Putting preemptive Time Petri Nets to work in a V-Model SW life cycle

Laura Carnevali, Lorenzo Ridi, Enrico Vicario
Dipartimento Sistemi e Informatica - Università di Firenze
{laura.carnevali, lorenzo.ridi, enrico.vicario}@unifi.it

Abstract—Preemptive Time Petri Nets (pTPNs) support modeling and analysis of concurrent timed SW components, running under fixed priority preemptive scheduling. The model is supported by a well-established theory based on symbolic state-space analysis through Difference Bounds Matrix (DBM) zones, with specific contributions on compositional modularization, trace analysis, and efficient over-approximation and clean-up in the management of suspension deriving from preemptive behavior.

In this paper, we devise and implement a framework that brings the theory to application. To this end, we cast the theory into an organic tailoring of design, coding, and testing activities within a V-Model SW life cycle in respect of the principles of regulatory standards applied to the construction of safety-critical SW components. To implement the toolchain subtended by the overall approach into a Model Driven Development (MDD) framework, we complement the theory of state-space-analysis with methods and techniques supporting semi-formal specification and automated compilation into pTPN models and real-time code, measurement-based Execution Time estimation, test-case selection and execution, coverage evaluation.

Index Terms—Real-time systems, safety-critical SW components, SW life cycle, V-Model, preemptive Time Petri Nets, symbolic state-space analysis, Model Driven Development, automated model transformation, automated code generation, Execution Time estimation, real-time testing, test-case selection and execution, coverage analysis.

I. INTRODUCTION

In the development of safety-critical SW, formal methods may provide a crucial help both in design and in verification activities. This is explicitly encouraged in certification standards [60], [13], [56] with specific reference to SW with complex behavior deriving from concurrency, synchronization, and distributed processing, provided that proposed methods are smoothly integrated within a defined and documented SW life cycle [12]. In this context, the adoption of formal methods may serve to attain various degrees of rigor, ranging from nonambiguous specification to support for manual verification and even to automated verification [60]. The integration of formal methods along multiple activities of the SW life cycle is also typical in Model Driven Development (MDD), where Domain Specific Modeling Languages (DSMLs) enable the formalization of system requirements and design, and transformation engines support the development of generators of concrete artifacts including code, documentation, complementary models, and tests [29], [16].

The concepts of the MDD have been practiced in various tools [67], [7], [68] with different goals in the aspects of model formalization, implementation, and verification. Simulink [67] supports mathematical modeling and simulation of complex control systems through block diagrams, which can then be translated into C-code for various Real-Time Operating Systems (RTOSs), emphasizing performance and signal-flow over correctness, and hiding the effects of concurrency and resource usage. Charon [7] specifies interacting systems through modular hybrid state machines, which capture architectural and behavioral aspects and enable formal validation [6]; code generation experiences are mainly focused on the preservation of properties that might be jeopardized in the mapping of continuous/concurrent behavior of the model onto a discrete/preemptive target platform [61]. The Giotto Language [68] introduces an intermediate layer of abstraction between the mathematical model and the corresponding generated code, defining an embedded software model which takes into account timing and concurrency constraints while neglecting functionality concerns and platform-dependent choices such as scheduling or preemption policies [32]; the actual execution of the generated SW components on a target platform relies on the use of a Virtual Machine that accounts for scheduling, preemption, and resource allocation policies.

In [41], an on-line controller based on the structure of a discrete time state-space keeps the execution of the system within a range of correct behavior. A similar approach in continuous time is reported in [18]. In both cases, exhaustive enumeration of the state-space becomes a precondition for the implementation stage, and on-line control puts an overhead on timing resources. Moreover, the resulting code is closer to the perspective of the analysis than to usual patterns of real-time programming. In the Uppaal tool [23], the formal nucleus of Timed Automata is exploited to support state-space analysis of non-preemptive models, test-case selection and execution [8], [43], and synthesis of a controller that drives the system along selected behaviors in the state-space [19]. Extension to systems running under preemptive scheduling is addressed by Timed Automata with Tasks (TATs) [17] through the composition of asynchronous task releases with the analysis technique of [38], under the assumption that the model does not include both nondeterministic Execution Times and dependencies among release and completion times of tasks [22].

Preemptive Time Petri Nets (pTPNs) [26], [25] extend Time Petri Nets (TPNs) with resources, representing suspension in the advancement of clocks. This realizes the so-called Preemptive ReSuMe execution policy (PRs) [51] and enables convenient modeling of the usual patterns of real-time concurrency [28], with an expressivity that compares with...
StopWatch Automata [20], Petri Nets with hyper-arcs [55], and Scheduling-TPNs [50]. The analysis technique of [26] enumerates an over-approximation of the state-space that maintains the efficient encoding of Difference Bounds Matrix (DBM) zones and supports exact identification of feasible timings of selected behaviors through a clean-up algorithm, enabling efficient verification of reachability properties under timing constraints and real-time deadlines. The over-approximation is applied also in [50], where the state class graph is translated into a bisimilar Linear Hybrid Automaton [6] with a reduced number of clocks and analyzed through the HyTech model checker [31]. In [69], symbolic analysis of TPNs with inhibitor arcs is extended with temporal parameters, supporting design decisions through the automated derivation of min-max values of parameters.

In this paper, we propose a comprehensive methodology that brings the theory of pTPNs to concrete application. To this end, we consolidate and extend preliminary results on code-generation and real-time testing reported in [44], [45], [46], [47], so as to devise an organic tailoring of the V-Model SW life cycle [12] following the principles of standards for safety critical SW [60], [13], [56] and the experience gained in a pilot project of industrial development. To this end, state-space analysis of [26], [25] is integrated within a full-fledged MDD approach supporting: modeling of real-time SW through a semi-formal specification and automated translation into a corresponding pTPN; generation of real-time code preserving sequencing and timing properties of the model; Execution Time profiling; test-case selection and execution, oracle verdict, and coverage evaluation. With reference to the framework of [14], this can be regarded as a hybrid approach combining model-to-model and model-to-text transformations.

The paper is organized as follows. In Sect.II, we present the concept of the proposed tailoring, we introduce a running case study reflecting our industrial experimentation, and we define the conceptual model of a task-set extending [44], [45], [46], [47] so as to encompass the patterns of real-time concurrency encountered in that industrial practice. In Sect.III, we describe the modeling of task-sets through pTPNs (Sect.III-A); we illustrate application of the theory reported in [26], [25] with reference to the running case study (Sects.III-B and III-C); and, we show how pTPNs can be derived from a semi-formal specification based on timelines with semaphores and messages, extending [44], [45], [46] with a detailed description of the translation process, and with an example of application on the running case study (Sects.III-D and III-E). In Sect.IV, we consolidate and extend preliminarily results reported in [44], [45], [46] about the implementation of pTPN models, adding new contribution on: generation of code that complies with regulatory standards for safety-critical SW and encompasses message passing communication to include a pattern frequently encountered in the industrial practice (Sects.IV-A and IV-B); a measurement-based approach to Execution Time estimation, which was systematically applied to the verification of time-frame requirements of low-level modules (Sects.IV-C and IV-D); and, an experimental assessment for the evaluation of the accuracy of measurements (Sect.IV-E).

In Sect. VI, we expand concepts reported in [45], [46] adding: an explicit model of failures and defects addressed by the proposed approach (Sect.V-A); a comprehensive discussion about the application of pTPNs to test-case selection, coverage analysis, and evaluation of executed tests (Sects.V-B and V-C); experimental results on the application of the test-case execution approach of [47] to the running case study (Sect.V-E). Conclusions are finally drawn in Sect.VI.

II. MAPPING pTPN THEORY AND TOOLS ONTO A V-MODEL SW LIFE CYCLE

Fig.1 reports a scheme of the System Development (SD) Submodel of the V-Model framework [12]. We illustrate the process referring to the specific tailoring applied to the development of a real-time controller for an electro-optical system at the Finmeccanica site of Galileo Avionica in Florence. To avoid disclosure of classified details, concepts are transposed into a case of equivalent complexity that addresses an Intelligent Visual Surveillance System (IVSS) performing real-time 3D tracking of people moving within a target area.

System-level User Requirements are determined in SD1 (System Requirements Analysis) and mapped to the Units of a System Architecture in SD2 (System Design). SD3 (Software/Hardware Requirements Analysis) allocates Technical Requirements to CSCIs (Computer Software Configuration Items) and HCIs (Hardware Configuration Items) that comprise each Unit. In the specific case of study, User Requirements define parameters of the vision system, such as the frame rate (i.e., 25 frames per second), the image size (i.e., 160 x 120 pixels), the compression format (i.e., JPEG), the operating temperature (i.e., from 0°C to 40°C), the camera mass (i.e., up to 1.5 Kg). In SD2, these requirements are allocated to 3 Units, shown in the diagram of Fig.2: a Control Unit (CU), a Pan-Tilt-Zoom (PTZ) Camera, and a Video Processing Unit (VPU). The CU sends images acquired by
the PTZ Camera to the VPU, receives the elaborated images back, and consequently sets values for parameters of image acquisition (e.g., the levels of pan, tilt, and zoom) and image processing (e.g., the window size of non-linear filters), sending them to the PTZ Camera and the VPU, respectively. In SD3,

![Diagram of System Architecture](image)

Fig. 2. Case study: SD2. UML (Unified Modeling Language) component diagram of the System Architecture in the MARTE (Modeling and Analysis of Real-Time and Embedded systems) profile [54]: the ProcessingResource stereotype models an active, protected, executing-type resource allocated to the execution of schedulable resources; the DeviceResource stereotype specializes the concept of processing resource to represent an external device that may be manipulated/invoked by the platform through specific services, with internal behavior not relevant for the model under consideration; a CommunicationMedia represents a mean to transport information from one location to another.

CSCIs and HCIs are specified through more detailed UML-MARTE component diagrams. As an example, Fig.3 shows the diagram for the VPU, which is made by 3 CSCIs (i.e., Images Acquisition (IA), Basic Features Extraction (BFE), and Multiple Target Tracking (MTT)) and 4 HCIs (i.e., 3 boards running the RTAI (Real-Time Application Interface) operating system [15] and a battery): each CSCI is implemented as a different real-time task-set and allocated to a different board.
Fig. 3. Case study: SD3. UML component diagram of the VPU in the MARTE profile: the rtUnit stereotype models a real-time application that owns one or more schedulable resources; a SchedulableResource is an active resource that performs actions using the processing capacity brought from a processing resource managed by a scheduler; a Hw_Card symbolizes a printed circuit board typically made by chips and electrical devices; the allocate stereotype identifies an allocation relation between elements of the application model and of the execution platform, represented through the stereotypes ApplicationAllocationEnd and ExecutionPlatformAllocationEnd, respectively; a Hw_PowerSupply is a hardware component supplying the hardware platform with power; the Hw_Bus stereotype represents a particular wired channel with specific functional properties.
Our proposed methodology comes into play with SD4-SW (Preliminary Software Design) and SD5-SW (Detailed Software Design), where each CSCI is mapped on a set of concurrent and interacting tasks according to the following model derived from the practice of real-time systems [28], [36] and described by the UML diagram of Fig.4:

- A task releases jobs in a recurrent manner with periodic, sporadic, or jittering policy, depending on whether the release time is deterministic, bounded by a minimum but not a maximum value, or constrained between a minimum and a maximum value, respectively.
- A job is internally structured as a sequence of chunks, each characterized by a non-deterministic Execution Time constrained between a minimum and a maximum value. Chunks representing a computation are also associated with an entry-point function for the attachment of functional behavior to the corresponding low-level module.
- Chunks belonging to jobs of different tasks may have dependencies (e.g., semaphore synchronization, message passing communication through a mailbox) and may require resources (e.g., one or more processors), in which case they are associated with a priority level and run under static priority preemptive scheduling. Wait and signal semaphore operations are constrained to occur at the beginning and at the end of chunks, respectively, but a semaphore may be held across multiple subsequent chunks of the same task. Receipt and dispatch mailbox operations are constrained to occur at the beginning and at the end of chunks, respectively.

In SD4-SW, the Software Architecture of each CSCI is specified as a task-set with assigned functional low-level modules, prescribed release times and deadlines. Minimum inter-release times and deadlines are directly prescribed by Technical Requirements, while task periods are design choices. The Software Design is completed in SD5-SW with the allocation of resources and time requirements to SW modules through the specification of target processors, task priorities, and allowed time-frames of computation chunks. In the usual practice, the latter design choice is initially based on analogy with previous realizations and may require iterations along the development process. In the proposed methodology, in both the stages of SD4-SW and SD5-SW, task-sets are modeled through a semi-formal specification and automatically translated into pTPN models through the Oris Tool [27], [44]. According to this, the sub-activity SD5.2-SW (Analysis of Resources and Time Requirements, not shown in Fig.1) is supported by pTPN theory and the Oris Tool through state-space analysis, which drive two nested cycles of feedback on design choice of temporal parameters until the structure of the dynamic architecture of each task-set proves to be adequate to meet sequencing and timeliness requirements. Referring again to the RTCA/DO-178B standard [60], state-space analysis verifies the design with the level of rigor of full coverage, which is recommended there with explicit reference to the case of concurrent and timed behavior. When state-explosion prevents complete enumeration and analysis, the rigor degree of the approach downgrades to the level of a method that supports manual verification, but it is in any case able to verify a significant part of the state-space, by far beyond the limits that could be attained through code inspection and testing.

SD6-SW (Software Implementation) automatically derives the implementation of the dynamic architecture of each CSCI from its semi-formal specification through the Oris Tool [27], [44]. The structure of the code closely follows readable patterns of preemptive real-time programming and relies on conventional RTOS primitives. Again, this meets the RTCA/DO-178B [60] recommendation that the introduction of formal methods does not change the essential nature of development processes and, moreover, avoids legacy constraints on tools for automated generation, which also has relevance for industrial acceptance. SD6-SW also includes implementation of functional code of low-level modules attached to the entry-points. The sub-activity SD6.3-SW (Self Assessment of the Software Module, not shown in Fig.1) performs unit-testing of low-level modules and it is supported by pTPN theory in the verification of timing requirements through a measurement-based approach to Execution Time profiling.

SD7-SW (Software Integration) verifies the integration of low-level modules within each CSCI (SW Component Level in Fig.1) and then the integration of CSCIs within each Unit (Unit Level in Fig.1). The theory of pTPNs impacts on the verification of the integration at the SW Component Level supporting test-case selection, test-case execution, oracle verdict, and coverage evaluation.

Out of the scope of impact of the methodology proposed here, SD8 (System Integration) composes units and performs self-assessment of the system, while SD9 (Transition To Utilization) operates the transition that leads to install the completed system at the intended application site and to put it into operation.

III. SPECIFICATION AND ARCHITECTURAL VERIFICATION OF REAL-TIME TASK-SETS

In SD4-SW and SD5-SW, the representation of the dynamic architecture of a CSCI through pTPN models enables schedulability analysis and sequencing verification through symbolic analysis. We recall here the salient aspects of state-space analysis of pTPNs [26], [25] and report on its application in the case of study.

pTPNs [25], [26] extend Petri Nets with the timing semantics of TPNs [58], [10], [18] and with an additional mechanism of resource assignment, that makes the progress
of timed transitions dependent on the availability of a set of preemptable resources. As in Petri Nets [65], the model is a bipartite graph where nodes are places and transitions, accounting for conditions and events, respectively: a transition is enabled iff all its input places contain at least one token; an enabled transition may fire and in this case a token is removed from each of its input places and a token is added to each of its output places. As in TPNs [58], [10], each transition is associated with a static firing interval made of an earliest and a latest 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 can it 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. The expressivity of the model compares with StopWatch Automata [20], Petri Nets with hyper-arcs [55], and Scheduling-TPNs [50], providing a convenient setting for the representation of the patterns of concurrency and inter-task communication described in the task-set model of Sect.II.

A. Case study: SD4-SW and SD5-SW (formal specification)

The task-set of the BFE CSCI is made by 5 tasks: \( Tsk_1 \) performs noise reduction through a median filter; \( Tsk_2 \) manipulates parameters employed by the noise reduction algorithm; \( Tsk_3 \) and \( Tsk_4 \) accomplish edge and corner detection, respectively; \( Tsk_5 \) manipulates parameters employed by edge and corner detection algorithms. In particular, \( Tsk_2 \) sends values for parameters of the noise reduction algorithm to \( Tsk_1 \); \( Tsk_3 \), \( Tsk_4 \), and \( Tsk_5 \) are synchronized in order to access a shared memory, where \( Tsk_3 \) and \( Tsk_4 \) read values for parameters of edge and corner detection algorithms written by \( Tsk_5 \); \( Tsk_3 \) and \( Tsk_4 \) are also synchronized in order to write in a second shared memory the results of edge and corner detection algorithms, respectively. Fig.5 reports the pTPN model of the dynamic architecture of the BFE CSCI: \( Tsk_1 \), \( Tsk_2 \), \( Tsk_3 \) and \( Tsk_4 \) are periodic tasks, with period equal to 40, 40, 80, and 100 time units, respectively; \( Tsk_5 \) is a sporadic task with minimum inter-release time equal to 120 time units. Transitions \( t_{10} \), \( t_{20} \), \( t_{30} \), \( t_{40} \), and \( t_{50} \) model recurrent job releases of \( Tsk_1 \), \( Tsk_2 \), \( Tsk_3 \), \( Tsk_4 \), and \( Tsk_5 \), respectively: they have neither input places nor resource requests, so as to fire repeatedly with inter-firing times falling within their respective firing intervals. Job chunks are modeled by transitions with static firing intervals corresponding to the min-max range of Execution Time, associated with resource requests and static priorities (low priority numbers run first). For instance, transition \( t_{12} \) represents the completion of the unique chunk of each job of \( Tsk_1 \), which requires resource \text{cpu} \ with priority 1 for an Execution Time ranging between 5 and 10 time units.

In general, we assume that the access to shared resources is regulated by the priority ceiling emulation protocol [49], which raises the priority of any locking chunk to the highest priority of any chunk that ever uses that lock (i.e., its priority ceiling). Although the protocol does not require the use of semaphores on single-core systems, we assume semaphore synchronization in compliance with the general case of multi-core systems, which are in fact encompassed in modeling and analysis of the proposed approach. Binary semaphores are modeled as places initially marked with 1 token. For instance, place \text{mutex} \ models a binary semaphore synchronizing the first chunk of \( Tsk_3 \), the first chunk of \( Tsk_4 \), and the second chunk of \( Tsk_5 \): \text{wait} operations are modeled by the immediate transitions (i.e., with firing interval equal to \([0, 0]\)) \( t_{31} \), \( t_{42} \), and \( t_{53} \); \text{signal} operations are represented by transitions \( t_{32} \), \( t_{43} \), and \( t_{54} \), which also account for the completion of the three synchronized chunks. According to the priority ceiling emulation protocol, priorities of tasks \( Tsk_4 \) and \( Tsk_5 \) are raised in the sections where they hold a resource. Priority \text{boost} operations are modeled in explicit manner by the immediate transitions \( t_{41} \), \( t_{45} \), and \( t_{52} \), which precede \text{semaphore wait} operations. Conversely, the corresponding \text{deboost} operations are allocated to transitions \( t_{43} \), \( t_{47} \), and \( t_{54} \), which also account for the completion of computation chunks. In fact, preemption by a different task within the priority ceiling can occur at the boost operation but not at deboost, and thus the model does not need to distinguish chunk completion and priority deboost.

Mailboxes are modeled by empty places. For instance, place \text{mbx} \ represents a mailbox used in message passing communication between the first chunk of \( Tsk_2 \) (sender) and the unique chunk of \( Tsk_4 \) (receiver): message \text{dispatch} operations are modeled by transition \( t_{21} \), which also accounts for the completion of the first chunk of \( Tsk_2 \); message \text{receipt} operations are modeled by the immediate transition \( t_{11} \).
Fig. 5. Case study: SD4-SW and SD5-SW. The pTPN model of the dynamic architecture of the BFE CSCI, made by 5 tasks synchronized by 2 binary semaphores and 1 mailbox.
B. Architectural verification through state-space enumeration of the pTPN model

In state-space analysis, the state of a pTPN is a pair \( s = (M, \tau) \), where \( M \) is a marking and \( \tau \) is a vector of times to fire for enabled transitions. Since \( \tau \) takes values in a dense domain, the state-space of a pTPN is covered using state-classes, each comprised of a pair \( S = (M, D) \), 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 [18], [26]: a class \( S^c \) is reachable from class \( S^p \) through transition \( t_0 \), and we write \( S^p \xrightarrow{t_0} S^c \), if and only if \( S^c \) contains all and only the states that are reachable from some state collected in \( S^p \) through some feasible firing of \( t_0 \) (i.e., \( s_c \in S^c \iff \exists s_p \in S^p \mid s_p \xrightarrow{t_0} s_c \)). This relation defines a graph of reachability among classes called state-class-graph (SCG) [18], [26].

The reachability relation among classes preserves the EA condition [70] among states (i.e., \( \forall s_c \in S^c, \exists s_p \in S^p \mid s_p \xrightarrow{t_0} s_c \)) and guarantees that states reached through the same firing sequence but with different times are collected in the same class [10], [11], [18]. 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 with 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 fireable 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 DBM zones, which enable efficient derivation and encoding of successor classes in time \( O(N^2) \) with respect to the number \( N \) of enabled transitions [18], [26]. Moreover, the set of timings for the transitions fired along a symbolic run can also be encoded as a DBM zone, thus providing an effective and compact profile for the range of timings that let the model run along a given firing sequence [18].

When the model includes preemptive behavior, derivation of the successor class breaks the structure of DBM zone and takes the form of a linear convex polyhedron, resulting in exponential complexity for the derivation of classes and, more importantly, for their encoding [25], [26], [55], [9]. To avoid the complexity, [26] replaces classes with their tightest enclosing DBM zone, thus yielding 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. In particular, the algorithm provides a tight bound on the maximum time that can be spent along any 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.

C. Case study: SD5.2-SW

The Oris Tool [27] supports enumeration of the SCG, selection of symbolic runs attaining specific sequencing and timing conditions, and tight evaluation of their range of timings. For the pTPN model of the dynamic architecture of the BCE CSCI, the state-space includes 433 reachable markings, covered by 51079 state-classes. For each task, the analysis of the SCG identifies the paths between release and completion of each job and the corresponding execution sequences. Specifically, tasks \( Tsk_1, Tsk_2, Tsk_3, Tsk_4, \) and \( Tsk_5 \) have 15295, 10967, 21170, 491703, and 156574 symbolic runs, and 497, 113, 935, 25377, and 14044 symbolic execution sequences, respectively. The analysis provides a Worst Case Completion Time (WCCT) of 12, 14, 30, 58, and 60 time units for \( Tsk_1, Tsk_2, Tsk_3, Tsk_4, \) and \( Tsk_5, \) respectively, thus verifying that all deadlines are met with minimum laxity of 28, 26, 50, 42, and 60 time units, respectively.

As an example, Fig.6 reports the results of trace analysis on a symbolic run of \( Tsk_4 \) that was selected through state-space analysis as a case in which \( Tsk_4 \) may attain its WCCT of 58 time units. After the initial job release (i.e., the firing of transition \( t_{40} \), not shown in the schema of Fig.6), task \( Tsk_4 \) is suspended until time +30: first, it undergoes preemption by the releases of tasks \( Tsk_1, Tsk_2, \) and \( Tsk_3 \), then, it is overtaken by the first chunk of \( Tsk_2 \), by the unique chunk of \( Tsk_3 \), and by the second chunk of \( Tsk_2; \) next, it undergoes preemption by the second chunk of \( Tsk_5 \) and by the three chunks of \( Tsk_3, Tsk_4, \) and \( Tsk_5 \). Finally, after the completion of its first chunk, \( Tsk_4 \) is overtaken again by the releases of tasks \( Tsk_1, Tsk_2, \) and \( Tsk_3 \), by the first chunk of \( Tsk_2 \), by the unique chunk of \( Tsk_3 \), and by the second chunk of \( Tsk_3 \). Note that \( Tsk_5 \) is the lowest priority task, but its priority is raised to the level of \( Tsk_4 \) before the acquisition of the binary semaphore \( nulex \).

Trace analysis detects mutual dependencies among transition firing times along the sequence (e.g., transition \( t_{44} \) can fire in \([+47, +56]\) and transition \( t_{47} \) can fire in \([+48, +58]\), but the assumption that transition \( t_{44} \) fires at time +56 restricts \( t_{47} \) to fire in \([+57, +58]\)). This may serve to identify the timings that let the system run along that sequence and the specific values that yield the WCCT. In so doing, anomalies are detected as the cases where the WCCT along the run requires that some previous firing occurs before its latest feasible value.
Fig. 6. Case study: SD5.2-SW. A schema illustrating the range of feasible timings for the symbolic run \( \rho = S_{5605} t_{40} S_{5795} t_{20} S_{6092} t_{45} S_{6591} t_{24} S_{7106} t_{21} S_{7741} t_{10} S_{8120} t_{13} S_{8406} t_{15} S_{8711} t_{10} S_{9030} t_{13} S_{9313} t_{13} S_{9521} t_{15} S_{9687} t_{15} S_{9873} t_{15} S_{10121} t_{15} S_{10396} t_{13} S_{10670} t_{15} S_{10969} t_{15} S_{11222} t_{12} S_{11729} t_{15} S_{12164} t_{15} S_{12607} t_{12} S_{13028} t_{12} S_{13454} t_{15} S_{13907} t_{15} S_{14344} t_{12} S_{14739} \) in the state-space of the pTPN model of the dynamic architecture of the BFE CSII. The run starts with the release of a job of Task 4 (\( t_{40} \) entering \( S_{5605} \), not shown) and terminates with its completion (\( t_{47} \)). Time advances along the horizontal axis. Transition firings are represented by arrows and positioned at their latest feasible time, while striped rectangles indicate their allowed range of variation (e.g., \( t_{12} \) fires twice along the sequence: first within \([+10,+12]\) and then within \([+46,+52]\); the two firings are displayed at time \(+12\) and \(+52\), respectively). Transitions enabling-periods are marked through grey or white rectangles depending on whether the transition is progressing or suspended, respectively (e.g., \( t_{22} \) fires twice along the sequence and in both cases is suspended until the firing of \( t_{12} \)). Classes entered at transition firings are listed over the schema (e.g., \( t_{12} \) enters \( S_{5795} \); in case of simultaneous firings, classes are listed from the top in their order (e.g., \( t_{10} \) enters \( S_{5795} \), which is left at the firing of \( t_{20} \) to enter \( S_{6092} \)).
D. Timeline specifications and derivation of pTPNs

In SD4-SW and SD5-SW, the dynamic architecture of a CSCI is modeled through a semi-formal specification based on the concept of timeline schemas and it is automatically translated into a corresponding pTPN model. Timeline schemas restrict the expressivity of pTPNs in a manner that gives explicit representation to the structural concepts of task, job, chunk, hyperperiod, completion-time deadline, and entry-point. This enables a direct formalization of the task-set at a higher level of abstraction which smooths the complexities of pTPN domain, provides modeling convenience, and facilitates industrial acceptance.

1) Tasks and jobs: A task is specified by a name \(Tsk\) and a release interval \([\text{min, max}]\) (see Fig.7a); in the equivalent pTPN model (see Fig.7b), it is accounted by a transition with neither input places nor resource requests. A task can also be associated with a deadline, which has no counterpart in the equivalent pTPN model, being part of the descriptive requirements rather than of the operational specification of the task-set design. When not specified, the deadline is implicitly assumed to be coincident with the minimum inter-release time.

Fig. 7. The timeline schema for a task (a) and the transition modeling its job releases in the equivalent pTPN model (b).

2) Chunks: A chunk is specified by a name \(C\), an execution interval \([\text{beet, wcet}]\), a set of resource requests specifying the processors to which the chunk is statically allocated, and the level of priority of its scheduling (see Fig.8a). The set of resource requests is empty in case the chunk accounts for a waiting state rather than for a computation. In the equivalent pTPN model (see Fig.8b), each chunk \(C\) is translated into a transition with an input place \(C_{\text{in}}\). The sequence of chunks that form a task is represented by chaining input places of individual chunks with the initial task release transition. Entry-point functions associated with computation chunks have no counterpart in the pTPN model, but serve in the later stages of coding and testing to attach functional behavior to low-level modules.

3) Semaphore synchronization and priority ceiling: Semaphore operations are specified by decorating chunks with circles annotated with a (non-atomic) sequence of semaphore operations, each referred to a semaphore name (see Fig.9a). In the equivalent pTPN model (see Fig.9b); each semaphore is translated into a place initially marked with 1 token; a wait operation is accounted by an immediate transition preconditioned by the semaphore place; a signal operation is collapsed with the completion of the chunk that releases the semaphore. Note that the translation process incorporates the priority ceiling emulation protocol, and thus a boost transition is added, while priority deboost is collapsed with the completion of the chunk that releases the semaphore.

Fig. 8. The timeline schema for a task with two chunks (a) and the equivalent pTPN model (b).

4) Message passing communication through mailboxes: Mailbox operations are specified by decorating chunks with a (non-atomic) sequence of mailbox operations, each referred to a mailbox name (see Fig.10a). In the equivalent pTPN model (see Fig.10b): each mailbox is translated into a place with no tokens; a send operation is collapsed with the completion of the chunk that dispatches the message; a receive operation is accounted by an immediate transition preconditioned by the mailbox place.

E. Case study: SD4-SW and SD5-SW (semi-formal specification)

Fig.11 shows the timeline schema of the dynamic architecture of the BFE CSCI: entry-point \(f1\) performs noise reduction; entry-points \(f21\) and \(f22\) manipulate parameters of the noise reduction algorithm; entry-points \(f31\) and \(f41\) acquire values for parameters of the edge and corner detection algorithms, respectively, from a memory shared with task \(Tsk_3\); entry-points \(f32\) and \(f42\) perform edge and corner detection, respectively, and entry-points \(f33\) and \(f43\) write the corresponding results on a second shared memory; entry-point \(f51\) manipulates parameters of the edge and corner detection algorithms, and entry-point \(f52\) writes them on the memory shared with tasks \(Tsk_3\) and \(Tsk_4\).

IV. CODING AND EXECUTION TIME PROFILING

In SD6-SW, the model of the dynamic architecture of each CSCI is implemented into code that assumes the fol-
following responsibilities: release task jobs according to their policies; invoke semaphore, mailbox, and priority handling operations; enforce sequenced invocation of entry-points. The implementation is derived from the timeline model, either automatically or through manual programming guided by the model structure. As a salient trait, in both approaches the resulting code has a readable structure, which follows natural and readable patterns of concurrent programming and closely mirrors the structure of the corresponding pTPN model. This enables a measurement-based approach to Execution Time profiling [59]. In [44], we described the SW architecture of an MDD engine supporting agile transformation of models in the Oris Tool. We report here on the structure of the code produced by the engine in the automated translation of timeline schemas into real-time code running on RTAI version 3.6 [15]. Translation towards VxWorks 5.5 and 6.2 [71] was also experimented.

A. Implementation of the dynamic architecture of a CSCI on RTAI

Fig.12 describes the architecture of the implementation. Salient aspects of the code implementing the dynamic architecture of the BFE CSCI of Fig.3 are reported in and Figs.13 and 14 (times expressed in ms).
Fig. 12. UML class diagram for the implementation structure of the dynamic architecture of a CSCI on RTAI in the MARTE profile: the stereotype SwResource represents a software structural entity provided to the user by the execution support; SwMutualExclusionResource, MessageComResource, and SharedDataComResource are subtypes of SwResource describing a resource commonly used to synchronize access to shared variables, a communication resource used to exchange messages, and a specific resource used to share the same area of memory among concurrent resources, respectively; the stereotype SwConcurrentResource specializes SwResource to represent an entity that provides a routine with an executing context; the stereotype SwSchedulableResource specializes SwConcurrentResource to describe a resource that executes concurrently with other resources under the supervision of a scheduler, modeled by the stereotype Scheduler; the stereotype EntryPoint represents a routine executed in the context of a SwConcurrentResource; the stereotype BehavioralFeature represents the routine to be executed.
1) Implementation of a task-set: The task-set is implemented as a kernel module, loaded into the kernel space through the entry-point init_module (class LoadModule in Fig.12) and unloaded at the end of the execution through the exit-point cleanup_module (class UnloadModule in Fig.12). The entry-point init_module allocates data structures that are used in the task-set and the exit-point cleanup_module deletes them (in Fig.13). These include two binary semaphores mux1 and mux2 and a mailbox mbx, which are explicitly represented in the timeline model of Fig.11. A binary semaphore named asynch_mux5 and a shared memory named ASYNCH_SHM, which are instrumental to the implementation of the sporadic task Tsk5).

Binary semaphores are implemented as RTAI binary semaphores (class BinarySemaphore in Fig.12), created and deleted through the primitives rtTypedSemInit and rtSemDelete, respectively. RTAI binary semaphores require explicit priority handling operations to implement the priority ceiling emulation protocol [49]. Actually, RTAI resource semaphores natively implement priority inheritance and they could be encompassed in the model by extending the semantics of pTPNs with marking-dependent priorities without significantly impacting on state-space analysis techniques [25]; however, the construct was not used to facilitate portability across different RTOs. Mailboxes are implemented as RTAI Mailboxes (class Mailbox in Fig.12), created and deleted through the primitives rtTypedMbxInit and rtMbxDelete, respectively. Shared memory spaces (class SharedMemory in Fig.12) are created through the RTAI primitive rtaiKmalloc, which allocates in the kernel space a chunk of memory to be shared among kernel modules and Linux tasks; they are freed by means of the RTAI primitive rtaiKfree.

Before activating any real-time task, the entry-point init_module starts the timer in periodic mode with a period of 500 µs through the RTAI primitives rtSetPeriodicMode and startRtTimer. The exit-point cleanup_module stops the timer through the RTAI primitive stopRtTimer.

2) Implementation of each individual task: For each task Tskx in the timeline model, the entry-point init_module creates a real-time task tskx (class Task in Fig.12) through the RTAI primitive rtTaskInit, which associates a real-time task with an entry-point function, a priority level, and a stack. The exit-point cleanup_module deletes the real-time tasks through the RTAI primitive rtTaskDelete. In Fig.13, the exit-point function associated with a real-time task tskx is named tskx_job and it is responsible for the execution of task jobs (class Job in Fig.12). For each periodic task of the specification (e.g., tasks Tsk1, Tskg, Tsk3, and Tsk4 in the timeline model of Fig.11), init_module also defines a start time and a period, through the RTAI primitive rtTaskMakePeriodic (class PeriodicTask in Fig.12). The primitive does not make the real-time...
task actually recurrent and periodic, but only guarantees that an invocation of the RTAI primitive \texttt{rt\_task\_wait\_period} will suspend the task until the start-time of the next period. To make the task recurrent, an explicit loop control structure is programmed in the function \texttt{tsk\_}\texttt{x\_}\texttt{job} that performs job executions, and each repetition of the loop is completed with an invocation of \texttt{rt\_task\_wait\_period}.

The loop body of \texttt{tsk\_}\texttt{x\_}\texttt{job} implements a single job by performing: invocation of entry-point functions that execute chunk computations (in Fig.14, \texttt{tsk4\_}\texttt{job} invokes the entry-points \texttt{f41}, \texttt{f42}, and \texttt{f43}); wait and signal semaphore operations (through the RTAI primitives \texttt{rt\_sem\_wait} and \texttt{rt\_sem\_signal}, respectively); priority boost and deboost operations according to the priority ceiling emulation protocol (through the RTAI primitive \texttt{rt\_change\_prio}, which is passed the pointer to the current real-time task returned by the RTAI primitive \texttt{rt\_whoami}); mailbox send and receive operations (by means of the RTAI primitives \texttt{rt\_mbx\_send} and \texttt{rt\_mbx\_receive}).

Jittering and sporadic tasks of the specification (e.g., the sporadic task \texttt{Tsk5} in the timeline model of Fig.11) account for asynchronous jobs that must be scheduled on reaction to external stimuli, whose timing is not under control of the task-set but only subject to an expected restriction on the minimum and maximum time that can elapse between subsequent occurrences (class \texttt{Sporadic\_Task} in Fig.12). In the dynamic architecture, they are thus implemented as jobs scheduled on reaction to the arrival of a signal. Synchronization is performed by the way of a semaphore (binary semaphore \texttt{tsk5\_pux} in Fig.13) which is supposed to receive a signal on each release.

Possibly reentrant job releases (which occur whenever the completion time of a job exceeds the time between subsequent releases) are encompassed by the semantics of pTPNs, but they are not supported by the proposed implementation architecture. They could be included by employing a separate task for each job release: the body loop of \texttt{tsk\_}\texttt{x\_}\texttt{job} would spawn a real-time task at each repetition, using a mechanism of multiple buffering to associate each real-time task with a different task handler. This would keep the Execution Time of each loop in \texttt{tsk\_}\texttt{x\_}\texttt{job} short, thus avoiding the case in which the invocation of \texttt{rt\_task\_make\_periodic} comes after the end of the task release period; however, it would break the recommendation of main regulatory standards (e.g., the Ravenscar profile [2]) that rule out dynamic task creation.

Yet, this implementation pattern can be useful in the early implementation stage to detect reentrant job releases, until verification guarantees that all jobs are completed within the end of their task period.

### B. Executable Architecture of a CSCI

The code produced in the implementation of the dynamic architecture of a CSCI concretely realizes the concept of \textit{Executable Architecture} of Unified Process (UP) [57]. To this end, chunk computations attached to the entry-points and job releases of asynchronous tasks can be replaced through calls to busy-sleep and wait functions providing a scaffolding implementation conforming with expected timing requirements: this yields an executable code that implements the architecture before the construction of any of its functional capabilities. As usual in UP development, this provides a baseline for incremental integration and testing of low-level modules. In particular, in SD6.3-SW, this supports separate \textit{Execution Time measurement} and \textit{unit-testing} of low-level modules embedded within their expected operation environment, providing a number of relevant advantages: the scaffolding effort needed to support unit testing of each module is largely reduced; Execution-Time measurement accounts for pipeline and cache damage induced by the execution in preemptive interrupted mode; unit testing is carried out under the general setting of the specification model rather than under a specific integration scenario where each module implements only a subset of behaviors accepted by the specification. In so doing, the effects of changes in individual modules are confined, thus reducing the complexity in the subsequent activity of integration testing.

#### C. Execution Time Profiling

The specification of the task-set allocated to a CSCI includes temporal parameters playing different roles that subdivide different hurdles in the development process: task periods are design choices that can be easily enforced in the implementation; minimum inter-release times and deadlines are derived from high-level requirements and do not have a direct counterpart in the implementation; whereas, chunk Execution Times account for the time spent by entry-points of the specification and thus comprise requirements for low-level SW components, which are not easy at all to predict and control.

In the early stages of development, Execution Times are determined through a tentative approach which mainly relies on analogy with previous realizations of comparable functionalities on comparable platforms. As the development process advances and chunk entry-points become available, actual Best Case Execution Times (BCETs) and Worst Case Execution Times (WCETs) must be evaluated to verify the satisfaction of allocated bounds or to relax/tighten the initial estimates. In general, this can be done either through a static or through a measurement-based method [59]. Static tools (e.g., Heptane [3], aIT [5], Bound-T [53]) provide lower and upper bounds for Execution Times by combining control-flow analysis of code structure to a model of the hardware architecture; measurement-based tools (e.g., RapiTime [24], MTime [34], Synta/P [66], [39]) derive estimates through the conjoint use of analysis of the code architecture and measurements of the Execution Time of sections of code by means of partial/full hardware tracing support or simulation.

In our proposed methodology, precise estimates on Execution Times appear to be more relevant than safe bounds, since inaccurate valuations may jeopardize test-case execution and coverage analysis, and, moreover, verification is in any case supported by the evidence of testing. We thus assume a measurement-based method which is implemented in a simple yet effective manner by developing on the formal basis of pTPNs, providing: estimates for BCETs and WCETs; the distribution of observed Execution Times; and, a measure of coverage of the state-space. In particular, the approach
is independent of the complexity of the code and of the corresponding pTPN model. As a salient trait, unlike RapiTime and MTime tools, our method carries out measurements by letting chunks run in the Executable Architecture of the task-set, thus accounting for possible dataflow dependencies as well as for cache and pipeline damages due to the execution in interrupted mode within a concurrent preemptive context.

The proposed approach is efficiently implemented through automated code instrumentation that produces a time-stamped log for each event corresponding to each transition in the pTPN. By construction, these are: job releases, chunk completions, semaphore wait operations, priority boost operations, and mailbox receive operations. The initial state of an execution is easily identified, either through the implementation of a reset function, or though a technique of state identification [62]; the sequence of time-stamped logs supports reconstruction of the sequence of states visited during the execution, enabling evaluation of the sojourn time in each state; in turn, each state also determines which transitions are newly enabled or persistent and which are progressing or suspended. The Execution Time of a chunk is thus evaluated as the time during which the transition representing the chunk completion has been progressing since it was newly enabled, which is derived off-line as the sum of sojourn times in the visited states where the transition is progressing:

$$ET(\tau_i) = \sum_{k=n}^{K-1} c_{i,k} \cdot (\tau_{k+1} - \tau_k)$$

where: \(\tau_{n}^i\) is the instance of transition \(t_i\) newly enabled in the \(n\)-th state visited by the trace; \(c_{i,k}\) is either 1 or 0 whether \(t_i\) is progressing or suspended in the \(k\)-th state visited by the trace; \(K_{i,n}\) is the index (in the trace) of the state reached through the firing of \(\tau_{n}^i\); \(\tau_k\) is the time of entrance into the \(k\)-th state visited by the trace, i.e., the \(k\)-th time-stamp in the log.

The responsibility of logging time-stamped events is conveniently allocated to the implementation of the dynamic architecture, supporting reuse and avoiding perturbation of chunk entry-points. To this end, the implementation architecture is extended to support the observation of the actual release time of jobs, using a separate real-time task for each job and assigning highest priority to recurrent real-time tasks that release them. Logs are written on an RTAI FIFO queue and transferred on a file by a low-priority task running in the user space, since file operations are not available in the kernel space and would in any case take time beyond the acceptable limits.

D. Case study: SD6.3-SW

As mentioned throughout Sect.IV-A, the application of the proposed programming method to the implementation of the running case study is illustrated in Figs.13 and 14.

The proposed approach to Execution Time profiling was implemented in the Oris Tool [27] and systematically applied to unit-testing of low-level modules in the development of the running case study. Figs.15 and 16 show experimental results obtained on two different implementations of the edge detection module of the BFE CSCI (corresponding to entry-point \(f_{32}\) in Fig.11): the first histogram of observed Execution Times has a wider spectrum with various peaks within \([7.771, 8.620]\) ms, due to a set of input-data dependent alternatives in the implementation of the entry-point; whereas, the second histogram has a narrower spectrum within \([7.279, 7.371]\) ms, thus evidencing a substantial independence from input-data. Both implementations accomplish the time-frame requirement of \([5, 10]\) ms.

E. Assessment of the accuracy of measurements

An experimental assessment was carried out to evaluate the perturbation induced by time-stamped logging and the accuracy of measurements. While our proposed methodology is platform-independent, reported results were obtained on an Intel Core 2 Quad Q6600 desktop processor.

As a preliminary step, we developed a function \texttt{busy\_sleep} that uses the cpu for a controlled Execution Time. Actually, RTAI [15] natively provides the function \texttt{rt\_busy\_sleep}, but this tends to jam the operating system clock when applied for timings over few hundreds of
microseconds and, concretely, it only controls the Completion Time (CT). In fact, \texttt{rt\_busy\_sleep} embeds a while loop that exits when the current time overtakes the time at which the loop was entered plus the specified duration, causing the actual Execution Time to be lower than that duration if a task is suspended while executing the loop.

A \texttt{busy\_sleep} fitting our needs was thus implemented as a simple deterministic loop, with the usual expediens that avoid compiler optimizations. It was experimentally tuned on the specific HW platform, assuming that its actual duration is a linear function with a constant start-up time plus a duration proportional to the number of cycles. Timings were measured using the RTAI primitive \texttt{rt\_get\_time\_ns}, whose overhead was made negligible through the repetition of multiple calls of \texttt{busy\_sleep} between subsequent time measurements. Fig.17 plots the results of the experimentation, showing that a linear approximation effectively fits the relation between the number of cycles of the deterministic loop and its duration.

We then experimented to evaluate accuracy and confidence of logged time-stamps. To this end, we repeatedly applied a constant Execution Time through the function \texttt{busy\_sleep}, logging the completion time of each execution on an RTAI FIFO queue. Experiments were repeated for several values of the nominal Execution Time. Fig.18 plots the histogram of the nominal Execution Time minus the difference between subsequent logged time-stamps for an Execution Time of 25 ms, reporting also the min-max delay measured with respect to the nominal value, its mean value and standard deviation. Note that the error sums up various factors: the (variable) overhead due to the acquisition of a time measure through \texttt{rt\_get\_time\_ns}; the overhead for the log onto the RTAI FIFO queue; the finite accuracy of \texttt{busy\_sleep} in controlling the applied Execution Time.

Observed results show that 99.4\% of cases fall within [214, 1120] ns, with recurrent error peaks within [3174, 3710] ns occurring in 0.6\% of the cases. These peaks turn out to have negligible impact on our application context where the precision of temporal parameters is in the order of various hundreds of µs. In any case, this appears to be related to the RTOS implementation rather than to the structure of our implementation: according to experimental results obtained by the RTAI development team, error peaks are interpreted as timing uncertainties due to processor and bus effects, which are reported to cause jitters on the order of 1 µs to a few tens of µs on general purpose CPUs running a hard RTOS [48], [21].

V. UNIT AND INTEGRATION TESTING

In SD7-SW, conformance of the implementation to design specification is assured through a combined use of static code inspection and dynamic functional testing: static inspection is used to ascertain that inter-process communication (IPC) primitives are not invoked within the code of chunk entry-points; under this assumption, conformance is verified through the exercise of the implementation and the observation of the times of execution of the events represented in the pTPN abstraction of the task-set.

A. Failure and defect model

According to [35], we define failures as deviations of a component or system from its expected delivery, service, or result. In the abstraction of the proposed model, these can be:

- un-sequenced execution: a run breaking sequencing requirements (e.g., a priority inversion);
- time-frame violation: a temporal parameter assuming a value out of its nominal interval (e.g., untimely job release, a computation chunk breaking its expected Execution Time interval);
- deadline miss: a job breaking its end-to-end timing requirement.

Still according to [35], defects are defined as flaws in a component or system that can cause failures in performing the required function. In the proposed programming model, these can consist of:

- task programming defect: flaw in concurrency control and task interactions due to semaphore operation not properly combined with priority handling, wrong priority assignment, flawed usage of any IPC mechanism, un-sequenced invocation of entry-points;
• cycle stealing: detraction of computational resources due to tasks intentionally not represented in the specification as considered not critical for real-time behavior, unexpected overheads of the Executable Architecture or the RTOS.

Defects in the sequential code of entry-points are not considered here as they are more conveniently addressed through consolidated control-flow and data-flow strategies [63], [64] at the level of unit-testing [60].

B. Test-case selection

Certification standards explicitly prescribe structural coverage criteria on the control-flow graph of the code such as all-statements and all-decisions, with modified condition decision coverage (MCDC) [37], and more demanding criteria can be applied to improve coverage beyond the contractual certification needs [63]. While effectively exerting control structures and data-flow dependencies, these criteria provide a limited coverage of the variety of behaviors that may result from the concurrent and interrupted execution of the dynamic architecture of a task-set. The SCG provides an effective abstraction to account for this variety, that can be interpreted in different perspectives: if the code generator has been previously qualified as correct, then the SCG reflects the actual implementation of the dynamic architecture of the task-set and can be used to evaluate a structural measure of coverage and to drive architectural testing of entry-points and asynchronous task releases; whereas, if the code generator is not qualified or if the code is manually written, then the SCG provides a functional abstraction for test-case selection in the verification of conformance of the implementation [33], [30] composed by the dynamic architecture along with entry-points and asynchronous task releases. In both perspective, test-cases can be selected as symbolic runs satisfying a specific test purpose determined through a model checking technique [4] or as part of a test suite defined through various criteria to automate test-case selection and/or coverage analysis [8].

We illustrate here the application of few relevant criteria to the SCG abstraction, highlighting the trade-off between the subtended complexity and the attained type of coverage.

1) All-Markings: All-Markings is satisfied when each reachable marking has been visited by at least one test-case. This covers all the acceptable states of concurrency, i.e., all the possible combinations of chunks that are concurrently ready/runningBlocked. The criterion is a kind of all-nodes with complexity proportional to the number of reachable markings. It is efficiently resolved on the SCG of the pTPN model: for each reachable marking \( m \), select any class \( S_m \) with marking \( m \) and include in the test-suite any path from a controllable starting point to \( S \).

2) All-Marking-Edges: All-Marking-Edges is satisfied when each edge between any two reachable markings has been traversed by at least one test-case. This includes All-Markings and guarantees to observe the transitions between subsequent states of concurrency (e.g., the events of preemption following to asynchronous releases, chunk completions, semaphore and mailbox operations, and priority boost/deboost). The criterion is a kind of all-edges with complexity proportional to the number of reachable markings multiplied by the maximum number of events occurring within a marking, which in turn is limited by the number of tasks in the set. As for All-Markings, selection is performed on the SCG: for each edge between any two markings \( m_1 \) and \( m_2 \), select any class \( S_1 \) with marking \( m_1 \) that accepts an event leading to a class \( S_2 \) with marking \( m_2 \), and include in the test-suite any path starting from a controllable starting point and covering the edge.

3) All-Classes and All-Class-Edges: All-Markings and All-Marking-Edges can be refined into All-Classes and All-Class-Edges, requiring coverage of all reachable state-classes and all edges between reachable state-classes, respectively, and adding the capability to partially account for the difference among paths whenever this results in different timing restrictions. However, they also largely increase the actual complexity of coverage (e.g., in the state-space of the task-set of Fig.5, 433 markings are refined into 51079 classes).

4) All-Symbolic-Runs: All-Symbolic-Runs covers every path in the SCG that starts with a job release and terminates with its completion or deadline miss. This includes All-Class-Edges and thus also All-Marking-Edges, adding the capability to cover all the possible concurrency states and transitions within the context of the same job execution, which could be relevant to account for possible implicit data-flow and timeliness dependencies among chunks of the same job.

If deadlines of synchronous and asynchronous jobs are not higher than task periods and minimum inter-release times, respectively, task releases and completions are directly represented by specific transitions, while deadline misses are identified with tasks releases occurring when a task-job is still pending. The test-suite thus includes every path that starts from any task release and terminates either with the task completion or with a new release (which indicates a deadline miss). If a deadline is anticipated with respect to the period or minimum inter-release time, deadline misses are not directly represented on the SCG, but they can be made observable by extending the pTPN model with watch transitions as proposed in [11].

The resulting number of test-cases is guaranteed to be finite under the reasonable assumption that all tasks have a finite deadline and that all tasks require a non-null minimum Execution Time. In fact, if the number of paths would be infinite, there should be a marking \( m \) that can be visited an infinite number of times between a release and the completion/deadline of some task \( T \). Since the time between release and completion/deadline is bounded, the Infimum of the time spent between two subsequent visits of \( m \) should be equal to 0. However, every sequence between two subsequent visits of \( m \) must include an event that advances the state of a task \( T \) and thus changes the marking in a place \( p_T \) belonging to \( T' \); to bring back the marking of \( p_T \) to its value in \( m \), \( T' \) must complete its execution, restart a new release and execute all the steps prior to \( m \), which requires a time not lower than the minimum Execution Time of the task which is supposed to be strictly higher than 0.

5) All-Symbolic-Executions: All-Symbolic-Executions covers any feasible events sequence that starts with the release of a job and terminates with its completion or deadline-miss. This
C. Oracle verdict

Time-stamped logs can be analyzed by different oracles evaluating conformance of the implementation to the specification with different levels of abstraction, each implying a different run-time overhead for logging and a different off-line complexity for the decision.

1) End-to-end Oracle: The oracle verifies that end-to-end task deadlines are met. This requires time-stamped logging of job releases and completions, and it is implemented in a straightforward manner by comparing the time elapsed between release and completion of any job against the deadline of its task (without explicit reference to the structure of the pTPN).

2) Sequencing Oracle: The oracle verifies that the qualitative ordering of events conforms with design specification (i.e., that there exists at least one timing that makes the logged sequence feasible for the specification). This requires logging of all the events corresponding to a transition in the pTPN model (i.e., job releases; chunk completions; semaphore wait, priority boost, and mailbox receive operations), avoiding the invocation of a time-measuring primitive of the RTOS (e.g., rt_get_time_ns on RTAI [15]) to associate each logged event with a time-stamp.

The decision algorithm consists in verifying whether the logged sequence is a symbolic execution sequence in the SCG of the pTPN model (i.e., whether the SCG contains a path that follows the sequence of logged events). If the SCG is not finite, or if it does not fit in the available memory, then conformance can be tested by reconstructing the fragment of the state-space composed by the classes visited by the logged sequence. To this end, we assume that the logged sequence begins with the start-up of the task-set. This identifies the initial marking, guarantees that all enabled transitions are newly enabled, and thus identifies a reachable state-class that contains the initial state of the logged sequence. Since a state-class is derived in time $O(N^2)$ with respect to the number $N$ of enabled transitions [18], [26], the entire fragment is computed in time $O(N^2 \cdot L)$, where $L$ is the length of the logged sequence.

3) Timing Oracle: The oracle verifies the timed trace inclusion relation of [4] (i.e., it verifies that the logged sequence is a feasible execution for the specification), requiring a time-stamped log for each event corresponding to a transition in the pTPN model. To guarantee identification of the initial state, we assume that the logged sequence starts from a state with a known marking where all enabled transitions are newly enabled. This can be the initial state reached through a reset function, or any intermediate controllable state reached through an ad-hoc function where no computation is ongoing and all times to the next job releases are known.

The decision algorithm does not require state-space analysis and relies on a simulation of the specification model: starting from an initial state $s_0 = \langle M_0, F_{\psi_0} \rangle$, the algorithm checks the feasibility of the first time-stamped event $⟨ t_1, τ_1⟩$ by verifying whether $t_1$ can be fired at time $τ_1$ from state $s_0$ and computes the subsequent state $s_1$; at the $n$-th step, the algorithm checks whether $t_n$ can be fired at time $τ_n − τ_{n−1}$ from state $s_{n−1}$ and computes the resulting state $s_n$. The oracle emits a failure verdict as soon as any time-stamped event $⟨ t_n, τ_n⟩$ is not accepted by the simulator; a pass verdict when the run terminates; an inconclusive verdict when the trace diverges from the test-case sequence with an event that is accepted by the simulator. In so doing, the decision algorithm verifies off-line whether the sequence of time-stamped events produced by an implementation is a subset of the dynamic behavior that the semantics of the specification model may accept, thus distinguishing from the technique reported in [4], where an automaton is employed online during the testing process to collect auxiliary information for coverage evaluation, and also from the approach proposed in [9], where an observer is used to evaluate quantitative properties through state-space enumeration of the specification model augmented with additional places and transitions.

4) Failure detection: All the failures detected by the Sequencing Oracle are also detected by the Timing Oracle, while the vice-versa is of course not true: any un-sequenced execution is detected by the Sequencing Oracle (and thus by the Timing Oracle); any time-frame violation is detected by the Timing Oracle, but not by the Sequencing Oracle unless timing causes the execution to diverge from the expected sequencing. Failure detection capability of the End-to-end Oracle is not comparable with that of the Sequencing Oracle and the Timing Oracle, since an execution may break the expected sequencing and/or an internal timeframe and satisfy its deadline, and vice-versa. However, if state-space analysis of the specification model has verified the absence of symbolic runs breaking any task deadline, then all the failures observed by the End-to-end Oracle are also observed by the Sequencing Oracle (and thus by the Timing Oracle).

Coverage metrics can be obtained by mapping on the SCG the sequence of events reproduced by the Timing Oracle or by the Sequencing Oracle, providing a measure of confidence in the absence of residual defects regardless of the number of identified failures. The sequence of events produced by the End-to-end Oracle cannot be mapped on the SCG because the oracle does not observe all actions.

D. Test-case execution

Test-case execution determines timed inputs that force the Implementation Under Test (IUT) to execute selected test-cases. In [4], [42], test-cases are deterministically timed event sequences, executed under the assumption that all events can be observed and that all inputs can be controlled so as to assume a deterministic value and time. However, periodic and asynchronous release times can be effectively controlled through conventional RTOS primitives; whereas, computation times are often impractical to handle. Moreover, timers may take values within a subset of their nominal range of variation due to the determinization of the implementation with...
respective to the specification model to be unfeasible in the IUT [1] due to various practical factors: i) assumptions taken in the specification model cannot exactly correspond to the actual implementation (e.g., synchronous releases may suffer of some jitter, semaphore and priority handling operations require a non-null Execution Time); ii) temporal parameters of the specification are usually associated with a nondeterministic range of variation, to make the model robust with respect to possible variations of the IUT; iii) in a re-engineering process, temporal parameters are associated with intervals which overapproximate those of the actual implementation, to circumvent the difficulty in obtaining precise estimates of Execution Times and release times; iv) specification usually neglects dependencies among Execution Times of computation chunks, which are difficult to quantify and definitely unreliable as an assumption for schedulability; v) the RTOS may contribute to the partial determinization of the implementation (e.g., RTAI [15] turns out to assign a fixed order to the releases of periodic tasks with the same priority at times multiple of their hyperperiod, insensitively to the order in which tasks are initially started).

Assumptions on deterministic behavior and observability of the IUT are relaxed in [52], which introduces a timed version of the input output conformance relation of [40] and proposes a framework for black-box conformance testing of real-time systems specified as nondeterministic and partially-observable timed automata. 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 [42], the timed input output conformance is relativized to explicitly take environment assumptions into account and an on-the-fly testing algorithm is defined to support online test-case generation, test-case execution, and conformance verdict assignment. The approach is based on randomized testing and does not provide any coverage evaluation on the variety of timed behaviors. As a relevant trait, the specification automaton adopted in [52], [42] does not encompass preemptive behavior.

The formal approach to test-case execution proposed in [47] nicely exploits state-space enumeration and trace analysis of a pTPN model of the specification to identify timing restrictions that can attain path execution. As a characterizing trait, the technique distinguishes between controllable and non-controllable timers, identifying state-classes that can be selected as starting points for the IUT. The approach was integrated in the MDD framework presented in this paper and systematically applied to the development of the running case study.

E. Case study: SD7-SW

We report here experimental results on the application of the test-case execution approach of [47] to testing of the implementation of the BFE CSCI, comparing randomized and guided testing in the execution of a symbolic run $\rho$ of $Tsk_4$ that may attain its WCCT of 58 ms.

In randomized testing, the IUT was run for 1 hour (which corresponds to 90000 releases of the shortest-period task $Tsk_1$) releasing the sporadic task $Tsk_5$ according to a uniform distribution within $[120, N_{max}/10^9]$ ms, being $N_{max}$ the maximum representable integer value. Coverage evaluation evidenced that the test-case $\rho$ was never covered. Since $\rho$ comprises the execution of a job of $Tsk_5$, a second experiment was performed by activating jobs of $Tsk_5$ according to a uniform distribution within $[120, 1000]$ ms. However, also in this case, $\rho$ was never covered.

In guided testing, the IUT was repeatedly started from a controllable class, releasing $Tsk_5$ within $[0, 10]$ ms and starting $Tsk_1$, $Tsk_2$, $Tsk_3$, and $Tsk_4$ at the same time within $[4, 11]$ ms. One thousand executions were performed, which correspond to an overall Execution Time of a little more than 1 minute, and $\rho$ was executed 207 times, evidencing the effectiveness of the approach with respect to randomized testing.

VI. CONCLUSIONS

In this paper, we drew a comprehensive methodology that integrates the theory of pTPNs into a tailoring of the V-Model SW life cycle [12], engineering the steps of design, implementation and testing, in compliance with our industrial experience and with the principles of process standards for safety-critical SW [60], [13], [56]. To effectively bring the theory to application, the overall toolchain is implemented in the Oris Tool [27] according to an MDD approach, supporting automated derivation of pTPN models from a semi-formal specification, automated compilation of models into real-time code running on RTAI [15], measurement-based Execution Time evaluation, test-case selection and execution, and coverage evaluation.

As a salient trait, the proposed methodology has a smooth impact on the development life cycle, as explicitly recommended by the RTCA/DO-178B standard [60]. In fact, the main difficulty that the approach brings along is the need of a specification of the dynamic architecture through pTPNs. This is largely eased by the use of the semi-formal specification of timelines and by their automated translation into pTPN models and real-time code. Without any additional effort on the part of the developer, the pTPN model provides a formal abstraction that supports verification of the dynamic architecture, Execution Time profiling of entry-points, unit and integration testing.

Explosion of the timed state-space of the pTPN model may impair exhaustive verification of SW design, but it does not prevent the overall development approach: on the one hand, partial enumeration of the state-space still enables the verification of a significant part of possible behaviors, reaching the level of rigor that can be attained through simulation; on the other hand, conformance of the implementation with respect to the specification is in any case achieved through testing requested for certification purposes.

The code of the implementation follows in straightforward manner the structure of the pTPN model. Since each component of the pTPN specification accepts a context-free translation, the code can be generated automatically by structural induction, thus providing a full fledged Model Driven
Development method. At the same time, and perhaps more importantly, the implementation code has a readable structure, which follows common patterns of concurrent programming, leaving the developer full insight and control over the final code and even allowing manual programming guided by the model structure. This is a crucial point to support industrial acceptance and avoids legacy constraints on tools that support editing and translation of timeline schemas and pTPN models.

A clean and readable code structure also prevents erroneous understanding of the semantics of pTPNs and timeline schemas, which in industrial practice may be less understood and trusted than conventional programming languages and operating system primitives.

ACKNOWLEDGEMENTS

This work was supported by the SW-Initiative of the FinMecanica Group. We kindly thank Paolo Villiani and Simonetta Orsi of Galileo Avionica for their help in bridging the gap between theory and practice. We also thank Giacomo Bucci for advise and Irene Bicchierai and Dario D’Amico for help in the experimentation stage.

REFERENCES
