Sensitization of Symbolic Runs in Real-Time Testing Using the ORIS Tool

Laura Carnevali, Luigi Sassoli, Enrico Vicario
Dipartimento Sistemi e Informatica - Università di Firenze
carnevali@dsi.unifi.it, sassoli@dsi.unifi.it, vicario@dsi.unifi.it

Abstract

We address the problem of test case selection and path sensitization in the process of testing real-time preemptive systems, following a formal methodology based on the theory of preemptive Time Petri Nets (pTPN) implemented in the Oris tool. We discuss practical factors that limit feasible behaviors in the implementation of a nondeterministic specification and we motivate the assumption of test cases defined as paths selected in the symbolic state space of a pTPN specification. Feasibility and effectiveness of the proposed sensitization technique are demonstrated through experimentation on a real-time operating system.

1 Introduction

Testing is the process of verifying the correctness of a system through the exercise of its components and functionalities [1]. As a part of this process, *test case selection* consists in the determination of a suite of test cases attaining some degree of coverage with respect to some functional or structural abstraction of the system. *Sensitization* is the subsequent activity that determines the inputs that let the system run along selected cases.

Both activities have their specialization in the verification of reactive and real-time software. On the one hand, test case selection emphasizes the relevance of abstractions focusing on finite state behavior [20][12][16] and quantitative timing [15][19][18], with the aim of exercising the system along runs that can reveal faults related to concurrency, communication and timeliness. On the other hand, sensitization becomes a matter of determining sequencing and temporal parameters of controllable events and tasks much more than functional values.

Test case selection and sensitization can largely benefit from the application of formal methods. This adds rigor to the process of testing and reduces the possible gap between the verified abstraction and its actual implementation; the approach can provide an effective support even when formal analysis cannot be exhaustively completed, thus avoiding the limits of state space explosion; last but not least, the practice can be smoothly integrated with development processes without changing their essential nature, as required for industrial acceptance and explicitly recommended by software life cycle certification standards [11][8].

Formal methods in test case selection for reactive and real-time systems have been reported in various experiences, mostly following a functional approach. In the partial-WpMethod [12], a deterministic Finite State Machine (FSM) specification is used to derive a test suite, which achieves state identification and guarantees full fault coverage under the assumption of a correctly implemented reset function and of an upper bound on the number of states in the implementation. The approach is extended to real-time systems in [9] where a test suite is derived by applying the Wp-method on the FSM obtained through a finite sampling of the Region Graph of a specification model expressed as a Timed Input Output Automaton (TIOA). The method still guarantees full fault detection, but the complexity of the test suite tends to explode and the assumptions on the number of states in the implementation become much less realistic than in the untimed context. In [21], an untimed system is specified in a gray-box style as a SW architecture which is supposed to be translatable into a Labeled Transition System (LTS). A test suite is derived by achieving some FSM coverage criterion over a bisimulation reduction of the LTS which hides unobservable events.

In [15][19], a real-time system is specified as a deterministic and output-urgent Timed Automaton. Test cases are deterministically timed event sequences, which can be selected either as witnesses of real-time logic expressions capturing specific testing purposes or as elements of a test suite achieving some coverage over the locations of the specification automaton. In particular, all-nodes and all-edges criteria [22] are extended with a nice timed interpretation of dataflow testing principles [26] covering paths between the definition and the usage of clock variables. The approach assumes that all events can be observed and that all inputs can be controlled so as to assume a deterministic value and time. More importantly, since the specification is deterministic, the approach assumes that the system itself reacts with

1-4244-0826-1/07/$20.00 © 2007 IEEE

deterministic actions and timings.

Assumptions on deterministic behavior and observability of the implementation under test (IUT) are relaxed in [18], where the specification timed automaton may include nondeterministic choices and delays. In the framework, a tester is regarded as an adaptive strategy reacting to nondeterministic choices and delays of the IUT. Two kinds of testers are proposed: in analog testers, interaction with the IUT occurs at dense-valued times and adaptation is performed by restricting the symbolic state space of the specification with respect to observed events; in digital testers, interaction occurs at discrete times and adaptation is performed by tracing system behavior across a discrete representation of the timed state space. The construction of the tester is described as a general algorithm, but it is not instantiated with respect to specific coverage strategies or testing purposes.

In [19] the timed input output conformance (tioco) of [18] is relativized by taking environment assumptions explicitly into account (rtioco). An on-the-fly algorithm is described to test the rtioco conformance of a IUT through the generation of inputs derived from the composition of the environment and the specification. The approach is based on randomized testing and does not provide any coverage evaluation on the variety of timed behaviors.

In [7] we proposed the adoption of preemptive Time Petri Nets (pTPNs) [31][14][6] to support the verification process of real-time software, using a methodology that can be smoothly integrated within activities and practices of the industrial life cycle. In particular, in that paper we focused on the usage of pTPNs in the specification, formal verification and disciplined coding of a SW architecture, and in the support to testing activities through a time-sensitive Oracle which emits a verdict about sequencing and timeliness conformance of an implementation.

In this paper, we extend the methodology of [7] by proposing and experimenting a technique of path sensitization of selected behaviors. With this intent, we provide here four main contributions:

- We consider the case of a system running under preemptive scheduling. This is the most common practice in RT development, but involves the concept of suspension, which is not represented in specification models expressed as Timed Automata or Time Petri Nets and which poses complexities in symbolic analysis that have been addressed only recently [14][27][29][10]. In particular, we show how symbolic analysis in the Oris tool [28] supports determination of the weakest timings restriction that can attain path sensitization.

- We propose a testing framework addressing aspects related to the feasibility of control models defined by the tester component. On the one hand, sensitization of test cases must face the problem of partial controllability of events and temporal parameters of the implementation. On the other hand, it must cope with the unfeasibility of test cases for nondeterministic specifications, which mainly arises because nondeterministic temporal parameters are modeled through conservative ranges of variation and dependencies among them are often neglected by the specification model. Therefore, coverage criteria cannot be based on the selection of deterministic test cases, which may be not feasible for the implementation.

- We propose a test case to be a path in the state space resulting from symbolic analysis, because such a path represents the dense variety of runs associated to a set of events in a given qualitative order, with a dense variety of timings between subsequent events. We also define a strategy to force the IUT to run along selected cases.

- We assess feasibility and impact of the proposed framework through experimentation on a testbed running on conventional primitives of a real-time operating system (Linux RTAI).

2 Problem formulation

2.1 Specification of Task Sets with Preemptive Time Petri Nets

We consider a task set comprised of a set of recurrent *tasks* releasing *jobs* with three possible policies: i) *periodic*, in which tasks have a deterministic periodic release time; ii) *sporadic*, in which tasks have a minimum but not a maximum release time; and iii) *jittering*, in which tasks have a release time constrained between a minimum and a maximum. Each job can be internally structured as a sequence of *chunks*, each characterized by a nondeterministic execution time ranging within a finite interval. Each chunk may require a set of *preemptable resources*, and in this case it is associated with a priority level and runs under static priority preemptive scheduling. Chunks may have inter-task dependencies such as *semaphore* synchronizations and *message* passing precedences.

Task sets can be effectively modeled using preemptive Time Petri Nets (pTPN) [13][14], which extend Time Petri Nets (TPN) [25][2] with an additional mechanism of resource assignment making the progress of timed transitions be dependent on the availability of a set of preemptable resources. Syntax and semantics are formally expounded in [14], and we report here only an informal description. As in TPN, each transition is associated with a static firing interval made up of an earliest and a latest static firing time, and

each enabled transition is associated with a clock evaluating the time elapsed since it was newly enabled: a transition cannot fire before its clock has reached the static earliest firing time, neither it can let time pass without firing when its clock has reached the static latest firing time. In addition, each transition may request a set of preemptable resources, each request being associated with a priority level: an enabled transition is progressing and advances its clock if no other enabled transition requires any of its resources with a higher priority level; otherwise, it is suspended and maintains the value of its clock. This supports representation of the suspension mechanism and thus of preemptive behavior, attaining an expressivity that compares to that of stopwatch automata [10][27][14]. The pTPN in Fig.1 specifies two

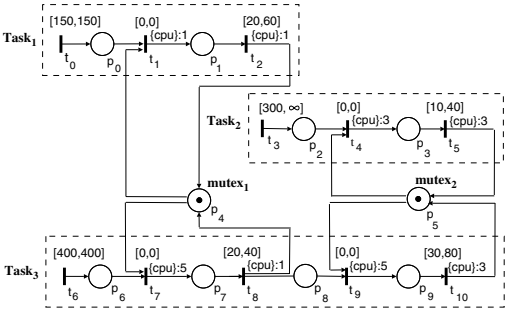


Figure 1. A pTPN specification model.

periodic tasks $Task_1$ and $Task_3$, with period equal to 150 and 400 time units, respectively, and a sporadic task $Task_2$ with minimum inter-arrival time of 300 time units. Transitions t_0 represents a job release for $Task_1$. Since it has no input place, its time to fire is reset at each firing. Computational chunks of each job are modeled by transitions with static firing intervals corresponding to the min-max range of execution time, with their resource requests and static priorities (high priority numbers correspond to low priority levels). For instance, transition t_2 represents the completion of the unique chunk of each job of $Task_1$, which requires cpu for a time ranging between 20 and 60 time units. Computations in different jobs may contend for a preemptable resource. For instance, both t_2 and t_5 require resource cpu, with priority 1 and 3, respectively; if t_2 becomes enabled while t_5 is progressing, then t_2 preempts t_5 and t_5 becomes suspended. Semaphores are modeled as places and their acquisition operations as immediate transitions, respectively. For instance, place p_4 is a semaphore, transitions t_1 and t_7 are *wait* operations performed by jobs of $Task_1$ and $Task_3$, respectively, and transitions t_2 and t_8 are the subsequent *signal* operations. Note that, according to a priority ceiling emulation protocol, the priority of $Task_3$ is modified after the acquisition of a semaphore.

2.2 Symbolic analysis of pTPN models

The state of a pTPN can be represented as a pair $s = \langle M, \tau \rangle$, where M is a marking and τ is a vector of times to fire for enabled transitions. Since τ takes values in a dense domain, the state space of a pTPN is covered using *state classes*, each comprised of a pair $S = \langle M, D \rangle$, where M is a marking and D is a firing domain encoded as the space of solutions for the set of constraints limiting the times to fire of enabled transitions. A reachability relation is established among classes: a state class S' is reachable from class S through transition t_0 , and we write $S \xrightarrow{t_0} S'$, if and only if S' contains all and only the states that are reachable from some state collected in S through some feasible firing of t_0. This reachability relation, sometimes called AE relation [24], defines a graph of reachability among classes that we call *state class graph* (SCG).

The AE reachability relation turns out to collect together the states that are reached through the same firing sequence but with different times [2][4][31]. A path in the SCG thus assumes the meaning of *symbolic run*, representing the dense variety of runs that fire a given set of transitions in a given qualitative order with a dense variety of timings between subsequent firings. A symbolic run is then identified by a sequence of transitions starting from a state class in the SCG, and it is associated to a completion interval, calculated over the set of completion times of the dense variety of runs it represents. Note that the same sequence of firings may be firable from different starting classes. According to this, we call *symbolic execution sequence* the finite set of symbolic runs with the same sequence of firing but with different starting classes.

If the model does not include preemptive behavior, i.e. if it can be represented as a TPN, firing domains can be encoded as Difference Bound Matrixes (DBM), which enable efficient derivation and encoding of successor classes in time $O(N^2)$ with respect to the number of enabled transitions N. Moreover, the set of timings for the transitions fired along a symbolic run can also be encoded as a DBM, thus providing an effective and compact profile for the range of timings that let the model run along a given firing sequence [31].

When the model includes preemptive behavior, then derivation of the successor class breaks the structure of DBM, and takes the form of a linear convex polyhedron. This results in exponential complexity for the derivation of classes and, more importantly, for their encoding [13][14][27][3]. To avoid the complexity, [14] enumerates the state space through a semi-algorithm which replaces classes with their tightest enclosing DBM, thus yielding to an over-approximation of the SCG. For any selected path in the over-approximated SCG, the exact set of constraints limiting the set of feasible timings can be recovered, thus

supporting clean-up of false behaviors and derivation of exact tightening durational bounds along selected critical runs [14]. In particular, the algorithm provides a tight bound on the maximum time that can be spent along the symbolic run and provides an encoding of the linear convex polyhedron enclosing all and only the timings that let the model execute along a symbolic run.

The Oris tool [28] implements the theory proposed in [14], thus supporting enumeration of the SCG using the tightest DBM encoding, clean-up of false behaviors and selection of symbolic runs attaining specific sequencing and timing conditions, together with the exact tightening of their range of timings. For the example of Fig.1, the state space enumeration leads to 37 reachable markings, covered by 2310 state classes. For each task, the analysis of the SCG allows the identification of the paths starting with the release of a job and ending with its completion, which we call *task symbolic runs*, and of the corresponding execution sequences, which we call *task execution sequences*. Specifically, the analysis identifies 465, 494, 870 symbolic runs for $Task_1$, $Task_2$ and $Task_3$, respectively. Their analysis provides the worst case completion time for each task (100, 240 and 280 for $Task_1$, $Task_2$ and $Task_3$, respectively), thus verifying that deadlines are met and with which minimum laxity (50, 160 and 20 for $Task_1$, $Task_2$ and $Task_3$, respectively). Task symbolic runs turn out to be collected into 15, 37 and 118 task execution sequences for $Task_1$, $Task_2$ and $Task_3$, respectively.

2.3 On the feasibility of test cases for nondeterministic specifications

While the analysis of the specification model supports early verification of the process architecture, confidence in the conformance of the implementation to the specification can be achieved through testing. In this step, which is in any case requested for certification purposes [11][8], the state space of the specification model can be exploited to select test cases and to identify timed inputs that let the system run along selected cases [15][19][18][16]. Both steps face the existence of behaviors that are legal in the specification model but cannot be observed in a real implementation. In principle, the problem applies to any process of conformance testing where the specification model may include nondeterministic behavior [30]. However, in the case of timed systems, nondeterminism in the specification model assumes a specific and major relevance, which may basically invalidate the coverage objectives assumed in test case selection [5].

In general, when the specification of a SW component is developed, various temporal parameters (e.g. execution times of computation chunks, jittering delays, interarrival times) are necessarily associated with a nondeterministic range of variation, both to accommodate changes in the embedding context and to allow a margin of laxity for the implementation.

Also, specification models usually neglect dependencies among temporal parameters, to keep the specification model relatively simple and to avoid the difficulty in quantifying dependencies. However, in practice, various dependencies may exist among computation times of subsequent chunks of the same job, among jobs of the same task or even among jobs of different tasks. Just to give a concrete idea of the concept, consider the case of a task performing MPEG compression of the chunks of a video stream: since compression time depends on contents, subsequent jobs will have correlated execution times. Taking this correlation into account in the specification would be difficult, and making schedulability be dependent on its effects would definitely threaten robustness. However, when dependency is neglected, no implementation can exhibit all the behaviors of the specification model. As a notable limit case, dependency between subsequent executions includes the case of a deterministic implementation, which is systematic in software requiring higher Integrity Levels, and which can be detected by static analysis of the code: in such an implementation, no selected test case assuming two different values for any two instances of the same temporal parameter will be actually feasible.

The operating system may also contribute to the partial determinization of the implementation, sometimes in subtle and unexpected manner. For instance, in the semantics of the specification model of Fig.1, synchronous tasks $Task_1$ and $Task_3$ at any multiple of their hyper-period may release a job in any order. However, in the implementation on top of Linux RTAI (discussed later in this paper), even if the two tasks are programmed with the same priority level, contemporary releases are always ordered so as to give precedence to the shortest period task $Task_1$, following a kind of monotonic policy, which is not affected by the order in which $Task_1$ and $Task_3$ are initially started. As a concrete effect, this prevents observation of a run in which $Task_3$ is released at the same time as $Task_1$ and overtakes it.

It is worth noting that the same problem occurs also when a model is developed as the description of an existing implementation (rather than as specification of an implementation to be built). This may happen in a variety of contexts, and notably within a re-engineering process or as a step to support reuse of a component within a modular architecture. Also in this case, the model will include nondeterministic temporal parameters ranging within boundaries which over-approximate those of the actual implementation. This is in fact necessary both to make the model robust with respect to possible variations and to circumvent the difficulty in obtaining a reliable estimate of execution times and other temporal parameters.

Also note that the problem of unfeasible paths also arises in structural testing, where test cases are selected on a control flow graph which represents the actual implementation but hides variable dependencies replacing functional conditions with nondeterministic choices [1]. As in that case, unfeasible paths in the control flow graph may invalidate coverage objectives, suggesting the opportunity that test case selection and sensitization be somehow integrated [5][17].

2.4 Assuming symbolic runs as test cases

Due to the mismatch between the set of timings of the specification and that of the IUT, coverage criteria cannot effectively rely on the selection of deterministic test cases, as many of these could be actually unfeasible. We thus propose that test cases be specified as symbolic runs, and that a test case be considered covered when the IUT has executed any of its runs.

As proposed in [15], a symbolic run can be selected as the witness of a specific test purpose determined trough a model checking technique, or it can be part of a test suite identified through a coverage criterion.

The Oris tool supports selection of a suite of symbolic runs enforcing various coverage criteria: *all markings* and *all state classes* guarantee that each reachable marking and each state class is visited by at least one selected symbolic run, respectively; *all marking edges* and *all class edges* guarantee that selected symbolic runs cover every edge between markings and classes, which corresponds to executing tasks under different logical conditions and timings, respectively; *all task sequences* and *all task runs* guarantee that all task execution sequences and all task symbolic runs are tested, respectively, thus providing confidence in the absence of faults in concurrency control and tasks interactions.

The possibility to effectively detect faults resulting in timeliness failures largely depends on the characteristics and assumptions on the oracle which performs off-line verification of time-stamped logs produced during testing. If each action (i.e. job releases, chunk completions, semaphore operations) is observable, the operational semantics of pTPNs can be exploited as a *time-sensitive Oracle*: this permits to detect failures pertaining both the sequencing and the timing of individual actions and provides a functional measure of coverage by comparing testing logs against the SCG [7].

3 Sensitization of symbolic runs

The dense variety of timings of the symbolic run representing a test case results from the combination of a number of nondeterministic temporal parameters which determine the dwelling times in the classes along the run. They range within intervals that may be mutually dependent, thus resulting in a linear convex polyhedron domain, which we call *symbolic run profile* and which is defined by a set of linear inequalities derived through the inspection of classes visited by the run itself [31][14]. In principle, a IUT can be forced to cover a test case by imposing temporal parameters corresponding to any point within the symbolic run profile. However, as mentioned in Sect.2.4, not all the points in the profile are feasible, and some temporal parameters may be actually not controllable. Periodic and asynchronous release times can be effectively controlled through conventional primitives of a real-time operating system. Whereas, controlling computation times requires that inputs can be selected and that the execution time that they produce can be estimated, which is often impractical. This gives a major relevance to state classes of the specification where no computational chunk is pending and all jobs are waiting for their next release, that we call *idle classes*, as the IUT can be controlled to start its execution from any state collected within one such class.

3.1 Sensitization procedure

We consider the problem of testing a symbolic run ρ originating from a class S_{target} (see Fig.2). This could be obtained by letting the IUT start from any state in the subset of the initial class S_0 which admits ρ^z as a feasible run, being ρ^z a symbolic run that reaches S_{target} and terminates with ρ. However, when S_{target} is distant from S_0, this results in a major computational complexity, and incurs in a high probability that uncontrollable actions let the IUT diverge from ρ^z before reaching S_{target}. To reduce the problem, we start the test from a state collected in any idle class S^w_{idle} which can reach S_{target} without visiting any intermediate idle class and can be efficiently identified by the dominance relation mentioned in [5]. Once S^w_{idle} has been identified, the IUT is repeatedly started from states sampled in the weakest restriction $\bar{S}^w_{idle} \subseteq S^w_{idle}$ which admits ρ as a feasible run, and it is stopped after the maximum duration of ρ^w has elapsed. During the execution, the IUT logs its activity on a real-time queue, which is analyzed off-line to verify whether the test is inconclusive (i.e. the execution has diverged from the selected run) or it is passed or failed. To sum up, this results in the following testing procedure:

1. Given a test case ρ that starts from state class S_{target}, the SCG is inspected to identify a symbolic run ρ^w that *(i)* starts from an idle class S^w_{idle}, *(ii)* reaches S_{target} without visiting any intermediate idle class and *(iii)* executes the symbolic run ρ.

2. The timing profile of ρ^w is derived through the algorithm reported in [31][14]. This provides the necessary condition to execute ρ^w, identifying *(i)* the subset $\bar{S}^w_{idle}$ of S^w_{idle} collecting all and only the states of S^w_{idle}

which admit ρ^w as a feasible run, and *(ii)* a set of timing constraints for transitions that are newly–enabled in classes visited by the run.

3. The IUT is started from a state in $\bar{S}^w_{idle}$ and each controllable action is forced to occur within the constraints of the timing profile of ρ^w. The IUT is restarted after a time corresponding to the worst completion time of ρ^w and an execution log is obtained for each restart, reporting sequencing and timing of observable actions. The log can be off-line analyzed in order to emit a verdict and to provide an evaluation of attained coverage, as performed in [7].

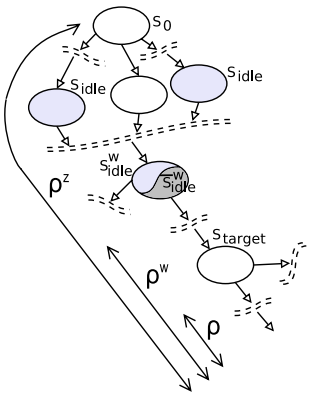


Figure 2. Sensitization procedure.

Steps 1 and 2 pertain to the analysis of the specification and their application will be exemplified through the Oris tool in Sect.3.2. Point 3 mainly involves instrumentation and testing of the implementation, which will be discussed and experimented in Sect.4.

3.2 Strategy enforcement with Oris

In this section we illustrate application of the sensitization procedure for a symbolic run selected as a specific testing purpose [15]. We first use the model checker of the Oris tool [28], which can generate all the symbolic paths in the SCG that are witnesses of a branching-time temporal logic formula, with state and action formulae expressing conditions on both visited markings and traversed transitions. Referring to the specification of Fig.1, we select symbolic runs of $Task_3$ with completion time higher than 250 time units.

To this end, we check whether the SCG contains any symbolic run that: (i) starts from an idle class, identified by the marking M_{idle} assigning a token to places p_4 and p_5 and no tokens to all the other places; (ii) reaches any state class that fires t_6 without visiting any intermediate idle class; (iii) terminates after more than 250 time units with the firing of t_{10} without including any intermediate firing of t_{10}.

Among the witnesses provided by the model checker, we then select ρ^w as the symbolic run that starts from state class $S^w_{idle} = \langle M_{idle}, D^w_{idle} \rangle$, where

$$D^w_{idle} = \begin{cases} 90 \leq \tau(t_0) \leq 130 \\ 90 \leq \tau(t_6) \leq 130 \\ 0 \leq \tau(t_3) \leq +\infty \\ 0 \leq \tau(t_0) - \tau(t_6) \leq 0, \end{cases} \quad (1)$$

and executes the firing sequence $\{t_0, t_6, t_1, t_2, t_7, t_3, t_8, t_4, t_5, t_9, t_0, t_1, t_2, t_{10}\}$ in a time comprised between 260 and 410 time units. This includes the test case ρ executing the sequence $\{t_6, t_1, t_2, t_7, t_3, t_8, t_4, t_5, t_9, t_0, t_1, t_2, t_{10}\}$ within 170 and 280 time units. The state class S^w_{idle} collects states in which periodic tasks are constrained to have the same release time (within the interval $[90, 130]$), while no constraints exist on the release of the sporadic task.

We then derive the timing profile of ρ^w using the trace viewer module of the Oris tool [31][14][28]. In our example, the application of sequencing constraints imposed by ρ^w restricts the time to fire of t_3 in class $\bar{S}^w_{idle}$ to be within 110 and 230 time units. According to this, the necessary condition to execute the test case consists in releasing the sporadic task $Task_2$ between 20 and 100 time units after the contemporary release of $Task_1$ and $Task_3$. Note that $[20, 100]$ is the largest time interval representing the weakest condition for $Task_2$ to be released between the completion of $Task_1$ and the completion of the first chunk of $Task_3$. The condition is necessary but not sufficient, since the execution of ρ also depends on computation times of $Task_1$ and $Task_3$.

The analysis also identifies the maximum execution time of 410 time units for ρ^w, which gives a limit after which the test can be stopped.

4 Computational Experience

The sensitization procedure has been experimented on top of the Linux RTAI operating system [23], to evaluate its feasibility and its effectiveness with respect to a randomized testing approach.

Randomized testing: We implemented the specification of Fig.1 as a kernel module following the disciplined (and straightforward) coding described in [7], which also reports a method to generate nondeterministic timings and to control duration of task computations. Temporal parameters of the specification are assumed to be expressed in milliseconds. Two different implementations were run for an hour, which corresponds to 24000 releases of the shortest period task. The first implementation releases $Task_2$

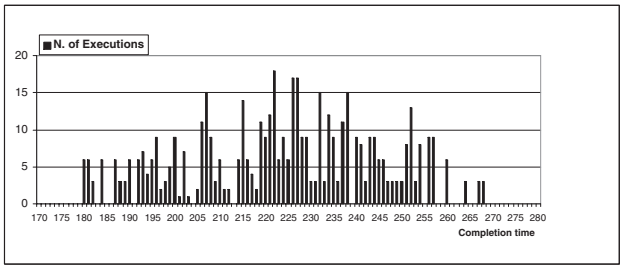
according to a uniform distribution between 300 ms and $N_{max}/10^6$ ms, being N_{max} the maximum representable integer value. Coverage evaluation evidenced that its execution covered 20.0%, 2.7% and 5.1% of execution sequences of $Task_1$, $Task_2$ and $Task_3$, respectively; in particular, the test case ρ was never covered. This is mainly due to the fact that ρ comprises a release of $Task_2$, which was activated only once during execution (as reported by testing log). The second implementation releases $Task_2$ according to a uniform distribution within the interval $[300, 1200]$ ms. Its execution reached a higher level of coverage: 66.7%, 62.2% and 37.3% of execution sequences of $Task_1$, $Task_2$ and $Task_3$, respectively. In particular, ρ was covered 51 times, with a completion time comprised between 175 ms and 249 ms.

Sensitized testing: We then forced the IUT to cover the test case ρ by controlling tasks release times. Results obtained in Sect.3.2 indicate that the system should be started from a state in $\bar{S}^w_{idle} \subset S^w_{idle}$, releasing $Task_2$ in a time comprised between 20 and 100 ms after the contemporary release of $Task_1$ and $Task_3$. Since S^w_{idle} collects idle states of the IUT, $Task_1$ and $Task_3$ can be released just after the system is started, without waiting the minimum time of 90 ms. In addition, the restart time of 410 ms derived by the analysis can be decremented by 130 ms, as the completion time of ρ^w is measured from the time of entrance in class S^w_{idle}. The reset function is obtained through an external script which iteratively loads the module into the kernel space and unloads it at the end of its execution, and through an additional high priority task, which is started after 280 ms to delete all the specification tasks. Within 280 ms, $Task_2$ can be released only once, having a minimum interarrival time of 300 ms. According to this, it is emulated through a one-shot task whose release time randomly varies within $[20, 100]$ ms at each restart.

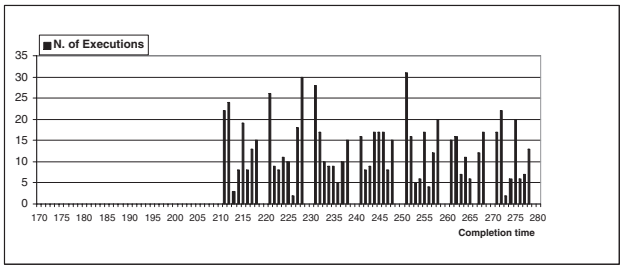
We run the modified implementation so as to perform 1000 restarts. With respect to randomized testing, less that 5 minutes execution were sufficient to cover the test case ρ 514 times, with a completion time comprised between 171 ms and 268 ms (see Fig. 3). Finally, we assumed that also computation time of $Task_1$ was controllable. This is achieved by replacing its code with an invocation to the function *void busy_sleep(int nanosecs)*, which iteratively executes the increment of a variable so that its execution lasts for *nanosecs* nanoseconds. On an AMD Athlon XP 2000+, we implemented this function with a precision of 100 μs, under the assumption of a bound of 10^{10} on the *nanosecs* parameter. We selected the maximum value of 60 ms for the computation time of $Task_1$, in order to obtain a high completion time when ρ is covered. Imposing this additional constraint to the timing profile of ρ^w reduces the range of values for the release time of $Task_2$ to the interval $[60, 100]$ ms since the contemporary release of $Task_1$ and $Task_3$. Again, the implementation was run 1000 times. With respect to the previous case, ρ was covered 725 times, which corresponds to an increment of 41%, and with higher completion times, comprised within the interval $[211, 278]$ ms (see Fig. 3).

	Testing Time	N. of executions covering ρ
Randomized testing (1)	3600 s	0
Randomized testing (2)	3600 s	51
Sensitized testing (1)	280 s	514
Sensitized testing (2)	280 s	725

(a)



(b)



(c)

Figure 3. (a) Number of executions of the test case ρ in random and sensitized testing. (b)-(c) Number of executions of ρ as a function of its completion time in sensitized testing, controlling only releases or both releases and $Task_1$ computation time, respectively.

5 Conclusions

We proposed a path sensitization technique for real-time preemptive systems, using pTPN theory in the testing process of real-time task sets. The technique assumes symbolic runs as test cases to overtake the problem of the unfeasibility of test cases being associated to a deterministic timing profile. The sensitization procedure is enforced taking advantage of the Oris tool, which allows state space enumeration of the specification model and selection of test cases, together with the derivation of those timing conditions that are necessary to execute them. Feasibility and effectiveness

of the sensitization strategy have been evaluated by considering the partial controllability of events of an implementation running on a real-time operating system.

For large models, architecture verification of the task set through state space enumeration may become unfeasible due to state space explosion. In this case, partial verification limited to a portion of the state space can still provide a relevant support in testing activities and, in particular, in sensitization techniques. In fact the state space, even if uncomplete, can still be employed to select critical behaviors to be tested, following the sensitization procedure proposed in this paper. In addition, it is worth noting that the pTPN model of the specification can still be employed as an Oracle in the evaluation of activity logs, also providing a level of coverage with respect to the portion of the state space which has been enumerated.

References

[1] B. Beizer. Black-box testing: Techniques for functional testing of software and systems. *Wiley*, 1995.

[2] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time petri nets. *IEEE Trans. on Soft. Eng.*, 17(3), March 1991.

[3] B. Berthomieu, D. Lime, O. H. Roux, and F. Vernadat. Reachability problems and abstract state spaces for time petri nets with stopwatches. *LAAS Report 04483*, 2004.

[4] B. Berthomieu and M. Menasche. An enumerative approach for analyzing time Petri nets. In R. E. A. Mason, editor, *Information Processing: proceedings of the IFIP congress 1983*, volume 9, pages 41–46. Elsevier Science, 1983.

[5] A. Bertolino and M. Marré. Automatic generation of path covers based on the control flow analysis of computer programs. *IEEE Trans. on Soft. Eng.*, 20(12):885–899, 1994.

[6] G. Bucci, L. Sassoli, and E. Vicario. Correctness verification and performance analysis of real time systems using stochastic preemptive time petri nets. *IEEE Trans. on Soft. Eng.*, (11):913–927, November 2005.

[7] L. Carnevali, L. Sassoli, and E. Vicario. Casting preemptive time petri nets in the development life cycle of real-time software. 19^{th} *Euromicro Conf. on Real-Time Sys.*, 2007.

[8] CENELEC-prEN50128. Railway applications: Sw for railway control and protection systems. 1997.

[9] A. En-Nouaary, R. Dssouli, and F. Khendek. Timed wp-method: Testing real-time systems. *IEEE Trans. on Soft. Eng.*, 28(11), 2002.

[10] F.Cassez and K.G.Larsen. *The Impressive Power of Stopwatches*, volume 1877. LNCS, August, 2000.

[11] R. T. C. for Aeronautics. Do-178b, software considerations in airborne systems and equipment certification. *http://www.rtca.org/*.

[12] S. Fujiwara, G. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite-state models. *IEEE Trans. on Soft. Eng.*, 17(2):591–603, 1991.

[13] G.Bucci, A. Fedeli, L. Sassoli, and E. Vicario. Modeling flexible real time systems with preemptive time petri nets. *Proceedings of the 15-th Euromicro Conference on Real-Time Systems (ECRTS03)*, July 2003.

[14] G.Bucci, A. Fedeli, L. Sassoli, and E. Vicario. Timed state space analysis of real time preemptive systems. *IEEE Trans. on Soft. Eng.*, 30(2):97–111, February 2004.

[15] A. Hessel, K. Larsen, B. Nielsen, P. Pettersson, and A. Skou. Time-optimal realtime test case generation using uppaal. *International Workshop on Formal Approaches to Testing of Software (FATES03)*, 2003.

[16] C. Jard and T. Jéron. Tgv: theory, principles and algorithms, a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Software Tools for Technology Transfer (STTT)*, 6, October 2004.

[17] B. Jeannet, T. Jéron, V. Rusu, and E. Zinovieva. Symbolic test selection based on approximate analysis. pages 349–364, April 2005.

[18] M. Krichen and S. Tripakis. Black-box conformance testing for real–time systems. *SPIN'04 Workshop on Model Checking Software*, 2004.

[19] K. Larsen, M. Mikucionis, and B. Nielsen. Online testing of real-time systems using uppaal: Status and future work. (04371), 2005.

[20] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines – a survey. *Proc. IEEE*, 84(8):1090–1123, 1996.

[21] H. Muccini, A. Bertolino, and P. Inverardi. Using software architecture for code testing. *IEEE Trans. on Soft. Eng.*, 30(3):160–171, 2004.

[22] S. Ntafos. A comparison of some structural testing strategies. *IEEE Trans. on Soft. Eng.*, 14(6):868–874, 1988.

[23] D. of Aerospace Eng. of the Polytechnic of Milan. Rtai: Real time application interface for linux. *https://www.rtai.org*.

[24] W. Penczek and A. Polrola. Specification and model checking of temporal properties in time petri nets and timed automata. *Proocedings of the* 25^{th} *Int. Conf on Application and Theory of Petri Nets, ICATPN2004*, June 2004.

[25] P.Merlin and D.J.Farber. Recoverability of communication protocols. *IEEE Trans. on Communications*, 24(9), 1976.

[26] S. Rapps and E.J.Weyuker. Selecting software test data using data flow information. *IEEE Trans. on Soft. Eng.*, 11(8), 1985.

[27] O. H. Roux and D. Lime. Time petri nets with inhibitor hyperarcs: formal semantics and state-space computation. *25th Int. Conf. on Theory and Application of Petri nets*, 3099:371–390, 2004.

[28] L. Sassoli and E. Vicario. Analysis of real time systems through the oris tool. *Proc. of the* 3^{rd} *Int. Conf. on the Quant. Evaluation of Sys.(QEST)*, Sept., 2006.

[29] T.Amnell, E.Fersman, L.Mokrushin, P.Pettersson, and W.Yi. Times: a tool for schedulability analysis and code generation of real-time systems. *Proc. of the* 1^{st} *Int. Workshop on Formal Modeling and Analysis of Timed Systems*, France, 2003.

[30] J. Tretmans. Test Generation with Inputs, Outputs, and Quiescence. 1055:127–146, 1996.

[31] E. Vicario. Static analysis and dynamic steering of time dependent systems using time petri nets. *IEEE Trans. on Soft. Eng.*, August 2001.