A tailored V-Model exploiting the theory of preemptive Time Petri Nets

Laura Carnevali, Leonardo Grassi, Enrico Vicario

Dipartimento di Sistemi e Informatica - Università di Firenze
`{carnevali,grassi,vicario}@dsi.unifi.it`
`http://www.dsi.unifi.it`

Abstract. We describe a methodology that embeds the theory of preemptive Time Petri Nets (pTPN) along development and verification activities of a V-Model lifecycle to support the construction of concurrent real time SW components. Design activities leverage on a pTPN specification of the set of concurrent timed tasks. This supports design validation through simulation and state space analysis, and drives disciplined coding based on conventional primitives of a real-time operating system. In verification activities, the pTPN model comprises an Oracle for unit and integration testing and its symbolic state space supports test case selection, test sensitization and coverage evaluation.

Keywords: *concurrent real-time systems, V-Model, preemptive Time Petri Nets, formal methods, state space analysis.*

1 Introduction

Intertwined effects of concurrency and timing comprise one of the most challenging factors of complexity in the development of safety critical SW components. Formal methods may provide a crucial help in facing this complexity, supporting both design and verification activities, reducing the effort of development, and providing a higher degree of confidence in the correctness of products.

Integration of formal methods in the industrial practice is explicitly encouraged in certification standards such as RTCA/DO-178B [1], with specific reference to software with complex behavior deriving from concurrency, synchronization, and distributed processing, under the recommendation that proposed methods are smoothly integrated with design and testing activities prescribed by a defined and documented SW lifecyle. This recommendation can be effectively referred to the framework of the V-Model [22], which is often adopted by process oriented standards ruling the development of safety critical software subject to explicit certification requirements, such as airborne systems [1], railway and transport applications [17], medical control devices [21].

In this paper, we describe a tailoring of the V-Model life cycle that leverages on the theory of preemptive Time Petri Nets (pTPN) [5] to support design, coding and testing of complex concurrent and real time task sets. The proposed tailoring spans over the activities of Preliminary and Detailed SW Design, SW Implementation, and SW Integration. During SW design, the architecture of the task set is specified using the intuitive formalism of timelines, which can be automatically translated into a pTPN model. In turn, this supports validation of design with respect to sequencing and timeliness requirements through timed simulation and/or timed state space analysis. During SW implementation, the pTPN specification of the task set is implemented through a disciplined coding approach relying on conventional primitives of a real time operating system. The implementation produces a so-called operational architecture which supports incremental integration and testing of low-level SW components. During SW Integration, the pTPN specification comprises an Oracle for integration testing, while symbolic state space analysis supports test-case-selection, test-sensitization and coverage-evaluation.

The rest of the paper is organized in six sections. The specification of the real-time task set architecture and its validation are discussed in Sect.2, while Sect.3 describes how to implement the specification model on top of LinuX RTAI APIs. Sect.4 illustrates how pTPNs support testing activities, both in test case selection/sensitization and in the evaluation of executed tests. Sect.5 organizes all the activities to outline a tailoring of the V-Model SW life cycle. Conclusions are drawn in Sect.6.

2 Preemptive Time Petri Nets in the specification and architectural validation of real-time task sets using the Oris Tool

In this section, we introduce preemptive Time Petri Nets (pTPN) [8][5], showing how they support the modeling of real-time task sets and how simulation and analysis of the specification model are employed in the architectural validation of the task set itself, supporting tight schedulability analysis and verification of the correctness of logical sequencing.

2.1 Specification of real-time task sets through timelines

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 [9].

The task set is comprised by tasks. Tasks release jobs in recurrent manner with three different possible release policies: i) *periodic*, in which tasks have a deterministic release time; ii) *sporadic*, in which tasks have a minimum but not a maximum release time; and iii) *jittering*, in which tasks have a release time constrained between a minimum and a maximum. Task deadline is usually coincident with its minimum release period.

Jobs can be internally structured as a sequence of chunks, each characterized by a nondeterministic execution time constrained within a minimum and a maximum value. Chunks may require preemptable resources, notably one or more processors. In this case, they are associated with a priority level and run under static priority preemptive scheduling. Chunks may be synchronized through binary semaphores, to guarantee mutual exclusion in the use of shared resources: a chunk acquires a semaphore before starting its execution and releases it at the end of its execution.

Task sets can be conveniently specified through timelines, which represent a temporal scheme annotated with parameters of tasks and chunks (release period, deadline, resource request, priority). Fig.4 reports an example with 4 tasks. T_1 is a periodic task synchronized by semaphore m_1 to the sporadic task T_4; tasks T_2 and T_3 are periodic and synchronized through semaphore m_2.

2.2 Preemptive Time Petri Nets

A timeline schema can be easily translated into an equivalent pTPN. Preemptive Time Petri Nets [8][5] extend Time Petri Nets (TPN) [10][11] with an additional mechanism of resource assignment, making the progress of timed transitions be dependent on the availability of a set of preemptable resources. Syntax and semantics are formally expounded in [5], and we report here only an informal description. As in TPN, each transition is associated with a static firing interval made up of an earliest and a latest static firing time, and each enabled transition is associated with a clock evaluating the time elapsed since it was newly enabled: a transition cannot fire before its clock has reached the static earliest firing time, neither it can let time pass without firing when its clock has reached the static latest firing time. In addition, each transition may request a set of preemptable resources, each request being associated with a priority level: an enabled transition is progressing and advances its clock if no other enabled transition requires any of its resources with a higher priority level; otherwise, it is suspended and maintains the value of its clock. This supports representation of the suspension mechanism and thus of preemptive behavior, attaining an expressivity that compares to that of stopwatch automata [7][6][5].

Translating a timeline schema into a pTPN model: The Oris Tool supports both the editing of a timeline schema and its automatic translation into an equivalent preemptive Time Petri Net. Repetitive job releases performed by a task are modeled as an always-enabled transition with static firing interval equal to the task release range; chunks are modeled as transitions with static firing intervals corresponding to their min-max range of execution time and with the same resource requests and priority. A binary semaphore is modeled by a place containing one token, which represents the permission to acquire the semaphore itself.

Priority inversion frequently occurs in practical systems and limiting its adverse effects is extremely important in a system where any kind of predictable response is required. For instance, priority inversion can occur when a high priority chunk requires exclusive access on a resource that is being currently accessed by a low priority chunk: if one or more medium priority chunk then run while the resource is locked by the low priority chunk, the high priority chunk can be delayed indefinitely. To avoid priority inversion, we extend pTPN formalism to assume priority ceiling emulation protocol, 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).

Fig.5 reports the pTPN modeling the timeline schema of Fig.4.

Timeliness expressivity: In principle, the specification model could be expressed directly using pTPNs. However the usage of timelines, that represent an intuitive formalism, augments modeling convenience and facilitates industrial acceptance and interoperability. In addition, timelines provide a structural restriction on the expressivity of pTPNs which gives meaning to some relevant concepts in the theory of real-time systems such as task, job, chunk, hyperperiod and idle state. As a drawback, this restriction prevents an explicit representation of priority ceiling emulation protocol within timeline formalism.

2.3 Architectural validation through simulation or state space enumeration of the pTPN model

The pTPN specification model can be simulated or analyzed to perform architectural validation of the real-time task set.

The state of a pTPN can be represented as a pair $s = \langle M, \tau \rangle$, where M is a marking and τ is a vector of times to fire for enabled transitions. Since τ takes values in a dense domain, the state space of a pTPN is covered using *state classes*, each comprised of a pair $S = \langle M, D \rangle$, where M is a marking and D is a firing domain encoded as the space of solutions for the set of constraints limiting the times to fire of enabled transitions. A reachability relation is established among classes: a state class S' is reachable from class S through transition t_0 , and we write $S \xrightarrow{t_0} S'$, if and only if S' contains all and only the states that are reachable from some state collected in S through some feasible firing of t_0. This reachability relation, sometimes called AE relation [12], defines a graph of reachability among classes that we call *state class graph* (SCG).

The AE reachability relation turns out to collect together the states that are reached through the same firing sequence but with different times [11][13][14]. A path in the SCG thus assumes the meaning of *symbolic run*, representing the dense variety of runs that fire a given set of transitions in a given qualitative order with a dense variety of timings between subsequent firings. A symbolic run is then identified by a sequence of transitions starting from a state class in the SCG, and it is associated to a completion interval, calculated over the set of completion times of the dense variety of runs it represents. Note that the same sequence of firings may be firable from different starting classes. According to this, we call *symbolic execution sequence* the finite set of symbolic runs with the same sequence of firing but with different starting classes.

If the model does not include preemptive behavior, i.e. if it can be represented as a TPN, firing domains can be encoded as Difference Bound Matrixes (DBM), which enable efficient derivation and encoding of successor classes in time $O(N^2)$

with respect to the number of enabled transitions N. Moreover, the set of timings for the transitions fired along a symbolic run can also be encoded as a DBM, thus providing an effective and compact profile for the range of timings that let the model run along a given firing sequence [14].

When the model includes preemptive behavior, then derivation of the successor class breaks the structure of DBM, and takes the form of a linear convex polyhedron. This results in exponential complexity for the derivation of classes and, more importantly, for their encoding [8][5][6][15]. To avoid the complexity, [5] replaces classes with their tightest enclosing DBM, thus yielding to an over-approximation of the SCG. For any selected path in the over-approximated SCG, the exact set of constraints limiting the set of feasible timings can be recovered, thus supporting clean-up of false behaviors and derivation of exact tightening durational bounds along selected critical runs. In particular, the algorithm provides a tight bound on the maximum time that can be spent along the symbolic run and provides an encoding of the linear convex polyhedron enclosing all and only the timings that let the model execute along a symbolic run.

The Oris tool [16] supports enumeration of the SCG, selection of symbolic runs attaining specific sequencing and timing conditions and tightening of their range of timings. The example of Fig.5 has a symbolic state space comprised by 29141 state classes, having 134 different markings. For each task, the analysis of the SCG allows the identification of the paths starting with the release of a job and ending with its completion, which we call *task symbolic runs*, and of the corresponding execution sequences, which we call *task execution sequences*. Specifically, tasks T_1, T_2, T_3 and T_4 have 6915, 10816, 22093 and 13837 symbolic runs and 244, 951, 2823 and 1935 symbolic execution sequences, respectively. The analysis provides the worst case completion time for each task (70, 100, 170 and 140 time units for T_1, T_2, T_3 and T_4, respectively), thus verifying that deadlines are met and with which minimum laxity (80, 80, 70 and 220 time units for T_1, T_2, T_3 and T_4, respectively).

3 Coding Process

The pTPN specification model enables a disciplined coding of the task set architecture on top of conventional primitives of a real-time operating system. The procedure is described with reference to the APIs of Linux RTAI, a patch for the Linux kernel which introduces a hardware abstraction layer and an application interface supporting the development of real-time applications for several processor architectures.

The task set is implemented as a kernel module, with functions *init_module*() and *cleanup_module*() as entry points for loading and unloading. Tasks are created in *init_module*() through *rt_task_init*() and they are started by calling *rt_task_make_periodic*() or *rt_task_resume*() depending on they are recurrent or one-shot tasks, respectively; they are destroyed in *cleanup_module*() by invoking *rt_task_delete*(). Chunks are implemented as C functions, invoked by their respective tasks. Semaphore operations must be appropriately combined with priority handling, to guarantee proper implementation of the specification model. RTAI provides resource semaphores, which implement priority inheritance, and binary semaphores, which instead leave 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 [8]. More specifically, when a low priority task acquires a semaphore, priority boost requested for priority ceiling emulation must precede semaphore wait operation; viceversa, at release, priority must be restored to the previous level after semaphore signal operation. Data structures of the application, such as semaphores and real-time FIFO queues, are created and destroyed in *init_module*() and *cleanup_module*(), respectively.

In our approach, a disciplined manual translation has been preferred to non-supervised, model-based code generation. By leaving the programmer the responsibility of the coding process, we ensure human control over the implementation and we preserve the readability and maintainability of the output source code. Disciplined coding enables to fully exploit the flexibility of programming languages in the realization of real time design patterns; besides, the axiomatic semantics of programming languages and IPC primitives ensure construction (i.e. types) and procedural consistency and are still to be retained as better specified than the semantics and notation of formal modeling languages.

However, we believe that automatic code generation is achievable without considerable efforts on the part of the developers and without critically impacting the proposed approach. As a proof, let's consider the structural decomposition of a pTPN specification into its semantic components, i.e. tasks, jobs chunks and synchronization structures. In our approach, each model component has a context-free translation into a corresponding code element, i.e. a C function or a OS IPC primitive; according to this partitioning, the entire specification model can be modularly implemented by composing code structures inductively derived from individual pTPN elements.

This seems to greatly reduces the complexities related to the generation of code and to the verification of its correctness thus allowing to seamlessly integrate automated model-to-code translation within the development process.

4 Supporting the testing process through preemptive Time Petri Nets and the Oris Tool

We address the testing phase and, in order to detect failures in the implementation, we show how the pTPN specification model can be employed in the evaluation of logs produced during testing and in test case selection and execution.

Fault model and failure detection: We consider failures deriving from types of fault that do not guarantee a proper implementation of the specification model with respect to the sequencing and the timing of individual actions (i.e. job releases, chunk completions, signal and wait operations): *i) time frame violation fault*, i.e. a fault in the chunk implementation leading a an action to assume values out of its nominal interval; *ii) cycle stealing fault*, i.e. the presence of additional tasks which steal computational resources (these can be unexpected tasks, services provided by the operating system, or tasks intentionally not represented in the specification because considered not critical for the real-time application to be realized); *iii) faults in concurrency control and task interactions*, i.e. a wrong priority assignment, a semaphore operation which is not appropriately combined with priority handling, or in general, a wrong implementation of any IPC mechanism.

We assume that the implementation is instrumented so as to provide a time-stamped log of actions represented in the specification model [20]. Therefore, each run provides a finite sequence of timed actions $tr = \{\langle a_n, \tau_n \rangle\}_{n=1}^{N}$, where a_n is an action corresponding to a unique transition t_n in the pTPN model and τ_n represents the time at which a_n has occurred. According to this, the operational semantics of the pTPN model can be exploited as a *time-sensitive Oracle* in order to evaluate an execution run. The Oracle off-line simulates the execution of the sequence of timed actions and emits a failure verdict as soon as any timed action is not accepted by the simulator; a pass verdict is emitted when the run is finished.

It can be easily verified that any time frame violation fault, as well as any fault in concurrency control and task interaction, is detected as a failure by the time-sensitive Oracle, either because a transition is not firable or because it is firable but not with the observed timing. Viceversa, a cycle stealing fault is recognized provided that its duration exceeds the laxity between an actual computation and its expected upper bound.

The time-sensitive Oracle somehow performs the function of the observers proposed in [2] [15]. In [2], an observer is an automaton employed online during the testing process to collect auxiliary information that is used for coverage evaluation. In [15], 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 the concepts of observer, our Oracle evaluates off-line 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.

Test case selection and execution: While the analysis of the specification model supports early validation of the process architecture, confidence in the conformance of the implementation to the specification can be achieved through testing. In this step, which is in any case requested for certification purposes [1][17], the state space of the specification model can be exploited to select test cases and to identify timed inputs that let the system run along selected cases [2][3][4][18]. Both steps face the existence of behaviors that are legal in the specification model but cannot be observed in a real implementation. In fact, when the specification of a software component is developed, various temporal parameters are necessarily associated with a nondeterministic range of variation, not only to accommodate changes in the embedding context and allow a margin of laxity for the implementation, but also to support a re-engineering process or reuse of a component within a modular architecture. Also, specification models usually neglect dependencies among temporal parameters, both to keep the specification model relatively simple and to avoid the difficulty in quantifying these dependencies. Besides, when software requires high Integrity Levels, the implementation must be deterministic.

According to this, coverage criteria cannot rely on the selection of deterministic test cases, as many of these could be not feasible. We thus propose that test cases be specified as symbolic runs, as they represent the dense variety of runs that follow the same sequence of actions with a dense variety of timings. A test case is considered covered when any of its runs has been executed. As proposed in [2], a symbolic run can be selected as the witness of a specific test purpose determined trough a model checking technique, or it can be part of a test suite identified through a coverage criterion defined on the state class graph (i.e. all nodes, all edges, all paths). Regardless of the number of identified failures, a metric of coverage is needed to provide a measure of confidence in the absence of residual faults and it can be derived by mapping on the state class graph the sequence of actions reproduced by the time-sensitive Oracle.

A procedure to sensitize a selected test case has been proposed in [19]. It is based on the observation that not all temporal parameters are controllable: in fact, periodic and asynchronous release times can be effectively controlled through conventional primitives of a real-time operating system, whereas controlling computation times is often impractical. This gives a major relevance to state classes of the specification where no computational chunk is pending and all jobs are waiting for their next release, that we call *idle classes*. Given a test case ρ with initial class S_{target}, the procedure identifies the temporal constraints representing the necessary condition to first reach S_{target} starting from an idle class S_{idle} and then execute ρ. Therefore, the IUT is started from any state within S_{idle} and controllable actions are forced to occur within the identified constraints.

5 Using preemptive Time Petri Nets within the software life cycle V-Model

In this section we show how the theory of preemptive Time Petri Nets can be smoothly integrated as a formal method in the V-Model of the software life cycle, also with reference to the RTCA/DO-178B standard, which provides guidelines for the production of software for avionic systems. In particular, we illustrate how the effort at modeling a real-time task set through pTPNs provides relevant advantages in the subsequent stages of software development: in fact, as also evidenced in the previous sections, analysis and simulation of pTPN models support both design and verification activities.

The V-Model of the German Federal Administration regulates the processes of system development, maintenance and modification in the software life cycle. The standard describes the development process from a functional point of view, defining a set of activities and products (results) that have to be produced. Since it has general validity and it is publicly available, it has been adopted by many companies and *tailored* to a variety of specific application contexts. Fig.1 reports

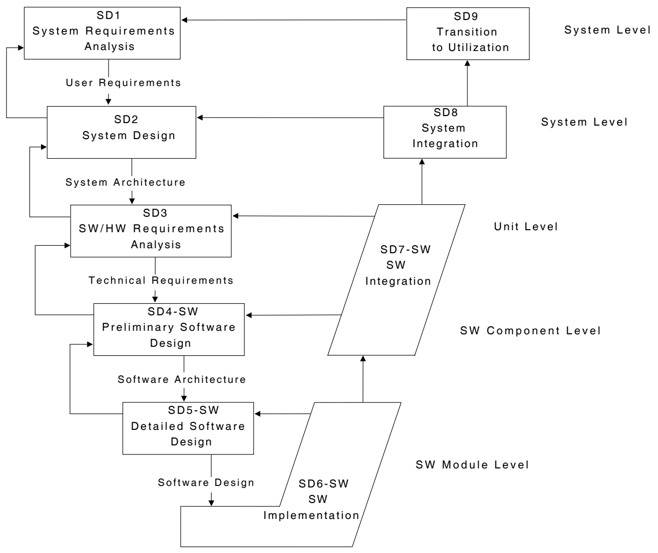


Fig. 1. Overview of Activities of Submodel System Development of the V-Model

a graphical representation of the activities pertaining the System Development (SD) submodel, emphasizing the integration between design and verification activities (left/right) and the hierarchical decomposition from System to Module Levels (top/down). Even if the order of activities appears sequential, iterations are very common during the development process.

With reference to a case example, we illustrate how pTPNs can be casted in the development life cycle of real-time software, focusing on those activities of the V-Model which the adoption of pTPNs as a formal method mainly supports.

5.1 Casting pTPNs within the V-Model of the software life cycle

As a case example, we consider the activities pertaining the development of an avionic radar system. Fig.2 evidences the first three design activities. *System Requirements Analysis* (SD1) defines User Requirements, specifying both functional requirements and non-functional requirements, such as transmission radius, power and frequencies. *System Design* (SD2) identifies main system units (a receiving/transmission antenna unit, a receiver unit, a converter unit, a signal/data processing unit) and allocates User Requirements to each of them. *Software/Hardware Requirements Analysis* (SD3) examines both software and hardware resources of each unit, decomposing them into software and hardware components which, according to the notation of Software Configuration Management, are referred to as Computer Software Configuration Items (CSCIs) and Hardware Configuration Items (HCIs). Referring to the example, requirements pertaining the signal/data processing unit are allocated to three separate CSCIs: a raw-image elaborator, a tracker and a central processor. It is worth noting that SD1 and SD2 pertain the entire system under development, while subsequent design activities are repeated for each unit.

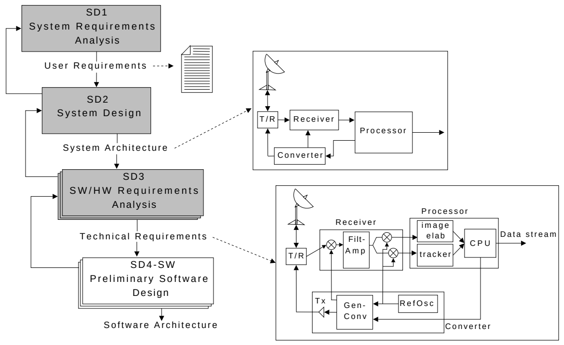


Fig. 2. The first three design activities in the development of an avionic radar system.

Preliminary Software Design (SD4-SW, see Fig.3) defines the *Software Architecture* of each CSCI, allocating it to a task set defined in terms of communicating tasks with assigned functional modules and prescribed release times and deadlines. *Detailed Software Design* (SD5-SW, see Fig.3) allocates resources and time requirements to software modules and produces the *Software Design* of each CSCI; in particular, the sub-activity of *Analysis of Resources and Time Requirements* (SD5.2-SW, not shown in Fig.1) addresses the evaluation of architecture feasibility. pTPNs allow the description of a shared resource environment with concurrent tasks subject to temporal constraints and running under priority preemptive scheduling, thus supporting both design activities. More specifically, Software Architecture of a CSCI can be modeled through a pTPN where timing requirements on computational chunks are left unspecified. This pTPN model can then be refined through the definition of low-level requirements, by associating each computational chunk with a minimum and a maximum execution time. It is worth noting that constraints on computation times can be assigned conservatively through estimations based on emulators or by attained experience with reused components and prototypes. However, they can also be assigned without

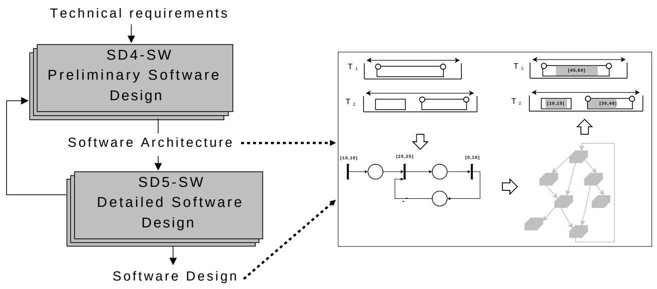

Technical requirements
SD4-SW Preliminary Software Design
Software Architecture
SD5-SW Detailed Software Design
Software Design

Fig. 3. Preliminary and Detailed Design activities in the development of an avionic radar system.

any measurement on code but only according to a resource allocation policy. Hence, pTPNs as a formal method can be smoothly integrated within design activities; in addition, modeling convenience can be enhanced by considering the equivalent timeline schema.

Referring to the example, Fig.4 and Fig.5 show the timeline and the pTPN model, respectively, pertaining the Software Design of the tracker CSCI. Simu-

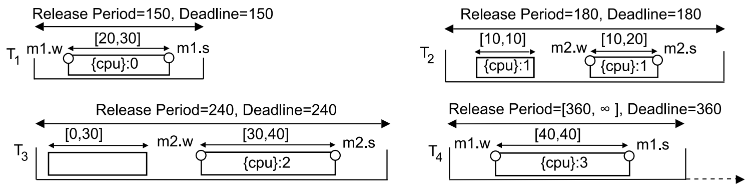

Release Period=150, Deadline=150
[20,30]
T1
m1.w
m1.s
{cpu}:0
Release Period=180, Deadline=180
[10,10]
[10,20]
T2
m2.w
m2.s
{cpu}:1
{cpu}:1
Release Period=240, Deadline=240
[0,30]
[30,40]
T3
m2.w
m2.s
{cpu}:2
Release Period=[360, ∞], Deadline=360
[40,40]
T4
m1.w
m1.s
{cpu}:3

Fig. 4. The timeline schema of the Tracker CSCI, composed of four tasks synchronized by two semaphores. T_1 is a periodic task (period 150 ms) for the retrieval of radar data from Raw Image Processor CSCI. T_2 is a periodic task (period 180 ms) for the transmission of track data to the CPU CSCI. T_3 is a periodic task (period 240) representing plot extraction, fusion and Track-while-Scan (TWS) tracking. T_4 is a sporadic task (minimum interarrival time 360 ms) for the management of console commands (i.e. the request of radar pulse change). T_1 releases jobs made of a unique chunk, having an execution time constrained between 20 and 30 ms and requiring resource *cpu* with priority level 0 (low priority numbers correspond to high priority levels). This chunk is synchronized through semaphore m_1 with the unique chunk of T_4. The acquisition (*wait*) and the release (*signal*) of a semaphore performed by a chunk are represented through two circles embracing the rectangle which represents that chunk.

lation and analysis of the pTPN specification model is employed in the architectural validation of the task set (*Analysis of Resources and Time Requirements*,

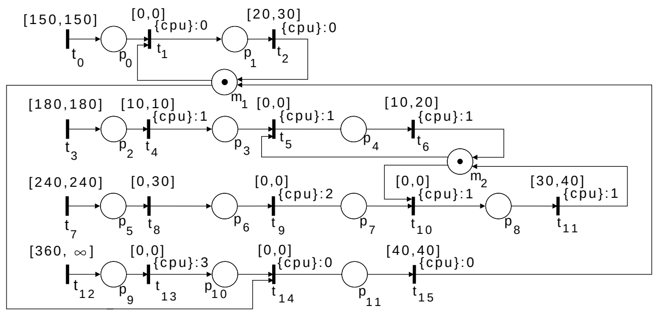
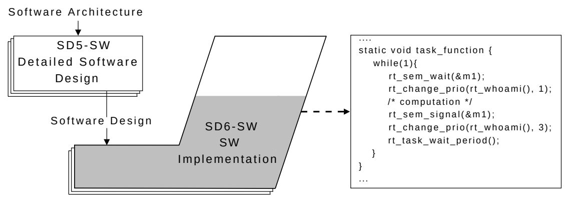


Fig. 5. The pTPN model for the timeline schema of Fig.4. Transitions t_0, t_3, t_7 and t_{12} account for repetitive job releases for tasks T_1, T_2, T_3 and T_4, respectively. They all have an output place, enabling the transition modeling the first chunk of the corresponding task. Subsequent chunks are modeled by chaining the transition representing the chunk and its input place. Places m_1 and m_2 model the two semaphores synchronizing the tasks. Transition t_1 models the acquisition of semaphore m_1 performed by the first chunk of T_1 with null execution time. Its firing enables transition t_2, which represents the nondeterministic execution time of the chunk and also performs release of the semaphore, having place m_1 as an output place. Since priority ceiling emulation protocol is assumed, in the translation from the timeline schema to the pTPN model, priority of tasks T_3 and T_4 is modified at the acquisition of semaphores m_2 and m_1, respectively. In particular, immediate transitions t_9 and t_{13} are added to T_3 and T_4, respectively, to model priority boost operations. The corresponding de-boost operations are represented by transitions t_{11} and t_{15}, which also model signal operations on semaphores m_2 and m_1, respectively.

SD5.2-SW, not shown in Fig.1), supporting tight schedulability analysis and verification of the correctness of logical sequencing.

The refined and validated pTPN model enables a disciplined coding of the CSCI (*Software Implementation*, SD6-SW, see Fig.6) which relies on conventional primitives of a real-time operating system, as reported in Sect.3.

Verification processes proceed from Module to System Levels through subsequent integrations, which may provide a feedback to the corresponding design activity. Software Implementation (SD6-SW) includes an activity of testing on single modules (*Self Assessment of the Software Module*, SD6.3-SW, not shown in Fig.1), which is aimed at testing single modules within an emulated environment. *Software Integration* (SD7-SW) achieves the integration of CSCIs and their modules into a software unit, performing self-assessment of both CSCIs and units, whereas *System Integration* (SD8) composes units and performs self-assessment of the system. *Transition To Utilization* (SD9) comprises tasks that are required to install a completed system at the intended application site and to put it into operation. Note that the integration process (SD7-SW and SD8) is carried out iteratively, due to the replacement of dummies (such as simulators, prototypes and emulators) with operative software and due to the exchange



```
....
static void task_function {
    while(1){
        rt_sem_wait(&m1);
        rt_change_prio(rt_whoami(), 1);
        /* computation */
        rt_sem_signal(&m1);
        rt_change_prio(rt_whoami(), 3);
        rt_task_wait_period();
    }
}
...
```

Fig. 6. Implementation activity in the development of an avionic radar system.

of modules/components/units with improved versions or off-the-shelf products: therefore, during integration, software components and units can be tested in isolation or within an emulated environment, and integration may be run until all dummies in the system have been replaced. As described in Sect.4, pTPNs can be effectively integrated within verification activities, to enable test case selection and to support test case execution and subsequent evaluation.

6 Conclusions

In this paper, we described how preemptive Time Petri Nets can be smoothly integrated into the life cycle of real-time software, supporting both design and verification stages. Preemptive Time Petri Nets permit modeling of a task set subject to temporal constraints, enabling architectural validation of the specification through its analysis and simulation. The pTPN specification model drives the implementation stage, supporting a disciplined coding of the task set architecture, and enables the definition of oracles which can be employed in the evaluation of time-stamped logs produced during the execution. We pointed out how coverage criteria can be defined on the symbolic state space of the pTPN model and motivated the adoption of paths in the state space as test cases, illustrating a procedure to sensitize them.

For large models, validation of process architecture through state space enumeration may become unfeasible due to state space explosion. However, partial verification limited to a portion of the state space can provide a relevant support in testing activities. In fact, the pTPN model of the specification can still be employed as an oracle in failures detection, also providing a level of coverage with respect to the portion of the state space which has been enumerated. In addition, the state space, even if uncomplete, can be employed to select critical behaviors to be tested and sensitized.

Acknowledgments. This research was supported as a part of *Iniziativa Software* FINMECCANICA. We are grateful to R. Loiacono, S. Orsi and P. Viliani of GA - Firenze, for their precious contribution in filling the gap between theory and practice.

References

1. RTCA (Radio Technical Commission for Aeronautics). Do-178b, software considerations in airborne systems and equipment certification. (http://www.rtca.org/).
2. A. Hessel, K. Larsen, B. Nielsen, P. Pettersson, A. Skou. Time-optimal realtime test case generation using UPPAAL. International Workshop on Formal Approaches to Testing of Software (FATES03), 2003.
3. K. Larsen, M. Mikucionis, B. Nielsen. Online Testing of Real-Time Systems Using UPPAAL: Status and Future Work. Perspectives of Model-Based Testing. 2005.
4. M. Krichen, S. Tripakis. Black-box conformance testing for real–time systems. SPIN'04 Workshop on Model Checking Software, 2004.
5. G. Bucci, A. Fedeli, L. Sassoli, E. Vicario. Timed state space analysis of real time preemptive systems. IEEE Trans.on Soft.Eng. **30**(2), 97–111, 2004.
6. O.H. Roux, D. Lime. Time petri nets with inhibitor hyperarcs: formal semantics and state-space computation. 25th Int. Conf. on Theory and Application of Petri nets, **3099**, 371–390, 2004.
7. F.Cassez, K.G.Larsen. The Impressive Power of Stopwatches. **1877**, LNCS, 2000.
8. G. Bucci, A. Fedeli, L. Sassoli, E. Vicario. Modeling flexible real time systems with preemptive time petri nets. Proceedings of the 15-th Euromicro Conference on Real-Time Systems (ECRTS03), 2003.
9. G. Buttazzo. Hard Real-Time Computing Systems. Springer, 2005.
10. P.Merlin, D.J.Farber. Recoverability of communication protocols. IEEE Trans.on Communications, **24**(9), 1976.
11. B. Berthomieu, M.Diaz. Modeling and verification of time dependent systems using time petri nets. IEEE Trans. on Soft. Eng., **17**(3), 1991.
12. W. Penczek, A. Polrola. Specification and model checking of temporal properties in time petri nets and timed automata. Proocedings of the 25^{th} Int. Conf on Application and Theory of Petri Nets (ICATPN2004), 2004.
13. B. Berthomieu, M. Menasche. An enumerative approach for analyzing time Petri nets. Information Processing: proc. of the IFIP congress 1983. **9**, 41–46, 1983.
14. E. Vicario. Static analysis and dynamic steering of time dependent systems using time petri nets. IEEE Trans. on Soft. Eng., 2001.
15. B. Berthomieu, D. Lime, O.H. Roux, F. Vernadat. Reachability problems and abstract state spaces for time petri nets with stopwatches. LAAS Report, 2004.
16. L. Sassoli, E. Vicario. Analysis of real time systems through the oris tool. Proc. of the 3^{rd} Int. Conf. on the Quant. Evaluation of Sys. (QEST), 2006.
17. CENELEC-prEN50128: Railway applications: Sw for railway control and protection systems. 1997.
18. C. Jard, T. Jéron. Tgv: theory, principles and algorithms, a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. Software Tools for Technology Transfer (STTT), **6**, 2004.
19. L. Carnevali, L. Sassoli, E. Vicario. Sensitization of symbolic runs in real-time testing using the oris tool. Proc. of the 12th IEEE Conference on Emerging Technologies and Factory Automation (ETFA), 2007.
20. L. Carnevali, L. Sassoli, E. Vicario. Casting preemptive time petri nets in the development life cycle of real-time software. Proc. of the 19-th Euromicro Conference on Real-Time Systems (ECRTS), 2007.
21. IEC 62304 International Standard Edition 1.0 Medical device software - Software life cycle processes, 2006.
22. Developing Standard for IT Systems of the Federal Republic of Germany. Lifecycle Process Model General Directive No. 250., 1997.