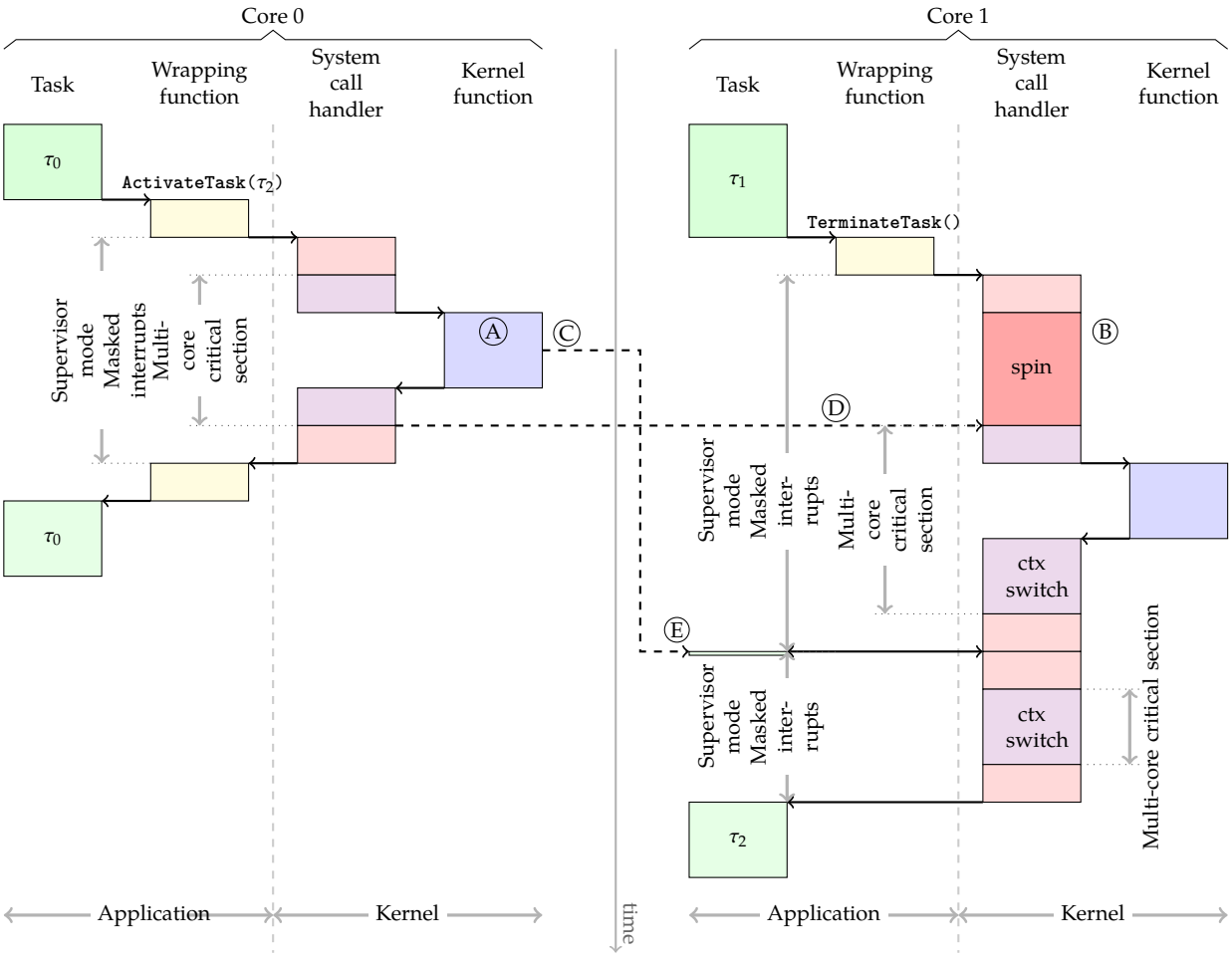


Figure 3.12: The activation of a task τ_2 on core 1 by a task τ_0 running on core 0 in parallel with the termination of the task τ_1 on core 1.

Part II

Contribution

Chapter 4

High-level Colored Time Petri Nets for multi-core concurrency

4.1 Introduction

Implementing multi-core real-time systems requires concurrent access in true parallelism to shared resources. Time Petri nets do not capture these features directly and are unsuitable for modeling systems where data affects the system's behavior. High-level Petri nets [80] have been proposed for modeling scientific problems with complex structures allowing the description of both system data and control. We propose extending time Petri nets with color and high-level functionality in this chapter. We present in detail the High-level Colored Time Petri Nets (HCTPN) used to describe the Trampoline real-time operating system. HCTPN with stopwatches are used to model preemptive real-time systems.

4.2 Informal presentation

4.2.1 Petri nets

Petri nets are a mathematical formalism and one of the many modeling languages used to describe distributed concurrent systems. A Petri net is a directed bipartite graph whose vertices are places and transitions. 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 of its input places and produces tokens in each of its output places.

4.2.2 High-level Petri nets

Petri nets can be classified into two classes: ordinary Petri nets and high–level Petri nets. High-level Petri nets [80] are proposed for modeling scientific problems with complex structures and manipulating different types of expressions made up of variables and written in terms of a predefined syntax. In high-level nets, each token can carry complex information which, e. g., may describe the entire state of a process or a database and handle different expressions and data structures.

The 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.

4.2.3 Colored Petri nets

The colored extension of Petri nets allows 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 its firing will consume it. An arc from a transition to a place (TP) specifies the token 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 in a TP arc that the color consumed in the input place will be the one produced in the output place.

A marking M of a colored Petri Net represents not only the number of tokens in each place but also their respective colors. That is represented either by a multiset or by a matrix.

4.2.4 Time Petri Nets

Time Petri nets (TPN) extend Petri nets with temporal intervals (such as $[\alpha, \beta]$ or $[\alpha, +\infty[$) 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.

To describe the semantics of TPN, we usually consider that a clock is associated with each transition. This clock is set to zero when the transition is newly enabled, and the transition fires when the value of the clock is in the firing interval.

4.2.5 Colored Time Petri Nets

For real parallelism or with interleaving semantics of timed systems, the notion of multiple enableness is needed. It refers to the fact that a transition is enabled at least twice in the same state, which implies a dynamic number of timers. Multiple enableness in time Petri nets is a natural way for modeling paradigms like multiple servers and multiple instances of codes [81].

For Colored Time Petri Nets, multiple enableness occurs when several combinations of colors enable a transition at a given time. In this case, there can be at most one clock per color and per transition.

4.2.6 Time Petri Nets with stopwatches

Time Petri nets with stopwatches, extend TPN by adding the notion of stopwatch: instead of the clocks, 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.

Since the clocks are replaced by stopwatches, in the case of Colored Time Petri Nets with stopwatches, there is at most one stopwatch per color and per transition.

4.3 Formal definition

We consider a Petri Nets model which encompasses both colors, high-level functionalities and stopwatches. We now give the formal definition.

4.3.1 High-level Colored Time Petri Net

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.

4.3.1.1 Definition and semantics

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 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.

We consider a set C of colors. An arc is either associated with a color of C or can take on the particular color called *any*. For the firing of a transition, all its arcs associated with the *any* color must match to instantiate *any* at the same color taken from C.

If several values of *any* allow its enabling, the transition is multi-enabled, and in this case, several clocks (one per color) are associated with the transition, allowing several firing dates depending on the enabling date and the time interval.

The formal definition is as follows.

Definition 1 (High-level Colored Time Petri Net) *A* High-level Colored Time Petri Net *(HCTPN) is a tuple* $\mathcal{N} = (P, T, X, C, \text{pre}, \text{post}, (m_0, x_0), guard, update, I)$ *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,*
- pre : $P \times T \to \mathbb{N}^{C_{any}}$ *is the backward incidence mapping,*
- 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.*

4.3.1.2 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.

4.3.1.2.1 ***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 $\text{pre}(p, t) \in \mathbb{N}^{C_{any}}$ is a vector such that $\text{pre}(p, t)[c]$ is the number of tokens of color $c \in C$ in place p needed to enable the transition t and $\text{pre}(p, t)[any] > 0$ 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. $T_{any} = \{t \in T, \exists p \in P, \text{ s.t. } \text{pre}(p, t)[any] > 0\}$. 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:

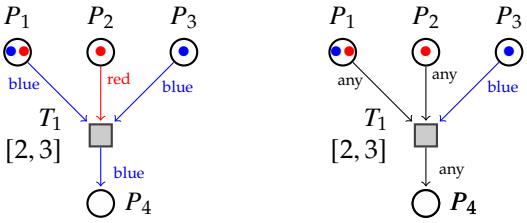
- if $t \in T_{\overline{any}}$, and $\forall p \in P$ and $\forall c \in C$, $m(p)[c] \geq \text{pre}(p, t)[c]$. We denote $\text{en}(m, t) \in \{true, false\}$, the true value of this condition.

- if $t \in T_{any}$, and $\exists c_a \in C$ such that $\forall p \in P, m(p)[c_a] \geq \text{pre}(p,t)[any]$ and $\forall c \in C \setminus \{c_a\}, m(p)[c] \geq \text{pre}(p,t)[c]$. The corresponding set of color c_a is noted $\text{colorSet}_{\text{any}}(m,t) \subseteq C$

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 $\text{en}(m,t) = true$ and either $\text{colorSet}_{\text{any}}(m,t) = \emptyset$ and $guard(m,t,x,\bullet) = true$ or $\exists c_a \in \text{colorSet}_{\text{any}}(m,t) \neq \emptyset$ 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 4.1.a, the transition $T_1 \in T_{\overline{any}}$.

We have $\text{pre}(T_1) = \begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \\ P_4 \end{array} \begin{array}{c} \begin{array}{ccc} red & blue & any \end{array} \\ \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{array}$. 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 $\text{en}(m_0, T_1) = true$.



4.1.a: $T_1 \in T_{\overline{any}}$ 4.1.b: $T_1 \in T_{any}$

Figure 4.1: Enabling transition

Now we consider the HCTPN given in Figure 4.1.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 $\text{pre}(T_1) = \begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \\ P_4 \end{array} \begin{array}{c} \begin{array}{ccc} red & blue & any \end{array} \\ \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{array}$. The transition is enabled only if any takes the red value then $\text{colorSet}_{\text{any}}(m_0, T_1) = \{red\}$. If place $P2$ had two tokens with one token per color, then the transition would be multi-enabled by the two colors leading to $\text{colorSet}_{\text{any}}(m_0, T_1) = \{blue, red\}$.

4.3.1.2.2 ***The firing of a transition.*** An arc $\text{post}(p,t) \in \mathbb{N}^{C_{any}}$ is a vector such that $\text{post}(p,t)[c]$ is the number of tokens of color $c \in C$ produced in place p by the firing of the transition t, and $\text{post}(p,t)[any]$ gives the number of tokens produced in p with the color $c \in \text{colorSet}_{\text{any}}(m,t)$ 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 $\text{en}(m,t) = true$ and $guard(m,t,x,\bullet) = true$ leads to a new marking m' defined by $\forall c \in C, \forall p \in P, m'(p)[c] = m(p)[c] - \text{pre}(p,t)[c] + \text{post}(p,t)[c]$ and a new valuation $x' = update(m,t,x,\bullet)$. This new marking is denoted $m' = \text{firing}(m,t,\bullet)$ 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 $c_a \in \text{colorSet}_{\text{any}}(m,t)$ leads to a new marking defined by $\forall c \in C \setminus \{c_a\}, \forall p \in P, m'(p)[c] = m(p)[c] - \text{pre}(p,t)[c] + \text{post}(p,t)[c]$ and $\forall p \in P, m'(p)[c_a] = m(p)[c_a] - \text{pre}(p,t)[c_a] - \text{pre}(p,t)[any] + \text{post}(p,t)[c_a] + \text{post}(p,t)[any]$ and a new valuation $x' = update(m,t,x,c_a)$. This new marking is denoted $m' = \text{firing}(m,t,c_a)$.

We denote by $\text{newen}((m,x),t,c)$ 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 4.1.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 4.1.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. $m'_0 = \begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \\ P_4 \end{array} \begin{array}{c} \begin{array}{cc} red & blue \end{array} \\ \begin{pmatrix} 1 & 1 \\ 1 & 1 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \end{array} \xrightarrow{(T_1,blue)}$

$$m_3 = \begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \\ P_4 \end{array} \begin{array}{c} \begin{array}{cc} red & blue \end{array} \\ \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix} \end{array}.$$

4.3.1.3 High-level functionalities

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

The model in Figure 4.2 is an HCTPNwith 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[1].

```
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 {
                return c[firedColor] -1 ;
        }
}
```

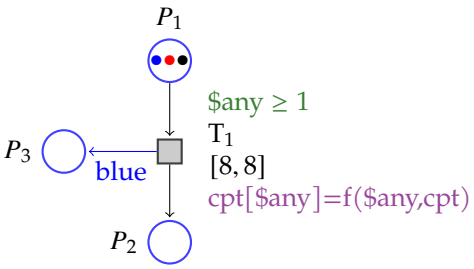


Figure 4.2: High-level manipulation of variables.

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 produce 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 blue

1. 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.

token because of the guard \$any ≥ 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 a red and a black tokens leading to a marking with a red and a black tokens in P_2 and 2 blue tokens in P_3. It remains a blue token in P_1 and the final value of cpt is {2,4,4}.

4.3.1.4 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 $c \in \text{colorSet}_{\text{any}}(m,t)$. 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 $\forall t \in T, \forall c \in C \cup \{\bullet\}, \bar{0}(t,c) = 0$.

As an example, if we keep only the useful clocks, the initial valuation of the HCTPN of Figure 4.1, is $v_0 = T_1 \begin{pmatrix} \bullet & red & blue \\ 0 & & \end{pmatrix}$ for Figure 4.1.a, $v_0 = T_1 \begin{pmatrix} \bullet & red & blue \\ & 0 & 0 \end{pmatrix}$ for Figure 4.1.b, and $v_0 = T_1 \begin{pmatrix} \bullet & blue & red & black \\ & 0 & 0 & \end{pmatrix}$ for Figure 4.2.

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

Definition 2 (Semantics of a HCTPN) *The semantics of a HCTPN is a timed transition system* $(Q, Q_0, \rightarrow)$ *where:*

- $Q \subseteq \mathbb{N}^{P\times C} \times \mathbb{X}^X \times {\mathbb{R}_{\geq 0}}^{T\times C}$
- $Q_0 = ((m_0, x_0), \bar{0})$
- $\rightarrow \in Q \times ((T \times C \cup \{\bullet\}) \cup \mathbb{R}_{\geq 0}) \times Q$ *consists of two types of transitions:*
  - *discrete transitions (firing t from* $((m,x),v)$*) iff:*
    - $((m,x),v) \xrightarrow{(t\in T_{\overline{any}},\bullet)} ((m',x'),v')$ *with*
      - $\text{en}(m,t) = true$ *and* $v(t) \in I(t)$,
      - $m' = \text{firing}(m,t,\bullet)$
    - $((m,x),v) \xrightarrow{(t\in T_{any},c)} ((m',x'),v')$ *with*
      - $c \in \text{colorSet}_{\text{any}}(m,t)$ *and* $v(t,c) \in I(t)$,

· $m' = \text{firing}(m, t, c)$

* $guard(t, x) = true$ *and* $x' = update(t, x)$
* $\forall t' \in T_{\overline{any}}$ *s.t.* $\text{en}(m', t') = true$
  · $v'(t', \bullet) = v(t', \bullet)$ *if* $t' \notin \text{newen}((m, x), t, \bullet)$,
  · $v'(t', \bullet) = 0$ *otherwise*
* $\forall t' \in T_{any}$ *and* $\forall c \in \text{colorSet}_{\text{any}}(m', t')$
  · $v'(t', c) = v(t', c)$ *if* $t' \notin \text{newen}((m, x), t, c)$,
  · $v'(t', c) = 0$ *otherwise*

– *time transitions:* $((m, x), v) \xrightarrow{d \in \mathbb{R}_{\geq 0}} ((m, x), v')$, *iff:*

* $\forall t \in T_{\overline{any}}$ *s.t.* $\text{en}(m, t) = true$,
  · $v'(t, \bullet) \leq I(t)^{\downarrow}$
  · $v'(t, \bullet) = v(t, \bullet) + d$
* $\forall t \in T_{any}$ *and* $\forall c \in \text{colorSet}_{\text{any}}(m, t)$,
  · $v'(t, c) \leq I(t)^{\downarrow}$
  · $v'(t, c) = v(t, c) + d$

We now illustrate the main features of HCTPN in an example. The guards are in green in the Figures and the update in purple.

4.3.1.5 Examples of HCTPN

We give three examples. The first two examples illustrate the high-level functionalities, and the third illustrates the notion of color and multi-enableness.

Example 1 Let's go back to the HCTPN given in Figure 4.2, page 63.

The initial marking $m_0 = \begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \end{array} \begin{array}{c} \begin{array}{ccc} blue & red & black \end{array} \\ \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{array}$ enables the transition T_1. The valuations of the clocks are given by the matrix such that the initial valuation is $v_0 = \begin{array}{c} \\ T_1 \end{array} \begin{array}{c} \begin{array}{cc} \bullet & blue \quad red \quad black \end{array} \\ \begin{pmatrix} & 0 \quad 0 \end{pmatrix} \end{array}$. 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: $\left(m_0, \{2,2,5\}, \begin{array}{c} \bullet \\ \\ \end{array}\begin{array}{ccc} blue & red & black \\ 0 & 0 \end{array}\right) \xrightarrow{8}$

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

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

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

Example 2 The HCTPN given in Figure 4.3, page 66, illustrates time behavior and high-level manipulation of variables. This HCTPN has only one color and a single variable cpt, and is part of a larger HCTPN. We assume that $g()$ returns an integer between 1 and 10, handled by the other part of the net.

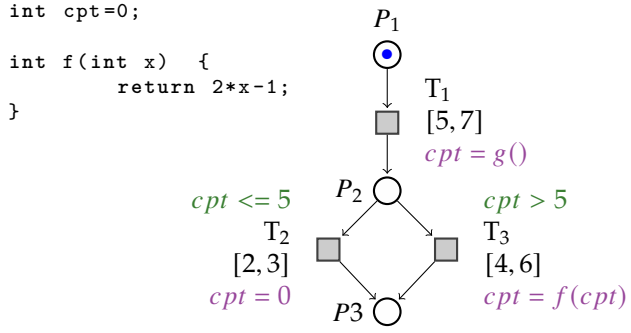


Figure 4.3: HCTPN illustrating high-level manipulation of variables

A marking is written by the matrix $(|P|, |C|)$. Since there is only one color, the marking is a vector and the initial marking is then $m_0 = \begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \end{array}\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$ and enables the transi-

tion T_1. The valuations of the clocks are given by the matrix (here a vector) such that the initial valuation is $v_0 = \begin{matrix} T_1 \\ T_2 \\ T_3 \end{matrix} \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$. Since the set of variables is $X = \{cpt\}$, we note a state $s = (m, cpt, v)$. The initial state is $q_0 = (m_0, 0, v_0)$. The transition T_1 can fire after elapsing 5 time units. We now consider the run where the function $g()$ called by the update of the firing of T_1 returned the value 7. Then the transition's guard T_2 is false, and the transition T_3 is enabled. We assume that the transition T_3 took 4.6 time units for this run. The firing of the transition T_3 executes the corresponding update and calls the function f that returns 13. The corresponding run is as follows: $\left(\begin{pmatrix}1\\0\\0\end{pmatrix}, 0, \begin{pmatrix}0\\0\\0\end{pmatrix}\right) \xrightarrow{5} \left(\begin{pmatrix}1\\0\\0\end{pmatrix}, 0, \begin{pmatrix}5\\0\\0\end{pmatrix}\right) \xrightarrow{(T_1,\bullet)} \left(\begin{pmatrix}0\\1\\0\end{pmatrix}, 7, \begin{pmatrix}0\\0\\0\end{pmatrix}\right) \xrightarrow{4.6} \left(\begin{pmatrix}0\\1\\0\end{pmatrix}, 7, \begin{pmatrix}0\\0\\4.6\end{pmatrix}\right) \xrightarrow{(T_3,\bullet)} \left(\begin{pmatrix}0\\0\\1\end{pmatrix}, 13, \begin{pmatrix}0\\0\\0\end{pmatrix}\right)$

Example 3 The model given in Figure 4.4, page 67, is a HCTPN with a set of two colors $C = \{red, blue\}$. Several combinations of color usage, on guards and in updates, via the $\$any$ variable are presented.

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

int f(int color, int[2] c) {
        if (color == blue) {
                return c[color] + 1;
        }
        else if (color == red) {
                return c[color] * 2;
        }
}
```

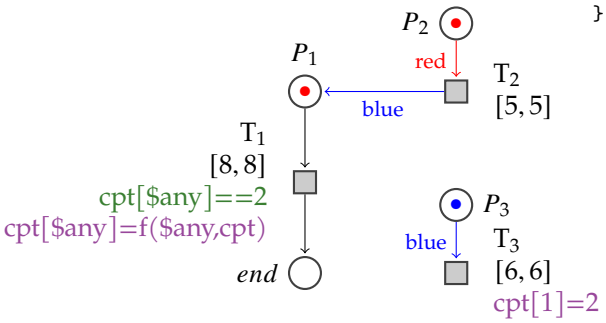


Figure 4.4: HCTPN model illustrating colored multi-enableness

In the sequel a marking is written by the matrix $(|P|, |C|)$. The initial marking is

then $m_0 = \begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \\ end \end{array} \begin{array}{c} \begin{array}{cc} red & blue \end{array} \\ \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \end{array}$ and enables the transitions T_1, T_2 and T_3. The variable cpt of the model is an array indexed by the color. Its initial value is $x_0 = cpt \begin{array}{c} \begin{array}{cc} red & blue \end{array} \\ \begin{pmatrix} 2 & 1 \end{pmatrix} \end{array}$ The valuations of the clocks are given by the matrix such that the initial valuation is $v_0 = \begin{array}{c} \\ T_1 \\ T_2 \\ T_3 \end{array} \begin{array}{c} \begin{array}{cc} \bullet & \begin{array}{cc} red & blue \end{array} \end{array} \\ \begin{pmatrix} 0 & 0 \\ 0 & \\ 0 & \end{pmatrix} \end{array}$ (We omit the insignificant values). We note a state $s = (m, x, v)$. The initial state is $q_0 = (m_0, x_0, v_0)$. Since the time intervals are points, we have an unique run:

$$\left(\begin{pmatrix} 1&0\\1&0\\0&1\\0&0 \end{pmatrix}, \begin{pmatrix} 2&1 \end{pmatrix}, \begin{pmatrix} 0&0\\0&0 \end{pmatrix}\right) \xrightarrow{5} \left(\begin{pmatrix} 1&0\\1&0\\0&1\\0&0 \end{pmatrix}, \begin{pmatrix} 2&1 \end{pmatrix}, \begin{pmatrix} 5&0\\5&0 \end{pmatrix}\right) \xrightarrow{(T2,\bullet)} \left(\begin{pmatrix} 1&1\\0&0\\0&1\\0&0 \end{pmatrix}, \begin{pmatrix} 2&1 \end{pmatrix}, \begin{pmatrix} 0&0\\5&0 \end{pmatrix}\right) \xrightarrow{1} \left(\begin{pmatrix} 1&1\\0&0\\0&1\\0&0 \end{pmatrix}, \begin{pmatrix} 2&1 \end{pmatrix}, \begin{pmatrix} 0&6\\6&0 \end{pmatrix}\right) \xrightarrow{(T3,\bullet)}$$
$$\left(\begin{pmatrix} 1&1\\0&0\\0&0\\0&0 \end{pmatrix}, \begin{pmatrix} 2&2 \end{pmatrix}, \begin{pmatrix} 0&6\\0&0 \end{pmatrix}\right) \xrightarrow{2} \left(\begin{pmatrix} 1&1\\0&0\\0&0\\0&0 \end{pmatrix}, \begin{pmatrix} 2&2 \end{pmatrix}, \begin{pmatrix} 0&8\\0&2 \end{pmatrix}\right) \xrightarrow{(T1,red)} \left(\begin{pmatrix} 0&1\\0&0\\0&0\\1&0 \end{pmatrix}, \begin{pmatrix} 4&2 \end{pmatrix}, \begin{pmatrix} 0&2\\0&0 \end{pmatrix}\right) \xrightarrow{6} \left(\begin{pmatrix} 0&1\\0&0\\0&0\\1&0 \end{pmatrix}, \begin{pmatrix} 4&2 \end{pmatrix}, \begin{pmatrix} 0&8\\0&0 \end{pmatrix}\right) \xrightarrow{(T1,blue)}$$
$$\left(\begin{pmatrix} 0&0\\0&0\\0&0\\1&1 \end{pmatrix}, \begin{pmatrix} 4&3 \end{pmatrix}, \begin{pmatrix} 0&0\\0&0 \end{pmatrix}\right)$$

The time elapses from the initial marking until reaching date 5. T_2 is fired, and a blue token is dropped in the place P_1. The clock of T_1 associated with the red color has reached the value 5. The clock of T_1 associated with the blue color cannot start yet because the guard is false for this color. At date 6, T_3 is fired, causing a change in the variable `cpt` that makes the guard of T_1 true for the blue color. The clock associated with the blue color for T_1 can therefore start. Both colors enable the transition T_1, and the corresponding clocks give the time from the two enabling. After two more time units, T_1 is fired for the red color; at this moment, the clock of T_1 for the blue color has reached 2. Finally, after 6-time units, T_1 is fired for the blue color, ending the run.

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 atomic from the HCTPN 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 $x' = update(t, x)$.

4.3.2 High-level Colored Time Petri Net with stopwatches

We now consider stopwatches instead of clocks. Hence, for Colored Time Petri Nets with stopwatches, there is at most one stopwatch per color and per transition.

When using stopwatches with the HCTPN formalism, the temporal behavior differs and depends on the time derivative function $\dot{v}(t, \bullet)$ when transitions $t \in T_{\overline{any}}$ and $\dot{v}(t, c)$ for $t \in T_{any}$.

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 be fired. The time derivative of a stopwatch is in the rate set $\{0, 1\}$ and is given by a function from Markings.

Definition 3 (High-level Colored Time Petri Net with stopwatches)
A High-level Colored Time Petri Net with stopwatches *is a tuple*
$\mathcal{N} = (P, T, \text{pre}(.), \text{post}(.), m_0, guard, update, I, \dot{v})$ *where*
$(P, T, \text{pre}(.), \text{post}(.), m_0, guard, update, I)$ *is defined in Definition 1 and*
$\dot{v} : T \times \mathbb{N}^{P \times C} \times \mathbb{X}^X \to \{0, 1\}$ *is the* time derivative *function.*

4.3.2.1 Semantics

For the discrete transition of the semantics, the only difference with HCTPN is that a transition cannot be fired if its time derivative is not 1. For the time transition, the value of a stopwatch evolves according to its derivative as follows.

Definition 4 (Semantics of a HCTPN with stopwatches) *The semantics of a HCTPN with stopwatches is a timed transition system* $(Q, Q_0, \to)$ *where:*

- $Q \subseteq \mathbb{N}^{P \times C} \times \mathbb{X}^X \times {\mathbb{R}_{\geq 0}}^{T \times C}$
- $Q_0 = ((m_0, x_0), \bar{0})$
- $\to \in Q \times ((T \times C \cup \{\bullet\}) \cup \mathbb{R}_{\geq 0}) \times Q$ *consists of two types of transitions:*
  - *discrete transitions (firing t from $((m, x), v)$), as presented in Definition 2 with* $\dot{v}(t) = 1$
  - *time transitions:* $((m, x), v) \xrightarrow{d \in \mathbb{R}_{\geq 0}} ((m, x), v')$, *iff:*

* $\forall t \in T_{\overline{any}}$ *s.t.* $\text{en}(m,t) = true$,
  - $v'(t,\bullet) \leq I(t)^{\downarrow}$
  - $v'(t,\bullet) = v(t,\bullet) + d$ *if* $\dot{v}(t,\bullet) = 1$ *otherwise* $v'(t,\bullet) = v(t,\bullet)$
* $\forall t \in T_{\overline{any}}$ *s.t.* $\text{en}(m,t) = false$,
  - $v'(t,\bullet) = 0$
* $\forall t \in T_{any}$ *and* $\forall c \in \text{colorSet}_{\text{any}}(m,t)$,
  - $v'(t,c) \leq I(t)^{\downarrow}$
  - $v'(t,c) = v(t,c) + d$ *if* $\dot{v}(t,c) = 1$ *otherwise* $v'(t,c) = v(t,c)$
* $\forall t \in T_{any}$ *and* $\forall c \notin \text{colorSet}_{\text{any}}(m,t)$,
  - $v'(t,c) = 0$

We now illustrate the main features of HCTPN with stopwatches on an example.

4.3.2.2 Example of HCTPN with stopwatches

This example is the modeling of the preemptive scheduling of two tasks. The first task $task_1$ is a periodic task running on core 0, assigned to blue color. The second task $task_2$ is also periodic but is executed only 10 times on core 1, assigned to red color. The particular color *any* is used for enabling and firing all transitions. For the first two executions of $task_2$, the priority of $task_1$ is higher than $task_2$ priority, after which it becomes the opposite.

The model in Figure 4.5 is a HCTPN with stopwatches and has a single shared variable cpt and two colors. The initial value of cpt is zero. Only the transition T_2 has a guard and an update that manipulate the cpt variable. 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 increments 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 4.5.

In the sequel, a marking is written by the matrix $m = (|P|, |C|)$. The initial marking enables the transitions T_1 and T_2. The valuations of the stopwatches are given by the

```
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;
        }
}
```

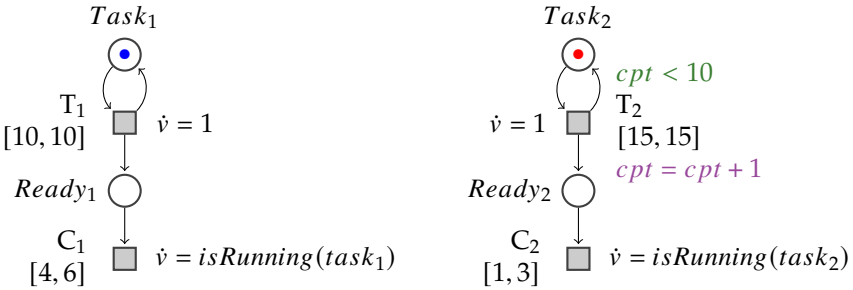


Figure 4.5: HCTPN model with stopwatches of two-task scheduling.

matrix $v = (|T|, |C|)$. Since all the transitions are in T_{any}, the bullet column of the stopwatch valuations is not used. We will therefore omit it in the states of this example in order to simplify the notation.

We note a state $s = (m, cpt, v)$ and the initial state is $q_0 = (m_0, 0, v_0)$, where:

$$m_0 = \begin{array}{c} \\ Task_1 \\ Task_2 \\ Ready_1 \\ Ready_2 \end{array} \begin{array}{c} \begin{array}{cc} red & blue \end{array} \\ \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \end{array}, \text{ and } v_0 = \begin{array}{c} \\ T_1 \\ T_2 \\ C_1 \\ C_2 \end{array} \begin{array}{c} \begin{array}{ccc} \bullet & red & blue \end{array} \\ \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{array} \text{ simply denoted } v_0 = \begin{array}{c} \\ T_1 \\ T_2 \\ C_1 \\ C_2 \end{array} \begin{array}{c} \begin{array}{cc} red & blue \end{array} \\ \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \end{array}.$$

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:

$$q_0 = \left(\begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}, 0, \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \right) \xrightarrow{10} \left(\begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}, 0, \begin{pmatrix} 0 & 10 \\ 10 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \right) \xrightarrow{(T_1, blue)} q_1 = \left(\begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}, 0, \begin{pmatrix} 0 & 0 \\ 10 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \right)$$

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

$$q_1 \xrightarrow{5} q_2 = \left(\begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}, 0, \begin{pmatrix} 0 & 5 \\ 15 & 0 \\ 0 & 5 \\ 0 & 0 \end{pmatrix} \right) \xrightarrow{(T_2, red)} q_3 = \left(\begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}, 1, \begin{pmatrix} 0 & 5 \\ 0 & 0 \\ 0 & 5 \\ 0 & 0 \end{pmatrix} \right)$$

In q_3, we have $\dot{v}(C_1, blue) = 1$ and $\dot{v}(C_2, red) = 0$ meaning that $task_2$ is preempted by $task_1$. Then $v(C_2, red)$ will keep its value 0 until the firing of C_1 that will change $\dot{v}(C_2, red)$.

$q_3 \xrightarrow{0.3} q_4 = \left(\begin{pmatrix}0&1\\1&0\\0&1\\1&0\end{pmatrix}, 1, \begin{pmatrix}0&5.3\\0.3&0\\0&5.3\\0&0\end{pmatrix}\right) \xrightarrow{(C_1,blue)} q_5 = \left(\begin{pmatrix}0&1\\1&0\\0&0\\1&0\end{pmatrix}, 1, \begin{pmatrix}0&5.3\\0.3&0\\0&0\\0&0\end{pmatrix}\right)$

In q_5, we have $\dot{v}(C_2, red) = 1$ hence

$q_5 \xrightarrow{2.4} q_6 = \left(\begin{pmatrix}0&1\\1&0\\0&0\\1&0\end{pmatrix}, 1, \begin{pmatrix}0&7.7\\2.7&0\\0&0\\2.4&0\end{pmatrix}\right) \xrightarrow{(C_2,red)} q_7 = \left(\begin{pmatrix}0&1\\1&0\\0&0\\0&0\end{pmatrix}, 1, \begin{pmatrix}0&7.7\\2.7&0\\0&0\\0&0\end{pmatrix}\right)$

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

$q_7 \xrightarrow{2.3} \xrightarrow{(T_1,blue)} \xrightarrow{6} \xrightarrow{(C_1,blue)} \xrightarrow{4} \xrightarrow{(T_1,blue)} \xrightarrow{(T_2,red)} \xrightarrow{6} \xrightarrow{(C_1,blue)} \xrightarrow{3} \xrightarrow{(C_2,red)} \xrightarrow{1} \xrightarrow{(T_1,blue)} \xrightarrow{5} \xrightarrow{(T_2,red)} q_{13}$

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

$q_{13} = \left(\begin{pmatrix}0&1\\1&0\\0&1\\1&0\end{pmatrix}, 3, \begin{pmatrix}0&5\\0&0\\0&5\\0&0\end{pmatrix}\right)$ then we have $\dot{v}(C_1, blue) = 0$ and $\dot{v}(C_2, red) = 1$ meaning that the task $task_2$ is not preempted by the task $task_1$.

Hence we have :

$q_{13} \xrightarrow{2.4} q_{14} = \left(\begin{pmatrix}0&1\\1&0\\0&1\\1&0\end{pmatrix}, 3, \begin{pmatrix}0&7.4\\2.4&0\\0&5\\2.4&0\end{pmatrix}\right) \xrightarrow{(C_2,red)} q_{15} = \left(\begin{pmatrix}0&1\\1&0\\0&1\\0&0\end{pmatrix}, 3, \begin{pmatrix}0&7.4\\2.4&0\\0&5\\0&0\end{pmatrix}\right).$

4.4 Decidability, complexity and state space computation

Let us recall that a *High-level Colored Time Petri Net* (HCTPN) is a tuple $\mathcal{N} = (P, T, X, C, \text{pre}, \text{post}, (m_0, x_0), guard, update, I)$ such that the set C of *colors* is finite and X is a finite set of *variables* taking their value in a finite set $\mathbb{X}$.

Theorem 1 *Reachability problem for bounded High-level Colored Time Petri Net is decidable*

Proof: From HCTPN semantics, a transition can be multi-enabled a maximum of $|C|$ times at a given time. Hence, firing domains can be symbolically abstracted with state classes using Difference Bound Matrix (DBM) over $|C| \times T$ variables. As in [41, 82], the number of DBM is finite. Moreover, the number of markings of a k-bounded Petri net is bounded by $(k+1)^{|P|}$ then the number of discrete states of a k-bounded HCTPN is bounded by $(k+1)^{|P|} \times \mathbb{X}^X$. Hence, computable finite abstractions of the state space exist, and the reachability problem is decidable. □

State Space Computation A discrete *state* of the net $\mathcal{N}$ is a tuple $((m, x), v)$ in $\mathbb{N}^{P \times C} \times \mathbb{X}^X \times {\mathbb{R}_{\geq 0}}^{T \times C}$, where: m is a marking, x is a variable valuation and v is a valuation of the clocks. Roméo computes the state-class graph (SCG) that preserve LTL properties of bounded nets [41]. It performs translations from HCTPNs to Timed Automata (TAs) that preserve the behavioural semantics (timed bisimilarity) of HCTPNs. Two distinct

methods are implemented: The state-class graph [41] and the zone-based graph [92]. The zone-based graph method is derived from the TA framework [92], while the state-class graph method is based on the classical state class graph approach [93].

Temporal logics were introduced by Pnueli [83] as specification languages to express the behaviors of sequential and concurrent systems and TCTL (Timed Computation Tree Logic), introduced in [28], is a real-time extension of the branching-time temporal logic CTL (Computation Tree Logic).

We can prove, as in [84] for bounded Time Petri Nets, that the theoretical complexity of TCTL model-checking for bounded High-level Colored Time Petri Nets is PSPACE-complete. However, as for Timed Automata and Time Petri Nets, no effective PSPACE algorithm exists in practice, and real implementations are with exponential algorithms.

In practice, on-the-fly TCTL model-checking for bounded High-level Colored Time Petri Nets is proposed in the Roméo tool, used to model the Trampoline RTOS and the application in Section 4.6, page 74.

Moreover, as shown in the previous section, HCTPN can be extended with stopwatches allowing the modeling of preemptive scheduling. In the stopwatch setting, the reachability problem is undecidable but efficient semi-algorithms are implemented in Roméo [85] that converges for almost all practical cases.

4.5 Roméo tool

The Roméo tool [86] is a free and open-source software developed by the real-time systems team of LS2N at École Centrale de Nantes. It allows the modeling of complex and preemptive real-time systems using the HCTPN formalism with stopwatches. It consists of a Graphical User Interface GUI (written in TCL/Tk) to edit and design TPNs and computation modules (written in C++). Roméo provides a variable *any* that gives the integer value of the color used for the transition firing. The requirements in Roméo are expressed in a subclass of TCTL temporal logic [28] and verified with an on-the-fly efficient algorithm over bounded HCTPN. It includes parameter synthesis for the model-checking [48], allowing the computing of parameter values that guarantee the satisfaction of the property and the addition of linear constraints on the parameters to limit their domain. Roméo implements on-line simulation and reachability model-checking of HCTPN with stopwatches.

Roméo has been used to model and analyze several interesting problems [87–89]. The authors in [87] used parametric time model-checking to verify the time behavior of biological oscillatory systems. They focused on resilience properties which they formalized in the TCTL logic and applied to the oscillatory system of the mammalian circadian clock. Their verification is done in the Roméo tool where the properties are represented using observers modeled as parametric time Petri nets.

4.6 Application

The application chosen as an example is the modeling of the spinlocks mechanism present in the PowerPC MPC5643L dual-core microcontroller from NXP [94] and used to build critical sections for parallel program executions. This mechanism is based on a hardware unit, the SEMA4 for *Semaphore Unit*. For the software, this unit is materialized as an array of 16 registers implementing 16 locks. The exclusive access to the bus regulates the concurrent accesses to one of these registers. If a register contains the value 0, the lock is available, and it is possible to write to it. If the value contained is different from 0, the lock is occupied, and it is only possible to write the value 0 to it, and writing any other value has no effect. Therefore, getting a lock consists in writing a value different from 0 and releasing it consists in writing 0. Thus, using this unit requires respect of a protocol, and an example of implementation is given on page 1322 of [94]. Algorithm 1 reproduces it.

Algorithm 1 Lock acquisition protocol. *gate* is one of the hardware registers of the SEMA4 unit.

```
cn ← core_number                                   ▷ (1 .. N)
do
    lock ← gate
while lock ≠ 0
do
    gate ← cn
    lock ← gate
while lock ≠ cn
```

4.6.1 Modeling the spinlocks mechanism

The modeling takes advantage of the possibilities of the HCTPN. The hardware part, which by virtue of the exclusive access to the bus allows operations that are intrinsically atomic, is modeled using functions. To simplify the presentation, only one register of the SEMA4 unit, *gate*, is modeled but the model could just as well use an array to accurately model the hardware. The listing 4.1, page 76, shows this part of the model. *gate* is initialized to the UNLOCKED state (line 2) and is accessible through three functions. *lock* (line 4) mimics the behavior of the hardware by only allowing writing to *gate* if its value is UNLOCKED. The core number corresponding to a color and a color among N being coded by an integer from 0 to N-1, a core locks by writing $color + 1$ in *gate*. *unlock* (line 10) simply writes the value UNLOCKED into *gate*. *isLocked* (line 14 returns 1 if the *gate* is locked, 0 otherwise). Finally, *isLockedBy* (line 22) returns 1 if *core* holds the lock and returns 0 otherwise.

Each function of the software is modeled by an HCTPN reproducing the control flow graph of the function. Two HCTPNs model the functions *GetSpinLock* and *RelSpinLock* (see Figure 4.6, page 77). *GetSpinLock* corresponds to the algorithm 1, page 74, and *$any* allows to represent on which core the function is executed. The call of a function modeled by a HCTPN is done by dropping a token of the core color in the initial place. Thus, "calling" the function *GetSpinLock* is performed by the update `GetSpinLock[color] = 1` on a transition of the HCTPN of the calling function. This is identical to drawing an arc of the corresponding color between the transition and the initial place of *GetSpinLock*. The function return requires a synchronization. This one is implemented by a variable of type array and of size equal to the number of colors and indexed by the color, i.e. the core on which the function call is made. We have therefore for our two function models the two variables `endOfGSL` and `endOfRSL`, see listing 4.2, page 77.

4.6.2 Verification of the system

The spinlock model is completed by an application model. Two tasks, τ_0, running on core 0 (red color) and τ_1, running on core 1 (blue color), are modeled as shown in Figure 4.7, page 78. The task τ_0 takes then releases the spinlock while the task τ_1 has the possibility to take it, as τ_0 does, or to reach the final state without taking the spinlock. We want to check that τ_0 and τ_1 cannot occupy simultaneously and respectively the places P_{12} and P_{22} by the CTL formula $A\Box(\neg(P_{12}[0] == 1 \wedge P_{22}[1] == 1))$. Here $P_{12}[0]$

Listing 4.1: Modeling of the SEMA4 hardware

```
1 const int UNLOCKED = 0;
2 int gate = UNLOCKED;
3
4 void lock(int core, int &ioGate) {
5 if (ioGate == UNLOCKED) {
6 ioGate = core + 1;
7 }
8 }
9
10 void unlock(int &ioGate) {
11 ioGate = UNLOCKED;
12 }
13
14 int isLocked(int &inGate) {
15 if (inGate == UNLOCKED) {
16 return 0;
17 } else {
18 return 1;
19 }
20 }
21
22 int isLockedBy(int core, int &inGate) {
23 if (inGate == core + 1) {
24 return 1;
25 } else {
26 return 0;
27 }
28 }
```

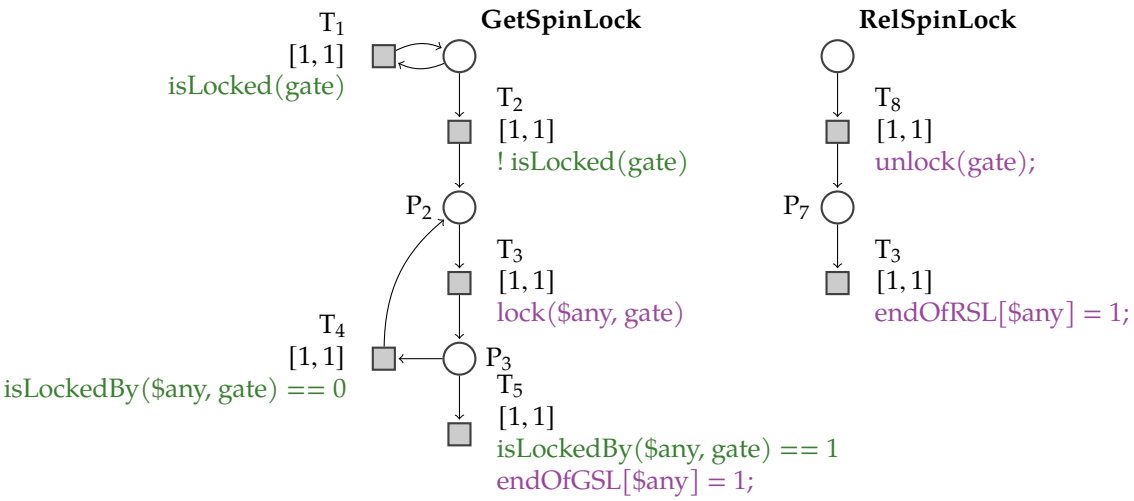


Figure 4.6: The GetSpinLock and RelSpinLock function models

denotes the marking of P_{12} for the red color and $P_{22}[1]$ denotes the marking of P_{22} for the blue color. Roméo answers *true* for this CTL formula.

4.7 Conclusion

This chapter has presented High-level Colored Time Petri Nets. This formalism allows to model complex systems and is well adapted to multi-core hardware and software modeling, as shown in the case study. The high-level features allow the modeling of the software, the timed transitions model the execution times, the colors specify the hardware where the software is executed, and preemption is supported by means of stopwatches. A timed transition enabled by more than one color allows true concurrency modeling. The model-checking of this extended formalism is implemented in the Roméo tool. The next chapter focuses on using this formalism to model the RTOS.

Listing 4.2: Synchronization variables for the function return

```
1 int[2] endOfGSL = {0, 0};
2 int[2] endOfRSL = {0, 0};
```

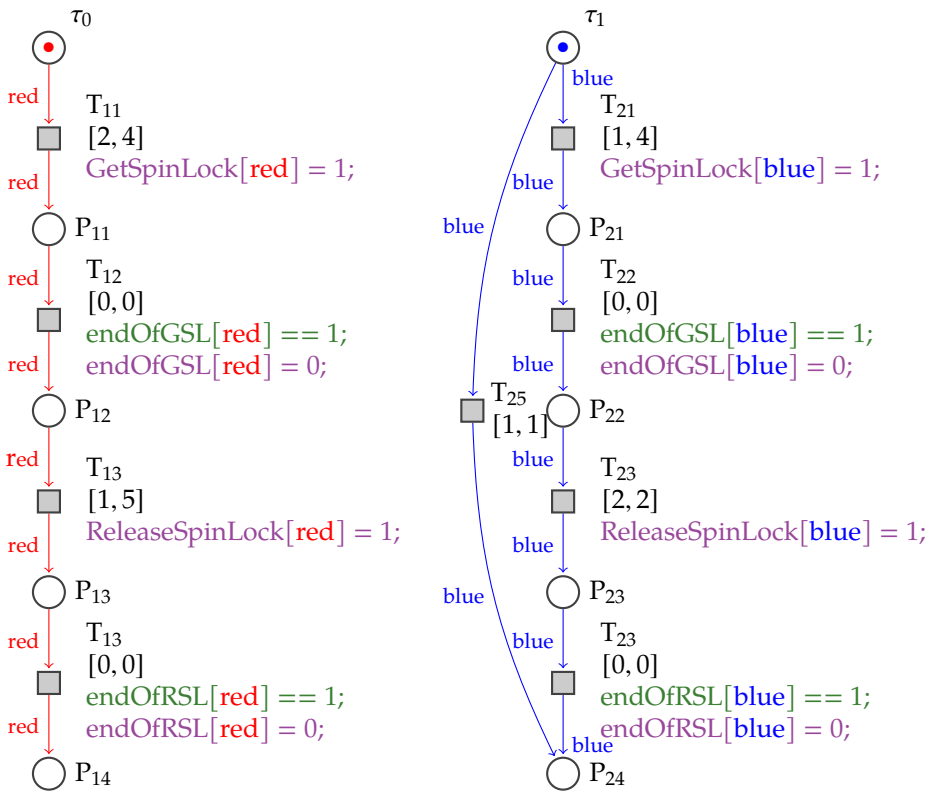


Figure 4.7: The tasks models

Chapter 5

Modeling with High Level Colored Petri Nets

5.1 Introduction

This chapter presents the Trampoline model built in its multi-core version with fixed priority partitioned scheduling for the tasks. The model consists of the same variables and data structures found in the Trampoline source code as well as a HCTPN representing the control flow. The source code that includes 180 functions for the target-independent part is modeled by a systematic approach based on the HCTPN translation rules in Roméo tool 4.5. The model contains 115 Petri sub-networks that form a single one. It includes 600 transitions, 550 places, 162 constants, and other data structures and variables generated automatically in the GOIL compilation phase, as shown in figure 3.6, page 44, using a developed Goil Template Language (GTL) module.

5.2 Modeling rules

We apply the following modeling principles:

- Each Petri subnet describes a function of the operating system and faithfully describes its 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;
- The Petri net transitions include the same assignment instructions and other imperative operations as those present in the source code;

- Pointers are replaced by arrays in the model;
- The processor cores are represented one-to-one by the token colors in the model;
- All kernel 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 the model would be lost, (iii) it is necessary to capture all the possible interleavings and some of them could be cancelled by real-time constraints;
- The application model is described by API function calls;
- Functions that do not require fine-grained modeling 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.

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 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. The Figure 5.1 presents the mechanism. Without changing the semantics of HCTPN, 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.

Atomic modeling 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. An update can

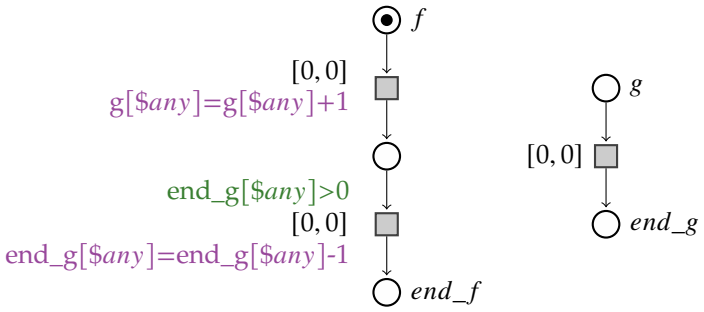


Figure 5.1: Function call mechanism.

read and/or write variables. When it is a variable of the modeled system, one must be careful to reproduce the competition situations 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.

5.3 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 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. 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, and once the transition is fired, the variable is set to true when the transition is fired.

5.3.1 API 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. As we presented in 3.5.2, a service is called on a core by entering kernel mode. The kernel

is locked during its execution → 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` and `ActivateTask` services as examples.

GetAlarmBase service This service allows to obtain the requested information on the alarm base and store it in a global variable. An error is returned if the alarm identifier is invalid. This service call model is shown in Figure 5.2. `GetAlarmBase` calls the kernel function `tpl_get_alarm_base_service model`[1], represented in details in 5.3.2 (Figure 5.8).

ActivateTask service This API service allows the activation of a task. Figure 5.3 shows its modeling. The Petri subnet's first place represents the function's initial location, and each transition describes the execution state of the function. Upon calling this Petri subnet, it calls the kernel function `tpl_activate_task_service` to execute the requested service. `tpl_activate_task_service` function is itself modeled by a Petri subnet.

5.3.2 Kernel modeling

The Kernel contains all the low-level functions on which the Trampoline services are based as represented in 3.5.2. It ensures the start and shutdown of the OS and allows the activation of tasks, their scheduling, and their synchronization. The kernel model

1. 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.

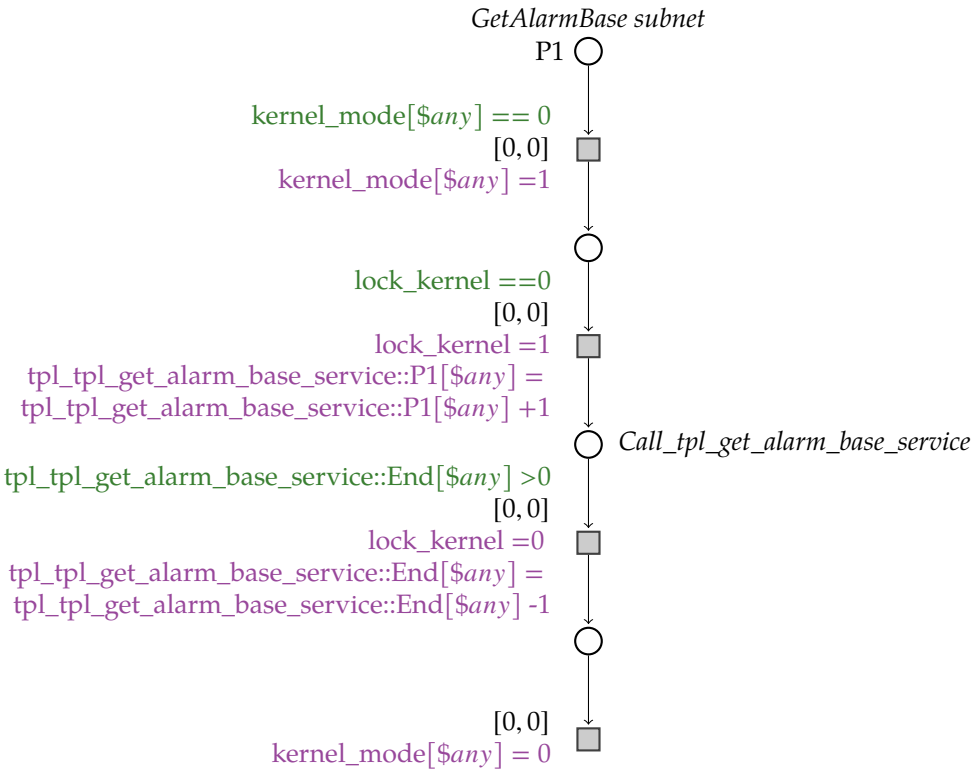


Figure 5.2: GetAlarmBase service model.

contains four components that contain the functions required by the Trampoline services (Figure 3.10, page 50). 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 switches if necessary. All the functions contained in the 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. To terminate the running task, the function `tpl_terminate_task_service` first performs some checks about interrupts, spinlocks, the task's level, and resource, then decrements the activation count, calls the kernel function `tpl_terminate_task`, performs the rescheduling and context switch and finally ends the task being executed.

The modeling of the task manager function `tpl_terminate` is shown in Figure 5.4. This kernel function performs a release of the internal resource held by the task by

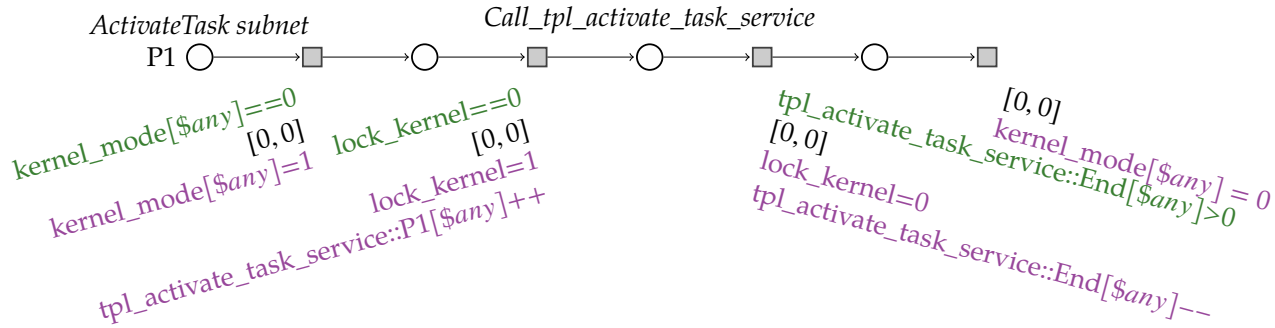


Figure 5.3: Example of the `ActivateTask` service modeling.

calling the `tpl_release_internal_resource` function. A guard on the activation counter allows the running task state to be set either to `READY_AND_NEW`[2] or `SUSPENDED` state. Also, the events associated with the extended tasks are initialized.

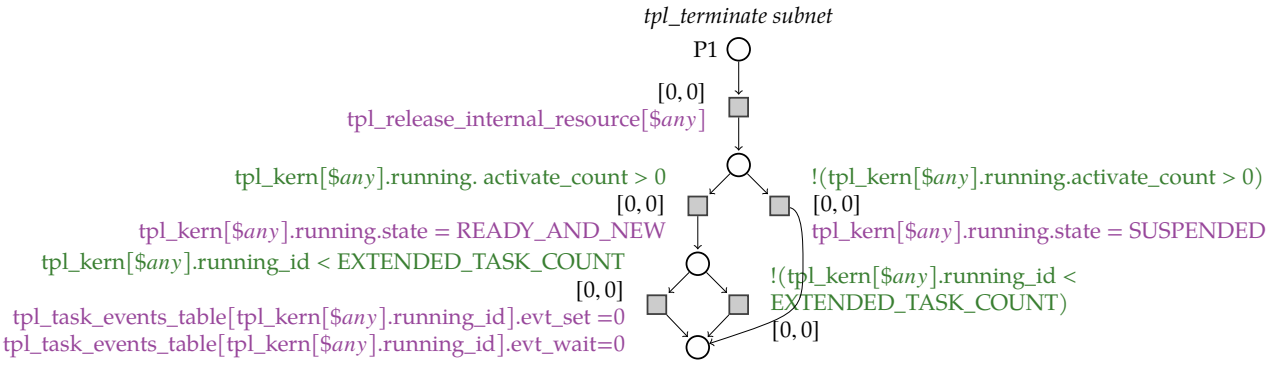


Figure 5.4: The kernel function `tpl_terminate`.

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 multi-core version of Trampoline implements a fixed priority partitioned scheduler. The scheduler uses functions to handle the list of ready tasks and ISRs of category 2. Among the functions,

2. The state `READY_AND_NEW` is identical to the state `READY` but the execution context of the task is not yet initialized.

we can highlight the `tpl_put_new_proc` function, explained in 3.5.2. Its model is presented in Figure 5.5. The function starts its execution by calling some initialization functions written in the Roméo language. `GET_PROC_CORE_ID` (Figure 5.6) initializes the variable `core_id_var` with the `core_id` assigned to the process passed as an argument. The `core_id` is obtained from the process static descriptor stored in a table. `GET_CORE_READY_LIST` initializes the `ready_list` belonging to the core `core_id`. The structure `ready_list` is a binary heap. Each element has two fields, the dynamic process priority (key) and the process identifier (id). The key is the concatenation of the priority and the rank of the job. Finally, `GET_TAIL_FOR_PRIO` initializes the `tail_for_prio` with the rank table of core `core_id`. `tail_for_prio` stores the last rank used to store a process. The variable $*any* represents the `core_id` of the running core. The second transition represents the dynamic priority calculation, obtained by concatenating the static priority and an order number per priority level. `PRIORITY_SHIFT` is used to shift the key part used to store its priority and `RANK_MASK` to get the part of the key used to store its rank. Then the new entry is added at the end of the ready list, and the `tpl_bubble_up` [3] subnet is called to bubble the entry at the index place up in a heap.

Interrupt dispatcher The model of this component is built with a subnet, allowing the management of software and hardware interrupts. To notify an interruption, a token with a color corresponding to the target core is put in the `tpl_it_handler` first place, represented in Figure 5.7. Then interrupts are masked on the core (`kernel_mode[$any]=1`), the global lock is taken if possible (`lock_kernel=1`). The different requests from the interrupts are not considered until the called services leave the kernel mode. The counter manager is called when the interrupt source is the counter tick. The inter-core interrupt manager is called to handle the interactions and communication between the cores. The central interrupt handler is called to manage the ISR. Next, the interrupt handler interacts with the scheduler and the context switch manager if the triggered interrupt causes the running task to lose the processor. Finally, the global lock is released, and the core returns to user mode, unmasking interrupt consequently.

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

3. The `tpl_bubble_up` function compares the added element with its parent; if they are in the correct order, the operation stops. If not, the element is swapped with its parent, and the operation returned to the previous comparison step.

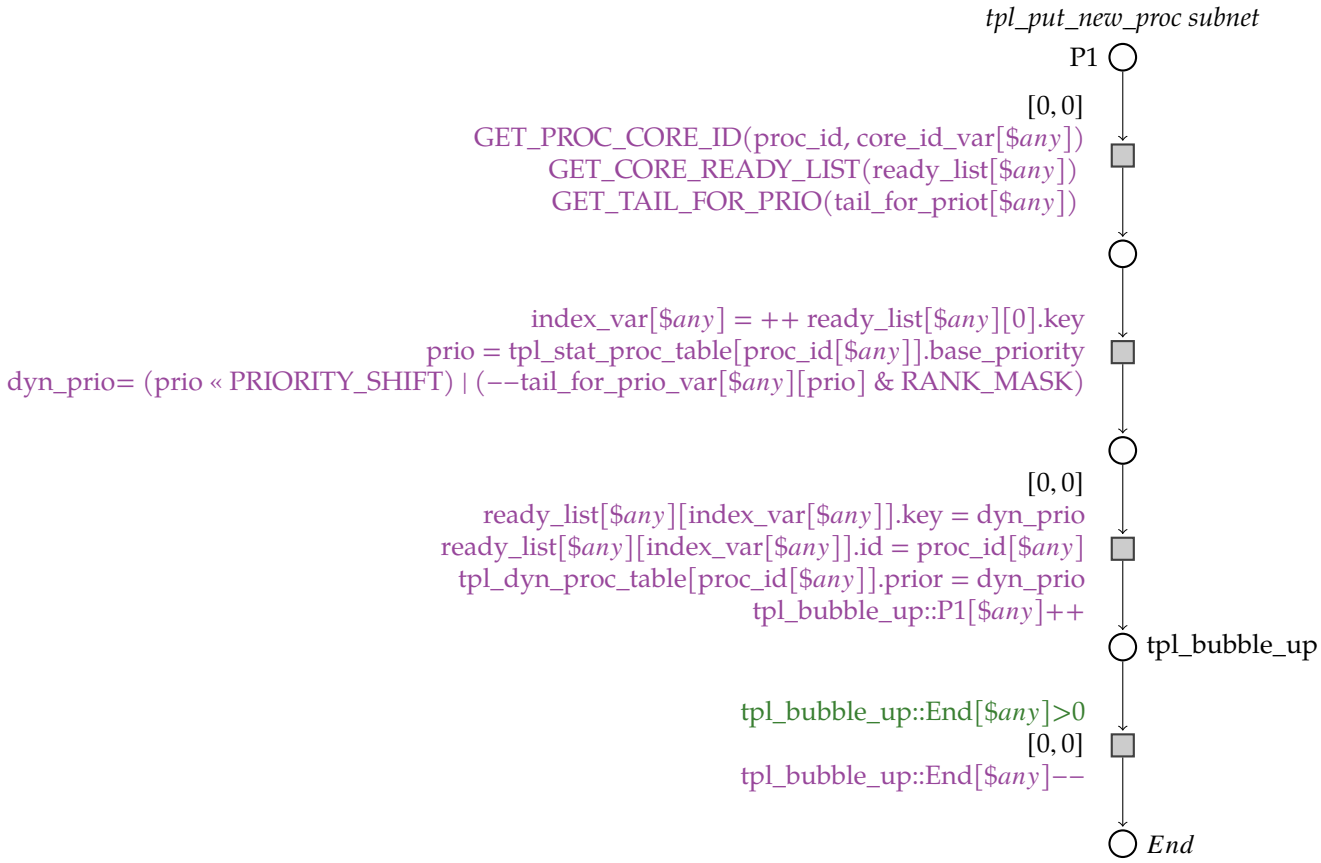


Figure 5.5: tpl_put_new_proc model.

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 describes when an alarm time object is raised.

Alarm base information can be obtained through the `tpl_get_alarm_base_service` kernel function called by the GetAlarmBase API service (Figure 5.2). Figure 5.8 presents this HCTPN model that provides 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 the Roméo language.

```
void GET_PROC_CORE_ID (tpl_proc_id a_proc_id, tpl_core_id &a_core_id)
{
  if (N == 1)
  {
    a_core_id = 0;
  }
  else
  {
    a_core_id = tpl_stat_proc_table[a_proc_id].core_id;
  }
}
```

Figure 5.6: GET_PROC_CORE_ID C-like function. N is the number of cores.

5.3.3 Properties of the model

In the absence of an application, the model of the OS kernel remains in its initial state. 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 $\mathcal{N} = (P, T, X, C, \text{pre}, \text{post}, (m_0, x_0), guard, update, I)$ 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.

All the states would be observable by modeling the OS kernel with an assembler instruction per Petri net transition. We would get a perfectly equivalent net to the kernel but at the cost of a state space explosion.

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 the instructions associated with a transition $t \in T$ are executed sequentially (as the real code of the kernel on a given core) and considered as atomic in the state space. For observing a particular state enclosed in a sequence of instructions associated with a transition t, you only need to add a place in the Petri Net at the point you want to observe.

The use of colors allows the simultaneous enabling of transitions for different cores, but the kernel access is sequenced thanks to a global lock, and atomicity is applied on

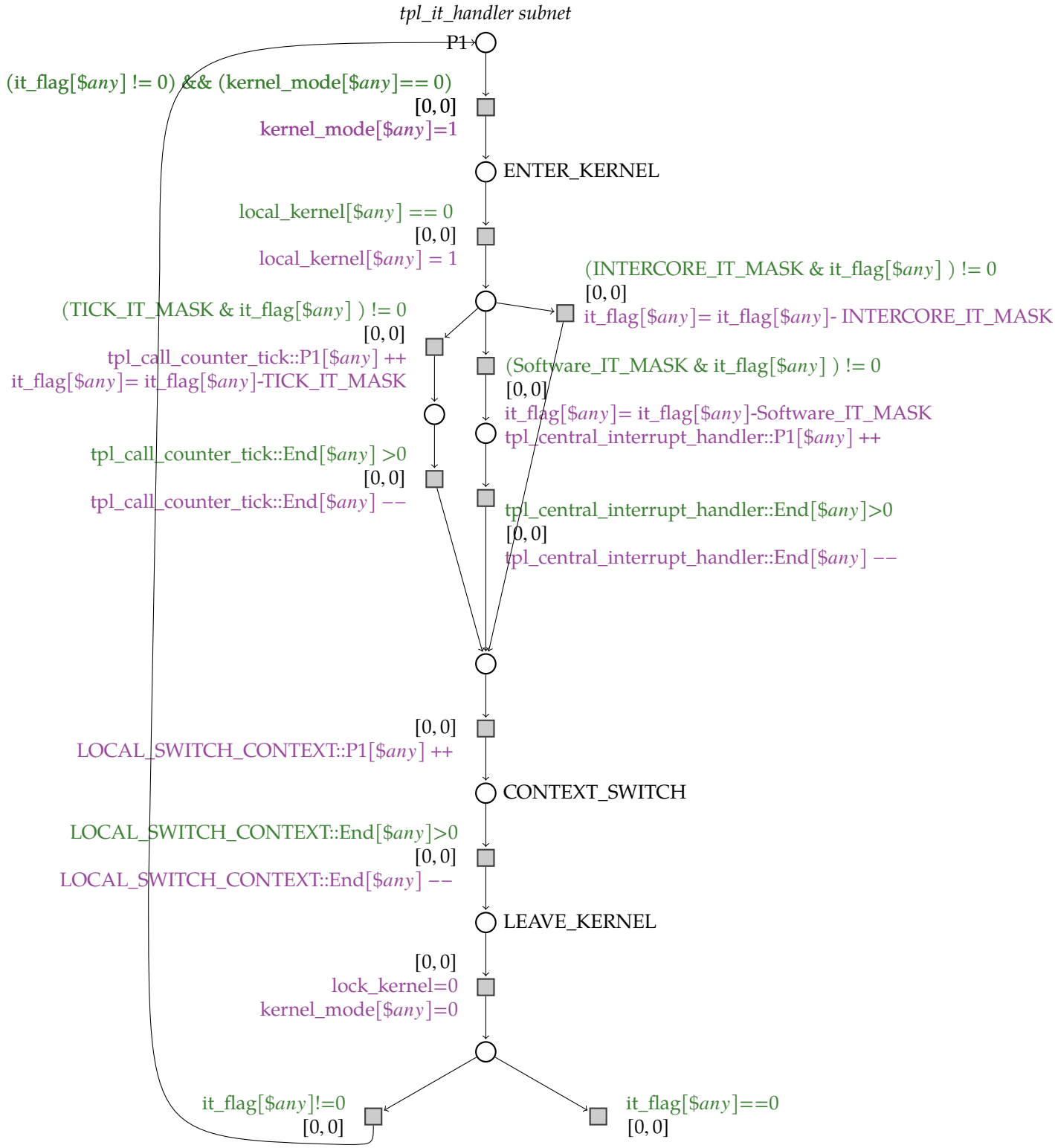


Figure 5.7: tpl_it_handler model.

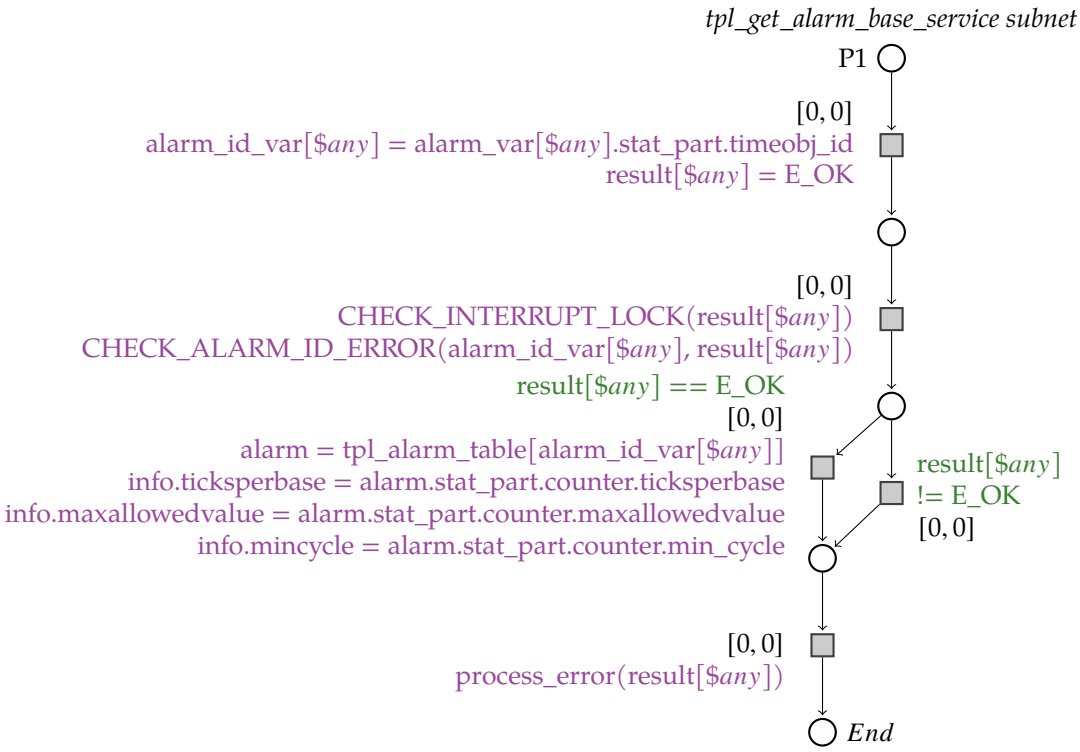


Figure 5.8: tpl_get_alarm_base_service model.

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 [10], 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 the reading and writing of a global variable are not on the same 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 $\mathcal{N}$. □

5.4 Application modeling

To perform verifications, it is necessary to add an application model to the RTOS model. An application contains a concurrent set of tasks that interact with the operating system through system calls such as `ActivateTask()` or `TerminateTask()`, interspersed with execution times given as an interval $[BCET, WCET]$ [4]. Its model consists of two parts.

The first part is the set of data structures corresponding to the descriptors of the different objects that appear in the application: task, alarms, spinlocks, etc. The attributes of the objects constitute part of the model variables. These data structures are transposed from the C language to ROMÉO's language, the model-checker that we use, which in its syntax is very close to the C language. The pointers sometimes used in these structures are translated into indexes in arrays. 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 5.9.

The second part is a set of Petri subnets, one per task or ISR of the application. The

4. $WCET$ 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.

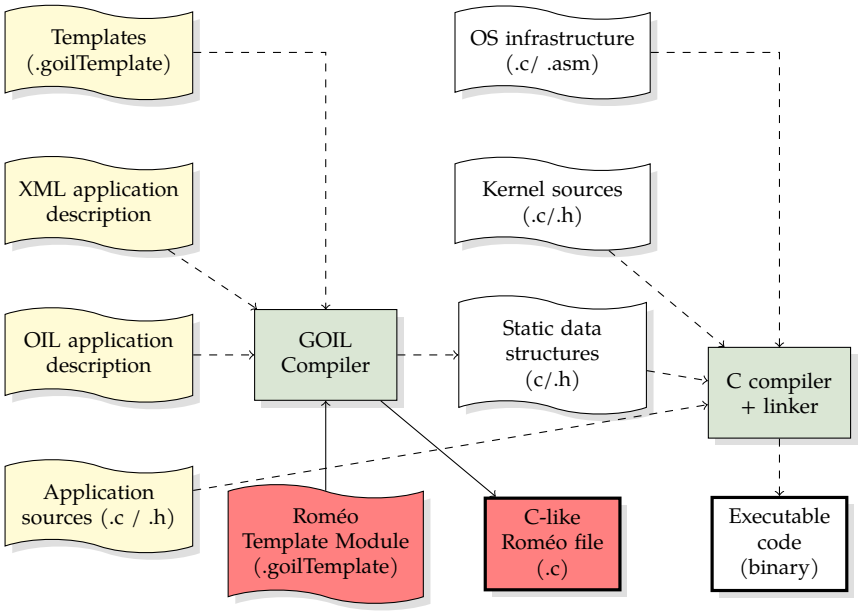


Figure 5.9: Trampoline application configuration with the added GTL module.

network reproduces the control flow graph of the task or ISR. The fact that a task runs in the model is controlled by the OS model by means of the derivative function $isRunning(task_i)$ 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.

5.4.1 The GTL module

Goil compiler includes a template interpreter for file generation. Figure 5.9 shows templates 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. GTL supports types (struct, boolean, list, integer, and string) and has readers to get variable information. The syntax of this language is detailed in the Trampoline OS documentation in the git [5].

The added GTL module is a set of template files that produces the Roméo file. A piece of code from the GTL module and the generated output file `tpl_init_romeo.c` after compilation are shown in Figure 5.10. The % character is used to switch from literal text mode

5. `https://github.com/TrampolineRTOS/GTL/blob/master/documentation/GTL.pdf`

GTL module

```
 %
  write to "tpl_init_romeo.c":
%
 /*
 * NUMBER OF CORES
 */
const int N= % !exists OS::NUMBER_OF_CORES default (1) %;
 /*
 * Priority of RES_SCHEDULER
 */

const int RES_SCHEDULER_PRIORITY= %
!OS::RESSCHEDULERPRIORITY %;

 /*
 * PRIORITY_MASK and RANK_MASK
 */

const int PRIORITY_SHIFT =% !PRIORITY_SHIFT %;
const int PRIORITY_MASK =% !PRIORITY_MASK  %;
const int RANK_MASK = % !RANK_MASK %;

/*
 * Definition and initialization of Task-related structures
 */
%
  do template task_descriptor_romeo
  end foreach
```

C-like output file

```
/*
* NUMBER OF CORES
*/
     const int N= 2;
/*
* Priority of RES_SCHEDULER
*/
     const int RES_SCHEDULER_PRIORITY= 2;
/*
* PRIORITY_MASK and RANK_MASK
*/
     const int PRIORITY_SHIFT =2;
     const int PRIORITY_MASK =12;
     const int RANK_MASK = 3;
/*
* Definition and initialization of Task-related structures
*/
/* Static descriptor of task t2 */
     tpl_proc_static t2_task_stat_desc;
     t2_task_stat_desc.internal_resource=
     INTERNAL_RES_SCHEDULER; /* internal ressource */
     t2_task_stat_desc.id = t2_id; /* task id */
     t2_task_stat_desc.core_id =    1; /* core id  */
     t2_task_stat_desc.base_priority = 1; /* task base priority */
     t2_task_stat_desc.max_activate_count = 1; /* max activation
     count  */
     t2_task_stat_desc.type = TASK_BASIC; /* task type */
     t2_task_stat_desc.app_id=SlaveApplication_id; /* OS
     application id */ ....
```

Figure 5.10: GTL module and the C-like output file.

to program mode. The module comprises descriptors templates in its root that output the C-like Roméo file. The `write` statement defines the output of the template processing in the specified file. Thus, we automatically extract the data structures and constants generated from the OIL description of the application and translate these structures into C-like language. Once the configuration file is defined, we model the application's source code by a Petri subnet which describes all the system calls performed.

5.4.2 Modeling examples

An example of modeling an application task is shown in Figure 5.11. The task calls the `GetAlarmBase` service, which provides information on the alarm base. The $IsReady(task_i)$ guards on Act_{task_i} means that the task model is ready for execution. The $IsRunning()$ derivative function associated with transitions returns 1 when the task $task_i$ is scheduled on the assigned core, and we are in the user mode (`kernel_mode==0`). The transitions with intervals of the form $[BCET, WCET]$ allows checking the application schedulability and temporal properties, as we will explain in chapter 7.
Figure 5.12 shows a modeling example of a two-tasks application. It represents a task

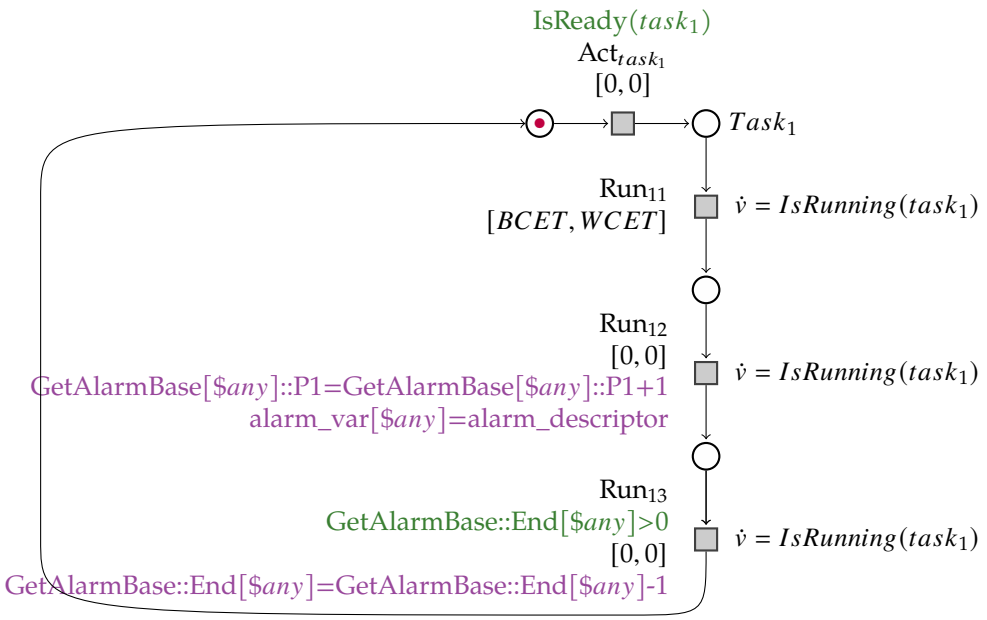


Figure 5.11: Application task model.

activation on another core. $task_1$ is automatically activated and runs on core 0, associated with the red color. $task_2$ runs on core 1, which is associated with the blue color. Applications are modeled with the same approach. The `IsRunning()` derivative function checks that the task passed in the parameter is running in user mode, and its boolean return allows the elapse of time when it is equal to 1; otherwise, it's blocked. $task_1$ is running on core 0 and after a time in the range $[2, 4]$, $task_1$ activates $task_2$ on core 1. In this case, the *Biglock* is taken by core 0 when the `ActivateTask` service is called, and core 1 waits actively for its release to take over the interrupt and make the context switch. In the model, handling the concurrency of service calls in parallel on different cores is represented by the *lock_kernel* variable. It is set to 1 and reset to 0 at the end of the execution of each service call. Since the interrupts are masked and can not run concurrently, a *kernel_mode* table is used, allowing simultaneous access through the *$any* variable.

5.5 Conclusion

At the end of our modeling work, we have a model of the Trampoline multi-core RTOS that we can complement with an application model. This model is described by HCTPNs and Roméo functions written in a syntax similar to the C language. The model is struc-

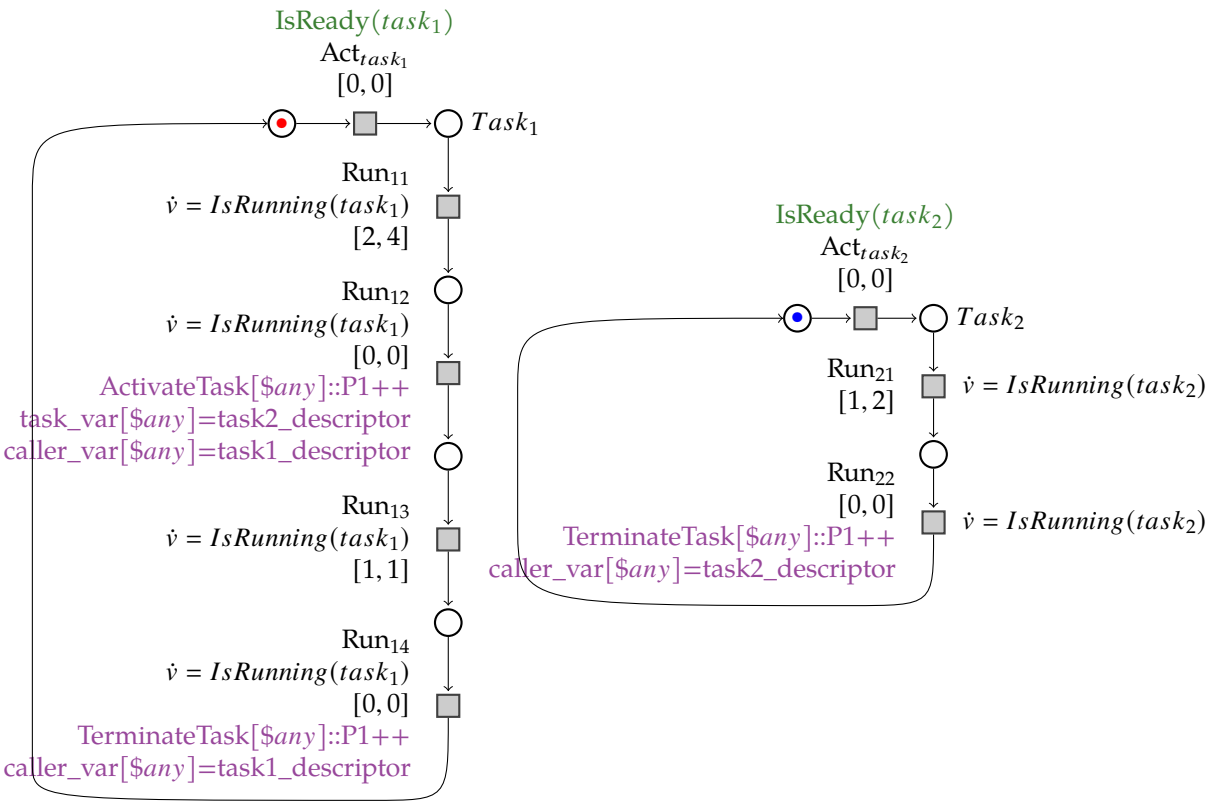


Figure 5.12: Application model.

tured to avoid state explosion with parts corresponding to a sequential code execution treated as an atomic transition w.r.t. the higher-level net. The entire model will be used in our verification approach presented in the following chapters.

Chapter 6

Formal verification of the multi-core Autosar OS compliance

6.1 Introduction

This chapter presents a formal process to verify multi-core OS compliance with the AUTOSAR standard and the synchronization mechanisms involved in the concurrent execution of OS services. AUTOSAR conformance testing is based on requirements verification by executing a test suite. The first part focuses on multi-core OS requirements, of which there are 80. Each requirement is formalized by an observer that evaluates compliance. The approach results conclude that the operating system model respects the AUTOSAR specifications. The second part of the verification is based on rare situations with simultaneous service calls in parallel on several cores that are almost impossible to test on real implementation. However, errors can be automatically identified with the model-checking method.

6.2 Formal verification of Autosar compliance

We propose an approach based on model-checking to verify the conformity of the operating system to the AUTOSAR standard. Figure 6.1 shows the two main stages of the process. The first phase involves developing a complete model that includes all the OS and application functions and services as represented in chapter 5. 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 as external observers able to evaluate the respect of the requirements. In the first case, the TCTL logic allows expressing requirements as properties in Roméo. However, these requirements can easily become difficult to express by involving several

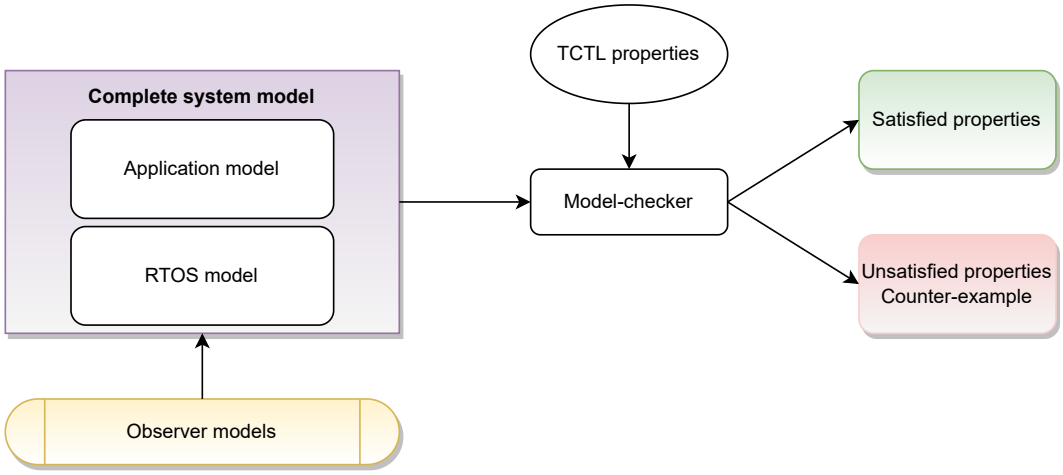


Figure 6.1: Verification approach.

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 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 AUTOSAR requirements are thus translated into observers describing the expected behavior. Then, with the help of reachability properties written in TCTL logic, we can verify by the model-checker their satisfaction or generate, on the contrary case, a counter-example trace.

6.2.1 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 completed correctly lead to the success of the test suite, thus checking the conformance.
We rely on the set of multi-core test cases developed by the Trampoline project to verify the OS compliance with the AUTOSAR standard. The project is available in the Trampoline repository[1], and it contains 75 OS-specific tests, of which 18 are dedicated to multi-

1. They are available in the Trampoline repository: `https://github.com/TrampolineRTOS/`

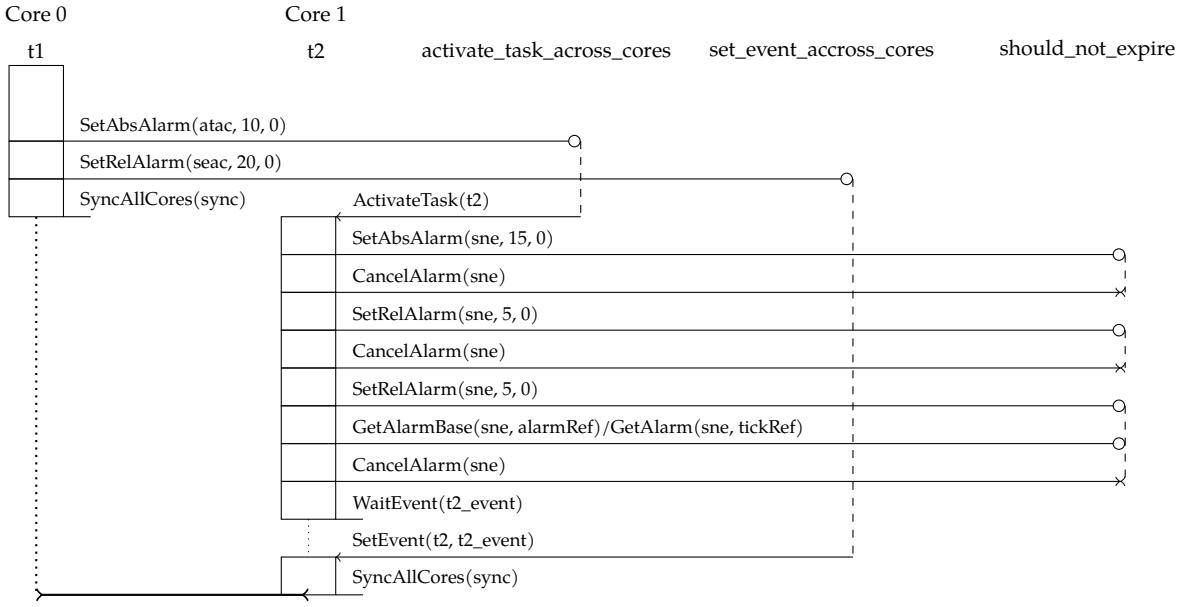


Figure 6.2: mc_alarm_s1 test sequence. ∘ represents the success of the system service call and × the canceling of the alarm.

core. These included test sequences have been run on several hardware targets showing Trampoline RTOS compliance. We illustrate the first AUTOSAR test sequence of the Trampoline repository, mc_alarm_s1, in Figure 6.2. 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 the figure 6.2. This sequence was developed to verify the AUTOSAR requirements from SWS_Os_00632 to SWS_Os_00640 in Table A.1, shown on page 147. We detail our verification approach on this application in 6.3.1.

6.2.2 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

`trampoline/tree/master/tests/functional`

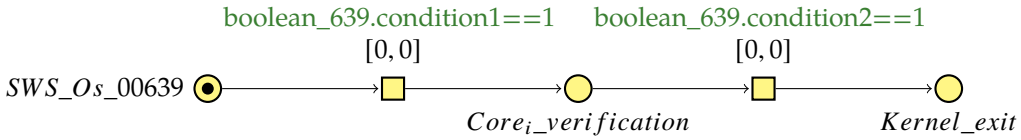
requirement.

How an observer works in the model? The observer relies on functions written in Roméo language, returning a boolean according to the satisfaction of the conditions forming the requirement. The observer waits in its initial state containing a token until the condition becomes true to evolve and fire its transitions. Thus, each requirement is translated by a test function that returns *true* for each satisfied condition. If the first condition is true, the observer model moves to the next state to check the second condition until reaching the final state. These functions are called in the RTOS model at locations updating the data structures involved in the verification. Correct verification of future states is ensured by resetting the rest of the conditions once the first one is true.

Requirement observer model Let's consider the observer model of the SWS_Os_00639 requirement (Figure 6.3), which verifies 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. They contain only committed transitions that are transitions with a priority to guarantee they are fired before all the other system transitions. Thus, if there are several fireable transitions at a given state of execution, the committed ones are fired first before all the others.

6.2.3 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 to be verified. Requirement in Roméo are expressed in a subclass of TCTL temporal logics [28] and verified with an on-the-fly efficient algorithm as represented in 4.5. An execution trace is automatically generated as a counter-example if the property violation is detected. The AUTOSAR 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. Thus, based on the observer model of the SWS_Os_00639 requirement (Figure 6.3), the




```
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;
}
```

Figure 6.3: SWS_Os_00639 Observer model.

corresponding verification formula is as follows: `(SWS_Os_00639[0])->(Kernel_exit[0])`. The token in the initial place of the model triggers the observer once the guard condition is satisfied.

6.3 Compliance of the AUTOSAR Trampoline OS

The enriched model with observers allows checking AUTOSAR requirements on any application, as each observer represents a specification. We focus on the set of multi-core test sequences proposed by the Trampoline project to conduct a formal verification with the ROMÉO model-checker. The observers are modeled following the procedure detailed in the previous section. We illustrates the application of our formal verification approach on the two applications mc_alarm_s1 and $mc\_spinlock\_s1$ of the Trampoline AUTOSAR test repository 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. The application model is constructed for each test sequence to verify the whole multi-core requirements.

6.3.1 mc_alarm_s1 application

This part focuses on the first multi-core test sequence of the Trampoline repository, mc_alarm_s1, represented in Figure 6.2. 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 5.2), 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.

6.3.1.1 Application model

The developed application model precisely describes the life cycle of each task through the performed system calls. Figure 6.4 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.

6.3.1.2 Verification results

We apply our verification approach to check the requirements covered by this example. We formalize each requirement by an observer model as presented in 6.2.2. For example, the first requirement *SWS_Os*_00632 is represented by the observer model in Figure 6.5. 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 6.4. 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.
Table 6.1 shows the verification results of the requirements covered by this application, listed in Table A.1. 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

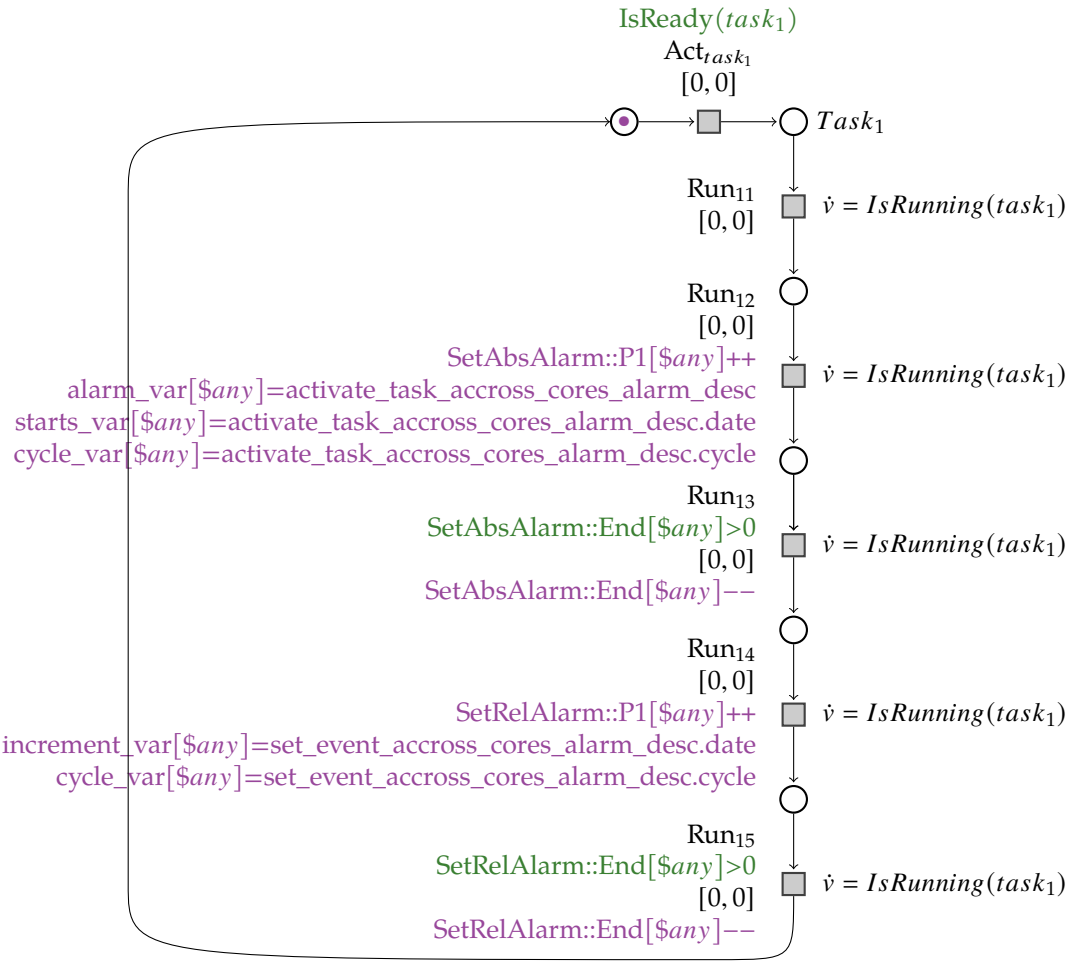
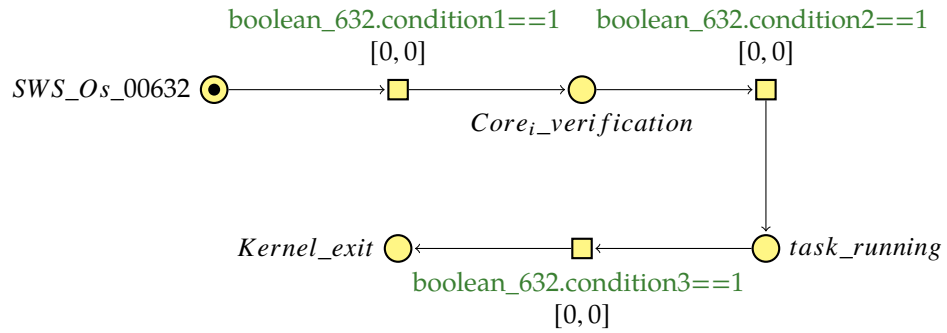


Figure 6.4: $Task_1$ model of the mc_alarm_s1 test sequence.




```
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;
}
```

Figure 6.5: SWS_Os_00632 Observer model.

column shows that the property is satisfied by the model. The requirements verification is performed in a similar time and memory. For all verifications, the result is *true*.

Table 6.1: Computing time and memory used for verification - mc_alarm_s1.

$(p) \rightarrow (q)$		
	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

6.3.2 mc_spinlock_s1 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 $RES\_SCHEDULER$ is obtained, 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 had already taken 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 6.6 the $mc\_spinlock\_s1$ multi-core test sequence of the Trampoline repository. The application contains four spinlocks handled by the services mentioned

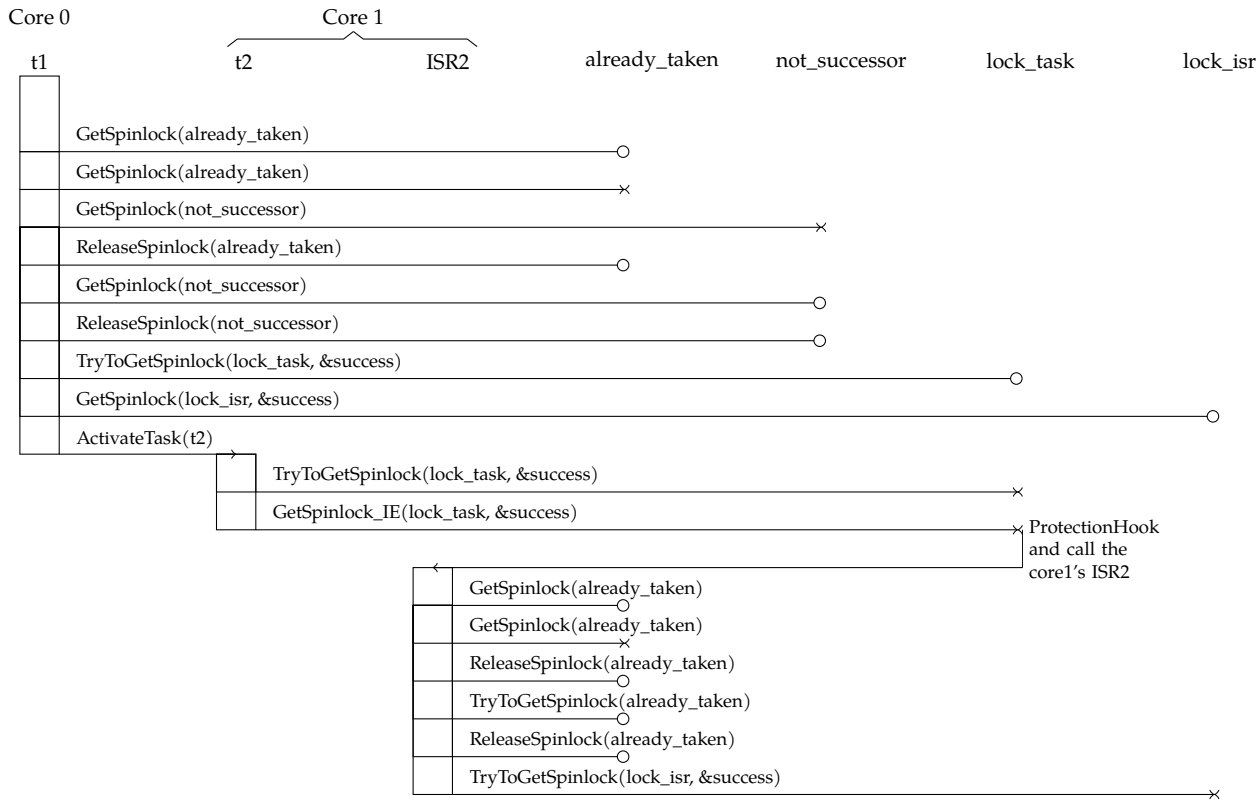


Figure 6.6: mc_spinlock_s1 test sequence.

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 6.6. 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 to enable the core1's $ISR2$. This application verifies the requirements from SWS_Os_00649 to SWS_Os_00661 listed in Table A.1.

6.3.2.1 Application model

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 6.7. 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.

6.3.2.2 Verification results

We conduct the verification on the elaborated application model by adding the observers. All the requirements tested by this application are formalized. Figure 6.8 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 A.1. 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 6.2. For all verifications, the result is *true*.

6.3.3 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 multi-core interleaving scenarios such that all

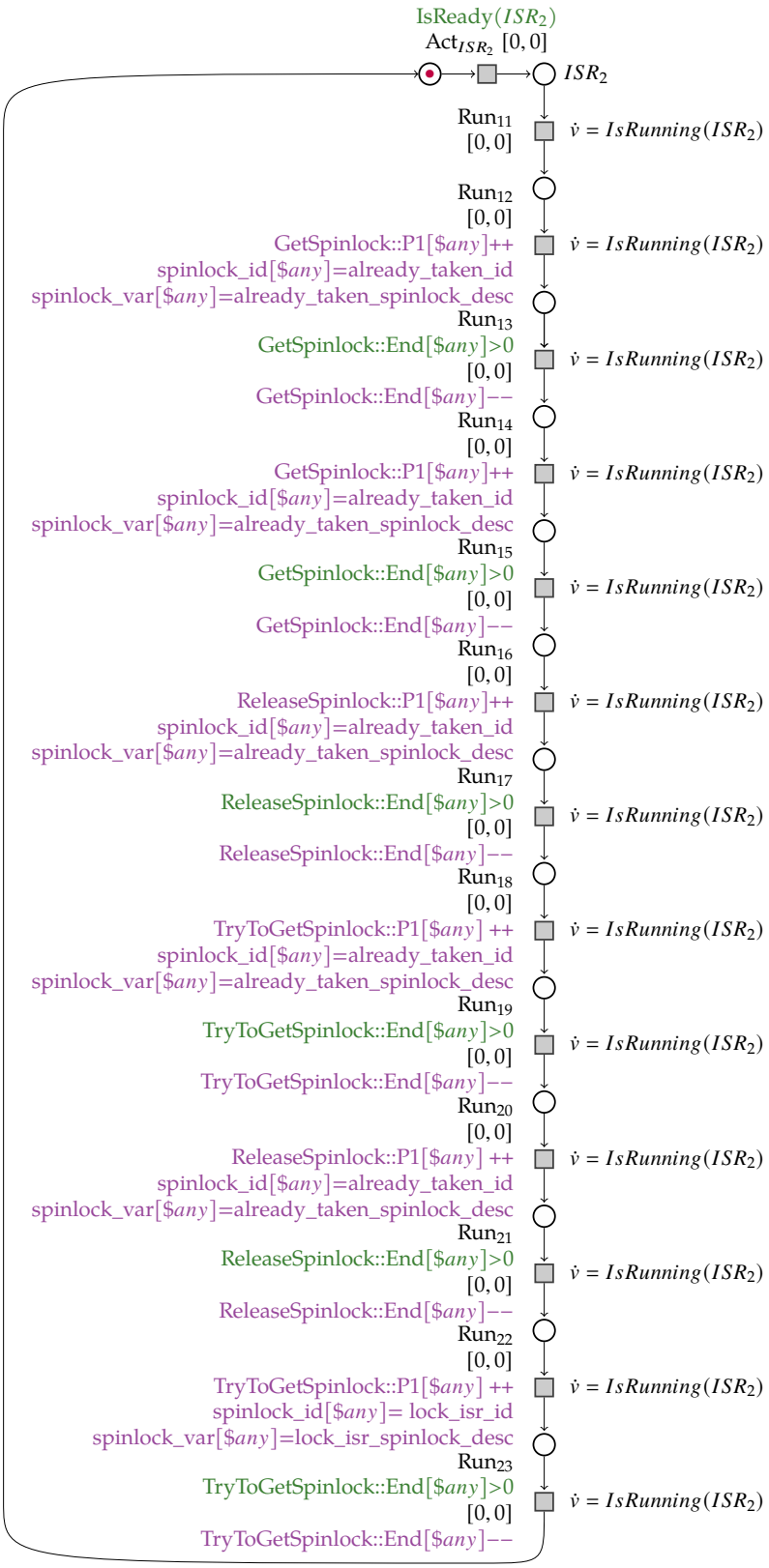
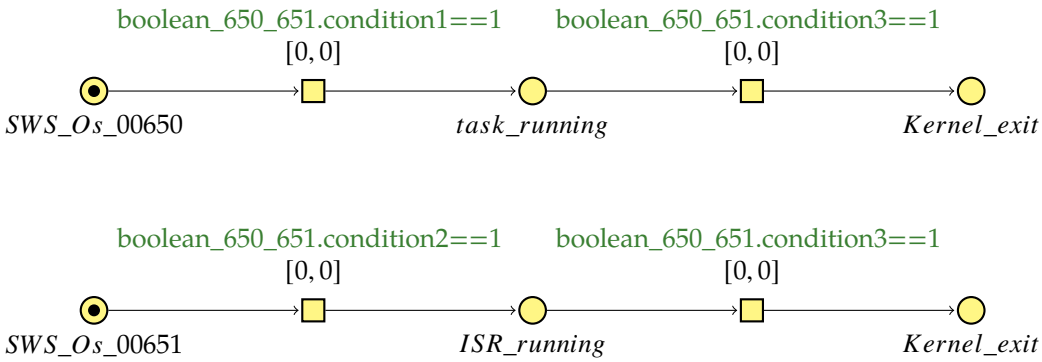


Figure 6.7: ISR_2 model of the $mc\_spinlock\_s1$ test sequence.




```
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;
}
```

Figure 6.8: SWS_Os_00650 and SWS_Os_00651 Observer models.

Table 6.2: Computing time and memory used for verification - mc_spinlock_s1.

$(p) \rightarrow (q)$		
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

concurrent events are enumerated in the state space, which makes its size vary exponentially. If this size exceeds the amount of computer memory, the exhaustive verification fails. The response time of the model-checker represents the time needed to explore the set of state spaces and check the property. All AUTOSAR multi-core operating system specifications were met for the modeled test applications [2], and the verification time is between 2.7 and 11 seconds consuming between 100 and 600 MB of memory. 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.

2. *mc_alarms_s*1 to *mc_taskTermination_s*2 except *mc_taskTermination_s*1, *mc_schedtables_s*1, and *mc_autostart_s*3: `https://github.com/TrampolineRTOS/trampoline/tree/master/tests/functional`

6.4 Formal verification of concurrency in multi-core implementation

This section presents the formal verification of the Trampoline multi-core RTOS in concurrent situations using the model-checker Roméo. Since the test sequences derived from the AUTOSAR requirements for Trampoline do not allow the testing of concurrency situations, they cannot be used as is. Indeed, the multi-core system call lets us suppose that a problem could occur when services are called simultaneously on two or more cores (3.5.2). We can verify that it is probably not enough to protect access to the OS data structures (the list of ready tasks and the structure `tpl_kern`) through the *Biglock*. Indeed, the structure `tpl_kern` allows communication between the core where the rescheduling is performed and the core where the context switch is performed, and the example of Figure 3.12 suggests that a problem could occur when services are called concurrently on two or more cores. We thus propose to use model-checking to verify the communication and synchronization mechanisms involved in the concurrent execution of OS services by an application: concurrent accesses to OS data structures, multi-core scheduling, and inter-core interrupt handling.
The application can implement, for example, a cruise control system in the vehicle. This system consists of several runnables responsible for data acquisition, diagnostics, and vehicle speed control. A task comprises runnables that must be performed according to a priority level. Its non-termination can cause severe consequences and should be avoided even in rare situations. Our goal is now to study situations with simultaneous service calls in parallel on several cores that are almost impossible to test on real implementation but that we will be able to obtain by our model-checking approach. Therefore, in order to prove that concurrent execution of services is error-free, we have designed several case studies. In the following, we present two of them that led us to identify two errors in the multi-core implementation of Trampoline.

6.4.1 Case study 1

To check for concurrent system calls on cores, we consider the system in Figure 6.9 which includes all possible interleaving by setting all transitions in the interval $[0, 0]$. It represents an activation of a higher priority task $Prio(task_1) < Prio(task_3) < Prio(task_2)$. Once the autostart $task_1$ is started on core 0 at system start-up time, it activates $task_2$

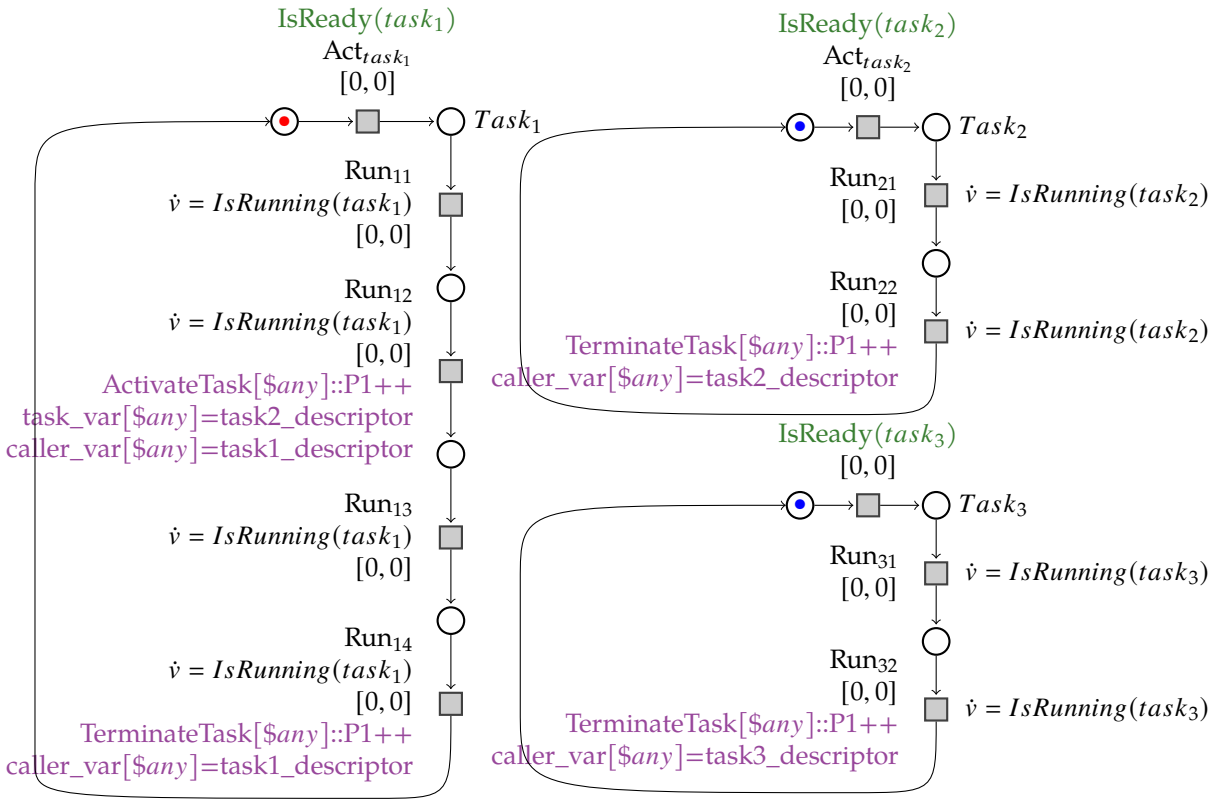


Figure 6.9: Three-tasks application model.

on core 1, and terminates afterwards. Core 0 is associated with the red color. $task_2$ and $task_3$ run on core 1, which is associated with the blue color. The purpose is to check that tasks will run and terminate their execution on cores, whatever the interleaving. $task_3$, which is an autostart task with a lower priority than $task_2$, is assumed to be preempted by $task_2$.

Formal analysis We run a complete analysis of the system, the application, and the RTOS, using Roméo. We check that all the application's tasks complete their execution in a concurrent context. We verify that the places $TerminateTask_i[task_i.core\_id]$ are always marked by a token with the property: $AF(TerminateTask_1[0] > 0\, and\, TerminateTask_2[1] > 0\ and\ TerminateTask_3[1] > 0)$. Roméo model-checker replies that the property is not satisfied, and a counter-example execution trace is generated, proving that $task_2$ never ends its execution and the place $TerminateTask_2[1]$ is never reached in a given case. The trace provided by the Roméo model-checker gives the following execution order:

1. StartOS: We start the operating system in the application mode. It first makes some

initializations, activates the autostart tasks and alarms, performs scheduling, and executes the highest priority task. Thus, the autostart $task_1$ and $task_3$ are run in this startup phase;

2. ActivateTask: The service is called by $task_1$ running on core 0 in order to activate $task_2$ on core 1. The rescheduling for core 1 is performed on core 0, where the service call occurs. Core 0 sends an inter-core interrupt request to core 1 by setting the variable `it_flag`. Figure 5.7 represents the interrupt handler model. The core 1 blocks on an active wait for the release of the local and global lock;

3. TerminateTask: The service is called by $task_3$ running on core 1. $task_3$ completes its execution on core 1 and the context switch is performed to the $idle$ task;

4. Handler: Core 1 enters the kernel to execute the inter-core interrupt and perform the context switch (Figure 5.7). Since the `need_switch` flag of the `tpl_kern` structure is set to `NO_NEED_SWITCH` after the execution of the idle task on core 1, the context switch is not achieved. Thus, we return to the execution of the $idle$ task;

5. TerminateTask: The service is called by $task_1$ running on core 0; $task_1$ ends its execution and the context switch is performed to the $idle$ on this core.

The rescheduling performed by core 0 for core 1 in step 2, elects $task_2$ and extracts it from the `ready_list`. When the inter-core interrupt is executed, the context switch is performed to the task extracted from the `ready_list` if the `need_switch` flag is true. By calling the `TerminateTask` service in step 3 before executing the interruption, the first element of the `ready_list` is extracted, which is the $idle$ task. Thus the problem occurs, and the context switch is made to $idle$ task instead of $task_2$. In other words, the activation of $task_2$ on core 1 is lost because the termination of task $task_3$ on core 1 elects task $idle$ without checking that the already elected task ($task_2$) has a higher priority.
This scenario is obtained thanks to the model-checking. In the process, the other possible paths in this concurrent situation are also verified and the model-checker finds that the property hold for these other paths.

6.4.2 Correction of the error

The ready list is modified when the executing OS call service leads to a rescheduling. To fix the problem of this case study, we have to test before the extraction of the task at

the front of the ready list that it has a higher priority than the already elected one. If this elected task has a priority equal to or higher than the one of the first task in the ready list, then the rescheduling is correct, and the extraction is useless. This modification guarantees a context switch to either the already elected task or the newly elected task with the highest priority.

Computing time Table 6.3 shows the computing time and the amount of memory used in this analysis. The second column gives the data for the application model shown in Figure 6.9. The third column corresponds to an application running on three cores. It is obtained by duplicating $task_3$ and assigning it to the third core to evaluate the model-checking computation time when increasing state space. Table 6.4 represents the computing time after fixing the error in the model. The verification time is longer because all possible interleavings are considered.

Table 6.3: Computing time and amount of memory used.

	$AF(TerminateTask_i[task_i.core\_id] > 0)$	
Number of cores	2	3
Model-checker result	false	false
Memory used	96.0MB	174.9MB
Computing time	7.1s	10.7s

Table 6.4: Computing time and memory used after correction.

	$AF(TerminateTask_i[task_i.core\_id] > 0)$	
Number of cores	2	3
Model-checker result	true	true
Memory used	117.6MB	127.8MB
Computing time	51.1s	2629.2s

6.4.3 Case study 2

Let us consider a modification of the application presented in figure 6.9 and obtained by replacing the call to the `TerminateTask` service with a call to the `GetTaskID` service in $task_3$. `GetTaskID` writes in the `TaskID` variable the identifier of the task currently running. If no task is running, for example if `GetTaskID` was called from an ISR (Interrupt Service Routine), `INVALID_TASK` is returned. We want to verify if the *Biglock* is enough

to protect the OS data structures access when we have concurrent calls to `ActivateTask` and `GetTaskID` services. Since `GetTaskID` is not a service that causes rescheduling, calling it should have no influence.

Formal analysis In Trampoline, a system call handler performs the operating system service call as explained in 3.5.1. During this process, the `need_switch` and `need_save` fields of the `tpl_kern` data structure belonging to the core on which the service is performed are reset before calling the kernel function.
We conduct the same reachability verification with the Roméo model-checker leading to the detection of the following problem: When the `GetTaskID` service call is made in concurrence with the execution of the inter-core interrupt on core 1, the `need_switch` and `need_save` flags are reset by the system call handler. The information is therefore lost for the inter-core interrupt handler. The trace provided by the Roméo model-checker is as follows:

1. StartOS: We start the operating system in the application mode, and the autostart $task_1$, $task_3$ and $task_4$ are run;
2. ActivateTask: The service is called by $task_1$ on core 0 to activate $task_2$ on core 1. Core 0 sends an inter-core interrupt request to core 1 after the rescheduling;
3. GetTaskID: The service is called by $task_3$ on core 1. The `need_switch` and `need_save` flags of the `tpl_kern` structure are reset (`tpl_kern[1].need_switch=0`);
4. Handler: The inter-core IT handler is called (Figure 5.7) to execute the inter-core interrupt on core 1 and effect the context switch. The context switch is not performed since the `need_switch` and `need_save` flags of the `tpl_kern` structure are reset after the execution of the GetTaskID on core 1. Thus, $task_2$ is never executed on core 1.

6.4.4 Correction of the error

The error found here holds for any application, independently of the number of tasks on two or more cores, as the system call handler always resets the kernel data structure before calling the corresponding kernel function, resulting in a loss of information.
The solution we adopted is to move, inside the `sc_handler`, the reset of the `need_switch` and `need_save` flag. Instead of resetting them before the kernel function is called, they

are reset when they are taken into account to perform the context switch. In this way, services that do not cause a context switch leave `need_switch` and `need_save` unchanged, and the information is not lost to the inter-core interrupt. This change was modeled, and verification was performed by Roméo. It showed that the property is now satisfied. This way, the right value is preserved in the concurrent cases, and the problem is solved.

Computing time The computation time and the amount of memory used in this analysis are given in Table 6.5. As in Case Study 1, the three-core system is obtained by duplicating Task 3 and assigning it to a 3rd core. After error correction, property checking requires exploring the entire state space, which involves more computing time, as shown in Table 6.6.

Table 6.5: Computing time and memory used.

	$AF(GetTaskID_2[task_2.core\_id] > 0)$	
Number of cores	2	3
Model-checker result	false	false
Memory used	56.6MB	114.0MB
Computing time	2.6s	3.7s

Table 6.6: Computing time and memory used after correction.

	$AF(GetTaskID_2[task_2.core\_id] > 0)$	
Number of cores	2	3
Model-checker result	true	true
Memory used	96.7MB	150.1MB
Computing time	13.3s	914.8s

6.4.5 Scalability

The scalability of the approach is based on the number of cores, the number of tasks and their time interval, and the number of system calls that can be simultaneous. All these factors can increase the computation time during verification. Additional verifications with four cores, more than what is currently found in automotive embedded systems, each executing a task making concurrent system calls, run in 1124.8 seconds with 1141.3 MB of used memory and show that the approach scales to realistic automotive systems.

6.5 Conclusion

This chapter presented an approach that determines the compliance of the AUTOSAR multi-core real-time operating system (RTOS) from its formal model. The application models constructed represent the AUTOSAR multi-core test sequences developed by the Trampoline project. The Trampoline RTOS and the application models form a complete model that allows performing verification. For each test sequence, 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 proves 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.
AUTOSAR test cases do not test the kernel in concurrency situations and are therefore unsuitable as a basis for verifying the correctness of the kernel's mutual exclusion and communication mechanisms. Consequently, we checked the rescheduling performed by the cores and the inter-core interrupt, considering case studies with simultaneous service calls on the cores. We found that the rescheduling and the context switching in concurrent situations are not functional. The model-checker provided counter-example traces. The problems have been reported to the developers, and solutions are proposed to fix them formally in the model. Verification showed the property satisfaction. A complete research door is opened to find implementation errors with several multi-core concurrent examples based on the verification approach.

Chapter 7

Formal schedulability analysis based on multi-core RTOS model

7.1 Introduction

Verification of real-time application schedulability is usually performed using a very abstract system representation, which poorly supports inter-task dependencies. This chapter presents the schedulability analysis of the application running on the multi-core operating system model using model-checking techniques. Verification is performed using observers and allows determining the multi-core application's schedulability. Using parameters on the firing intervals allows determining under which temporal conditions the application is schedulable.
The multi-core has considerably complicated the scheduling problem, and a greater confidence in new scheduling policies implementation would be achieved by verifying them formally within the RTOS model. Simulation approaches quickly reach their limit as there is parallelism, so model-checking is adapted to multi-core context considering the parallelism, concurrency, and preemption provided state space explosion can be avoided. We show the possibility of verifying new partitionned scheduling policies within the multi-core RTOS using the HCTPN model. To illustrate this possibility with an example, we modify the Trampoline scheduling policy with an ad-hoc scheduler and verify its specifications and the application schedulability. This approach can be used not only to test new scheduling policies, but also to verify a scheduler for its implementation.

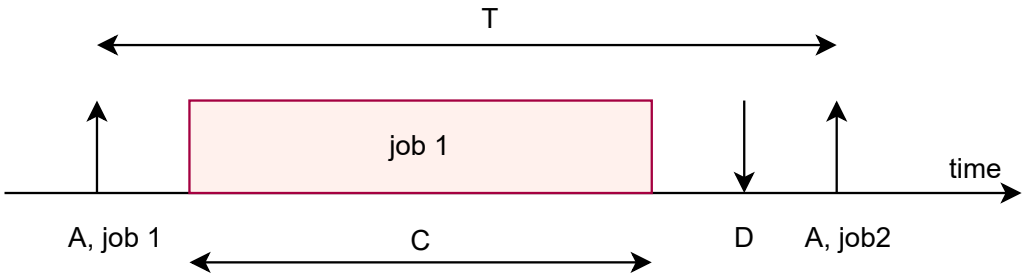


Figure 7.1: Task model.

7.2 Real-time schedulability analysis

The analysis of real-time systems seeks to ensure their temporal constraints. This means that the tasks that compose the system must respect the temporal deadlines according to the criticality, classified into different categories (hard real-time systems, soft real-time systems, firm real-time systems). These tasks can interact and share one or more resources. They can be:

- Periodic and be activated regularly with a fixed period;
- Sporadic and be started irregularly with a minimum duration between two task's instance (job) activations;
- Aperiodic and be activated irregularly.

Liu and Layland present the first model of periodic tasks [95] and describe a task using the parameters (C, T) shown in Figure 7.1, page 118. A is the activation date of the task's job. C corresponds to the $WCET$, which is the worst-case execution time. It is the duration considered the most in real-time scheduling analyses; in this study, we also consider the Best Case Execution Time ($BCET$) and the dependencies between tasks in a multi-core context. T represents its arrival period, and its deadline D can be implicit ($D = T$). Thus, a periodic task executes for C time units every T time units without missing the deadline D. The response time is defined as the duration between the arrival time of the task's job and its completion.
In multi-processor real-time system scheduling, tasks are scheduled on each processor with a scheduling policy that defines the rules that choose the order and tasks to be executed. These scheduling decisions can be made online or offline, i.e., during or before execution. Since the founding work of Liu and Layland in 1973 [95], various

multi-processor (multi-core) scheduling policies have been proposed in three principal categories [96]: partitioned, global, and hybrid. Scheduling multi-core systems is a two-dimensional problem, with a temporal organization on each core and a spatial one that ensures the job is executed on which core. Partitioned scheduling consists in treating each of the two dimensions separately. Thus the tasks assigned to a core are scheduled with a mono-core scheduling policy and are not allowed to migrate to a different core. Global scheduling treats both the temporal and spatial dimensions together. It then applies single scheduling for the whole multi-core system. Finally, hybrid scheduling combines partitioned and global scheduling and allows better control of tasks' migration.

Several research works have well-studied scheduling analyses in mono-core systems [97–99], and many studies illustrate the lack of guarantees for their analytical results by finding flaws [100–102]. In the multi-core systems, the existing methods for the schedulability analysis are studied in different research works [103–105]. As multi-core systems are more complicated, the schedulability analysis is affected by significant complexity and pessimism, especially for global multi-core fixed priority analysis [106, 107]. This motivates the use of formal methods to provide confidence in these scheduling analyses.

Most real-time operating systems in the multi-core case are based on partitioned schedulers with fixed priorities, even though several scheduling policies exist in theory. Indeed, scheduling policies are based on simplified assumptions and are sometimes described abstractly. However, the real multi-core context considers several aspects, such as parallelism, interrupt management, and possible interleaving due to concurrency. It would be interesting to study and verify other scheduling policies considering the RTOS. Model-checking is well adapted for verifying a scheduler within the RTOS that needs to consider all these aspects to check its policy or eventually implement it. However, model-checking using the HCTPN formalism with stopwatches is limited to partitioned scheduling where the temporal and spatial organization is separated. Indeed, if a task is preempted, the time associated with its stopwatch is frozen. Modeling the migration of this task to another core would lead to the color change and the stopwatch's reset, which explains the limitation of global and hybrid preemptive scheduling. We only consider in this work the preemptive partitioned scheduling.

7.3 Formal verification of schedulability analysis

Formal verification with models for schedulability analysis of real-time systems has been studied for several years, as presented in Section 2.3.4. This section presents our method for verifying quantitative temporal properties for real-time applications with preemptive scheduling based on the complete RTOS model built with HCTPN(Chapter 5). The defined scheduling policy of the AUTOSAR multi-core RTOS is based on a partitioned scheduling policy with fixed priorities. Each task is statically assigned to a core, can be preempted, and resumed at the same point later.
In our work, we propose a verification chain that includes the steps presented in Figure 6.1, page 96. 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. We illustrate the observer model associated with each task to check its respect or missing deadline. This process will be detailed in the following for the schedulability analysis.

7.3.1 Schedulability observer

A classical method for schedulability analysis is to rely on the use of observers [108], 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 7.2, page 121 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 that leads to an unbounded system. The schedulability property is then written for this observer by the CTL logic formula $AG(Obs_i < 1)$; if it is not satisfied, a counter-example execution trace is generated.

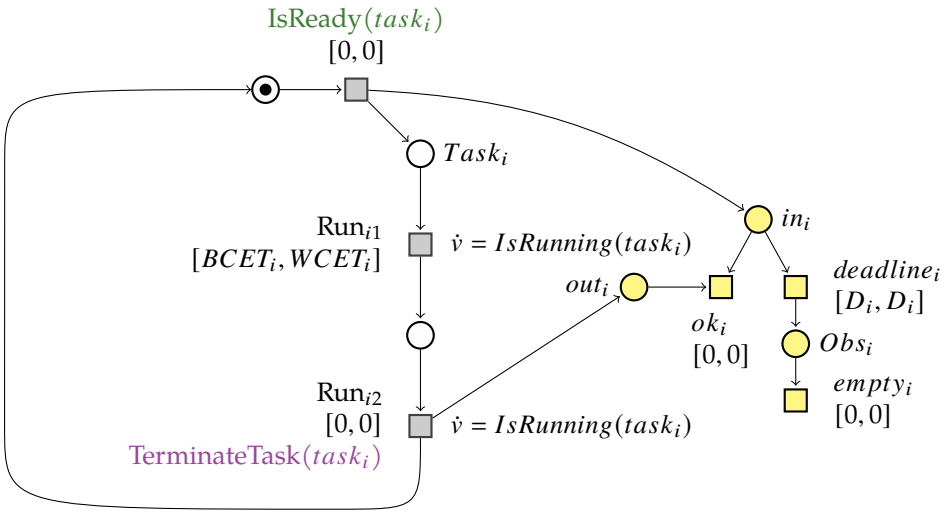


Figure 7.2: Observer model (in yellow) linked to a task model.

7.3.2 Parameters synthesis

The parametric analysis of HCTPNs with stopwatches using Roméo allows synthesizing the parameter values used in the time bounds of the transitions, such that the model verifies the TCTL properties [48]. It also allows adding linear constraints on the parameters to restrict their domain. The reachability problem is undecidable for HCTPNs with stopwatches and parametric HCTPNs with stopwatches, and semi-algorithms are implemented in Roméo for these models with polyhedra using the Parma Polyhedra Library [109]. For parametric models, the semi-algorithms are based on the parametric state-class graph [110]. For a given formula, by computing the parametric state space, Roméo synthesize a set of linear constraints over parameters which represents the values of the parameters for which the formula is verified.

Parameters can replace any interval bound, such as an offset, the BCET, or the WCET of a task model. 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 BCET and WCET of a task $task_i$ and by using the previous observer, checking $AG(Obs_i < 1)$ will synthesize all the linear constraints over the parameters BCET and WCET such that $task_i$ respects its deadline.

7.3.3 Application of the scheduling analysis approach

To illustrate the approach presented above, let us consider the system of Figure 7.3, page 123, with the characteristics of Table 7.1, page 122. The example contains three task models considering time intervals of runnable between $[BCET, WCET]$. $task_1$ is executed between 8 and 11-time units, activates $task_2$ then runs for 2 time units before terminating. We consider the following priority relationship: $task_2$ has the highest priority, followed by $task_1$ and finally $task_3$. At start $task_1$ and $task_3$ are automatically activated and run on core 0 and core 1 respectively. Core 0 is associated with the red color. $task_2$ is activated by $task_1$ and runs on core 1, which is associated with the blue color. Priorities and assignment of cores are done statically. Considering the execution time of the first part of $task_1$ is its WCET (i.e. $WCET_{11} = 11$), we got the chronogram presented in Figure 7.4, page 124. In this case, the tasks meet their deadlines, and $task_2$ starts its execution, whereas the first job of $task_3$ is terminated on core 1. But we now ask for the whole execution time interval of $task_1$. Therefore, we apply our verification approach to check the schedulability of the application over the whole time interval $[8, 11]$. We 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 find the execution time interval of $task_1$.

Table 7.1: Three-tasks application set characteristics.

	A_i	D_i	T_i	C_i: [BCET,WCET]	Transitions
$task_1$	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}

Verification approach with observers Based on the application modeling approach presented in Figure 5.4, page 90, we construct the task models of Figure 7.3, page 123 with the timing values of Table 7.1, page 122. We add one observer per task, as shown in Figure 7.2, page 121. The schedulability analysis is conducted on the whole system containing the application and RTOS models with Roméo. We check that the place Obs_i are never marked by a token with the property: $AG(Obs_1 < 1 \; and \; Obs_2 < 1 \; and \; Obs_3 < 1)$. A counter-example execution trace is generated, proving this property is not satisfied, and $task_3$ may miss its deadline. Indeed, if the execution time of $task_1$ is its BCET (i.e.

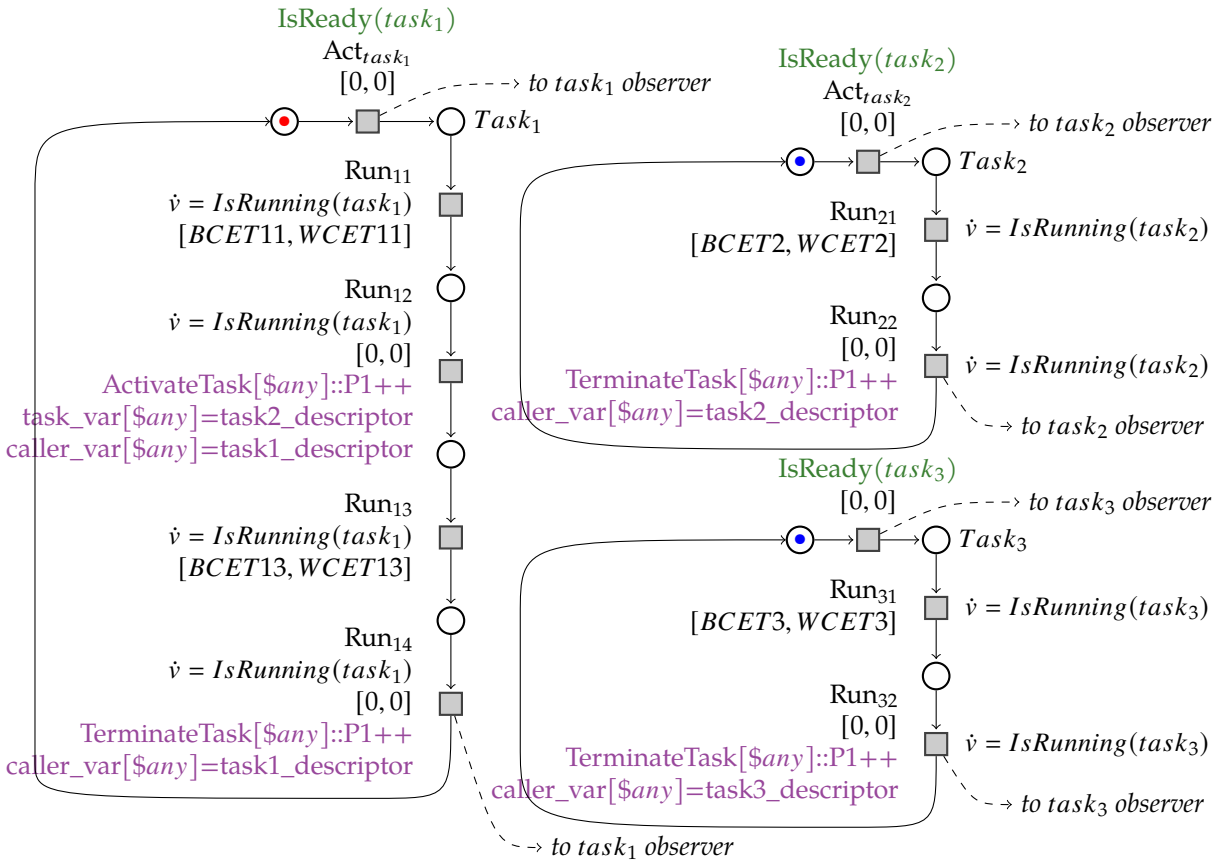


Figure 7.3: Three-tasks application model considering time intervals and observers.

$BCET_{11} = 8$), $task_3$, running on core 1 does not have the time to finish before its deadline as shown in Figure 7.5, page 124. 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 $task_1$.

Task parameters synthesis 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. Checking the property $AG(Obs_1 < 1 \; and \; Obs_2 < 1 \; and \; Obs_3 < 1)$ leads the parameter synthesis. The result of Roméo synthesis is $(10 < a \leq 11) \wedge (10 < b \leq 11) \wedge (a \leq b)$, thus the execution time of $task_1$ must be in the interval $]10, 11]$. The computing time and used memory for this analysis are shown in the table 7.2, page 125, and parametric model-

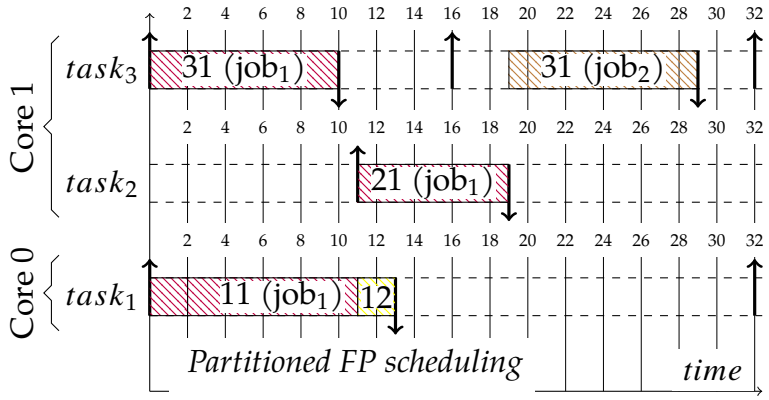


Figure 7.4: Schedule of tasks set with the $WCET_1$. The symbols ↑ and ↓ indicate activation and completion of tasks, respectively.

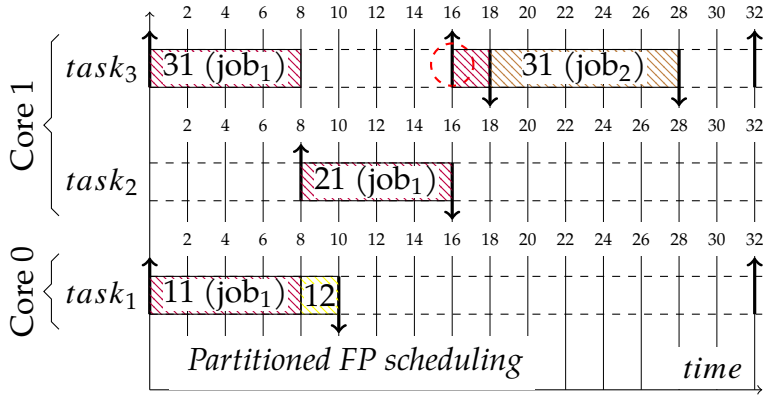


Figure 7.5: Schedule of tasks set with the $BCET_1$. The symbols ↑ and ↓ indicate activation and completion of tasks, respectively. Here job 1 of $task_3$ misses its deadline as indicated by the dashed red circle.

checking of HCTPNs with stopwatches consumes more time and memory than analysis using no parameters.

Table 7.2: Three-tasks application: 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

7.4 Ad-hoc scheduling system

The scheduler within Trampoline is kernel-based, which means there is no separation between the kernel and scheduler codes. Therefore, modifying its implementation requires expertise to identify the parts of the kernel that need to be replaced. Boukir et al. have already modified the scheduling policy of Trampoline in Global EDF and EDF-US[ξ] based on the RTOS model built by Tigori in Uppaal [111]. Significant modifications have been done, making the subject of a Ph.D. thesis [112]. However, the task execution time representation is considered discrete. The principal idea was in fact to verify the implementations of the schedulers.
Our main idea of this work is to show the possibility of efficiently verifying new scheduling policies within the multi-core RTOS, considering parallelism, concurrency, and preemption in compact time. To study the feasibility of this schedulability verification, we define an ad-hoc scheduler and modify the fixed priority partitioned scheduling of the Trampoline RTOS model. In principle, the scheduler should be called when an event occurs and requires rescheduling, for example, a task that activates another or terminates, as shown in Figure 3.12, page 54. This allowed us to determine the scheduler parts to change and perform a schedulability check on the system, considering the stopwatches. This section presents the specifications of the chosen ad-hoc scheduler, its implementation in the model, and the conducted verification using parameters synthesis.

7.4.1 Ad-hoc scheduler specifications

The scheduling policy consists of a four-tasks application: $task_1$, $task_2$, $task_3$, and $task_4$, statically assigned to two cores, as shown below:

- $task_1$ and $task_4$ are assigned to run on core 0;
- $task_2$ and $task_3$ are assigned to run on core 1;
- $Prio(task_1) < Prio(task_4)$;
- When $task_4$ runs on core 0, $Prio(task_2) < Prio(task_3)$ otherwise $Prio(task_3) < Prio(task_2)$.

The requirements of this ad-hoc policy lead to the rescheduling of $task_2$ and $task_3$ on core 1 once the activation or termination of $task_1$ and $task_4$ occur on core 0. When the scheduler is called for core 0, it is based on the job currently running on that core (i.e., $task_1$ or $task_4$) and calculates the priority of $task_2$ or $task_3$ jobs for core 1. It then decides whether a context switch is required on a core and have to be achieved. Thus, two events cause the rescheduling: (i) activation of new $task_1$ and $task_4$ jobs on core 0; (ii) termination of these tasks jobs running on core 0. That means, for example, $task_2$ continues to run on core 1 until $task_3$ preempts 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 only allow one possible activation of the job and use the Roméo model model-checker to analyze the system.

7.4.2 Ad-hoc scheduler implementation

The scheduler implementation within Trampoline relies on functions and data structures that are manipulated and involved in making decisions as presented in Section 3.5.1, page 46. Among these functions, we find the internal scheduler function tpl_start, called by the main scheduling function $tpl\_schedule\_from\_running$, that invokes the scheduler, whether in mono-core or multi-core. tpl_start allows updating the $elected\_id$ attribute of the tpl_kern structure with the identifier value of the job elected to run on the core. Since the two events that cause rescheduling are job activation and termination, our modification work focuses on the scheduler's internal function tpl_start called in both cases, at activation by the function $tpl\_schedule\_from\_running$ and termination by $tpl\_terminate\_service$, as shown in Figure 7.6, page 128. This figure also illustrates the other parts concerned with a change, presented in the following, to implement the ad-hoc scheduler in the model. Once the scheduler modified, the four-tasks applica-

tion is modeled based on the task model shown in Figure 7.2, page 121, linked with the observer.

tpl_start We extend the model of this function with the modification of tasks' dynamic priorities of core 1 according to the task currently elected on core 0. This means that after electing a task on core 0 and copying its information into the elected attributes of the tpl_kern structure, we rely on the identifier of this new elected task to compute the new dynamic priorities of $task_2$ and $task_3$ assigned to core 1. The dynamic priority concatenates the base priority and the activation order number as explained in the **scheduling** paragraph 3.5.2, page 51. The $ready\_list$ of core 1 is then updated with the new priorities, and the $need\_schedule$ flag is set to 1.

tpl_schedule_from_running This function invokes the scheduler only on one core. Since the scheduling decision on core 1 depends on the task running on core 0, we need to call the $tpl\_schedule\_from\_running$ function for core 1. We replace it with the $tpl\_multi\_schedule$ function, which loops its call over the cores. $tpl\_multi\_schedule$ performs several rescheduling when many tasks may be activated on multiple cores, indicated by the $need\_schedule$ flag of the core. Therefore, the rescheduling of core 1 from core 0 can be done, and if needed, core 0 sends an inter-core interrupt request to core 1 to trigger the context switch and execute the task.

tpl_terminate_task_service In the case of task termination on core 0, the rescheduling of core 1 from core 0 must be added explicitly by modifying the function $tpl\_terminate\_task\_service$. For that, $tpl\_schedule\_from\_running$ is called by core 0 to perform rescheduling for core 1, followed by a context switch to send the inter-core interrupt to core 1 and execute the elected task.

7.4.3 Task parameters synthesis

Let us consider this ad-hoc scheduling system with the characteristics of Table 7.3, page 128. The goal now is to synthesize the tasks execution time interval such that the tasks meet their deadlines. 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 7.3, page 128). We bound the parameters a and b in the interval $[3, 6]$. We use the non-

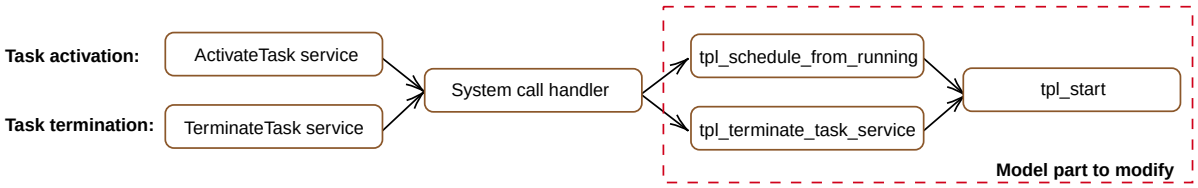


Figure 7.6: Model part to modify for the ad-hoc scheduler implementation. The symbol -> indicates a function call.

parameterized observer and we instantiate the D_i. Checking the property $AG(Obs_3 < 1 \; and \; Obs_4 < 1)$ with Roméo model-checker returns two resulats:

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

To test 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 7.7, page 129, 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 7.8, page 129. The tasks set is, therefore, schedulable.

Table 7.3: 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}

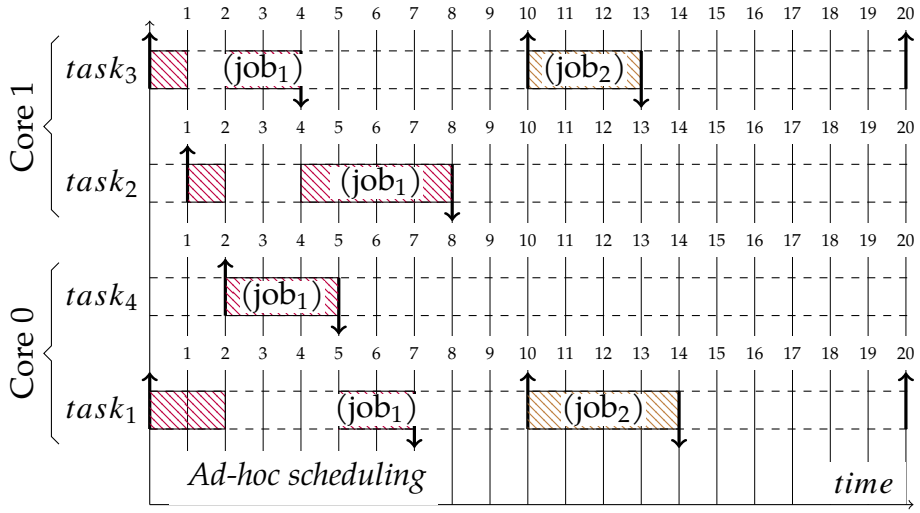


Figure 7.7: Schedule of tasks set with the $BCET_3$. The symbols ↑ and ↓ indicate activation and completion of tasks, respectively.

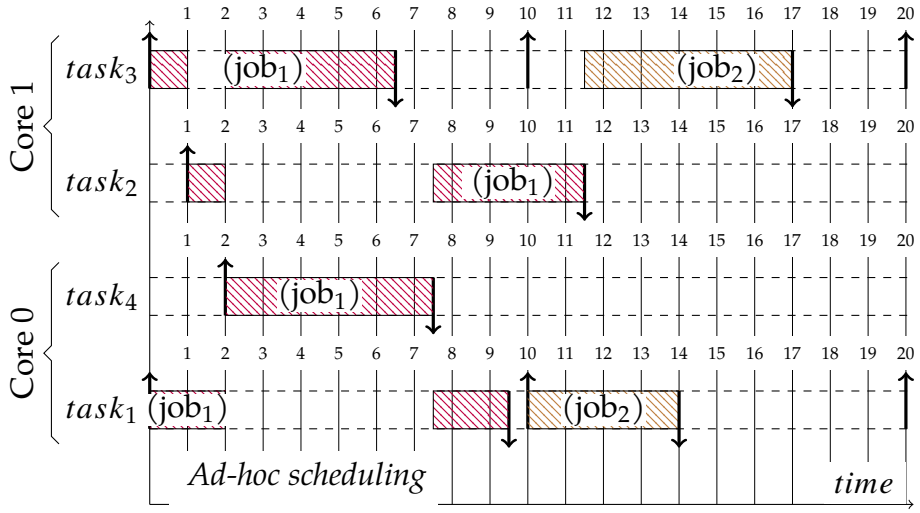


Figure 7.8: Schedule of tasks set with **Roméo**. The symbols ↑ and ↓ indicate activation and completion of tasks, respectively.

7.4.4 Response time analysis

The response time analysis allows calculating the response time, which represents the duration between the arrival time of the task's instance (job) and its completion. This analysis is performed using parameters in the same task-related observer, taking into account the dependency between the tasks. To automatically compute the response time of a $task_i$, 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 $task_i$.
Based on the same system characteristics of Table 7.3, page 128, we instantiate the task execution time with time intervals, and replace D_i with a parameter d_i in the task observers. Roméo synthesizes the values of the parameter d_i and analyzes the response time of $task_i$. Table 7.4, page 130, 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 7.3, page 128) 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 a counter-example represented by the chronogram in Figure 7.9, page 131. 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. $WCET_3 = 6$).

Table 7.4: 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

7.5 Conclusion

This chapter presented a method for efficiently analyzing real-time applications running on an RTOS with preemptive scheduling using the Roméo tool. Thanks to the capabilities of stopwatches, the application models include the possibility to block the runnable's elapsed time and to model the preemption. The complete model linked to

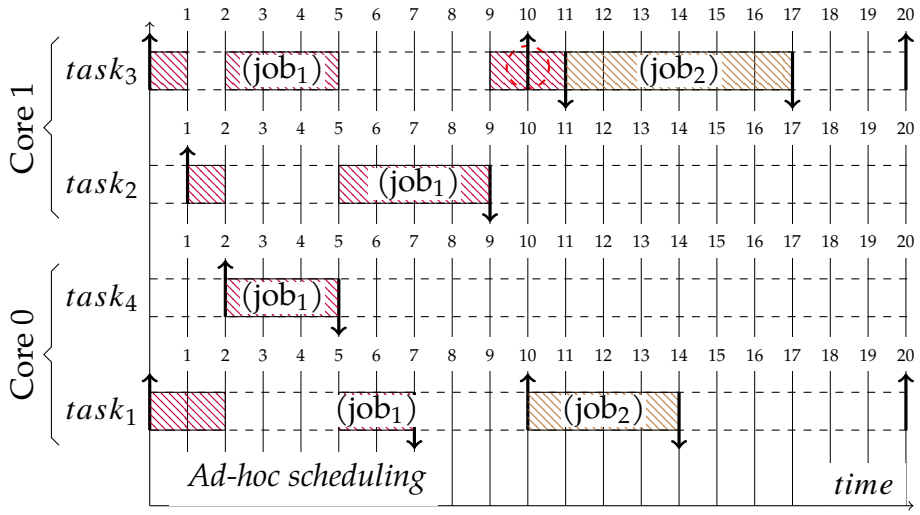


Figure 7.9: Schedule of tasks set with the $WCET_3$. The symbols ↑ and ↓ indicate activation and completion of tasks, respectively. Here job 1 of $task_3$ misses its deadline as indicated by the dashed red circle.

observers is used to check the schedulability of an application. The parameterized version calculates the execution times required to guarantee schedulability or response times.

We have also presented the possibility of changing the scheduling policy and efficiently verifying the temporal properties of dependent and preemptive tasks. We detailed the modifications required to support an ad-hoc scheduler and its verification with the same approach.

CONCLUSION AND PERSPECTIVES

Conclusion

Multi-core real-time embedded systems must respect several constraints, and their verification must consider real-time and concurrency aspects to guarantee an error-free system. Testing and simulation are methods to identify errors in software programs. However, they are not exhaustive and do not guarantee the elimination of all errors. Formal verification is a solution that uses a set of mathematical techniques to verify the correctness of the system behavior. To apply this method to multi-core Real-Time Operating Systems (RTOS), we focused on the model-checking approach, an automatic process that exhaustively checks if the model respects specific properties. Two questions are asked: The first one is how to ensure that the behavior of the multi-core RTOS conforms to its standard and requirements, taking into account concurrency and parallelism aspects. The second one concerns verifying the scheduling of a real-time application running on the RTOS, considering the preemptive aspect.

The thesis work proposed in this manuscript tries to answer these questions. First, we have defined the model of High-level Colored Time Petri nets (HCTPN) in which we can use temporal parameters on the transitions and consider parallelism and concurrency of cores. This formalism was used for modeling the multi-core RTOS, and the colors specify the hardware where the software is executed. Preemption is supported through stopwatches associated with timed transitions that can be activated by more than one color, thus allowing true concurrency modeling. Then, we proposed a formal verification approach where the RTOS model and the application are built with HCTPN based on a number of translation rules. The model-checking of the HCTPN formalism is implemented in the Roméo tool where we performed this approach. From the elaborated model, the verification of the properties has been performed using external observers modeled by an additional HCTPN associated in a non-intrusive way to the model without altering its behavior.

This approach has been applied to the Trampoline operating system, a multi-core RTOS compliant with OSEK/VDX and AUTOSAR standards used in automotive embedded

systems. It allowed us to verify with the model-checking of the extended formalism in Roméo:

- The multi-core RTOS conformity to the AUTOSAR standard;
- The inter-core synchronization mechanism involved in concurrent OS service execution;
- The schedulability of real-time applications with dependent preemptive tasks.

These obtained results showed the feasibility of our approach and the power of RTOS verification. It allowed the automatic identification of two possible OS errors in concurrent execution, considering simultaneous service calls on the cores. These errors illustrated that the rescheduling and the context switching in concurrent situations are not functional. Thus, the data protection is insufficient, and the synchronization of the cores is defective. The parametric analysis of the HCTPNs has also shown its effectiveness in synthesizing the parameter values that satisfy the property. It allowed calculating the necessary execution times to guarantee the scheduling or the response times.
However, some points remain to be discussed:

- The modeling work requires expertise and remains a heavy task that takes considerable time and effort;
- The combinatorial explosion problem can limit the verification of complex examples.

It, therefore, leads to perspectives on future works that we explore in what follows.

Perspectives

After the proposed multi-core RTOS verification approach with extended time Petri nets, one perspective of this work is to model the multi-core version that allows executing the kernel code on different cores in parallel and handling interrupts simultaneously. The current approach considers a global lock that prevents this parallel execution. Thanks to the formalism of HCTPN implemented in Roméo, modeling can be done using colors. The kernel can be executed simultaneously by the different cores, such as one color corresponding to one core. However, the combinatorial explosion problem must be examined as adding parallelism at the kernel level will considerably increase

the number of system states. One solution to this problem could be the abstraction of the OS model with a new model that preserves the behavior as much as possible and merges its states to reduce their number. This model abstraction must satisfy the same properties of the initial model we want to conserve. Thus, additional verification can be performed using a proof assistant, for example, and the approach would combine the two formal verification methods.

A second perspective is to implement a Domain Specific Language (DSL) to facilitate the modeling of the operating system and the construction of the model for several levels of abstraction. The DSL will automatically build a model of the operating system in HCTPN from its source code based on the rules proposed in Section 5.2, page 79. We can also apply these rules in reverse to generate a verified OS code by applying a reachability analysis on the model. The models built in Roméo are represented in XML files; therefore, an XML parser and a code translator are needed to elaborate the DSL.

Finally, a third perspective would be verifying several scheduling policies based on the RTOS model. A first approach would be to separate the scheduler code from the kernel by adding an intermediate model. Thus, instead of having the scheduler code distributed in the kernel, the scheduler model will be isolated while communicating with the system components. That will facilitate the modification of the scheduling policies for their verification, minimizing the changes within the kernel. The implementation of the DSL can also cover this perspective and describe the scheduler's implementation following a high-level-based approach. The DSL will allow the description of its behavior and the specification of the scheduler objects and parameters. Thus the language can generate its code to be integrated within the OS.

PUBLICATIONS

International conferences

- Imane Haur, Jean-Luc Béchennec, Olivier H. Roux, *Formal Schedulability Analysis Based on Multi-Core RTOS Model* in 29th International Conference on Real-Time Networks and Systems (RTNS 2021) (Association for Computing Machinery, Nantes, France, 2021), 216– 225.
- Imane Haur, Jean-Luc Béchennec, Olivier H. Roux, *High-level Colored Time Petri Nets for true concurrency modeling in real-time software* in International Conference on Control, Decision and Information Technologies (CODIT 2022) (IEEE, Istanbul, Turkey, May 2022). *Best Paper Award.*
- Imane Haur, Jean-Luc Béchennec, Olivier H. Roux, *Formal verification of the inter-core synchronization of a multi-core RTOS kernel* in 23th International Conference on Formal Engineering Methods (ICFEM 2022) 13478 (Springer, Madrid, Spain, Oct. 2022).

International journals

- Imane Imane Haur, Jean-Luc Béchennec, Olivier H. Roux. *Formal verification process of the compliance of a multicore AUTOSAR OS*. Software Quality Journal (Springer, 2022). *Under submission* (*Major revisions*).

BIBLIOGRAPHY

1. ARINC Group, *653P1-5 Avionics Application Software Standard Interface, Part 1, Required Services* 2019.
2. Group, O., *OSEK/VDX Operating System Specification* tech. rep. (2009).
3. AUTOSAR GbR, *Specification of operating system* 2009.
4. ISO, *ISO 26262:2018 Road vehicles — Functional safety* tech. rep. (2018).
5. Lamport, L., Proving the Correctness of Multiprocess Programs, *IEEE Transactions on Software Engineering* (1977).
6. OSEK Group, *OSEK/VDX OS Test Plan Version 2.0* 1999.
7. Clarke, E. M., Henzinger, T. A., Veith, H. & Bloem, R., *Handbook of Model Checking* (Springer Publishing Company, Incorporated, 2018).
8. Béchennec, J.-L., Briday, M., Faucou, S. & Trinquet, Y., *Trampoline an open source implementation of the OSEK/VDX RTOS specification in Emerging Technologies and Factory Automation, 2006. ETFA'06. IEEE Conference on* (2006).
9. Gadelha, M. R. *et al., ESBMC 5.0: An Industrial-Strength C Model Checker in Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (2018).
10. Tigori, K. T. G., Béchennec, J.-L., Faucou, S. & Roux, O. H., Formal Model-Based Synthesis of Application-Specific Static RTOS, *ACM Transactions on Embedded Computing Systems (TECS)* (2017).
11. Roux, O. H. & Lime, D., *Roméo: formal verification and synthesis for timed systems* http://romeo.rts-software.org.
12. Charette, R. N., *Automated to death in IEEE Spectrum* (2009).
13. Finkelstein, A. C. W., Report of the Inquiry into the London Ambulance Service (1993).
14. Cook, S. A., *The Complexity of Theorem-Proving Procedures in Proceedings of the Third Annual ACM Symposium on Theory of Computing* (1971).

15. Clarke, E. M., Filkorn, T. & Jha, S., *Exploiting Symmetry In Temporal Logic Model Checking in Proceedings of the 5th International Conference on Computer Aided Verification* (1993).

16. Gallier, J. H., *Logic for Computer Science: Foundations of Automatic Theorem Proving, Second Edition* (Dover Publications, Inc., 2015).

17. Rashid, A., Hasan, O., Siddique, U. & Tahar, S., Formal reasoning about systems biology using theorem proving, *PLOS ONE* (2017).

18. Kerber, M., Lange, C. & Rowat, C., An introduction to mechanized reasoning, *Journal of Mathematical Economics* (2016).

19. Kaliszyk, C., Chollet, F. & Szegedy, C., *HolStep: A Machine Learning Dataset for Higher-order Logic Theorem Proving* 2017.

20. Wang, H., *Computer Theorem Proving and Artificial Intelligence in Computation, Logic, Philosophy: A Collection of Essays* (1990).

21. Harrison, J., Urban, J. & Wiedijk, F., *History of Interactive Theorem Proving in Computational Logic* (2014).

22. Dowek, G. *et al., The Coq proof assistant user's guide : version 5.8* tech. rep. (1993).

23. Nipkow, T., Paulson, L. C. & Wenzel, M., *Isabelle/HOL: a proof assistant for higher-order logic* (2002).

24. Sutcliffe, G. & Suttner, C., Evaluating general purpose automated theorem proving systems, *Artificial Intelligence* (2001).

25. Mccune, W., Solution of the Robbins Problem, *Journal of Automated Reasoning* (1997).

26. Frege, G., Bynum, T. & Press, O. U., *Conceptual Notation, and Related Articles* (Clarendon Press, 1972).

27. Rozier, K. Y., Linear Temporal Logic Symbolic Model Checking, *Computer Science Review* (2011).

28. Alur, R., Courcoubetis, C. & Dill, D., Model-Checking in Dense Real-time, *Information and Computation* (1993).

29. Lamport, L., Proving the Correctness of Multiprocess Programs, *IEEE Transactions on Software Engineering* (1977).

30. Godefroid, P., *Using Partial Orders to Improve Automatic Verification Methods in Proceedings of the 2nd International Workshop on Computer Aided Verification* (1990).

31. Dhaussy, P., Roger, J.-C. & Boniol, F., *Reducing State Explosion with Context Modeling for Model-Checking in IEEE 13th International Symposium on High-Assurance Systems Engineering* (2011).

32. Alur, R. & Dill, D. L., A theory of timed automata, *Theoretical Computer Science* (1994).

33. Merlin, P. M., *A Study of the Recoverability of Computing Systems.* Theses (1974).

34. Henzinger, T., Nicollin, X., Sifakis, J. & Yovine, S., Symbolic Model Checking for Real-Time Systems, *Information and Computation* (1994).

35. Larsen, K. G., Pettersson, P. & Yi, W., Uppaal in a Nutshell, *Int. J. Softw. Tools Technol. Transf.* (1997).

36. Yovine, S., Kronos: A Verification Tool for Real-Time Systems. (Kronos User's Manual Release 2.2), *International Journal on Software Tools for Technology Transfer* (2001).

37. Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P. & Yi, W., *TIMES b— A Tool for Modelling and Implementation of Embedded Systems in Tools and Algorithms for the Construction and Analysis of Systems* (2002).

38. Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P. & Yi, W., *UPPAAL — a tool suite for automatic verification of real-time systems in Hybrid Systems III* (1996).

39. Merlin, P. & Farber, D., Recoverability of Communication Protocols - Implications of a Theoretical Study, *IEEE Transactions on Communications* (1976).

40. Ramchandani, C., *ANALYSIS OF ASYNCHRONOUS CONCURRENT SYSTEMS BY TIMED PETRI NETS* tech. rep. (USA, 1974).

41. Berthomieu, B. & Diaz, M., Modeling and Verification of Time Dependent Systems Using Time Petri Nets, *IEEE Trans. on Soft. Eng.* (1991).

42. Calvez, S., Aygalinc, P. & Khansa, W., P-Time Petri Nets for Manufacturing Systems with Staying Time Constraints, *IFAC Proceedings Volumes* (1997).

43. Bonhomme, P., *A symbolic schedulability technique of real-time systems modeled by P-Time Petri nets in IEEE International Conference on Automation Science and Engineering* (2011).

44. Rakkay, H., Boucheneb, H. & Roux, O. H., *Time Arc Petri Nets and Their Analysis in Ninth International Conference on Application of Concurrency to System Design* (2009).

45. Jones, N. D., Landweber, L. H. & Edmund Lien, Y., Complexity of some problems in Petri nets, *Theoretical Computer Science* (1977).

46. Liu, B. & Robbi, A., *TiPNet: a graphical tool for timed Petri nets in Proceedings 6th International Workshop on Petri Nets and Performance Models* (1995).

47. Vernadat, F., Berthomieu, B., Vernadat, F. & Berthomieu, B., *Time Petri Nets Analysis with TINA in Third International Conference on the Quantitative Evaluation of Systems - (QEST'06)* (2006).

48. Lime, D., Roux, O. H., Seidner, C. & Traonouez, L.-M., *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)* (2009).

49. Cassez, F. & Larsen, K., *The Impressive Power of Stopwatches in CONCUR 2000 — Concurrency Theory* (2000).

50. Roux, O. & Déplanche, A.-M., A T-time Petri net extension for real time-task scheduling modeling, *European Journal of Automation* (2002).

51. Lime, D. & Roux, O. H., Expressiveness and analysis of scheduling extended time Petri nets, *IFAC Proceedings Volumes* (2003).

52. Bucci, G., Fedeli, A., Sassoli, L. & Vicario, E., Time state space analysis of real-time preemptive systems, *IEEE transactions on software engineering* (2004).

53. Roux, O. H. & Lime, D., *Time Petri Nets with Inhibitor Hyperarcs. Formal Semantics and State Space Computation in Applications and Theory of Petri Nets 2004* (2004).

54. Behrmann, G., Larsen, K. & Rasmussen, J., Optimal scheduling using priced timed automata, *SIGMETRICS Performance Evaluation Review* (2005).

55. Zaharia, T. & Haller, P., *Formal verification and implementation of real time operating system based applications in 2008 4th International Conference on Intelligent Computer Communication and Processing* (2008).

56. Waszniowski, L. & Hanzálek, Z., Formal Verification of Multitasking Applications Based on Timed Automata Model, *Real-Time Syst.* (2008).

57. Pimkote, A. & Vatanawood, W., *Simulation of Preemptive Scheduling of the Independent Tasks Using Timed Automata in 2021 10th International Conference on Software and Computer Applications* (2021).

58. Grolleau, E. & Choquet-Geniet, A., Scheduling Real-Time Systems by Means of Petri Nets, *IFAC Proceedings Volumes* (2000).

59. Xu, D., He, X. & Deng, Y., Compositional schedulability analysis of real-time systems using time Petri nets, *IEEE Transactions on Software Engineering* (2002).

60. Lime, D. & Roux, O., Formal Verification of Real-time Systems with Preemptive Scheduling, *Real-Time Systems* (2009).

61. Jensen, K., Coloured petri nets and the invariant-method, *Theoretical Computer Science* (1981).

62. Hohmuth, M. & Tews, H., *The VFiasco approach for a verified operating system in 2nd PLOS* (2005).

63. Klein, G. *et al., SeL4: Formal Verification of an OS Kernel in Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (2009).

64. Espinosa, T. & Leon, G., Formal verification of a real-time operating system (2012).

65. Xu, F. *et al., A Practical Verification Framework for Preemptive OS Kernels in Computer Aided Verification* (2016).

66. Gu, R. *et al.,* Building Certified Concurrent OS Kernels, *ACM* (2019).

67. Xu, L. *et al., Towards Fault-Tolerant Real-Time Scheduling in the seL4 Microkernel in IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems* (*HPCC*/*SmartCity*/*DSS*) (2016).

68. Huang, Y. *et al., Modeling and Verifying the Code-Level OSEK/VDX Operating System with CSP in 2011 Fifth International Conference on Theoretical Aspects of Software Engineering* (2011).

69. Chen, J. & Aoki, T., *Conformance testing for OSEK/VDX operating system using model checking in Proceedings - Asia-Pacific Software Engineering Conference, APSEC* (2011).

70. Shi, J. *et al., ORIENTAIS: Formal Verified OSEK/VDX Real-Time Operating System in IEEE 17th International Conference on Engineering of Complex Computer Systems* (2012).

71. Béchennec, J., Roux, O. H. & Tigori, T., *Formal model-based conformance verification of an OSEK/VDX compliant RTOS in 2018 5th International Conference on Control, Decision and Information Technologies* (*CoDIT*) (2018).

72. Fang, L., Kitamura, T., Do, T. B. N. & Ohsaki, H., *Formal Model-Based Test for AUTOSAR Multicore RTOS in 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation* (2012).

73. Peng, Y., Huang, Y., Su, T. & Guo, J., *Modeling and Verification of AUTOSAR OS and EMS Application in International Symposium on Theoretical Aspects of Software Engineering* (2013).

74. Trinh, L. K., Chiba, Y. & Aoki, T., *Formalization and Verification of AUTOSAR OS Standard's Memory Protection in International Symposium on Theoretical Aspects of Software Engineering* (*TASE*) (2018).

75. Yan, R. & Guo, J., *Timing Modeling and Analysis for AUTOSAR Schedule Tables in IEEE 19th International Symposium on High Assurance Systems Engineering* (*HASE*) (2019).

76. Choi, Y., *Safety Analysis of Trampoline OS Using Model Checking: An Experience Report in 2011 IEEE 22nd International Symposium on Software Reliability Engineering* (2011).

77. Choi, Y., Model Checking an OSEK/VDX-Based Operating System for Automobile Safety Analysis, *IEICE Transactions on Information and Systems* (2013).

78. Boukir, K., Béchennec, J.-L. & Déplanche, A.-M., *Requirement Specification and Model-Checking of a Real-Time Scheduler Implementation in Proceedings of the 28th International Conference on Real-Time Networks and Systems* (2020).

79. Sha, L., Rajkumar, R. & Lehoczky, J. P., Priority inheritance protocols: an approach to real-time synchronization, *IEEE Transactions on Computers* (1990).

80. Hillah, L., Kordon, F., Petrucci, L. & Trèves, N., *PN Standardisation: A Survey in Formal Techniques for Networked and Distributed Systems - FORTE 2006* (2006).

81. Boyer, M. & Diaz, M., *Multiple Enabledness of Transitions in Petri Nets with Time in Proceedings of the 9th International Workshop on Petri Nets and Performance Models, PNPM 2001, Aachen, Germany, September 11-14, 2001* (2001).

82. Berthomieu, B. & Menasche, M., *An Enumerative Approach for Analyzing Time Petri Nets in Information Processing: proceedings of the IFIP congress 1983* (1983).

83. Pnueli, A., *The Temporal Logic of Programs in 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA* (1977).

84. Boucheneb, H., Gardey, G. & Roux, O. H., TCTL model checking of Time Petri Nets, *Journal of Logic and Computation* (2009).

85. Lime, D., Roux, O. H., Seidner, C. & Traonouez, L.-M., *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)* (2009).

86. Gardey, G., Lime, D., Magnin, M. & Roux, O. H., *Romeo: A Tool for Analyzing Time Petri Nets in Computer Aided Verification* (2005).

87. Andreychenko, A., Magnin, M. & Inoue, K., Analyzing resilience properties in oscillatory biological systems using parametric model checking, *Biosystems* (2016).

88. Parquier, B. *et al., Applying Parametric Model-Checking Techniques for Reusing Real-Time Critical Systems in Formal Techniques for Safety-Critical Systems* (2017).

89. Coullon, H., Jard, C. & Lime, D., *Integrated Model-Checking for the Design of Safe and Efficient Distributed Software Commissioning in Integrated Formal Methods* (2019).

90. Sun, Y., Lipari, G. & André, É., *Verification of Two Real-Time Systems Using Parametric Timed Automata in WATERS - International Workshop on Analysis Tools and Methodologies for Embedded Real-Time Systems* (2015).

91. André, É., *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* (2018).

92. Gardey, G., Roux, O. & Roux, O., State Space Computation and Analysis of Time Petri Nets, *TPLP* (2006).

93. Lime, D. & Roux, O. H., *State class Timed Automaton of a Time Petri Net in The 10th International Workshop on Petri Nets and Performance Models, (PNPM'03)* (2003).

94. Freescale Semiconductor, *MPC5643L Microcontroller Reference Manual* NXP (2013).

95. Liu, C. L. & Layland, J. W., Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment, *J. ACM* (1973).

96. Chéramy, M., Hladik, P.-E. & Déplanche, A.-M., Algorithmes pour l'ordonnancement temps réel multiprocesseur, *Journal Européen des Systèmes Automatisés (JESA)* (2015).

97. Atlas, A. & Bestavros, A., *Statistical rate monotonic scheduling in Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)* (1998).

98. Palencia, J. & Gonzalez Harbour, M., *Schedulability analysis for tasks with static and dynamic offsets in Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)* (1998).

99. Marti, P., Villa, R., Fuertes, J. & Fohle, G., *On real-time control tasks schedulability in 2001 European Control Conference (ECC)* (2001).

100. Devillers, R. & Goossens, J., Liu and Layland's schedulability test revisited, *Information Processing Letters* (2000).

101. Davis, R., Burns, A., Bril, R. & Lukkien, J., Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised, *Real-Time Systems* (2007).

102. Bletsas, K., Audsley, N. C., Huang, W.-H., Chen, J.-J. & Nelissen, G., Errata for Three Papers (2004-05) on Fixed-Priority Scheduling with Self-Suspensions, *Leibniz Transactions on Embedded Systems* (2018).

103. Bertogna, M., Cirinei, M. & Lipari, G., Schedulability Analysis of Global Scheduling Algorithms on Multiprocessor Platforms, *IEEE Transactions on Parallel and Distributed Systems* (2009).

104. Lee, J., Shin, K. G., Shin, I. & Easwaran, A., Composition of Schedulability Analyses for Real-Time Multiprocessor Systems, *IEEE Transactions on Computers* (2015).

105. Altmeyer, S. *et al., A Generic and Compositional Framework for Multicore Response Time Analysis in Proceedings of the 23rd International Conference on Real Time and Networks Systems* (2015).

106. Sun, Y. & Di Natale, M., *Assessing the Pessimism of Current Multicore Global Fixed-Priority Schedulability Analysis* tech. rep. (2017).

107. Sun, Y. & Di Natale, M., Pessimism in multicore global schedulability analysis, *Journal of Systems Architecture* (2019).

108. Dal-Zilio, S. & Berthomieu, B., *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* (2015).

109. Bagnara, R., Hill, P. M. & Zaffanella, E., The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems, *Sci. Comput. Program.* (2008).

110. Traonouez, L.-M., Lime, D. & Roux, O., *Parametric Model-Checking of Time Petri Nets with Stopwatches Using the State-Class Graph in 6th International Conference on Formal Modelling and Analysis of Timed Systems* (*FORMATS 2008* (2008).

111. Tigori, T. K. G., *Méthode de génération d'exécutifs temps-réel* Theses (2016).

112. Boukir, K., *Mise en oeuvre de politiques d'ordonnancement temps réel multiprocesseur prouvée* Theses (2020).

Appendix A

Requirements corresponding to multi-core OS

Table A.1: Subset of AUTOSAR OS requirements related to multi-core.

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 mechanisme
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

CENTRALE NANTES

Titre : Modélisation et vérification formelles d'un RTOS multicœur conforme à AUTOSAR

Mot clés : Vérification formelle, Model-checking, Réseaux de Petri colorés de haut niveau, Systèmes d'exploitation temps réel, Exécution multi-cœurs, Vérification d'OS AUTOSAR.

Résumé : La vérification formelle est une solution pour augmenter la fiabilité de l'implémentation du système. Dans notre travail de thèse, nous nous intéressons à l'utilisation de ces méthodes pour la vérification des systèmes d'exploitation multi-cœurs temps réel. Nous proposons une approche de model-checking utilisant les réseaux de Petri temporels, étendus avec des transitions colorées et des fonctionnalités de haut niveau. Nous utilisons ce formalisme pour modéliser le système d'exploitation multi-cœur Trampoline, conforme aux standards OSEK/VDX et AUTOSAR. Nous définissons dans un premier temps ce formalisme et montrons son adéquation avec la modélisation de systèmes concurrents temps reel. Nous utilisons ensuite ce formalisme pour modéliser le système d'exploitation multi-cœur Trampoline et vérifions par model-checking sa conformité avec le standard AUTOSAR. À partir de ce modèle, nous pouvons vérifier des propriétés aussi bien sur l'OS que sur l'application telles que l'ordonnançabilité d'un système temps-réel ainsi que les mécanismes de synchronisation : accès concurrents aux structures de données du système d'exploitation, ordonnancement multi-cœur et traitement des interruptions inter-cœur. À titre d'illustration, cette méthode a permis l'identification automatique de deux erreurs possibles de l'OS Trampoline dans l'exécution concurrente, montrant une protection insuffisante des données et une synchronisation défectueuse.

Title: AUTOSAR compliant multi-core RTOS formal modeling and verification

Keywords: Formal verification, Model-checking, High-level Colored Time Petri Nets, Real-Time Operating Systems (RTOS), Multi-core execution, AUTOSAR OS verification.

Abstract: Formal verification is a solution to increase the system's implementation reliability. In our thesis work, we are interested in using these methods to verify multi-core RTOS. We propose a model-checking approach using time Petri nets extended with colored transitions and high-level features. We use this formalism to model the Trampoline multi-core OS, compliant with the OSEK/VDX and AUTOSAR standards. We first define this formalism and show its suitability for modeling real-time concurrent systems. We then use this formalism to model the Trampoline multi-core RTOS and verify by model-checking its conformity with the AUTOSAR standard. From this model, we can verify properties of both the OS and the application, such as the schedulability of a real-time system and the synchronization mechanisms: concurrent access to the data structures of the OS, multi-core scheduling, and inter-core interrupt handling. As an illustration, this method allowed the automatic identification of two possible errors of the Trampoline OS in concurrent execution, showing insufficient data protection and faulty synchronization.