An Integrated Approach to Modeling and Analysis of Embedded Real-Time Systems Based on Timed Petri Nets [*]

Zonghua Gu and Kang G. Shin
Real-Time Computing Laboratory
EECS Department
University of Michigan
Ann Arbor, MI 48109-2122, USA
{zgu, kgshin}@eecs.umich.edu

Abstract

In computer-based control systems, embedded software is taking over what mechanical and dedicated electronic systems used to do, that is, to engage and control the physical world, interacting directly with sensors and actuators. Therefore, software running on a digital processor is tightly-coupled with its surrounding physical environment. We propose an integrated approach based on Timed Petri-Nets for modeling and analysis of embedded real-time systems where real-time scheduling behavior of the controller software is explicitly represented at the model-level, together with the physical environment that it interacts with. This enables the designer to have an integrated view of the entire system while analyzing the system and making design decisions. We also describe a syntax-directed, automated translation procedure from Timed Petri-Nets to Timed Automata, thus enabling the use of model checkers such as UPPAAL for analysis purposes. We consider the railroad crossing problem as an application example, and evaluate alternatives for controller implementation on either single-processor or distributed multi-processor platforms based on the integrated approach.

1 Introduction

Embedded software is the software controlling everything around us from telephones and pagers to cars and airplanes. Its main task is to take over what mechanical and dedicated electronic systems used to do, that is, to engage and control the physical world, interacting directly with sensors and actuators. Therefore, embedded real-time systems typically perform information processing tightly coupled with physical processes. The boundary between physical and software processes are often blurred. However, modeling tools tend to focus on either the system-level dynamics, or the real-time scheduling behavior of software processes.

Traditionally, the control engineer designs the control algorithms without consideration of controller platform issues, and then hands them over to the software engineer, who implements them on a minimum cost controller platform while guaranteeing system schedulability for a set of task execution frequency requirements. The authors in [18, 6] propose to break the rigid wall between controller design and software implementation, and adopt an integrated approach, thus opening up the possibility of applying a range of offline optimization and online adaptation techniques. For instance, instead of treating each task as having a rigid execution frequency requirement of 40Hz, we can relax it to an interval of [35Hz, 40Hz]. The controller suffers performance degradation with slower execution frequencies as long as it still maintains critical control objectives such as system stability. This enables the designer to perform cost-performance tradeoff analysis.

In this paper, we propose an integrated approach for modeling and analysis of embedded real-time systems with tight coupling between *embedded* software and *embedding* physical environment, and analyze the real-time scheduling behavior of the software together with the physical system that the software is controlling within the same formal framework of *Timed Petri-Nets* [17]. By adopting this approach, we enable the designer to have an integrated view of the entire system when making design decisions, so she can clearly see the effect of making a change in embedded software on the rest of the system, or a change in the physical system on embedded software design. She can also perform optimization analysis such as maximizing total system utility given resource constraints, or minimizing total

[*]The work described in this paper was supported in part by the US Army Research Office under contract DAAD 19-01-1-0473 and DARPA under the US AFRL grant F33615-00-1706.

system cost given safety and liveness requirements.

Various timed extensions to Petri Nets have been proposed, including Ramchandani's *Timed Petri Nets* [17], Merlin and Faber's *Time Petri Nets* [15], Little and Ghafoor's *Timed Petri Nets* [13], as well as stochastic extensions to Petri Nets [14]. (Here we use the term *PN with Time* to refer to the various timed extensions to PN). Even though analysis techniques [19] exist that can perform certain types of timing analysis on certain variants of PN with Time, tool support is generally either not available, or only offers limited analysis capabilities, despite an abundance of tools for analysis of various *untimed* Petri-Nets. For example, there is no tool support for the end-to-end timing analysis algorithms described in [22]. On the other hand, there are mature and scalable model checkers for Timed Automata (TA) [1], such as UPPAAL [2] and Kronos [10], that offer sophisticated analysis capabilities for temporal logic specifications of system property. We describe a simple translation algorithm from certain variants of PN with Time to semantically equivalent TA models, so that we can leverage the TA model checkers as a universal analysis back-end, instead of having to construct separate tools for each different variant of PN with Time. (Note that We do not deal with stochastic Petri Nets here.) In this paper we describe translation algorithms for two most popular types of PN with time, Ramchandani's *Timed PN* , and Merlin and Faber's *Time PN* [15]. In our opinion, PN has certain advantages over TA in terms of usability, since it has constructs for modeling system structure as well as behavior. It is easy to add in structural and behavioral hierarchy [5], which are absent in TA. However we can still map hierarchical PN models into TA models by flattening the hierarchy, at the expense of losing some clarity and understandability. Hence we propose to use PN with time as front-end interface for the designer and TA model checkers as back-end analysis engine.

In order to gain wider acceptance in industry, it is important to provide automated tool support instead of just algorithms described on paper. We use the Generic Modeling Environment (GME) [12] to provide the capabilities for modeling TPN and TA, as well as for implementation of translators from TPN to TA. GME is a configurable toolkit for creating domain-specific modeling and program synthesis environments. The configuration is accomplished through meta-models specifying the modeling paradigm of the application domain, which are used to generate the target domain-specific environment.

We use Ramchandani's *Timed Petri Nets* [17](hence referred to as *TPN*) as the modeling framework to illustrate the integrated modeling approach. TPN is well-suited for modeling distributed event-triggered software systems, since it is intuitive to map events to tokens, and execution of software components to transition firings. The translation procedure also gives a formal semantics for TPN in terms of TA, and clarifies some semantic ambiguities regarding multiple-enabled transitions. We use the well-known railroad crossing problem as an application example. Using the model checker UPPAAL, we were able to check the system safety and liveness properties, as well as schedulability of controller software within the same framework. In case a system timing property is violated, UPPAAL provides an error trace leading to the violation state and allows us to gain more insight into the cause of the violation.

Our contribution in this paper is two-fold: First, we propose an integrated modeling approach for embedded real-time systems with tight coupling between *embedded* software and *embedding* physical environment within the same modeling framework; next, we formally define a translation procedure from TPN models into TA models, thus enabling the use of mature model checkers for TA such as UPPAAL for TPN analysis. We also describe implementation of automated tool support for the TPN-to-TA translation within the Generic Modeling Environment(GME).

This paper is structured as follows. Section 2 provides a brief introduction to the two real-time formalisms considered, TPN and TA. Section 3 describes a simple algorithm for mapping TPN models into TA. Section 4 considers modeling and analysis of the railroad crossing problem. Section 5 describes related work, and the paper concludes with Section 6.

2 Introduction to TA and TPN

A timed automaton [1] is a standard finite-state automaton extended with a finite collection of real-valued clocks, which proceed at the same rate and measure the amount of time that has elapsed since they were last reset. The UPPAAL definition of TA has added a few extensions to the standard definition of [1], such as integer variables, CCS-style synchronous communication, urgent channels, etc. We refer interested readers to [2] for details.

Since there are many different variants of PN with Time, we provide a detailed description of our definition of TPN. A TPN is characterized by a 7-tuple $N = (P, T, B, F, I, M_0, D)$, where

- P is a finite set of *places* p_i.
- T is a finite set of *transitions* t_i.

- B is the *backward incidence* function $B : T \times P \rightarrow N$, where N is the set of nonnegative integers.
- F is the *forward incidence* function $F : T \times P \rightarrow N$.
- I is the *inhibitor edge incidence* function $I : T \times P \rightarrow \{0, 1\}$. The input place to an inhibitor edge is called an *inhibitor input place.*
- M_0 is the *initial marking* function $M_0 : P \rightarrow N$.
- D is a mapping $D : T \rightarrow Q^* \times (Q^* \cup \infty)$, which associates a *delay interval* $\tau = [lb, ub]$ with each transition $t \in T$, where Q^* is the set of rational numbers.

A transition t is said to be *enabled* when each of its input places p_i has at least $B(t, p_i)$ tokens, and each of its inhibitor input places p_j is empty. A transition T with delay interval $\tau = [lb, ub]$ is fired as soon as it is enabled, unless disabled by the firing of a conflicting transition that removes tokens from some of T's input places. The firing takes at least lb time units, but no more than ub time units. During the firing, the tokens at the input places have been consumed, but the tokens at the output places have not been produced. At the end of the firing, output tokens are produced. Note that our definition of TPN requires each transition to be *urgent*, that is, fired as soon as enabled, unless disabled by a conflicting transition at that instant, while in the original definition[17], no bound is imposed on when a transition may fire after it is enabled. Also, note the distinction between Timed Petri Nets and *Time Petri Nets* [15]. In a Time Petri Net, transition T has to be enabled continuously for $[lb, ub]$ time units before it can fire, and the firing is instantaneous: input tokens are consumed and output tokens are produced at the same time. During the time interval $[lb, ub]$, T may be disabled by the firing of a conflicting transition.

It is natural to use PN for representing resources and scheduling. CPU is an inherently sequential resource; that is, only one task can execute on the CPU at one time. This leads to an interleaving notion of concurrency, and priorities are used for arbitration of competing requests for the shared resource. TPN has *maximum parallelism* semantics, that is, independent transition firings can take place concurrently as if the number of processors available is unlimited. In order to model CPU scheduling, it is necessary to introduce shared places in order to sequentialize the execution of concurrent transitions. We can also easily model multiple physical resources such as multiple processors connected through a network, as well as logical resources such as semaphores.

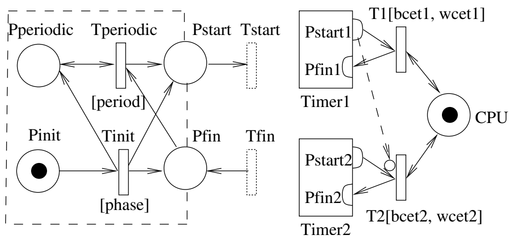


Figure 1: TPN models for periodic timers and real-time tasks.

On the left of Figure 1 is the TPN model for a periodic timer with *period* and initial *phase* that provides stimuli for a real-time task. The task is triggered when a token is deposited into P_{start}; at the completion of task execution, a token is deposited into P_{fin}. A *frame overrun* occurs if the task response time is greater than its period. In order to avoid frame overrun, the configuration $(P_{periodic} = 1, P_{fin} = 0)$ must not be reachable. On the right is the TPN model for static priority, *non-preemptive* scheduling of two periodic tasks. The blocks marked *Timer1* and *Timer2* are a syntactic shorthand for the timer model on the left. The upper part represents high-priority task $Task_1$; the lower part represents low-priority task $Task_2$. The place CPU denotes the shared resource of a single CPU. A triggered task executes if CPU is available, i.e., place CPU contains a token; otherwise it waits until the CPU becomes idle. The inhibitor edge from P_{start1} to T_2 represents the fact that $Task_1$ has priority over $Task_2$, since a non-empty P_{start1} prevents T_2 from firing.

Even though we use TPN in this paper, the integrated modeling approach is independent of the underlying modeling formalism. It is conceivable to adopt into our framework other formalisms that are more suitable for modeling real-time scheduling such as ACSR [7]. Since TPN lacks inherent notions of priority and preemption, we have to use a number of *ad hoc* techniques to work around these limitations. First, we use inhibitor arcs to simulate priorities. This approach prevents us from modeling dynamic priority scheduling algorithms such as Earliest Deadline First (EDF); it also becomes unwieldy when modeling a large number of tasks with distinct priorities. Second, although TPN is a dense-time formalism, we have to discretize the delay of a TPN transition in order to model preemptive scheduling, due to lack of a built-in notion of preemption like that in ACSR, or a stopwatch mechanism like that in Hybrid Automata [9]. (We omit the modeling of preemptive scheduling due to space limitations.)

3 TPN to TA Translation

We describe a translation algorithm for mapping a TPN model into a semantically equivalent TA model.[1]

1. Declare a global urgent channel *go*. A transition with an urgent channel as its synchronization label is an urgent transition, and has to be taken as soon as it is enabled without delay. Create an automaton named *Dummy* with a single location, and a transition with synchronization label *go*! starting and ending at that location, as in Figure 2.

2. For each TPN place $p \in P$, declare an integer global variable with the same name in the TA model.

3. Suppose a TPN transition $t \in T$ has an associated delay interval $[lb, ub]$, a pre-set of k input places $p_1^{in}, \ldots, p_k^{in}$, a post-set of m output places $p_1^{out}, \ldots, p_m^{out}$, and a set of n inhibitor input places $p_1^{inh}, \ldots, p_n^{inh}$. Classify each $t \in T$ according to its number of input, output and inhibitor places. For example, all transitions with 1 input place, 2 output places, and 1 inhibitor place are put into the same class. For each transition class:

   (a) Define a timed automaton template with two locations *disabled* and *firing*, one local clock c, and $k + m + n$ integer parameters named $p_1^{in}, \ldots, p_k^{in}$, $p_1^{out}, \ldots, p_m^{out}$, $p_1^{inh}, \ldots, p_n^{inh}$.

   (b) Add an invariant condition $c \leq ub$ at the location *firing*.

   (c) Add an edge from *disabled* to *firing* with guard condition $p_1^{in} \geq B(p_1^{in}, t), \ldots, p_k^{in} \geq B(p_k^{in}, t), p_1^{inh} == 0, \ldots, p_n^{inh} == 0$, synchronization label *go*?, and assignment label $c := 0, p_1^{in} := p_1^{in} - B(p_1^{in}, t), \ldots, p_k^{in} := p_k^{in} - B(p_k^{in}, t)$.

   (d) Add an edge from *firing* to *disabled* with guard condition $c \geq lb$, and assignment label $p_1^{out} := p_1^{out} + F(p_1^{out}, t), \ldots, p_m^{out} := p_m^{out} + F(p_m^{out}, t)$.

4. In the system configuration section, instantiate one automaton template for each TPN transition, with the appropriate global variables as parameters, representing the input, output and inhibitor places of that transition.

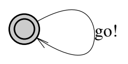


Figure 2: Automaton *Dummy* with an urgent transition *go*.

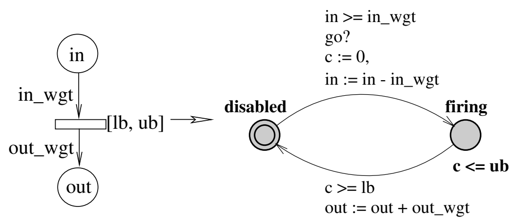


Figure 3: TA model of a TPN [17] transition t with 1 input place *in*, 1 output place *out*, and time bounds $[lb, ub]$. The process template has argument list (int *in*, *out*; const *in_wgt*, *out_wgt*; const lb, ub), and a local clock c.

Figure 3 shows the mapping for a TPN transition t with 1 input place *in* and 1 output place *out*. The urgent channel *go* ensures that the automaton changes its state from *disabled* to *enabled* as soon as $in \geq in\_wgt$, that is, the input place *in* contains *in_wgt* or more tokens. The number of tokens *in* is reduced by *in_wgt* representing the consumption of tokens in the input place. The TPN transition's firing duration $[lb, ub]$ is modeled by the state *firing* in the TA model, which has an invariant condition $c \leq ub$, and a guard condition $c \geq lb$ on the state change from *firing* to *disabled* that represents the end of transition firing. The resulting semantics is that the automaton has to change its state from *firing* to *disabled* if it has been staying in state *firing* continuously for at least lb time units, and at most ub time units. If the input place *in* contains more than $2 * in\_wgt$ tokens, and the TPN transition is still enabled after one firing, then the urgent channel *go* will immediately force a state change back to *firing* from *disabled*, and the clock is reset to start counting the delay interval $[lb, ub]$ all over again. That is, a new transition is freshly enabled after each firing.

Figure 5 shows a simple example taken from [22]. In order to translate this TPN model into a TA model, it is a simple matter of instantiating the TA templates for TPN transitions with 1 input/1 output, and 2 input/1 output, which happen to be the only two types

[1]This algorithm is based on that in [4]. Please refer to Section 5 for discussions on the differences between our algorithm and that of [4].

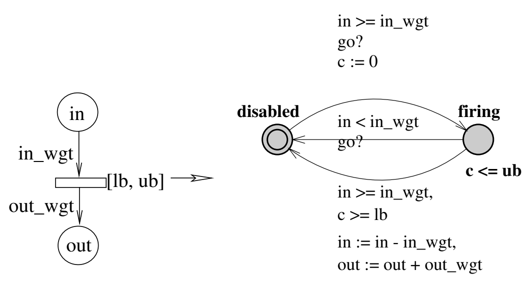


Figure 4: TA model of a *Time PN* [15] transition. Compare it to Figure 3 to see the difference in semantics between Merlin and Faber's *Time PN* [15] and Ramchandani's *Timed PN* [17]. Here a transition is eligible to be fired when it has been continuously enabled for *lb* time units, and it has to be fired when it has been continuously enabled for *ub* time units. But unlike Timed PN, the transition can be disabled if some other conflicting transition fires and removes tokens from its input place *in*, so that the number of tokens in place *in* drops below *in_wgt*.

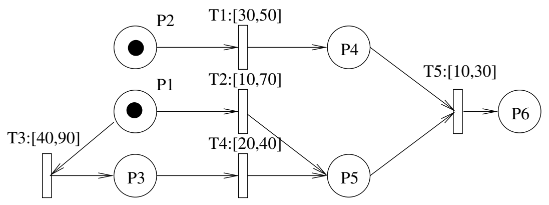


Figure 5: An example TPN model.

```
int P1:=1, P2:=1, P3:=0, P4:=0, P5:=0, P6:=0;
urgent chan go;
T1 := T1in_1out(P2, P4, 1, 1, 30, 50);
T2 := T1in_1out(P1, P5, 1, 1, 10, 70);
T3 := T1in_1out(P1, P3, 1, 1, 40, 90);
T4 := T1in_1out(P3, P5, 1, 1, 20, 40);
T5 := T2in_1out(P4, P5, P6, 1, 1, 1, 10, 30);
System T1, T2, T3, T4, T5, Dummy;
```

Figure 6: The UPPAAL system definition section that instantiates the templates for the TA model that is translated from the TPN model in Figure 5.

of transitions present, as shown in Figure 6.

The properties that can be verified through transformation from TPN to TA can also be directly verified through state space exploration of the TPN model itself, so we do not claim to add any analytical power by the TPN-to-TA mapping. We are merely proposing to take advantage of mature tools for TA such as UPPAAL for analysis of TPN, as well as other variants of timed extensions of Petri-Nets, as described in more detail in Section 1. Also note that reachability analysis for TA with variables is in general undecidable, so the model-checking procedure is not guaranteed to terminate. This corresponds to the fact that reachability analysis for TPN is in general undecidable.

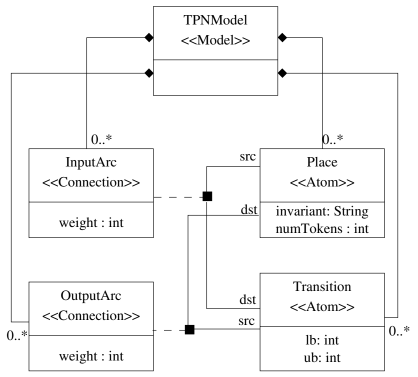


Figure 7: The UML-based meta-model for TPN.

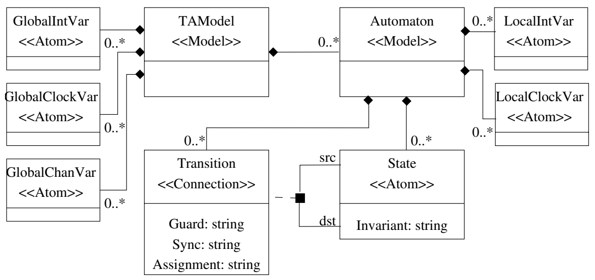


Figure 8: The UML-based meta-model for TA.

In order to implement automated tool support for the translation, we take advantage of the Generic Modeling Environment (GME) [12], which is a configurable toolset for creating domain-specific modeling and program synthesis environments through a *meta-model* that specifies the modeling paradigm of the application domain. The meta-model captures all the syntactic, semantic and presentation information regarding the application domain, and defines the family of models that can be created using the resulting modeling environment. Figure 7 and 8 show the UML-based meta-models for TPN and TA, respectively. For example, the meta-model in Fiture 7 specifies that a *TPNModel* consists of 0 or more *Places* and *Transitions*, connected by *InputArcs* and *OutputArcs*. Each *Place* has attributes such as *Invariant* and *numTokens*, and each *Transition* has a time interval specified by attributes *lb* (lower-bound) and *ub* (upper-bound). Domain-specific modeling environments for TPN and

TA can then be generated based on the meta-models. GME also provides a set of APIs for writing *model interpreters* that traverse the model elements and perform various processing tasks such as model translation and code generation. We have implemented an interpreter that performs TPN-to-TA translation by syntax-directed mapping of corresponding constructs in the two meta-models. For example, for each transition in the TPN model, an *Automaton* is generated for the TA model; for each place in the TPN model, a *GlobalIntVar* is generated for the TA model, etc. Another interpreter is needed to convert TA models in GME into the input file format to UPPAAL, which is based on XML.

4 Railroad Crossing Problem

Although the railroad crossing (RC) problem is a standard textbook problem in real-time specification and verification, there has been little discussion about the real-time scheduling behavior of the controller computer. That is, it is generally assumed that the controller is dedicated to a single task with no interference from higher-priority tasks or operating systems activity, so there is no need for real-time scheduling theory. This may well be true for the simple controller we are considering here, but in general designers have been putting more and more functionality on a single microcontroller in order to reduce costs. Furthermore, there is also a tendency to take advantage of distributed, multi-processor platforms. In these kinds of complex embedded systems, the real-time scheduling problem is non-trivial to solve, and it is desirable to model the scheduling and runtime platform issues explicitly.

4.1 RC without Scheduling

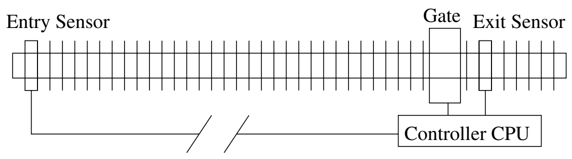
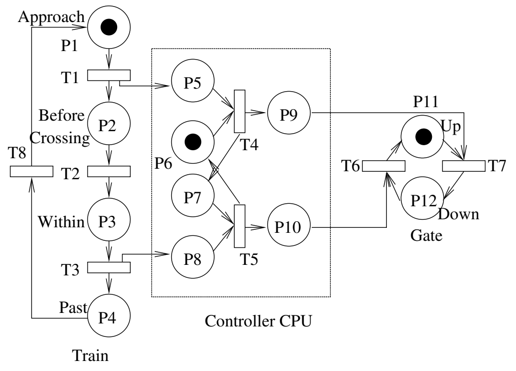


Figure 9: Railroad Crossing with a single controller CPU placed near the gate.

The RC problem describes a railroad crossing, whose physical layout is shown in Figure 9, and whose behavior is given by the TPN in Figure 10. Here we assume that the trains only travel from left to right. The system has to satisfy two properties:

- *safety*: Whenever the train is in the crossing, the gate has to be lowered.



Transition	T_1	T_2	T_3	T_4
[lb, ub]	[1,1]	[4,5]	[1,1]	[1,1]
Transition	T_5	T_6	T_7	T_8
[lb, ub]	[1,1]	[1,2]	[1,2]	[1,1]

Figure 10: TPN model of the RC system without consideration of real-time scheduling issues.

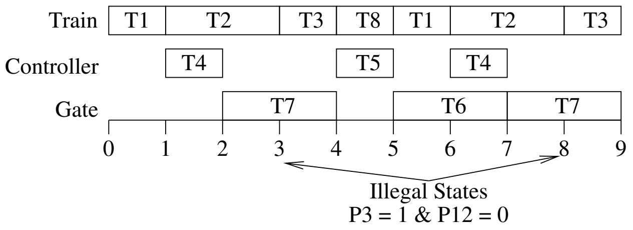


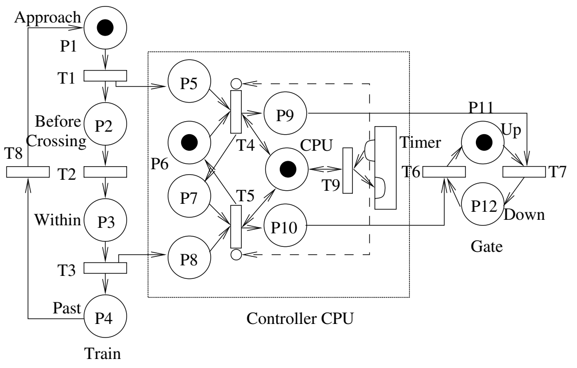
Figure 11: Execution trace of the TPN model in Figure 10, except T_2's delay interval is changed from [4,5] to [1,2]. That is, it takes shorter for the train to reach the crossing from entry sensor position.

- *bounded liveness*: Within a certain time limit δt after the train leaves the crossing, the gate has to be raised.

An entry sensor is placed some distance before the train reaches the crossing, and an exit sensor is placed a short distance after the train leaves the crossing. When the train crosses the position of the entry sensor (T_1 fires), a signal is sent from the sensor to the controller, which is typically placed near the gate. Upon receiving the signal, the controller sends a *lower-gate* command to the gate (T_4 fires). Upon receiving this command, the gate takes some time to lower itself (T_7 fires). Meanwhile, the train keeps going and enters the crossing (T_2 fires). In order to satisfy the safety requirement, the illegal state ($P_3 = 1$ & $P_{12} = 0$) should never be reached. That is, it should never be the case that the train is in the crossing and the gate is not lowered. After the train leaves the crossing (T_3 fires), the exit sensor sends a signal to the controller, which, in turn, sends a *raise-gate* command to the gate (T_5 fires). Upon receiving this command, the gate raises itself (T_6 fires). Note that we are not dealing with the *Generalized Railroad Crossing* [8] problem where multiple trains may be in the crossing at the same time. The TPN model in Figure 10 forces the gate to be raised and lowered once for each train going through the crossing.

Given the TPN specification of the RC system in Figure 10, we can map the TPN system into a TA model and use UPPAAL to check the system safety and liveness properties. For the safety property, it amounts to checking that (E<> $P_3 = 1$ and $P_{12} = 0$) is false. For the bounded liveness property, it is necessary to add an observer automaton that goes into an error state upon detecting a property violation. The system specified in Figure 10 satisfies both properties if $\delta t = 3$, that is, the gate has to be raised within 3 time units after the train leaves the crossing. However, if we change T_2's delay interval from [4,5] to [2,3], that is, the train travels faster and reaches the gate within [2,3] time units of tripping the entry sensor, then the safety property no longer holds. UPPAAL can provide us with an execution trace leading to the safety property violation, as shown in Figure 11.

4.2 RC with Single-Processor Scheduling



Transition	T_1	T_2	T_3	T_4	T_5
[lb, ub]	[1,1]	[4,5]	[1,1]	[1,1]	[1,1]
Transition	T_6	T_7	T_8	T_9	
[lb, ub]	[1,2]	[1,2]	[1,1]	[2,3]	

Figure 12: TPN model of the RC system with single CPU controller platform. A high-priority periodic task with period 10, execution time interval [2,3] and arbitrary release phase has been added. The timer block is a syntactical shorthand for the TPN model for a timer in Figure 1.

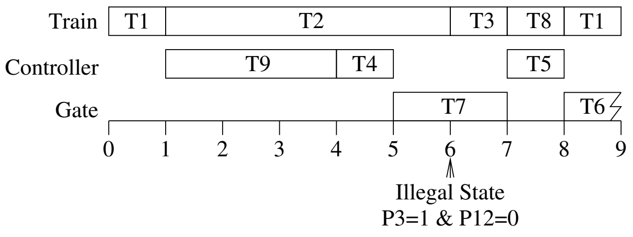


Figure 13: Execution trace of the TPN model in Figure 12.

In order to make the problem more interesting, we add a high-priority periodic task to the controller CPU. One can think of this task as a timer interrupt handler that demands immediate CPU processing. Figure 12 depicts the TPN model of the RC system for the single processor case. Note that we model non-preemptive scheduling for the sake of simplicity. Figure 13 shows that addition of the high-priority task results in the violation of safety property.

4.3 RC with Multi-Processor Scheduling

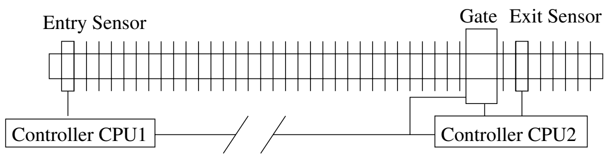
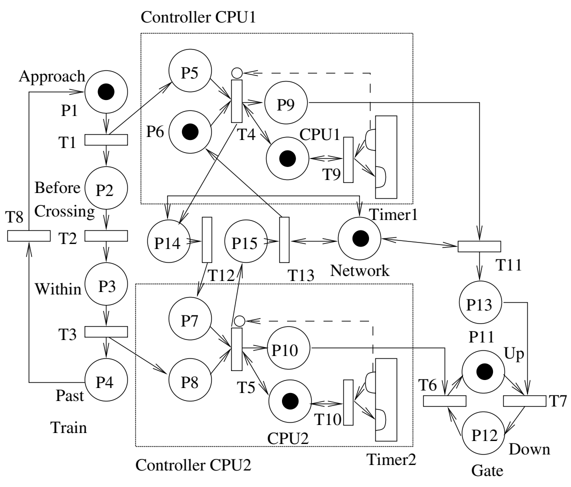


Figure 14: RC with a distributed multi-processor controller platform. CPU1 is placed near the entry sensor, and CPU2 is placed near the gate and exit sensor.

Figure 14 shows the layout for a multi-processor execution platform, where a controller is placed near the entry sensor that controls lowering of the gate, and another controller placed near the gate that controls raising of the gate. Figure 15 depicts TPN model of the RC system for the multi-processor case with two high-priority periodic tasks added, as well as explicit modeling of the shared network resource. Model-checking reveals that the safety property is again violated. Figure 16 shows an execution trace leading to the violation.

The designer has a number of options to remove the safety violation:

- Switch to preemptive scheduling, and assign lower priorities to the two interfering periodic tasks T_9,T_{10} on the controllers, as well as the network task T_{12}.
- Switch to a faster execution platform, including the CPUs and network. For example, reduce the



Transition	T_1	T_2	T_3	T_4	T_5
[lb, ub]	[1,1]	[4,5]	[1,1]	[1,1]	[1,1]
Transition	T_6	T_7	T_8	T_9	T_{10}
[lb, ub]	[1,2]	[1,2]	[1,1]	[1,1]	[1,1]
Transition	T_{11}	T_{12}	T_{13}		
[lb, ub]	[1,1]	[1,1]	[1,1]		

Figure 15: TPN model of the RC system with a multiprocessor execution platform. Two high-priority periodic tasks with period 10, execution time interval [1,2] and arbitrary release phase are added, one on each of the two CPUs. The place *Network* models the shared network connecting CPU_1 to CPU_2 and the gate. Here all the message transmission tasks on the network happen to have the same priority.

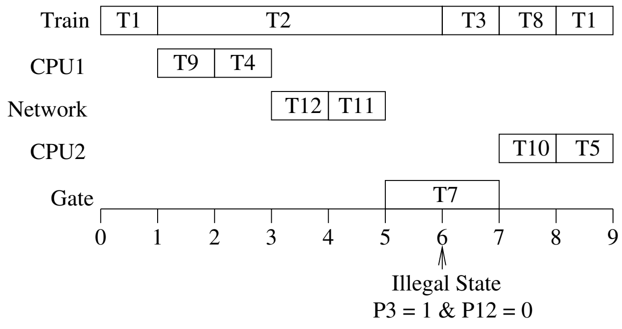


Figure 16: Execution trace of the TPN model in Figure 15.

WCET of T_4 to be below 0.5, and the network message transmission latency T_{11} to be below 0.5.

- Impose a reduced speed limit on incoming trains once they reach the entry sensor position, so that the minimum time the train takes to reach the crossing from the entry sensor position is above 6.
- Switch to a more responsive gate so that the time it takes to raise or lower the gate is below 1.

Of course we can adopt a combination of any subset of the above options. In general, model-checking can be used to derive certain timing parameters, whether those of the software or the physical environment, given timing specification for the rest of the system, in order to satisfy system-level requirements. Ideally this requires *parametric analysis* capability such as that provided by Hytech [9], which is not present in UPPAAL. Still, we can use a trial and error approach, and perform a binary search on the possible intervals of variable values to find out the answer.

The RC problem serves as an illustrating toy example for the integrated modeling approach, and we plan to model and analyze more realistic application examples and evaluate the scalability of our approach.

5 Related Work

Real-time scheduling theory based on the rate monotonic analysis (RMA) framework [11] is a mature research area whose results have been widely adopted by industry practitioners. However it also has certain limitations compared to formal analysis techniques. First, scheduling anomalies can occur when jobs have arbitrary release times and shared resources [20]. Even though algorithms have been developed to solve such scheduling problems, each algorithm is problem-specific and needs to be developed for each situation. Second, formal analysis tools such as ACSR [7] and Hytech [9] provide *parametric analysis* capabilities that can synthesize task timing parameters in order to satisfy system-level timing constraints, which is not straightforward to achieve with RMA. Third, scheduling theory focuses on the software system only and does not allow for an integrated modeling approach.

Model-Integrated Computing (MIC) [21] uses integrated, multi-view, domain-specific models to capture information relevant to the system under design. Models can represent the designer's understanding of an entire computer-based system, including information-processing architecture, physical architecture, and operating environment. Their modeling tool, Generic Modeling Environment (GME) [12], explicitly represents dependencies and constraints among various modeling views, and can be used for generating system implementation as well as specialized models that feed into various analysis tools such as model-checkers. Our approach can be viewed as a specific instance of the more general concept of MIC, using a particular formal model (TPN) and analysis method (model-checking), while MIC allows the designer to use meta-modeling techniques to construct arbitrary domain-specific modeling environments.

Cortes [4] proposed a mapping algorithm from *PRES+* model, which is a variant of *Time Petri Nets* with additional data handling capabilities, into HyTech [9] models. Our mapping is simpler and more compositional because we take advantage of UPPAAL's capability of having guard conditions on urgent transitions, which is not present in Hytech. Cortes' mapping algorithm can only deal with 1-safe PNs (each place can contain at most one token), while our algorithm can deal with non-1-safe PNs (each place can contain more than one token) and multiple-enabled transitions. Instead of assuming that the PN is 1-bounded, we can write temporal logic queries in UPPAAL to check for n-boundedness of any place or the entire PN. The ability to model non-1-safe TPNs is convenient for modeling task queuing and preemptive scheduling.

For PN with Time, some semantic ambiguities arise when multiple tokens are allowed in one place [3], since we can make a choice as to which tokens are chosen to enable transitions. In our TA-based semantics, instead of keeping track of the age of each individual token, we only require the total number of tokens in each input place to be above the enabling threshold in order for a transition to be enabled. This corresponds to a *threshold-based* instead of *age-based* [3] semantics. An example where this semantics is useful is Figure 17, which shows a model used to detect server overload, taken from [3]. Different semantics are useful in different situations, but the important point is that our translation algorithm gives a clear semantics to modeling constructs that are otherwise ambiguous.

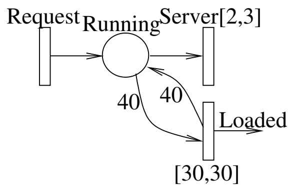


Figure 17: A model fragment used to detect server overload. The number of tokens in place *running* represents the number of outstanding requests to be processed at the server. If this number has been greater than 40 for more than 30 time units, an *overload* signal is generated by putting a token in the place *overload*. Note that this is a *Time PN* model [15] instead of a *Timed PN* model [17], and should be mapped to TA according to Figure 4.

Naedele [16] presented an approach *delegated execution*, which allows modeling and simulation of both functional and scheduling aspects of real-time systems with High-Level Petri Nets (HLPN). Due to high expressive power of HLPN, his approach is scalable to larger models, and can model preemptive scheduling more elegantly than our approach. However the analysis technique is limited to simulation; formal analysis via model-checking is not supported.

6 Conclusions and Future Work

In this paper we have proposed an integrated approach for modeling and analysis of embedded real-time systems with tight coupling between *embedded* software and *embedding* physical environment, where the physical system and the software artifacts are modeled within the same formal framework. We have also described a translation procedure from TPN models into TA models, thus enabling the use of mature model checkers for TA such as UPPAAL for TPN analysis. We describe implementation of automated tool support within the Generic Modeling Environment (GME). Our approach allows the designer to model and analyze the embedded system in an integrated manner, including the physical system and the software controlling it, and use model-checking to determine schedulability of the software together with system-level timing constraints.

Although we have used a specific modeling formalism (TPN), this approach is generic and can be applied together with other formal or informal models commonly used in the embedded systems domain. Formalisms like TPN and TA are used for analysis only, and are not *broad-spectrum* models that can be applied throughout the system development life-cycle. We plan to investigate integration of formal techniques with informal, widely-adopted techniques such as the Unified modeling Language(UML), which would allow the designer to gain the benefits of applying formal techniques without the overhead of learning "yet another modeling language".

References

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[2] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal - a tool suite for automatic verification of real-time systems. In *Proceedings of Workshop on Verification and Control of Hybrid Systems III*, pages 232–243, October 1995.

[3] Marc Boyer and Michel Diaz. Multiple enabledness of transtions in petri nets with time. In *Proceeding of the 9th International Workshop on Petri Nets and Performance Modeling*, 2001.

[4] L.A. Cortes, P. Eles, and Z. Peng. Verification of embedded systems using a petri net based representation. In *Proceedings of the 13th International Symposium on System Synthesis*, pages 149–155, 2000.

[5] Y. Deng, S. Lu, and M. Evangelist. A formal approach for architectural modeling and prototyping of distributed real-time systems. In *Proceedings of the 30th Hawaii International Conference on System Sciences*, pages 481–490, 1997.

[6] J. Eker and A Cervin. A matlab toolbox for real-time and control systems co-design. In *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, pages 320 –327, 1999.

[7] R. Gerber and I. Lee. A layered approach to automating the verification of real-time systems. *IEEE Transactions on Software Engineering*, 18(9):768–784, 1992.

[8] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. A benchmark for comparing different approaches for specifying and verifying real-time systems. In *Proceedings of the Tenth International Workshop on Real-Time Operating Systems and Software*, May 1993.

[9] T. Henzinger, P. Ho, and H. Wong-Toi. Hytech: A model checker for hybrid systems. *Software Tools for Technology Transfer, special issue on timed and hybrid systems*, pages 110–112, 1997.

[10] T. Henzinger, P. Ho, and H. Wong-Toi. Kronos: A verification tool for real-time systems. *Software Tools for Technology Transfer*, 1, 1997.

[11] Mark H. Klein, Thomas Ralya, Bill Pollak, and Ray Obenza. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems.* Kluwer Academic Publishers, 1993.

[12] Akos Ledeczi, Miklos Maroti, Arpad Bakay, and Gabor Karsai. The generic modeling environment. In *Proceedings of the IEEE International Workshop on Intelligent Signal Processing*, May 2001.

[13] T.D.C Little and A. Ghafoor. Spatial-temporal composition of distributed multimedia objects for value-added networks. *Computer*, 24(10):42–50, 1991.

[14] M. Ajmone Marsan. Stochastic petri nets: An elementary introduction. In *Advances in Petri-Nets, volumn 424 of Lecture Notes in Computer Science*, pages 281–305, 1989.

[15] Philip Merlin and David Farber. Recoverability of communication protocols – implications of a theoretical study. *IEEE Transactions on Communications*, 24, 1976.

[16] Martin Naedele. Modeling and simulating functional and timing aspects of real-time systems by delegated execution. In *Proceedings of International Conference on the Engineering of Computer Based Systems*, 2000.

[17] C. Ramchandani. Analysis of asynchronous concurrent systems by timed petri nets. In *Technical Report TR 120, Massachussets Institute of Technology*, February 1974.

[18] D. Seto, J.P. Lehoczky, L. Sha, and K.G. Shin. On task schedulability in real-time control systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 13–21, 1996.

[19] H. Storrle. A evaluation of high-end tools for petri nets. Technical report, University of Munich, 1998.

[20] Jun Sun and Jane W.S. Liu. Bounding the end-to-end response times of tasks in a distributed real-time system using the direct synchronization protocol. Technical report, University of Illinois, 1996.

[21] Janos Sztipanovits and Gabor Karsai. Model-integrated computing. *IEEE Computer*, 30(4):110–111, April 1997.

[22] J. Wang and Y. Deng. Reachability analysis of real-time systems using time petri nets. *IEEE Transactions on Systems, Man and Cybernetics*, 30(5), 2000.