©2007 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Casting Preemptive Time Petri Nets in the Development Life Cycle of Real-Time Software

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 describe a methodology for the construction of real-time tasking sets, which smoothly integrates a formal approach in both development and verification processes of the software life cycle. In the design stage, a timeline schema is used to specify concurrent processes with their dependencies and their expected temporal parameters. The schema is automatically translated into an equivalent preemptive Time Petri Net, which supports verification of the process architecture with respect to timeliness and sequencing requirements through state space analysis. The specification model drives the implementation stage enabling a disciplined coding of the process architecture on top of conventional primitives of a real-time operating system. At the same time, the preemptive Time Petri Net specification and the results of its state space analysis support functional testing enabling the construction of a time-sensitive Oracle and providing a metrics for coverage analysis. Computational experience in the Linux RTAI environment is reported to demonstrate the capability of the method to be effectively integrated in a practical approach.

1. Introduction

In the development of embedded software, the interaction between concurrency and timing comprises a factor of major complexity. This characterizes non-functional requirements and basically conditions the actual behavior of the system: on the one hand, correctness is concerned with both the logical sequencing and the quantitative timing of events [12] [21]; on the other hand, durational properties restrict the sequencing of events, which in turn conditions completion times.

Under adequate assumptions on the structure of the tasking set, sequencing correctness is basically simplified and timeliness can be verified with relatively low computational complexity using schedulability analysis techniques such as the extended rate monotonic theory [16][17]. However, these analytical techniques cannot be effectively applied in a wide class of systems with such complexities as: inter-tasks dependencies due to mutual exclusion or dataflow precedences; internal sequencing of tasks; non-deterministic temporal parameters; sporadic tasks or tasks with mutual dependencies in the time of release; multiple processors.

For this kind of systems, the verification of both sequencing and timing correctness may become sufficiently critical to motivate the use of formal methods which enable early detection of errors at design stage and which complement and support testing activities that are in any case required during verification and certification processes. This is recommended in various process-oriented standards and in particular in the RTCA-DO/178B standard [7], where the adoption of formal methods is advised as an approach complementary to testing that may augment the confidence about the absence or limited probability of anomalous behaviors. The standard explicitly encourages the proposition of formal approaches in the verification of software with complex behavior, deriving from concurrency, synchronization and distributed processing, provided that the assumption of the formal method maintains a limited impact on development processes.

Several experiences have been reported on the usage of formal methods in the derivation of abstractions supporting the test case selection and execution. In [5] a test suite is derived by applying the Wp-method [8] on the Finite State Machine obtained through a finite sampling of the Region Graph of a specification model expressed as a Timed Input Output Automaton (TIOA). In [13][15], 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. Assumptions on deterministic behavior are relaxed in [14], where the specification timed automaton may include non-deterministic
choices and delays. In the framework, a tester is regarded as an adaptive strategy reacting to non-deterministic choices and delays of the implementation under test. A similar approach has been proposed in [15], where the timed input output conformance (tioco) of [14] is relativized by taking environmental assumptions explicitly into account (rtioco).

In this paper, we describe a methodology which smoothly integrates preemptive Time Petri Nets (pTPN) [10] in both development and verification processes of the software life cycle of real-time tasking sets.

In the development process, the architecture of the tasking set is specified through a timeline schema and automatically translated into a pTPN [10]. As a characterizing trait, this represents systems running under preemptive scheduling, which are not encompassed in the expressive domain of Timed Automata and Time Petri Nets (TPN), and whose state space analysis has been only recently addressed [10] [19] [6]. The pTPN specification enables symbolic enumeration of the state space supporting early verification of the design, and can be translated into code through a disciplined coding process, which relies on conventional primitives of a real-time operating system and which maintains a full view and control of the designer over the actual code structure.

In the verification stage, the pTPN specification model is exploited as a time-sensitive Oracle to perform off-line verification of activity logs produced during testing, to detect failures in the implementation which pertain both the sequencing and the timing of individual events and end-to-end response times. Besides, the symbolic representation of the state space can be compared against testing logs to provide a functional measure of coverage with respect to the overall range of timed behaviors that the system specification subtends.

To demonstrate the capability of the proposed methodology to be effectively integrated in a practical approach, we report computational experience on Linux RTAI [18], describing the implementation of a generic specification model and the additional components supporting the testing process, and reporting experimental results which assess the correctness of the proposed implementation and its capability to keep real-time timing, the precision of the proposed Oracle in the detection of sequencing and timing failures, and the significance of coverage analysis measures.

The rest of the paper is organized in six sections. The specification of the process architecture of a real-time tasking set and its verification are discussed in Sect.2 and Sect.3, respectively. The testing methodology is introduced in Sect.4. In Sect.5, we discuss how to implement the method on the basis of RTAI APIs and we report benchmarks on a case example. Finally, conclusions are drawn in Sect.6.

2. Using timelines in the specification of real-time preemptive systems

2.1. Structure of the tasking set

We assume a general setting that includes the patterns of process concurrency and interaction which are commonly encountered in the context of real-time systems [11].

1. Processes release tasks in recurrent manner with three different possible release policies: i) periodic, in which processes have a deterministic release time; ii) sporadic, in which processes have a minimum but not a maximum release time; and iii) jittering, in which processes have a release time constrained between a minimum and a maximum.

2. Tasks can be internally structured as a sequence of steps, each characterized by a nondeterministic execution time constrained within a minimum and a maximum value. Steps may have dependencies such as semaphore synchronizations and message passing precedences.

3. Tasks may require preemptable resources. In this case, they are associated with a priority level and run under static priority preemptive scheduling.

Tasking sets belonging to this setting can be conveniently specified through timelines. For instance, the schema in Fig.1 specifies three processes $P_1$, $P_2$, $P_3$. $P_1$ and $P_3$ are periodic with release period of 150 and 400 time units, respectively, while $P_2$ is sporadic with a minimum inter-arrival time of 300 time units. Computational steps are represented by a segment annotated with minimum and maximum execution time, resource requests, and priorities: e.g. the unique step of $P_1$ has a computation time in [20,60] time units and requires resource $cpu$ with priority level 1 (low priorities run first). The circles embracing the step

![Figure 1. A timeline schema for three processes having nondeterministic computation times and sharing two semaphores.](image-url)
indicate the acquisition and the release of semaphore \( m_1 \). Additional notations may indicate process deadlines (in our example, all deadlines are coincident with the minimum release period), and priority ceiling emulation may be implicitly assumed.

3. Using preemptive Time Petri Nets in the verification of the process architecture

The Oris tool [4] supports the editing and the translation of a timeline specification into an equivalent preemptive Time Petri Net (pTPN) [9], which permits to verify its correctness with respect to logical sequencing and quantitative timing constraints [10]. In this paper, we employ pTPNs not only to verify the specification, but also to support the testing of the compliance of a software implementation with respect to requirements pertaining both sequencing and timing of events. To this end, we briefly recall here the syntax and the semantics of the formalism.

3.1. Syntax

A pTPN is a tuple \( pTPN = (P; T; A^+; A^-; M; FI^s; Res; Req; Prio) \). \( P \) and \( T \) are disjoint sets of places and transitions, respectively; \( A^- \subseteq P \times T \) and \( A^+ \subseteq T \times P \) are relations on places and transitions called preconditions and postconditions, respectively. A place \( p \) is said to be an input or an output place for a transition \( t \) if there exists a precondition or a postcondition from \( p \) to \( t \) (i.e. if \( (p, t) \in A^- \) or \( (t, p) \in A^+ \), respectively). \( M \) is the (initial) marking, associating each place with a natural number of tokens.

\( FT^s \) associates each transition \( t \) with a firing interval delimited by an earliest static firing time \( EFT^s(t) \in \mathbb{R}^+_0 \) and a latest static firing time \( LFT^s(t) \in \mathbb{R}^+_0 \cup \{+\infty\} \).

\( Res \) is a set of resources disjoint from \( T \) and \( P \). \( Req \) associates each transition with a subset of \( Res \) (i.e. \( Req : T \rightarrow 2^{Res} \)), and \( Prio \) associates each transition with a natural number.

3.2. Semantics

Following an interval semantics, the State of a pTPN is a pair \( s = (M, FI) \), where \( M \) is a marking and \( FI : T \rightarrow \mathbb{R}^+_0 \times \mathbb{R}^+_0 \cup \{+\infty\} \) is a dynamic firing interval associating each transition with an earliest and a latest dynamic firing time, denoted as \( EFT(t) \) and \( LFT(t) \), respectively.

Fireability: A transition \( t_0 \) is enabled if each of its input places contains at least one token, and it is progressing if every resource it requires is not required by any other enabled transition with a higher level of priority. Transitions that are enabled but not progressing are said to be suspended. A transition \( t_0 \) is firable with firing time \( \tau_0 \) if it is progressing and \( \tau_0 \) is neither lower than the earliest firing time \( EFT(t_0) \) of \( t_0 \) or longer than the latest firing time \( LFT(t) \) of any other transition \( t \) enabled by \( M \).

Firing: when a transition \( t_0 \) fires with firing time \( \tau_0 \), the state \( s = (M, FI) \) is replaced by a new state \( s' = (M', FI') \), and we write \( s \overset{t_0}{\rightarrow} s' \). The marking \( M' \) is derived from \( M \) by removing a token from each input place of \( t_0 \) and by adding a token to each output place of \( t_0 \):

\[
M_{tmp}(p) = M(p) - 1, \forall p \cdot (p, t_0) \in A^- \\
M'(p) = M_{tmp} + 1, \forall p \cdot (t_0, p) \in A^+
\]

Transitions that are enabled both by the temporary marking \( M_{tmp} \) and by the final marking \( M' \) are said persistent, while those that are enabled by \( M' \) but not by \( M_{tmp} \) are said newly enabled. If \( t_0 \) is still enabled after its own firing, it is always regarded as newly enabled. The new firing interval \( FI' \) is derived by computing the firing interval of any transition enabled by the new marking \( M' \): i) for any transition \( t_a \) which is newly enabled after the firing of \( t_0 \), the firing interval equals the static firing interval: \( EFT'(t_a) = EFT^s(t_a) \) and \( LFT'(t_a) = LFT^s(t_a) \); ii) for any transition \( t_t \) which was progressing in the previous state and is persistent after the firing of \( t_0 \), the firing interval is shifted left by the value of \( \tau_0 \): \( EFT'(t_t) = \max\{0, EFT(t_t) - \tau_0\} \) and \( LFT'(t_t) = \max\{0, LFT(t_t) - \tau_0\} \); iii) finally, for any persistent transition \( t_t \) that was suspended in the previous state, the firing interval remains unchanged \( EFT'(t_t) = EFT(t_t) \) and \( LFT'(t_t) = LFT(t_t) \).

3.3. Automated translation of a timeline specification into a pTPN model

Fig.2 reports the pTPN model derived from the timeline schema of Fig.1. Repetitive releases are modeled as

![Figure 2. The pTPN model for the timeline schema of Fig.1.](image-url)
t_3$ stands for a sporadic release of $P_2$. Computational steps are modeled as transitions with static firing intervals corresponding to the min-max range of execution time: transition $t_2$ represents the computation of process $P_1$, comprised between 20 and 60 time units. Transitions are associated with resource requests and static priorities determining preemptive behavior: 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. Additional immediate transitions account for semaphore synchronizations: $t_4$ and $t_7$ stand for the wait operation performed by processes $P_2$ and $P_3$ over semaphore $m_1$, modeled by place $p_4$; the subsequent signal operations are modeled by the execution of $t_2$ and $t_8$ for processes $P_1$ and $P_3$, respectively.

Note that, since a priority ceiling emulation protocol is assumed, in the translation from the timeline schema to the pTPN, the priority of $P_3$ is modified at the acquisition of each semaphore.

3.4 Verification of the process architecture through state space analysis

The state of a TPN/pTPN depends not only on the discrete marking but also on dense-valued timers associated with transitions. To obtain a discretely enumerable reachability relation, the state space is covered with equivalence classes, called state classes, each collecting a dense variety of states. This is obtained by collecting together the states that are reached through the same firing sequence but with different times [3]. A state class is thus represented as a tuple $S = (M, D)$, where $M$ is a marking and $D$ is a firing domain which identifies a (dense) set of values for the timers associated with enabled transitions. While the marking $M$ is represented in a straightforward manner, the firing domain $D$ must be encoded as the space of solutions for the set of constraints limiting the timers of enabled transitions. A state class $S'$ is reachable from class $S$ through transition $t_0$, and we write $S \rightarrow t_0 S'$, 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 $t_0$. Enumeration of the reachability relation among state classes leads to the construction of the so-called state class graph (SCG).

Furthermore, in the presence of preemptive behavior, constraints on timers can take the form of any kind of linear inequality, changing the nature of time and space complexity involved in the derivation and representation of the firing domains. An exact enumeration approach was proposed in [2], in which firing domains are encoded as general multi-dimensional polyhedra and the verification of timing properties on execution sequences is performed by augmenting the model with observer places and transitions. The complexity is avoided in [9],[10] through the enumeration of an over-approximate relation of succession among state classes which maintains inequalities in DBM form; the exact set of constraints limiting the set of feasible timings can then be recovered for any execution sequence, thus supporting clean-up of false behaviors and tightening of durational bounds along critical runs.

![Figure 3. Best and worst completion times for all execution sequences of process $P_2$.](image)

For the example of Fig.2, the state space enumeration leads to 37 reachable markings, encoded into 2310 state classes. In none of them any place contains more than one token. This indicates that it is never the case that a process releases a task while its previous instance is still pending. The analysis of the state class graph permits the identification of all different symbolic runs, i.e. all the paths that start with a release of a process and end with its completion: for instance, for process $P_2$ paths starting with the firing of $t_3$ and ending with the firing of $t_2$. This leads to 465, 494, 870 symbolic runs for process $P_1$, $P_2$ and $P_3$, respectively, each identified by a starting class, a firing sequence, and an interval representing the minimum and maximum completion time. This permits to derive the worst case completion time for each process (100, 240 and 280 for $P_1$, $P_2$ and $P_3$, respectively), thus verifying that deadlines are met. In addition, different firing sequences encoded into symbolic runs identify all execution sequences that each process may manifest, i.e. all the different successions of observable events that can occur between a task release and its completion (15, 37 and 118 for $P_1$, $P_2$ and $P_3$, respectively). As an example, Fig.3 shows best and worst completion times for execution sequences of process $P_2$.

4. Testing an implementation with respect to sequencing and timing requirements

The pTPN model derived from the timeline schema not only enables architectural verification of the tasking set but
also supports the testing stage in the identification of failures regarding event sequencing and timing. In particular, we consider failures deriving from two main types of fault:

- **time frame violation**: a fault in the unit implementation if leading a temporal parameter to assume values out of its nominal interval;
- **cycle stealing**: the presence of additional processes which steal computational resources. These can be unexpected processes, services provided by the operating system, or processes intentionally not represented in the specification because considered not critical for the real-time application to be realized.

In this section, we describe how to employ pTPNs as an oracle in the identification of failures originated by these kinds of fault and how to evaluate the level of coverage attained by testing activities by taking advantage of the analysis results obtained in the verification stage.

### 4.1. The time-sensitive Oracle

We assume that the implementation is instrumented so as to provide a timestamped log of the following actions represented in the timeline specification: the release of a process, the completion of a step, the acquisition and release of a semaphore. According to this, each run executed by an implementation is associated with a finite sequence of timed actions \( tr = \{(a_n, \tau_n)\}_{n=1}^N \), where \( a_n \) is an action represented in the timeline specification, which corresponds to a unique transition \( t_n \) in the pTPN model, and \( \tau_n \) represents the time at which \( a_n \) has occurred. This sequence comprises a test, which can be evaluated using a simulator of the specification model to decide whether the run is accepted by the semantics of the pTPN model. This is decided by a **time-sensitive Oracle** according to a timed trace inclusion relation [13]: the specification accepts \( tr \) from the initial state \( s_0 \) if: \( i \) \( s_0 \xrightarrow{t_0} s_1 \); and the specification accepts \( tr^{-1} = \{(a_n, \tau_n)\}_{n=2}^N \) from state \( s_1 \).

Concretely, the decision algorithm operates as follows: starting from the initial state \( s_0 =< M_0, F I_0 > \) accounting for conditions at which the system is started, the algorithm checks the feasibility of the first timed action \( (a_1, \tau_1) \) and computes the subsequent state \( s_1 \); at the \( n \)-th step, the algorithm checks whether \( t_n \) can be fired at time \( \tau_n - \tau_{n-1} \) from state \( s_{n-1} \) and computes the resulting state \( s_n \). The Oracle emits a failure verdict if a timed action \( (a_n, \tau_n) \) is not accepted by the simulator, otherwise a pass verdict is emitted.

The time-sensitive Oracle somehow performs the function of the observers proposed in [13] [2]. In [13], an observer is an automaton employed online during the testing process to collect auxiliary information that is used for coverage evaluation. In [2], an observer is used to evaluate quantitative properties through state space enumeration of the specification model augmented with additional places and transitions. Differently from both concepts of observer, our Oracle evaluates offline the execution logs produced by an implementation. This is done by verifying if the sequence of timed actions is a subset of the dynamic behavior that the semantics of the specification model may accept.

It can be easily verified that any time frame violation fault is detected as a failure by the time-sensitive Oracle. Viceversa, a cycle stealing fault is recognized provided that its duration exceeds the laxity between an actual computation and its expected upper bound. In both cases, the Oracle detects an execution sequence which is not feasible, either because a transition is not firable or because it is firable but not with the observed timing.

Specific contexts of application may pose a limitation on the logging function and thus on the observability of events. In this case, the time-sensitive Oracle can be replaced through a **sequence-sensitive Oracle** or a deadline-sensitive Oracle, obtained by relaxing the assumption that each action is observable. The former assumes that execution logs report the sequence of actions, disregarding the time at which they occur; according to this, the oracle can check if the reported sequence is feasible independently of timings, by mapping it on a path of the state class graph. The latter assumes that execution logs report timed actions associated with task releases and completions, disregarding any intermediate action; this only permits to evaluate the completion times of execution sequences, identifying a failure only when a deadline is missed. An experimental comparison among the sensitiveness of oracles developed on the basis of different observability assumptions is reported in Sect.5.3.

As a final remark, it is worth noting that, in principle, the time-sensitive Oracle can be used not only to verify that all temporal parameters fall within their specified range, but also to obtain a statistical evaluation of their actual values and distributions. In this perspective, the Oracle behaves as a tool of execution time profiling.

### 4.2. Coverage analysis

When testing does not detect any failure, a metric of coverage is needed to provide a measure of confidence in the absence of residual faults. To this end, we take advantage of the same results obtained in the analysis of the specification. Coverage analysis is performed by mapping on the state class graph the timed actions reproduced by the oracle. The following Lemma guarantees that any sequence of timed actions verified by the testing algorithm corresponds to one and only one path in the state class graph.
Lemma 4.1 Let $S_0$ be the state class containing the initial state of the system. Any sequence of timed actions $\{(a_i, \tau_i)\}_{i=1}^N$ verified by the time-sensitive Oracle corresponds to one and only one path $S_0 \xrightarrow{\tau_1} S_1 \xrightarrow{\tau_2} S_2 \ldots S_N-1 \xrightarrow{\tau_N} S_N$ in the state class graph independently of the values $\{\tau_i\}_{i=1}^N$.

The proof of the Lemma directly descends from the assumption that all the actions represented in the specification are observable and from the definition of the reachability relation among state classes [22]. In particular, the latter is defined so as to get rid of the dependency on the dense value of the firing time and directly implies a homomorphic relation between the set of firing sequences of the net and the set of paths in the reachability graph of its state classes.

This result permits to employ the state class graph in the derivation of metrics on the percentage of nodes, edges and paths covered by executed tests. Among measures based on nodes (i.e. markings or state classes), we reasoned that a high level of coverage evaluated on the basis of markings gives a little confidence about the absence of faults, since markings, differently from state classes, do not depend on time. Reaching a high level of coverage measured on edges indicates an high probability that tasks have been executed under different logical conditions (when edges are intended as transitions between markings) and different timings (when edges are intended as transitions between state classes). A high coverage measured on the basis of paths (symbolic runs or execution sequences) will provide confidence about the absence of failures related to tasks interactions such as preemption or synchronization mechanisms. The same is for failures induced by the presence of an additional process, which are manifested only along given execution sequences, depending on process parameters such as the period of release, its priority and its computation time.

5. Experimentation with Linux RTAI

We show how RTAI APIs [18] can be used to implement a pTPN specification model and to develop additional components supporting the testing process. We then report experience in testing a family of variations of the implementation, obtained through the injection of time frame and cycle stealing faults, to evaluate the effectiveness of the proposed oracles and the significance of coverage analysis measures.

5.1. Implementing a timeline specification on Linux RTAI

RTAI [18] is a patch for the Linux kernel introducing a HW abstraction layer and an API supporting development of real-time applications for several processor architectures. RTAI allows the creation of real-time tasks both in the kernel space and in the user space. While in the former case a real-time application runs as a kernel module in the kernel address space, in the latter case the application runs in the user space but with hard real-time execution capabilities. Real-time tasks created in the kernel space provide the very best performance (the time needed for a context switch is typically under 40 $\mu$s, depending on the hardware), while those in the user space take a marginal hit in context switch time (typically under 100 $\mu$s). We develop our experimentation in the kernel space as a kernel module, on top of RTAI version 3.3.

RTAI APIs enable the implementation of a timeline specification through a disciplined translation, which in principle could be performed also automatically. As a characterizing trait, this translation leaves the programmer full control and understanding of the structure of code.

- The entire specification is implemented as a kernel module, which must have two entry points for loading and unloading: function init_module(), which is invoked when the module is loaded into kernel memory, and function cleanup_module(), that is called just before the module is unloaded, removing its functions from the kernel.
- Processes are implemented as real-time tasks, they are created in init_module() through rt_task_init(...) and started by calling a RTAI primitive depending on their release policy (rt_task_make_periodic(...) and rt_task_resume(...) for recurrent and one-shot processes, respectively); they are destroyed in cleanup_module() by invoking rt_task_delete(...).
- Semaphore operations must be appropriately combined with priority handling, to guarantee proper implementation of the specification model. RTAI provides both resource and binary semaphores. While the former implements priority inheritance, the latter leaves the programmer control over priority handling. We use binary semaphores to obtain an implementation conforming to the semantics of pTPN models with static priorities, though dynamic priorities could also be encompassed in pTPN expressivity and analysis [9].
  In the pTPN model (see Fig.2), the acquisition of a semaphore by a low priority process and the corresponding priority boost requested for priority ceiling emulation are collapsed into a single event. In a correct implementation, priority must be boosted before the semaphore wait operation to avoid priority inversion. Viceversa, at release, priority must be restored to the previous level after the semaphore signal operation.
- Data structures of the application, such as semaphores and real-time FIFO queues, are created and destroyed
5.2. Validating the oracle sensitivity

Time-stamped log of timed actions is needed to support the testing methodology. In addition, to validate the oracles capability in detecting timing and sequencing failures, we also need to emulate both execution times of computations and times of asynchronous releases, so as to perform fault injection.

Logging timed actions: Testing an implementation through the time-sensitive Oracle described in Sect.4 requires a log of the sequence of timed actions produced by an implementation run. As file operations are not available in the kernel space, real-time processes use RTAI IPC mechanisms to communicate the sequence of timed actions to a process in the user space, which writes on a file the information about the execution log. This responsibility is assigned to userProcess. Time complexity of log activities must be kept negligible with respect to the time scale of the specification. In fact, the time-stamped log of a timed action requires an invocation to rt::get.time().ns(), two assignments in the process memory and an invocation to rt::put(...) which adds the log to a real-time FIFO queue: on an AMD Athlon XP 2000+, this requires 150 ns on average.

Controlling duration of task computations: For the experimentation, we need to emulate computations lasting for a controlled duration. To this end, RTAI provides function void rt::busy.sleep(int nanosecs) which delays the execution of the caller task for nanosecs nanoseconds. Unfortunately, this function burns away CPU cycles in a busy wait loop without returning control to the scheduler, and it can be used only for very short synchronization delays, to avoid substantial errors in the measurement of time elapsed since the start of the scheduler. To overcome the limitation, we implemented a function void busy.sleep(int nanosecs), which repeatedly executes the increment of a variable, so that its execution lasts for nanosecs nanoseconds. Note that parameter nanosecs is expressed in nanoseconds in compliance with RTAI primitives, but the precision of function busy.sleep(...) depends on the time scale of the specification we deal with. Assuming the specification of Fig.1 is expressed in milliseconds, on an AMD Athlon XP 2000+ function busy.sleep(...) was implemented with a precision of 100 μs, under the assumption of a bound of 10^{10} on parameter nanosecs.

Generation of timings compliant with the specification and fault injection: To reproduce nondeterministic computations and interarrival times compliant with the specification, random values could not be generated using function rand() which cannot be used within a kernel module. Pseudo-random values were thus pre-calculated by a process in the user space, which we refer to as userProcess. Started before any real-time process represented in the specification, userProcess produces a table of pseudo-random values for every nondeterministic temporal parameter and communicates these values to the kernel module through RTAI IPC mechanisms.

To support injection of time frame violation faults, data can be perturbed by forcing userProcess to put in a table values out of the corresponding nominal interval. It is worth noting that userProcess can also compute these values according to a given probability distribution, thus allowing the simulation of faults with a different probability of occurrence.

Besides, cycle-stealing faults are injected by simply adding to the implementation one or more processes not represented in the specification.

Implementing additional components to support the testing process: To enable the testing process, we provide additional components extending the implementation with respect to the disciplined coding described in Sec.5.1. We illustrate the process with reference to the pTPN model of Fig.2 and code fragments are reported at the end of this section.

- To obtain execution logs, in init_module() we create a real-time FIFO queue with identifier logFIFO which is used by our kernel module to give userProcess the sequence of timed actions, each represented by the data structure timedAction.
- Process releases are implemented as highest priority tasks, otherwise the actual release time may become not observable, as processes may be preempted before completing log operations. Steps are computational units requiring a set of resources with static priority during execution. The first step is started by the release process as a real-time task. Subsequent steps are also implemented as real-time tasks, though they could also be implemented as functions.

As an example, we take into consideration code fragments of the two real-time tasks responsible for the release and the first computational step of the periodic process P3, respectively. The former enters a while loop: at each step, it logs the release action, then it starts the task representing its first computational step and finally it calls RTAI primitive rt::task.wait_period(...) to suspend itself until the next period is reached. The latter first boosts its priority and waits for the acquisition of semaphore m1; then, it logs the acquisition action, emulates the computation time and releases the semaphore, logging the release action;
finally, it starts the real-time task representing the subsequent step and destroys itself. It is worth noting that, if the subsequent step is implemented as a function, it is necessary to restore the priority to the previous value after logging the release action.

- To emulate the release of a sporadic process (which is activated in response to an asynchronous event), the process is implemented as a periodic task which varies its release period at each activation by invoking `rt_set_period(...`, sampling values in a table generated by `userProcess`.

- To control the duration of a task computation, it is sufficient to replace task specific operations with a call to function `busy_sleep(...)`. To reproduce nondeterministic computations and asynchronous interarrival times, `userProcess` provides real-time tasks with tables of pseudo-random values by writing them in a shared chunk of memory, allocated in `init_module()`. As `userProcess` should be started before processes of the specification, in `init_module()` we create a highest priority process `initProcess`, which waits for `userProcess` to complete and then starts all the specification processes. This is obtained launching an external script which first loads the kernel module through the command `insmod moduleName.o` and then runs `userProcess` through `user_process`. The space memory and `initProcess` are deallocated and destroyed, respectively, in `cleanup_module()`.

- Finally, the overall test execution can be stopped after a predefined duration $T$ through an additional process which assumes responsibility of terminating all the processes of the specification.

```c
int init_module(void) {
  rtTypedSemInit(&m1, 1, BIN_SLM);
  rtTypedSemInit(&m2, 1, BIN_SEM);
  rtfCreate(logFIFO, size);
  pseudoRandTables=(long *)rtai_kmalloc(memoryName, tablesNumber*valuesPerTable* sizeof(long));
  int initProcessPriority=0;
  rt_task_init(&initProcess, initProcess_function, 0, 0, initProcessPriority, 0, 0);
  rt_set_periodic_mode();
  start_rt_time(nano2count(500000));
  rt_task_resume(&initProcess);
  return 0;
}

static void initProcess_function(int t) {
  while((writtenValuesNumber<tablesNumber*valuesPerTable)
  rt_sleep(delay);
  ...
  int plPrio = 0;
  int p2Prio = 0;
  int p3Prio = 0;
  rt_task_init(&p1, p1_function, 0, 2000, p1Prio, 0, 0);
  rt_task_init(&p2, p2_function, 0, 2000, p2Prio, 0, 0);
  rt_task_init(&p3, p3_function, 0, 2000, p3Prio, 0, 0);
  rt_task_make_periodic(&p1, p1Starttime, p1Period);
  rt_task_make_periodic(&p2, p2Starttime, p2Period);
  rt_task_make_periodic(&p3, p3Starttime, p3Period);
  rtTaskDelete(rtWhoami());
}

static void p3_step1_function(int p3SteplIndex) { 
  struct {int action; long long time};
  int action;
  long long time;
  int p3SteplPrio,
  rt_task_init(&p3Stepl, p3Stepl_function,
  rt_task_resume(&p3Stepl);
  rt_task_wait_period();
}

5.3. Results

We report experimental results obtained by testing two families of implementations with different injected faults. This permits to evaluate the capability of the time-sensitive Oracle in detecting timing and sequencing errors and in providing different measures on the attained coverage. In addition, it allows comparison among oracles developed on the basis of different assumptions on the observability of the actions of the specification model.

Mutations under test: We consider two mutations of a code implementing the specification of Fig.1. A first mutation includes a time frame violation obtained by introducing a probability of 1% for the second step of process $P_3$ to take a computational time comprised within $[140,200]$ ms, according to a uniform distribution. A second mutation concerns a cycle stealing effect due to the unexpected presence of an additional process having a sporadic release with a minimum interarrival time of $9 \times 10^5$ ms, a computation time comprised between $[0,10]$ ms and priority level 2.

Testing activities: In order to provide a set of execution logs, each faulty implementation was run 20 times, and each run was stopped after one hour. To obtain different runs for the same mutation, each run is started after a call to `userProcess`, which provides the generation of random values.
for nondeterministic timings of the implementation. The compliance of timed actions with respect to the specification has been checked offline through an ad-hoc extension of the Oris tool [20]. This takes in input the net of Fig.2, the corresponding state class graph, and the execution logs. The tool verifies the absence of sequencing and timing failures running the algorithm described in Sect.4 and evaluates the level of coverage on the basis of the state space.

**Oracle sensitiveness:** The evaluation of execution logs evidenced that every failure is detected by the time-sensitive Oracle: this happens on 20 executions of the first mutation and on 17 executions of the second mutation. In the first case, failures are observed as feasible execution sequences manifesting a late completion time of $P_3$ or as unfeasible execution sequences for $P_3$. In the second case, the fault is generated by an unobservable action (the execution of the additional process is not reported on the execution log), and it is observed only if it causes a sequencing or timing failure on an observable action. For instance, when the additional process releases a task that suspends a task released by process $P_2$, a failure is identified if the sum of computation times of the two tasks overruns 40 ms (the nominal interval for the computational step of $P_2$ is [10, 40] ms).

The time-sensitive Oracle did not detect any failure which is not due to faults injected in the implementation, thus evidencing that the coding discipline succeeded in implementing the specification. In addition, this indicates that the implementation is not affected by the overhead of logging timed actions when the timings of the specification are on the order of milliseconds.

The effectiveness of the time-sensitive Oracle has been evaluated with respect to a sequence-sensitive oracle and a deadline-sensitive oracle. Fig.4a reports the number of failures identified in the overall 40 execution logs by the three different oracles, as a function of execution time. With respect to the maximum value of 37 failures detected by the time-sensitive Oracle, the sequence-sensitive oracle detects only 20 failures, each manifested as an unfeasible execution sequence, while the deadline-sensitive oracle detects 4 failures as missed deadlines.

**Coverage evaluation:** The state class graph was employed in the evaluation of metrics on the percentage of nodes, edges and paths covered by executed tests. Results for the two types of faulty implementations are reported in Fig.4b and Fig.4c. The vertical axis reports the number of detected failures, while the horizontal axis indicates the attained percentage of coverage according to the different metrics. These include metrics based on nodes (markings or state classes), on edges (transitions between markings or state classes) and on paths (execution sequences or symbolic runs). Fig.4b and Fig.4c also indicate that an hour of test (corresponding to 24000 releases of the shortest period process) is sufficient to reach a high level of coverage only with respect to metrics based on markings and transitions.

![Figure 4](image_url)

(a) A comparison among three oracles developed under different assumptions on the observability of events. (b) The figure shows the number of identified failures as a function of the percentage of coverage, obtained by injecting a time frame fault in the second step of process $P_3$. Tests are referred to 20 executions of the implementation, each run for one hour. (c) The same plot obtained for an implementation containing an additional sporadic process.

---

among them.

6. Conclusions

In this paper, we described how preemptive Time Petri Nets can be smoothly integrated into the life cycle of real-time software. The architectural description of a tasking set through a pTPN model represents a rigorous specification, which is, by itself, one of the levels in usage of formal methods recommended by the RTCA-DO/178B standard. The analysis of the specification model can be employed for early architectural verification of a tasking set, supporting tight schedulability analysis and verification of the correctness of logical sequencing. The specification model also permits the definition of a time-sensitive Oracle supporting functional testing of a software implementation. This allows the identification of failures with respect to sequencing and timing constraints induced by the specification and the evaluation of the level of coverage attained through testing activities.

For large models, verification of process architecture 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 fault detection. It is worth remarking that, also in this case, the pTPN model of the specification can be employed as an oracle in failures detection. In fact, the time-sensitive Oracle emits its verdict through a simulation of execution logs in the specification model, and employs the state space only in the evaluation of the level of coverage reached by testing activities.

Practical application of the method has been demonstrated on top of conventional APIs of a real-time operating system and its effectiveness has been experimented through the injection of faults of different typologies. This experience points out the capability of the time-sensitive Oracle in detecting any failure breaking timing or sequencing constraints, thus evidencing the possibility of employing it not only in integration testing activities, but also to support unit testing.

References