Testing real-time systems with runtime enforcement

Jean-Luc Béchennec, Matthias Brun, Sébastien Faucou, Louis-Marie Givel, Olivier H. Roux

Testing real-time systems with runtime enforcement

Jean-Luc Béchennec*, Matthias Brun†, Sébastien Faucou*, Louis-Marie Givel*† and Olivier H. Roux*
* CNRS, École Centrale de Nantes, Université de Nantes
Laboratoire des Sciences du Numérique de Nantes
F-4400 Nantes, FR
† ESEO-TRAME, Groupe ESEO, Angers, FR

***Abstract*—When testing a time-critical system, some scenarios can be hard to run when acting only on the input sequence. The proper execution of a given scenario might require for instance a minimal execution time for a given piece of software. Execution times are notoriously difficult to control because they depend not only on the inputs, but also on the state of the micro-architecture. In this paper, we propose a method, based on runtime enforcement, which forces a system to run such a scenario. We also describe an implementation in the context of a RTOS for embedded control systems.**

Our method starts with a parametric formal model of the system where the parameters are delays that can be added to simulate longer execution times. The domain of acceptable parameter values to run the target scenario is computed offline. Online, a framework plugged in the RTOS observes the execution of the system and injects delays when needed.

***Index terms*—real-time systems, formal methods, runtime enforcement, real-time testing, model-based testing, RTOS, parametric timed model.**

I. Introduction

The Design and verification of modern SoCs requires a system-level approach encompassing both hardware and software [1]. Their increasing complexity makes it difficult to achieve a high level of test coverage with a black box approach based only on the control of the input sequence. It is sometimes mandatory to act on the internal behaviour as well. This is for instance required to test safety mechanisms used to handle timing faults, as found in mixed criticality real-time systems [2]. In the domain of formal methods, this problem is known as *runtime enforcement*. An enforcer is a (hardware or software) component that forces a running system to conform to a specified behaviour or policy. It is automatically generated from the specifications.

In this paper, we investigate the case for runtime enforcement in the test of time-critical systems. We focus on embedded control systems hosting a complex software layer typically built on top of a multitask RTOS, as found in automotive or avionics. We describe the design and implementation of an enforcer inserted at the interface between the application and the RTOS to inject delays in order to mimic longer execution times or to delay the notification of hardware events. We also briefly outline the model-based technique used offline to compute the delays in order to run a given test scenario.

A. Contribution

The problem solved in this paper is the following: given a multitask embedded control system and a test scenario that assumes a specific internal timed behaviour, provide a component to enforce this assumption at runtime. Starting with a model describing the real-time behavior of the system and a set of states that we want to reach, we briefly outline how to compute the set of lower bounds on timing parameters to enforce the reachability. Then, at runtime, delays are injected to enforce these bounds. We have introduced the overall approach in [3]. In this paper, we describe an implementation and we evaluate its performance for a system built with a Cortex-M4 based microcontroller running Trampoline RTOS [11]. The implementation is seamlessly integrated in the system, at the boundary between the application and the RTOS.

B. Related Works

A number of works deal with the test and the runtime enforcement of time-critical hardware-software systems. To the best of our knowledge, all these works assume a closed system: they only act upon its inputs (and outputs in the case of runtime enforcement) to produce the expected behaviour. In this paper, we give ourselves the possibility to act upon the internal behaviour of the system as well. Our goal is to increase the testability of safety functions used to handle rare or faulty timing behaviours. As a counterpart, our approach produces artificial runs of the system in the sense that these runs integrate artifical delays.

In [4], Briand *et al.* describe a stress testing strategy for real-time systems based on the seeding time of aperiodic tasks. The purpose of our offline phase is similar. In addition to the seeding time of aperiodic or sporadic tasks, our offline phase also computes constraints on the execution time of the tasks to enforce the execution of the scenario. Moreover, we use formal methods, while [4] use a genetic algorithm.

The work reported in [5] on the runtime enforcement of timed properties is also close from our offline phase. Our offline technique assumes a weaker enforcer that can only introduce delays but not shorten interval between events nor suppress events. However, their technique is based on timed automata, a formalism that does not allow to model general preemptive scheduling policy. In our application domain, we have to use a more expressive formalism to handle preemption.

In [6], Mueller and Wegener describe a method to compute input data to trigger a specific execution time for a task. This method could be used as a replacement of the online phase of our method with the benefit of running a concrete execution of the system rather than an enforced execution. Compared

to this approach, our online phase is easier to use because it requires no further computation. It also provides the possibility to enforce faulty behaviours, which is important for testing safety functions.

Lastly, as noted in [7], *"the research area lack implementation in practice"*. Their study covers the broad area of testability and software performance, that includes issues related to timeliness and response time. As explained above, the main contribution of this paper is the description and evaluation of an actual implementation.

C. Outline of the paper

In Section II we briefly outline the offline phase (modeling and analysis). We present in Section III the online phase (runtime enforcement) and its implementation in Trampoline RTOS. We showcase the whole approach on a simple case study in Section IV. We conclude in Section V.

II. Offline analysis

A. System modeling with piTPN

The goal of the offline step is to compute a runtime enforcer to drive the system toward a set of target states by adding delays. The corresponding workflow is shown in figure 1.

The computation is based on tools for the formal analysis of time Petri nets (TPN). More specifically, it is based on the analysis of parametric TPN with inhibitor arcs (piTPN). Inhibitor arcs [8] add to TPN the capacity to suspend and resume clocks. They can be used to model fixed priority preemptive scheduling policy typically used in embedded control systems. Parameter synthesis [9] can be used to compute delays that should be enforced at runtime. The tool ROMEO [10] supports both the modeling and the analysis of piTPN.

As an illustration, consider the model of figure 2. It is composed of two tasks. $Task_1$ is activated with an offset of 3 t.u. (time units) with regards to the start time of the system. Its execution time is within 6 and 9 t.u.. $Task_2$ is activated when the system starts. Its execution time is within 5 and 9 t.u.. The arc between the place $Ready_1$ and the transition Run_2 is an inhibitor arc: the transition Run_2 is frozen (the corresponding clock is paused) while the place $Ready_1$ has at least one token. In other words, in this model, the priority of $Task_1$ is greater than the priority of $Task_2$. Parameters a and b are introduced to model lower bounds on the execution times of the tasks. Their values are implicitly positive and constrained by the width of the execution time interval: $0 \leq a \leq 3 \wedge 0 \leq b \leq 4$.

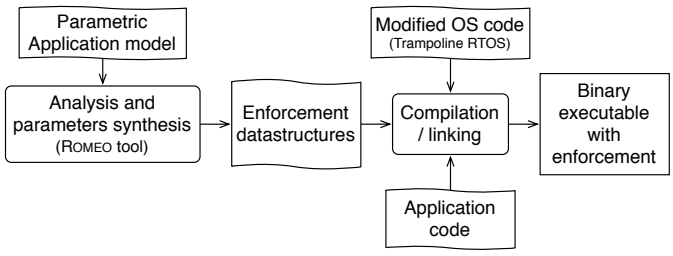

Figure 1. Workflow of the offline analysis.

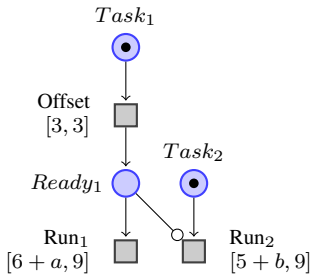

Figure 2. piTPN model of a system of two tasks using a fixed priority preemptive scheduling policy

Parameter synthesis can be used to compute the values of a and b such that a given state is reachable, for instance $Task_2$ terminates at least 12 t.u. after the start time. The result is $(a + b) \geq 1 \wedge 0 \leq a \leq 3 \wedge 0 \leq b \leq 4$.

B. Modeling a system for runtime enforcement

The single action of the enforcer on the behaviour of a task is to simulate a longer execution time by adding delays. From the point of view of the enforcer, the execution time of a task is observed as the duration between two subsequent system calls. Indeed, the behaviour between two system calls is internal to the task and difficult to act upon without invasive instrumentation. The model built for the offline analysis reflects this point of view.

In this model, the behaviour of each task is an alternation between system calls and internal actions. A system call $SysCall$ is modeled by a transition $SysCall$ of duration $[0, 0]$. An internal action is modeled by an ϵ-transition of duration [BCET,WCET] where BCET and WCET are respectively the best and the worst case execution time for this action. A branching is modeled as a non deterministic alternative (a fork) if at least one branch includes a system call. Otherwise it is abstracted by an ϵ-transition of duration [BCET,WCET] encompassing both branches.

Inhibitor arcs are used to represent the priority between the tasks. Given a pair of tasks t_1 and t_2 such that t_1 has a highest priority, inhibitor arcs are added between each place of t_1 and each transition of t_2.

At last, parameters are introduced in the time interval associated with transitions that can be delayed by the enforcer: the execution time of internal actions and the notification of hardware events to the application by the RTOS.

The domain of the parameters such that the target is reached for all the executions is computed by ROMEO.

III. Runtime enforcement

A. Principles

At runtime, the enforcer tracks the behavior of each task and insert delays if the observed behaviour does not conform to the required behaviour. To do so, data structures are generated from the offline model to describe the behaviour of each task independently from the others. The time used by the enforcer to update the data structure associated with a task is local to

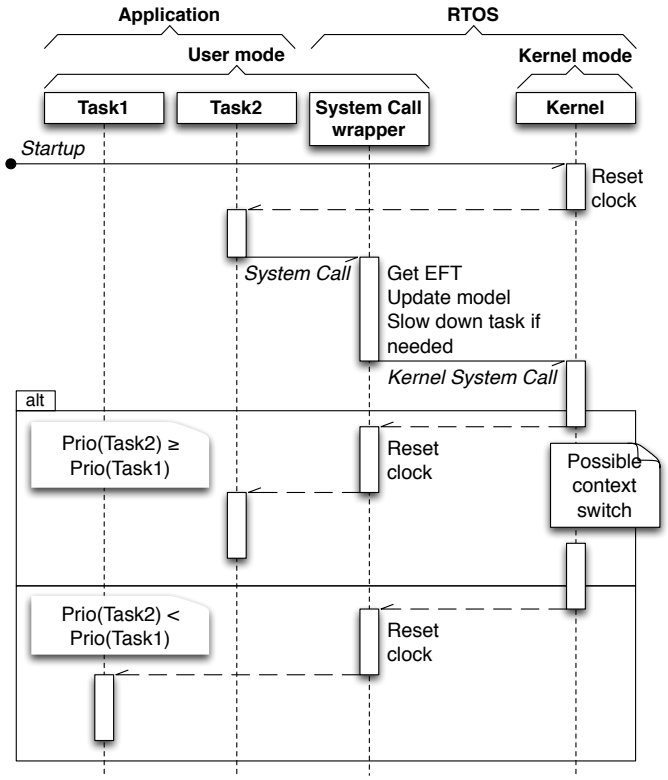

Figure 3. Interaction diagram of the control solution. When a task emits a System Call, the enforcer may delay this call while in user mode.

this task: it runs when the task is running and stops when the task is put back in the ready list because of a preemption or enters a waiting state because of a synchronization. This is consistent with the system-level analysis because preemption is modeled by the inhibitor arcs that are removed to obtain the local models [3].

Calls to the enforcer are inserted in system call wrappers[1], before the actual system call. The delays are added while the processor is still in user-mode to simulate a longer execution time of the tasks. It means that the task can be preempted while being delayed by the enforcer. As explained below, slight modifications of the kernel code are also required.

Thanks to the integration of the enforcer in the system call wrappers, it is possible to test systems without modifying the code of the tasks. In the context of a static RTOS where the software is composed of a single binary obtained by linking the object code of the tasks and the object code of the RTOS, it is even possible to test a task for which the source code is not provided. Trampoline, the RTOS used for this implementation, is static.

The detailed behaviour of the enforcer is described in figure 3. At system startup, the data structures are initialized. Then, the enforcer is triggered whenever a task emits a system call, *ie.* at the end of an internal action of the task. Most of the work is done in the wrapper before the kernel system call.

1) Identification of the transition: given the current state of the calling task and the information allowing to identify the call site, the transition in the local model is identified. In the current implementation, the identification of the call site can be based on the identity of the wrapper, which is sufficient if there is no state with two outgoing transitions labelled with the same system call, or based on the identity of the wrapper and the value of the arguments. Once the transition is identified, its Earliest Firing Time (EFT) is known. The EFT is computed offline as the sum between the BCET of the transition and the value picked for the parameter associated with this transition.
2) Slow task: the enforcer enters a busy loop as long as the execution time of the internal action is lower than the EFT.
3) Update model: the current state of the task is updated to the target state of the identified transition.
4) Kernel system call: the system call is passed to the kernel. If a task is activated during this call, its current state is set to its initial state and the execution time spent in this state is set to 0.
5) Timer reset: the timer used to measure the execution time spent in the new state of the elected task is reset. Please notice that during the previous step, a scheduling decision can trigger a context switch. Thus, the task elected at step 5 can be a different task than the one running at steps 1, 2, 3.

Handling asynchronous preemption: Besides system call, a task can be preempted asynchronously by the occurrence of a hardware event. In this case, the current value of the timer used to account for the execution time of this task is saved. Symmetrically, when a task is elected for running, this value is restored before to dispatch the task. Implementation of these steps requires to slightly modify the code of the kernel.

B. Implementation

The framework has been implemented on Trampoline RTOS, for the STM32F4Discovery board.

Trampoline RTOS: Trampoline [11] is an open-source implementation of the AUTOSAR OS standard. AUTOSAR OS defines the API of a static real-time operating system for automotive embedded systems. All objects are created and initialized at compile time. This configuration is described either in ArXML, a dedicated XML dialect, or in OIL, a plain text format. The kernel provides basic services for real-time control applications: static priority preemptive scheduling, inter-task communication and synchronization, real-time resource sharing protocol, time management.

STM32F4Discovery: this board developed by STMicroelectronics is built around a STM32F407VGT6 micro-controller with a 32 bit ARM Cortex-M4 core running at 168MHz. It hosts 1Mbyte of Flash Memory and 192Kbyte of (S)RAM on-board.

Configuration: the OIL compiler provided with Trampoline has been extended to support the description of the embedded model for each task. The model is given as a set of states with their outgoing transitions. Each transition is composed of a trigger (the system call and optionnaly its arguments), its EFT, and its target state.

According to this description, the OIL compiler generates the code to declare and initialize the following data structures.

[1]The user mode function that embeds the code performing the actual system call according to the platform application binary interface.

- An array of the current state of each task.
- An array of the execution time already spent by a task in its current state in case of asynchronous preemption.
- An array of pointers to the transition table of each task. If a task is not enforced, the pointer is null.
- For each task, a transition table given as an array of pointers to transition sets. It is indexed by task states.
- For each state of each task, a set of outgoing transitions.

Constant data are stored in flash to minimize the RAM overhead and variable data not needed when the enforcement code executes in user mode are stored in kernel space RAM to benefit from memory protection if available.

Runtime mechanism: The enforcer uses a 32 bit timer (TIM5). This timer is configured with a 1 microseconds granularity. At any time, the value of this timer reflects the execution time spent by the running task in its current state.

In Trampoline, the wrappers of the system calls are automatically generated from templates during the configuration of the system. The templates have been extended to insert calls to the enforcer before and after the actual kernel system call.

- Before: the enforcer searches the data structure to identify the transition and extract the EFT; then it loops while the value of the timer is lower than the EFT; at last, it changes the current state of the task to the target state of the transition.
- After: the enforcer resets the timer.

The scheduling service of the kernel has also been modified to account for task activation (reset of the timer) and asynchronous preemption (load/save the value of the timer).

Performances: Since the enforcement mechanism is inserted in the system call wrapper, it slows down each system call even if no delay is added. Table I gives the overhead of the enforcement mechanism and compare it to the execution time of some services of Trampoline. The application and Trampoline are compiled in EXTENDED configuration with GCC 5.4.1 and `-O3` optimization level. The overhead is between 11% (task is not enforced) and 50% (task is enforced). It is possible to take this overhead into account during the offline analysis by updating the implicit constraints on the parameter as follow: $\alpha \leq p \leq (WCET - BCET) - \alpha$, where p is a parameter and α is the overhead.

Service	Case	Cycles
ActivateTask	lower or equal priority task	466
ActivateTask	higher priority task (preemption)	844
SetEvent	non waiting task	346
SetEvent	waiting task of $\leq$ priority	546
SetEvent	waiting task of $>$ priority (preempt.)	886
Overhead when the task is not enforced		104
Overhead when the task is enforced		229

Table I
OVERHEAD OF THE ENFORCEMENT MECHANISM IN CPU CYCLES. The target is as described in III-B and has no cache nor branch prediction. A GPIO output is switched before and after the service call. Each service execution time has been executed and measured ten times with an oscilloscope and the variability of the execution time is lower than the measurement precision.

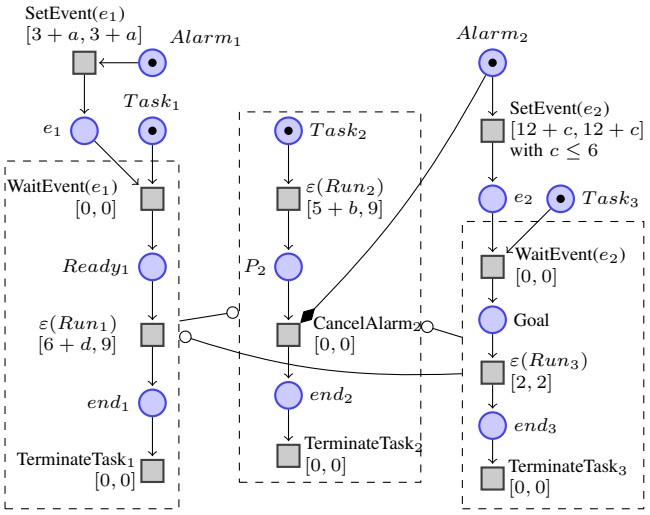

Figure 4. piPTN for tasks scheduling with fault tolerance

IV. APPLICATION TO AN EXAMPLE

A. Description of the application

To illustrate the whole approach, we have developed a small application with Trampoline RTOS.

It is composed of 3 tasks:

- $Task_1$ is activated by alarm $Alarm_1$ with an offset greater than or equal to 3ms. It runs for 6 to 9ms and terminates.
- $Task_2$ is activated at system startup. It runs for 5 to 9ms then tries to cancel alarm $Alarm_2$ before to terminate.
- $Task_3$ is activated by alarm $Alarm_2$ with an offset greater than or equal to 12ms. It runs for 2ms and terminates.

The complete model is given in figure 4. For briefness's sake, we define a shorthand notation between dashed box for inhibition between tasks: an inhibitor between two dashed boxes means that there is an inhibitor arc from each place of the source dashed box to each transition of the target dashed box. We also use reset arcs (also called flush arcs) represented by an arc with a black diamond. Those arcs empty their source places when their destination transition is fired. They are used here to model the cancellation of the timer. Parameters are introduced in the model on each transition that can be enforced at runtime.

B. Test scenario

In the scenario, we want to trigger the execution of $Task_3$. More precisely, we want to reach a state where the marking of place $Goal$ is not null. For this scenario, ROMEO computes the set of parameter values:

$$\begin{cases} d \leq 3 \\ b > 7 \\ b - c > 7 \end{cases} \cup \begin{cases} d \leq 3 \\ b > a - 2 \\ b - c + d > 1 \end{cases}$$

Any point in this polyhedron allows to run the scenario. We choose the following : $a = 0, b = 0, c = 0, d = 2$. At this point, the OIL file of the application is extended with the local model of each task integrating the chosen parameter values.

C. Results

The application has been built for the STM32F4Discovery board. The code of the application and the modified version of Trampoline are both available as open-source software at [12].

To visualize the execution of the application, waveforms have been plotted. The corresponding signal is produced by a slight instrumentation of the code of the system such that the signal is high when the task is executing, and low otherwise. In figure 5, each waveform corresponds to the execution of a tasks. From top to bottom, we have $Task_1$, $Task_2$ and $Task_3$.

At system startup, $Task_2$ runs for 3ms. Then it is preempted by $Task_1$. The enforcer delays the internal action $\epsilon(Run_1)$ of $Task_1$, with an EFT equal to 8ms ($6+d$, with $d=2$). $Task_2$ is resumed but too late to cancel $Alarm_2$ before that it triggers the execution of $Task_3$. Thus, 12ms after the startup, $Task_3$ is released, preempts $Task_2$, runs for $2ms$ and terminates. At last, $Task_2$ resumes and terminates its execution. This is the expected result.

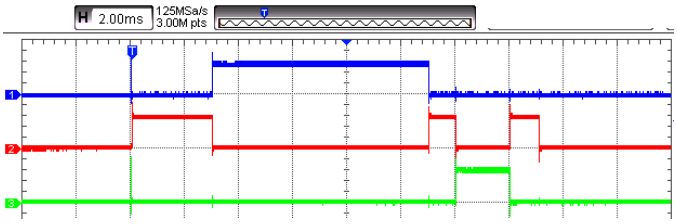

Figure 5. Enforcement of the execution of $Task_3$.

V. Conclusion

We have described a technique to enforce the execution of a test scenario assuming a specific internal timed behaviour of the system. In an offline phase, a model-based approach is used to compute a set of delays. Then in an online phase, a component observe the execution of the application and inserts the delays when needed to enforce the expected behaviour.

Concerning the offline phase, an existing modeling and analysis technique well suited for the domain of embedded control systems is used. However, the analysis is limited to the enforcement of reachability properties. The case for more complex properties, e.g. liveness, should be studied in the future.

Concerning the online phase, we have shown how to seamlessly integrate the enforcer in an AUTOSAR compliant RTOS. The corresponding code is available as open-source software. The measured overhead on the target hardware is compatible with the typical dynamics of embedded control systems. This overhead can also be accounted for during the analysis. Future works should investigate the indirect overhead caused by microarchitectural sideeffects (cache thrashing and other pollution of predictive execution mechanisms).

References

[1] W. Chen, S. Ray, M. Abadir, J. Bhadra, and L. C. Wang, "Challenges and trends in modern soc design verification," *IEEE Design Test*, vol. PP, 2017.

[2] R. Ernst and M. D. Natale, "Mixed criticality systems; a history of misconceptions?" *IEEE Design Test*, vol. 33, no. 5, pp. 65–74, 2016.

[3] L.-M. Givel, J.-L. Béchennec, M. Brun, S. Faucou, and O. H. Roux, "Testing real-time embedded software using runtime enforcement," in *Industrial Embedded Systems (SIES), 2016 11th IEEE Symposium on*. IEEE, 2016, pp. 1–6.

[4] L. C. Briand, Y. Labiche, and M. Shousha, "Using genetic algorithms for early schedulability analysis and stress testing in real-time systems," *Genetic Programming and Evolvable Machines*, vol. 7, no. 2, pp. 145–170, Jun 2006. [Online]. Available: https://doi.org/10.1007/s10710-006-9003-9

[5] Y. Falcone, T. Jéron, H. Marchand, and S. Pinisetty, "Runtime enforcement of regular timed properties by suppressing and delaying events," *Science of Computer Programming*, vol. 123, pp. 2–41, 2016.

[6] F. Mueller and J. Wegener, "A comparison of static analysis and evolutionary testing for the verification of timing constraints," in *Proceedings. Fourth IEEE Real-Time Technology and Applications Symposium (Cat. No.98TB100245)*, Jun 1998, pp. 144–154.

[7] M. M. Hassan, W. Afzal, B. Lindström, S. M. A. Shah, S. F. Andler, and M. Blom, "Testability and software performance: A systematic mapping study," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 2016, pp. 1566–1569.

[8] B. Berthomieu, D. Lime, O. H. Roux, and F. Vernadat, "Reachability problems and abstract state spaces for time Petri nets with stopwatches," *Discrete Event Dynamic Systems*, vol. 17, no. 2, 2007.

[9] L.-M. Traonouez, D. Lime, and O. H. Roux, "Parametric model-checking of stopwatch petri nets," *Journal of Universal Computer Science*, vol. 15, no. 17, pp. 3273–3304, 2009.

[10] D. Lime, O. H. Roux, C. Seidner, and L.-M. Traonouez, "Romeo: A parametric model-checker for Petri nets with stopwatches," in *TACAS 2009*, ser. LNCS, vol. 5505. Springer, 2009, pp. 54–57.

[11] J.-L. Bechennec, M. Briday, S. Faucou, and Y. Trinquet, "Trampoline an open source implementation of the osek/vdx rtos specification," in *IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, 2006.

[12] J.-L. Béchennec, S. Faucou, and O. H. Roux. Testing of a real-time Application - Trampoline with runtime enforcer. [Online]. Available: https://github.com/TrampolineRTOS/trampoline/tree/temporal-enforcement

Jean-Luc Béchennec earned a PhD degree in computer science at University of Paris VI in 1989. He is a full time researcher at CNRS (France) in the Laboratory of Digital Sciences of Nantes (LS2N, UMR 6004). He works in the field of system and RTOS implementation, runtime monitoring, modeling and verification.

Matthias Brun earned a PhD degree in Computer Science at University of Nantes (France) in 2010. He is associate professor at ESEO (France) in the Department of Software and Systems. He works on model driven software engineering with a particular interest in model transformation and model interpretation for real-time embedded systems.

Sébastien Faucou earned a PhD degree in Computer Science at University of Nantes (France) in 2002. He is associate professor at Université de Nantes. He is member of the laboratory of digital sciences of Nantes (LS2N, UMR 6004). He works on real-time embedded software systems: WCET analysis, runtime verification, and RTOS design.

Louis-Marie Givel defended his PhD at Ecole Centrale Nantes (France) in 2016. He works at Technology & Strategy company as an embedded systems engineer and does consulting for Magneti Marelli.

Olivier H. Roux earned a PHD degree in 1994. He is full Professor at Ecole Centrale Nantes (France) in the LS2N CNRS laboratory. His work deals with the verification and the control of timed systems. He has a particular interest in time Petri nets and their stopwatch and parametric extensions.