Towards the Exhaustive Verification of Real-Time Aspects in Controller Implementation

Carlo A. Furia, Marco Mazzucchelli, Paola Spoletini, Mara Tanelli

Abstract— **In industrial applications, the number of final products endowed with real-time automatic control systems that manage safety-critical situations has dramatically increased. Thus, it is of growing importance that the control system design flow encompasses also its translation into software code and its embedding into a hardware and software network. In this paper, a tool-supported approach to the formal analysis of real-time aspects in controller implementation is proposed. The analysis can ensure that some desired properties of the control loop are preserved in its implementation on a distributed architecture. Moreover, the tool provides as output information which can be used to approach straightforwardly some *design* problems, such as hardware sizing in the final implementation.**

I. Introduction and Motivation

In industrial applications, the number of final products endowed with real-time automatic control systems has dramatically increased in the last few years. Most of such systems, moreover, are responsible of managing critical situations as far as human safety is concerned, see *e.g.*, in the aeronautical and the automotive industries. Thus, it is of growing importance to complete the control system design flow taking care also of formally tackling its translation into real-time software code and its embedding in a larger hardware and software network, with particular attention to timing properties. As such, it seems particularly useful and promising to try and bridge the gap between system-theoretic and information-theoretic skills, devising new tools which allow one to carry out the whole design flow and provide new information to optimize the final system both in the controller design phase and in its code and hardware realization. Recently, in the control community, some preliminary considerations about the possible role for formal methods to tackle control software dependability properties have started to appear, see *e.g.*, [1], [2]. In [1], the authors discuss how probabilistic model checking can be used to analyze faults in controller-based systems. [2] offers a discussion on how formal methods may help to develop dependable manufacturing control systems. In [3], algebraic equivalence is used to verify the correct operation of control systems.

In the formal methods community, a vast gamut of techniques, methods, and tools has been studied by computer scientists for over two decades, [4]. In particular, since the early 1990's, several techniques have been developed to analyze *real-time* systems with respect to both behavioral and timing properties [5]. In particular, model-checking techniques for real-time systems, [6] have become a very popular solution to the system verification problem. Note that, in this context, full automation must always be heavily traded-off with expressiveness. In particular, it is usually impossible (or exceedingly complicated) to formalize complex properties that have to be analyzed in the design flow of embedded systems towards their implementation. In the same vein, formal techniques are usually mostly focused on *a posteriori* analysis, and provide instead little or no support to *design*. Some recent work addressing this problem is [7], which proposes mathematical models for embedded software that allow one to incorporate constraints and specifications at a high level of abstraction and to formally derive semantics-preserving software implementations.

In this paper, we propose a tool-supported approach to the formal analysis of real-time aspects in controller implementation. Our methodology builds upon an automated algorithmic enumeration technique of system behavior to pursue a guided examination of such real-time aspects. The analysis can ensure that some desired properties of the control loop are preserved in its implementation on a distributed architecture. Moreover, the information extracted automatically from the model can also be used to approach some *design* issues, such as hardware sizing in the final implementation. Our tool uses timed Petri nets as abstract system modeling paradigm. For illustration purposes, however, we employ the more abstract and essential model of state/transition systems (STSs). Specifically, we introduce an analysis technique to enumerate all possible timed behaviors of some considered STS, grouping them into equivalent classes, which share the same behavior. Afterward, we show how to extend the analysis technique for STS to timed Petri nets and how the results can be used proficiently in the iterative development of controller implementations.

II. State/Transition Systems

In this section, the notion of state/transition system (STS) and a technique to represent and analyze the behavior of a given STS are introduced, see also [8]. STSs are an abstract modeling paradigm suitable for a large variety of timed systems, based on the notions of *state* and *transition*. Informally, the evolution of an STS is characterized by an alternation between residences in states, where time elapses, and instantaneous occurrences of transitions, which trigger the system to a new state. At any instant of time, the current state of the system determines which transitions are *enabled*, *i.e.*, can *fire*; conversely, any fired transition triggers a change of state in the system.

As a simple example, consider a producer and a consumer which communicate through a (bounded) buffer, where the producer puts new items and the consumer takes them away.

C. A. Furia and M. Tanelli are with the Dipartimento di Elettronica e Informazione, Politecnico di Milano, Piazza Leonardo da Vinci 32, 20133 Milano, Italy. E-mail: {furia, tanelli}@elet.polimi.it. M. Tanelli is also with the Dipartimento di Ingegneria dell'Informazione e Metodi Matematici, Università degli studi di Bergamo, Via Marconi 5, 24044, Dalmine (BG), Italy. M. Mazzucchelli is with the Dipartimento di Matematica, Università di Pisa, Italy. E-mail: mazzucchelli@mail.dm.unipi.it P. Spoletini is with Università dell'Insubria, Como, Italy. E-mail: paola.spoletini@uninsubria.it. The work of M. Tanelli has been partially supported by MIUR project "New methods for Identification and Adaptive Control for Industrial Systems".

An STS modeling such a system would introduce a transition c which occurs whenever the consumer takes an item from the buffer. Note that c would be enabled only when the system is in a state where at least one item is in the buffer. Also, when c is taken and the buffer has some $n > 0$ items in it, the state of the buffer changes to represent the fact that $n - 1$ items are now stored in the buffer (see also Figure 1). When modeling *real-time* systems as STSs, the notion of state must include *timing* information about when the current state has been entered. Consequently, the triggering of transitions depends also on the time elapsed in the current state. Therefore, transitions are decorated with a lower bound and an upper bound denoting, respectively, the minimum and maximum time that should be consumed in a state enabling the transition before the transition is taken.

Analyzing the behavior of STSs. A fully formalized STS can be seen as an implicit, complete description of a set of timed behaviors, namely all those which are compatible with all the constraints included in the STS. For several analysis purposes, it would be extremely useful to have a characterization of the same behaviors which is explicit, rather than implicit. In other words, we would like to possess a representation which allows answering rather directly questions about the overall behavior of the system. For instance, going back to the example of the producer and consumer, one may consider questions such as:

- given certain producing and consuming rates, is it true that the buffer never overflows?
- is it true that every item stays in the buffer for at least 10 time units before it is consumed?
- if we double the consuming rate, can we halve the buffer size without risking overflow?

Simulation techniques are a way to answer such questions. However, simulation gives necessarily incomplete answers, as it considers only a finite subset of all possible behaviors of the system. Model-checking techniques, on the other hand, are exhaustive verification techniques where temporal logic formulas are checked against all behaviors of a finite-state system [6]. In this paper, we develop a third approach, in some sense intermediate between simulation and automated model-checking. On the one hand, we show how to extract and represent succinctly *all* possible behaviors of a given STS. On the other hand, we trade-off full automation with greater *flexibility* for the designer. In our method, in fact, rather than providing *a priori* a property in some (restricted) temporal logic language, the designer has to inspect the generated exhaustive behavioral representation of the system, looking for desired (or undesired) behaviors. In this sense, the designer can possibly make up for the algorithmic unfeasibility of verifying complex properties with his/her intuition and ingenuity. Finally, we believe that Petri net models are especially suitable to represent naturally systems with asynchronous components like those considered in our case study.

A. Definition of Transition Systems

The essential features of STSs can be further abstracted by representing explicitly only transitions, and by leaving the notion of state only implicit: such models are called transition systems (TSs). As the name suggests, TSs are based on the notion of transition, which are defined as follows.

Definition 2.1: A *transition* τ is a triple $\langle[\alpha,\beta],E,D\rangle$ where: $[\alpha,\beta]$ is an interval of time called *static firing interval*, here α and β are named respectively (static) earliest firing time and (static) latest firing time. They denote the lower and upper bound on the transition firing time, after it has become enabled; E (resp. D) is the set of transitions which become enabled (resp. disabled) when τ fires.

Note that, at a given time, more than one occurrence of the same transition can be enabled, each with its own enabling time. To represent this fact we introduce the following.

Definition 2.2: An *occurrence* or *instance* of a transition τ is a triple $\langle \mathrm{id}, x, [\hat{\alpha},\hat{\beta}]\rangle$ where: id is a unique identifier, to distinguish different instances of the same transition τ; x is the (absolute) enabling time; $[\hat{\alpha},\hat{\beta}]$, with $\hat{\alpha} = x + \alpha$ and $\hat{\beta} = x + \beta$, is the dynamic firing interval within which the transition *must* fire if it is not disabled.

In the following, we freely use the term "transition" to denote indifferently a transition or an instance thereof, whenever the actual meaning will be clear from the context. Also, we distinguish relative firing times from absolute ones by putting a hat over the latter. Finally, we introduce the notion of TS.

Definition 2.3: A *transition system* S is a pair $\langle T, T_0\rangle$ where: T is a set of transitions; $T_0 \subseteq T$ is the set of transitions which are enabled *initially*.

Any TS is an implicit representation of a set of behaviors, namely all those which respect the enabling/disabling relations and the firing time intervals. More precisely, we introduce the following.

Definition 2.4: An *evolution* e (or *trace*) of a TS $S = \langle T, T_0\rangle$ is given by two finite sequences: the sequence $\langle \tau_1, \tau_2, \ldots, \tau_k\rangle$ of instances of fired transitions; and the sequence $\langle \hat{\tau_1}, \hat{\tau_2}, \ldots, \hat{\tau_k}\rangle$ of firing times of the transitions. The two must respect all constraints arising from the transition firing semantics.

If we want to stress the fact that the number of taken transitions in e is bounded by some natural $k > 0$, we call e an *NT-evolution* (*i.e.*, "Number of Transitions").

Equivalence classes of evolutions. From the definitions above, it is clear that the number of NT-evolutions of a TS are, in general, infinite, when the time domain is a dense set. In practice, this is the case whenever there is at least one enabled transition whose firing time is non-deterministic, that is whose firing interval is not a singleton. To overcome this unpleasantness, we introduce a notion of equivalence between evolutions: two evolutions are *equivalent* if they have the same sequence of fired transitions, in the same order. Consequently, we introduce a notion of equivalence class as follows.

Definition 2.5: For a natural $k > 0$, a *k-NT-class* is an equivalence class over the set of all NT-evolutions of length k.

A k-NT-class can be represented as a tuple $\langle \tau_1, \ldots, \tau_k\rangle$ of the ordered sequence of transitions. It is not difficult to show that, regardless of the nature of the time domain, and in particular even if it is dense, the quotient set of the NT-evolutions of a TS by the equivalence relation is finite.

Theorem 2.6: Given a transition system S, for each integer $k > 0$, the number of k-NT-classes is finite.

In the rest of the paper, we assume the time domain to be the nonnegative rational numbers $\mathbb{Q}^+$. Intuitively, this restriction is necessary to ensure that all different time bounds have a common multiple, or, in other words, that a notion of "minimum time granularity" is definable. This would not be possible over the whole real domain, where rational and irrational numbers are incommensurate. For all practical purposes, however, the restriction to rational numbers is irrelevant, as any irrational number can be approximated by rational numbers with arbitrary precision. Note that the same choice was taken in [9].

B. Exhaustive Enumeration of TS Behavior

Theorem 2.6 suggests that an exhaustive enumeration of NT-classes may be feasible. The rest of this section presents an algorithm to build such an enumeration. The algorithm is based on techniques similar to those introduced in [9] for timed Petri nets. However, while [9] only provides enumeration techniques for NT-classes, we extend those techniques to build *real-time profiles* that characterize each enumerated NT-class. All NT-classes together with their real-time profiles provide a complete succinct description of all NT-evolutions of the system; this description can be usefully analyzed by the designer. Our technique has been implemented in a very prototypal tool used in analyzing the case study of Section III. In what follows, for brevity, we omit most technical details of the algorithm, focusing on its intuitive explanation. Let us consider a TS $S = \langle T, T_0 \rangle$. It is convenient to represent the NT-classes of S as a tree, named *relative tree*. Roughly, nodes represent states of the system evolutions, and every node is connected to all its children, which correspond to states that are reachable from the parent state by firing an enabled transition. More precisely, every node in the tree hosts a triple of the form $\langle ET_i^j, D_i^j, mM_i^j \rangle$, where the subscript i denotes the depth of the node within the tree; the superscript j denotes that the node has been reached by firing transition τ_j; ET_i^j is the set of the transitions that are currently enabled; D_i^j is the set of firing constraints of transitions in ET_i^j, described as a set of inequalities; mM_i^j is the least upper bounds among the transition times of transitions enabled in the current node, *i.e.*, $mM_i^j = \min\left\{\beta_x | (\alpha_x \leq \tau_x \leq \beta_x) \in D_i^j\right\}$. In extreme summary, the tree represents all NT-classes of the systems.

C. Real-Time Profiling of TS Behavior

We now build upon the relative tree an algorithm to create the *real-time profile* of any NT-class. In other words, we sketch how to construct intervals—of absolute time—for every transition in an NT-class, which describe the minimum and maximum absolute firing times that every evolution in the chosen NT-class may have. Once the relative tree has been built, real-time profiling is a two-phase process. First, the *absolute tree* for the TS is built; then the actual profiling is performed for a chosen NT-class of the TS. These two phases are succinctly described in the rest of this subsection.

Absolute Tree. For any TS S, the absolute tree is a tree isomorphic to the relative tree, but where every node hosts information about the absolute—rather than relative—firing times of transitions. To avoid ambiguities between the relative and absolute trees, and in accordance with the notation introduced in Subsection II-A, we decorate every symbol appearing in the absolute tree with a hat sign. As in the relative tree, every node in the absolute tree hosts a triple $\langle \hat{ET}_i^j, \hat{D}_i^j, \hat{mM}_i^j \rangle$, where $\hat{ET}_i^j$ is the same as ET_i^j in the relative tree; $\hat{D}_i^j$ is a set of absolute time constraints for the transitions in $\hat{ET}_i^j$, in the form $\hat{\alpha}_x \leq \hat{\tau}_x \leq \hat{\beta}_x$, with $\hat{\tau}_x \in \hat{ET}_i^j$; $\hat{mM}_i^j = \min\left\{\hat{\beta}_k | (\hat{\alpha}_i \leq \hat{\tau}_i \leq \hat{\beta}_i) \in \hat{D}_i^j\right\}$.

Properties of the Absolute Tree. The following theorem establishes the fundamental properties of the absolute tree construction.

Theorem 2.7: Let $\langle \tau_1, \ldots, \tau_k \rangle$ be any k-NT-class of some TS S, whose absolute tree $\mathcal{T}$ hosts the inequality $\hat{\alpha} \leq \hat{\tau}_k \leq \hat{\beta}$ for τ_k in $\hat{D}_{k-1}$. Then, for all (real) times t, there exists some NT-evolution in the chosen NT-class where τ_k fires at t if and only if $t \in [\hat{\alpha}_f, \hat{mM}_{k-1}]$.

Combining the Relative and Absolute Tree. The information hosted by the relative and absolute trees can be combined into a data structure called *class list*. This gives a complete real-time profiling of any NT-class. We introduce the operator $\sharp$ such that $\sharp(\tau) = i$ iff τ is the i-th transition firing in the considered NT-class.

The Class List. Let us consider some TS S, its relative and absolute trees $\mathcal{T}^r, \mathcal{T}^a$, and some k-NT-class $\langle \tau_1, \ldots, \tau_k \rangle$. The *class list* has size k, and every element is a triple $\langle \tau_f, DF_i^f, DV_i^f \rangle$, with $i = \sharp(\tau_f)$, where τ_f is the i-th transition in the NT-class; DF_i^f is a set of constraints on the absolute firing time of τ_f, in the form $m_f^i \leq \hat{\tau}_f \leq M_f^i$, with m_f^i and M_f^i rational constants; DV_i^f is a set of constraints on the absolute firing times of pairs of transitions, in the form: $\hat{\tau}_x + m_x^i \leq \hat{\tau}_f \leq \hat{\tau}_x + M_x^i$, with m_x^i and M_x^i rational constants, and τ_x such that $\sharp(\tau_x) < \sharp(\tau_f)$.

Building an NT-evolution. Any NT-evolution can be built relying on the information the class list provides. Starting from the first element in the list, where $\hat{\tau}_{f_1}$ is the only free variable in the inequalities in DF_1^1, we pick a firing time $\hat{\tau}_{f_1}$ which respects all such inequalities. Then, any occurrence of $\hat{\tau}_{f_1}$ in all inequalities in all DV sets in the class list is replaced with the chosen value, and the resulting inequality moved to the corresponding DF sets. Next, we move to the second element of the class list, and so on until we have built a complete set of firing times for our NT-evolution. The correctness of the class list construction is summarized in the following.

Theorem 2.8: Let $\mathcal{L}$ be the class list for a k-NT-class of a TS S. Any NT-evolution is valid for the chosen NT-class *if and only if* it respects all constraints in $\mathcal{L}$.

D. Timed Petri Nets

Timed Petri Nets (TPNs) [10] are a popular real-time extension of the classical Petri Nets [11], introduced by C. A. Petri in 1962 as a modeling tool for discrete event systems. TPNs are devoted to deal with real time systems and they can be seen as a form of STSs.

From TSs, to STSs, to TPNs. To represent TPNs semantics, TSs can be extended by introducing explicitly the concept of *state*.

Definition 2.9: A *state/transition system* (STS) S is a quadruple $\langle T, Q, q_0, \delta \rangle$, where T is the set of transitions of the system; Q is the set of states of the system (possibly infinite); $q_0 \in Q$ is the initial state of the system, *i.e.*, the state of the system at time $t = 0$; $\delta : Q \times T \times \mathbb{Q}^+ \to Q$ is the (partial) transition function; $\delta(q, \tau, t) = q'$ denotes that the system in state q, with transition τ firing at time t moves instantaneously to state q'.

The system evolves from a current state q_n to the next state q_{n+1} as a result of the firing of some transition τ_f, enabled in q_n. The firing time must be feasible, *i.e.*, it must be included in the dynamic firing interval of τ_f and it must be less than or equal to the minimum of the maximum firing times of the transitions enabled in q_n. The firing will disable τ_f, and possibly some other transitions enabled in q_n, and it will enable some new transitions.

Now, we introduce timed Petri nets and hint at how to describe them as STSs.

Definition 2.10: A *timed Petri net* is a tuple $\langle P, T, B, F, M_0, \mathrm{SIM}\rangle$, where P is a finite non-empty set of places (a.k.a. locations); T is a finite non-empty set of transitions; $B : P \times T \rightarrow \mathbb{N}$ is the backward incidence function (BIF); $F : T \times P \rightarrow \mathbb{N}$ is the forward incidence function (FIF); $M_0 : P \rightarrow \mathbb{N}$ is the initial marking; $\mathrm{SIM} : T \rightarrow \mathbb{Q}^+ \times \mathbb{Q}^+ \cup \{\infty\}$ is the static interval function.

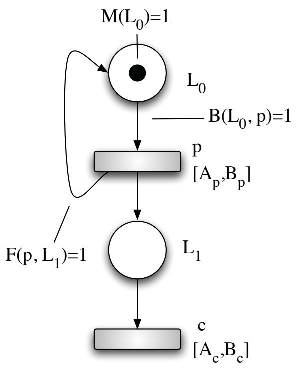


Fig. 1. TPN model of the producer/consumer example.

Figure 1 provides some intuition about the graphical representation of a TPN for the producer/consumer example. To formalize a TPN by means of an STS, for each transition $\tau_i \in T$, the static interval function gives the static firing interval, *i.e.*, $\mathrm{SIM}(\tau_i) = [\alpha, \beta]$. If x_i is the enabling instant of a transition τ_i, the dynamic firing interval is $[\alpha{+}x_i, \beta{+}x_i]$. Correspondingly, we can associate introduce a notion of state to describe the evolution of TPNs.

Definition 2.11: A *state* s of a TPN is defined as a pair $\langle M, I\rangle$, where: $M : P \rightarrow \mathbb{N}$ is the current marking function; I is the set of firing intervals of the transitions that are enabled in the current state, described as $\hat{\alpha}_i \leq \tau_i \leq \hat{\beta}_i$. These intervals are relative to the instant at which the TPN reached the current state; $\hat{\alpha}_i$ and $\hat{\beta}_i$ are the earliest and the latest firing time, respectively.

Correspondingly, a suitable notion of transition can be introduced to represent TPN state changes. As a consequence, the analysis technique of Section II-B can be applied to TPNs, with minimal modifications.

III. Case Study

In this section we employ the proposed technique to analyze the behavior of a TPN, modeling the implementation of an active braking control system. The idea is that, once the structural properties of the closed-loop system have been formally proved via control theory methods, before moving to the industrialization of such control system several other issues need to be taken into account. In our case, the control code would be sufficiently simple to guarantee that, once the controller is turned into control code, no structural problems, such as deadlocks, occur. In more complex cases this implementation step could be validated through techniques such as Model Checking. Then, it is necessary to fix the hardware lay-out of the Electronic Control Unit (ECU) which hosts the control code, and to design the whole vehicle network, *i.e.*, the physical connection between the sensors, the ECU itself and the actuators. Usually, in Automotive applications, a CAN bus is used for signal transmission.

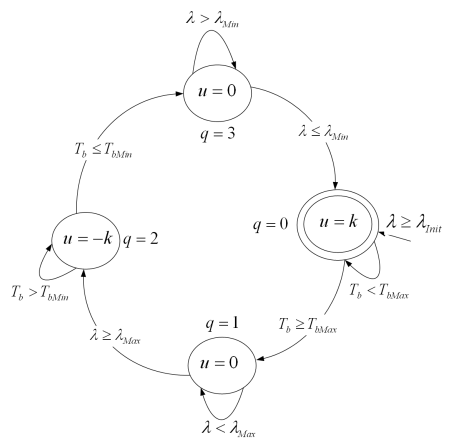
Note that, when a controller has been proved to ensure stability and robustness of the closed-loop system in a control-theoretic sense, these properties hold with the implicit assumption that the inputs and outputs of the control system are available synchronously at each sampling time, and that the controller can performs all the computations in a *nominal* predefined time interval. Thus, as both signal transmission and ECU processing can only be quantified to happen within a certain time interval, the TPN-based formal verification can provide crucial information on all the possible system behaviors. First of all it provides the equivalence classes to which the *nominal* behavior belongs. By looking at these classes in terms of upper-bounds on the time intervals, the designer is given a formal indication which may guide hardware sizing. If cost-constraints are tight, as it is usually the case in Automotive applications, the idea is to seek for the cheapest hardware which is capable of guaranteeing the needed performance level. Further, it is important to ensure that, when the control code routine starts, all the data provided as inputs refer to the same measurement interval (*input* consistency) and that the output sent to the actuators is also read with correct timing (*output* consistency).

A. Application Description

The considered application is an ABS controller based on a Hydraulic Braking System (HAB) actuator, which is the standard braking system on most commercial cars. ABS controllers are indeed a safety critical application, as they have to take care of managing panic-brakes which cannot be handled by common drivers. The design of an appropriate controller for these systems has been presented in [12], where it has been shown that the closed loop system has the desired stability properties.

System and Actuator Model. The braking dynamics have been modeled based on a quarter car model, [12]. The considered actuator is an HAB, which, according to its physical characteristics, is only capable of providing three control actions: *Increase* the brake pressure; *Hold* the brake pressure and *Decrease* the brake pressure. A static brake-pads friction model is assumed, *i.e.*, the braking torque T_b is computed from the measured brake pressure p_b as $T_b = r_d\, \chi\, A\, p_b$, where r_d is the brake disk radius; χ is the (constant) brake pad friction coefficient; A is the brake piston area and p_b is the measured brake pressure. The *increase* and *decrease* pressure actions are physically limited by the actuator rate limit, *i.e.*, $\frac{dT_b}{dt} = k$, where the rate limit $k \in \mathbb{R}^+$ is a known parameter. Accordingly, the controller design is based on the hybrid system made of the connection between the wheel dynamics and the hydraulic actuator, namely

$$\dot{\lambda} = -\frac{1-\lambda}{J\omega}\left(\Psi(\lambda) - T_b\right); \quad \dot{T}_b = u, \tag{1}$$



$\lambda > \lambda_{Min}$
$u = 0$
$q = 3$
$\lambda \le \lambda_{Min}$
$T_b \le T_{bMin}$
$q = 0$
$u = k$
$\lambda \ge \lambda_{Init}$
$u = -k$
$q = 2$
$T_b > T_{bMin}$
$T_b < T_{bMax}$
$q = 1$
$T_b \ge T_{bMax}$
$\lambda \ge \lambda_{Max}$
$u = 0$
$\lambda < \lambda_{Max}$

Fig. 2. Finite State Machine description of the hybrid controller.

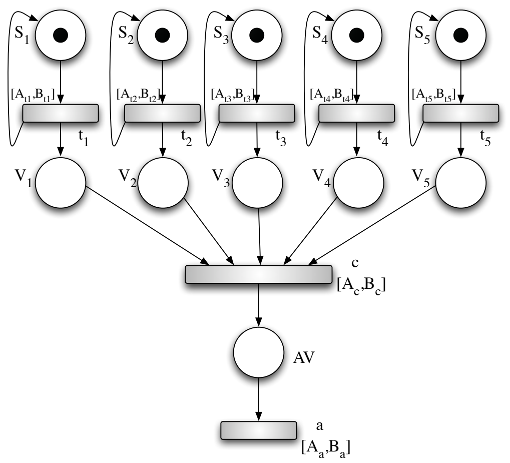


Fig. 3. TPN modeling the case study.

where $u = \{-k, 0, k\}$ and $\Psi(\lambda)$ is given by $\Psi(\lambda) = [r+(J/rm)(1-\lambda)]F_z\mu(\lambda)$, where λ is the longitudinal slip, which—during braking— is defined as $\lambda = (v-\omega r)/v$, $\lambda \in [0,1]$, $\omega\,[rad/s]$ is the angular speed of the wheel, $v\,[m/s]$ is the longitudinal speed of the vehicle body, the function $\mu(\lambda)$ describes the tire-road friction conditions, $T_b\,[Nm]$ is the braking torque, $F_z\,[N]$ is the vertical load, $J\,[kg\,m^2]$, $m\,[kg]$ and $r\,[m]$ are the moment of inertia of the wheel, the quarter-car mass, and the wheel radius, respectively. Note that $u = -k$ corresponds to the *Decrease* control action, $u = 0$ to the *Hold* control action and $u = k$ to the *Increase* control action. As shown in [12], a controller which guarantees closed-loop stability of the braking system is that shown in Figure 2. By analyzing Figure 2 one can see that we employ the braking torque T_b and the wheel slip λ as switching variables and that the switching conditions change according to the current discrete controller state $q = \{0,1,2,3\}$. The thresholds values T_{bMin}, T_{bMax}, λ_{Min}, and λ_{Max} which enable the transitions between the different states are chosen by the designer to guarantee a good trade-off between performance and safety in all driving conditions.

B. Formalization and Analysis

Let us now briefly describe the hardware/software lay out in the final vehicle implementation. The vehicle is equipped with 5 sensors, *i.e.*, 4 encoders to measure the wheel speeds, by means of which the wheel slip λ is obtained, and a pressure sensor to compute the braking torque T_b. By processing such sensor measurements, the control algorithm can be executed. As far as computational resources are concerned, the vehicle is equipped with an ECU which processes sensor outputs and executes the controller routine. Finally, the controller output is sent to the HAB which actuates it. Signal communication is managed by the vehicle bus. As for nominal performance — stated in terms of timing of the different actions — generally on car networks the CAN bus works with a nominal time interval which is either of 5 or 10 ms (100-200 Hz of sampling frequency), and the braking controller should complete the computations accordingly.

TPN Model. Considering the description of the hardware/software lay out of the vehicle and of the controller, the introduced case study can be modeled using the TPN shown in Figure 3. The TPN is composed of 11 places and 7 transitions. Each place labeled with S_i represents a sensor and a marking in S_i means that the correspondent sensor has just collected a datum. Each place S_i is connected with a transition t_i, enabled with a firing interval $[A_{t_i}, B_{t_i}]$, which models the datum transmission to the control system; once the datum is collected, t_i must fire after A_{t_i} time units and within B_{t_i} time units. When t_i fires, a fresh token is placed both in place S_i and in place V_i. Notice that we do not distinguish among different tokens. Thus, if a token is already present in V_i, once t_i fires and a new token reaches V_i, the two tokens in V_i are undistinguishable.

Transition c represents the control routine computation and this is why it needs a token in all V_i's to be enabled. The firing of c generates a token in place AV, meaning that the actuation value is ready and will be actuated when transition a fires. The static firing intervals are treated as symbolic variables where, for each transition k, $A_k \le B_k$.

TPN Analysis. As we cannot distinguish among tokens in a place according to their arrival time, the temporal behavior of the braking system components influences the correctness of its implementation. We now carry out a guided analysis of such correctness features by discussing the results of various runs of our enumeration algorithm on the presented TPN model. In what follows, all time units are in milliseconds, unless otherwise specified. Consider for example the following values for the marking function $M(V_1) = M(V_2) = M(V_3) = 1$, $M(V_4) = 1$, $M(V_5) = 2$ and M equal to 0 for all the other places. In this case, when transition c fires, it consumes the single tokens in V_1, V_2, V_3 and V_4, and one of the two tokens in V_5. However, these tokens correspond to two different data acquisitions, whereas the calculation performed by c would be coherent only if using the more recent one. As such, there is a need to introduce requirements on the firing intervals so that the following properties are met by all evolutions: (1) once a datum is collected, the control action is either performed or the datum is discarded and the actuation is eventually performed within a fixed amount of time; (2) the calculation of the value to be actuated is performed on coherent data (*input consistency*) and using the most recent ones.

A very simplified hypothesis that meets these requirements considers only deterministic firing intervals. Determinism is

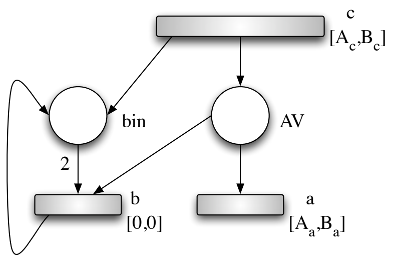


Fig. 4. A part of the TPN modified to implement the bin technique.

obtained by defining, for each interval, the lower bound equal to the upper bound. If, moreover, we let all sensors intervals to be equal, and set $A_{t_i} \geq A_c + A_a$, that is we assume sensors produce data more slowly than the whole sequential process of computing and performing the actuation, we can see that practically there is only one equivalence class where all the sensors fire at the same time instant.

However, while the hypothesis of identical sensors is realistic, that of determinism is not, as it implies that all the elements of the system are flawless instruments that always perform with exact timing. Hence, consider the case in which all the sensors and actuators behave equally with $[8.5,\ 10]$ as firing interval (the CAN bus works nominally at 100 Hz), and the computation (transition c) occurs within the time interval $[4,\ 5]$ (the control routine is assigned a nominal time slice of 5 ms by the operating system). We find NT-traces of length 8 such as $\langle t_1, t_2, t_3, t_4, t_5, c, t_1, t_2\rangle$, consuming from 17 to 18 time units overall. The absolute firing intervals for these example traces are: $[8.5,\ 9.5]$ (for t_1), $[8.5,\ 9.5]$ (for t_2), $[8.5,\ 10]$ (for t_3), $[8.5,\ 10]$ (for t_4), $[9,\ 10]$ (for t_5), $[14,\ 15]$ (for c), $[17,\ 18]$ (for the second occurrence of t_1), and $[17,\ 18]$ (for the second occurrence of t_2). In these cases, we have to guarantee that the accumulated extra (old) tokens are discarded. In terms of TPN, this solution can be modeled as shown in Figure 4. We call this approach *bin technique*, since the stale data is discarded through a new transition b which acts as a bin. This additional transition b is devoted to consuming the extra tokens in AV: the idea is to leave a enabled by one token in AV but, at the same time, additional tokens should be consumed by b as soon as possible. In fact, whenever a new token is produced by transition c, two copies of it simultaneously go in AV and bin. Then, the weight 2 on the arc that connects bin with b means that transition bin is enabled by, and consumes when it fires, exactly two tokens from bin. Since every token produced by c is in both bin and AV, b is enabled if and only if there are (at least) two tokens in both places bin and AV. When this is the case, b fires immediately, consumes two tokens from bin and one from AV, and puts one new token in bin. Overall, AV is left with one token and so is bin, so the two places always have the same number of tokens, and always contain at most two tokens. Hence, the bin technique guarantees a bound for the number of tokens in every place. Additional numeric experiments on the modified TPN show clearly that the desired properties of consistency are guaranteed in a system where $B_{t_i} = B_{t_j} = B_a > A_c$ for all $i, j = 1, \ldots, 5$, as long as we adopt the bin technique. This means that a realistic scenario, where: (1) the time needed to read data from sensors is roughly the same as that needed to send the control variable to the actuators, and (2) the time to compute one cycle of control action is smaller than the latter times, is compatible with a correct implementation of the control algorithm.

IV. Lessons Learned

Some remarks in the form of a critical summary of the overall analysis tools and of their expressiveness are in order. First of all it is worth noticing that, to the best of our knowledge, this is one of the first attempts to use formal methods (in the software-engineering sense) with the aim of supporting a real design step, rather than providing only yes/no answers on some specific properties of a system, or on the satisfaction of some requirements. Using the proposed tool, the information it provides may induce to reconsider the control design if certain timing constraints are proved to be more or less severe than others or if they need higher priority in the final system. Moreover, as it was shown by means of the case study analysis, it is rather straightforward to exploit the information provided by the analysis tool as a guide to hardware sizing. Section III revealed, in fact, that for the considered application the nominal controller behavior can be guaranteed by choosing an ECU which guarantees that the control routine executes in a shorter time with respect to the sensor and actuators signal transmission. This would suggest that more resources should be devoted to the microprocessor selection, while a cheaper (hence slower) bus can be selected. Of course, the final aim is that such tools can find place in industrial practice. To this end, a crucial step will be to work on the development of software interfaces which can provide the possibility of performing formal analysis of the control code at a high level of abstraction.

References

[1] M. Kwiatkowska, G. Norman, and D. Parker, “Controller dependability analysis by probabilistic model checking,” *Control Engineering Practice*, vol. 15, pp. 1427–1434, 2007.

[2] T. Johnson, “Improving automation software dependability: A role for formal methods?” *Control Engineering Practice*, vol. 15, pp. 1403–1415, 2007.

[3] M. Lawford and W. M. Wonham, “Equivalence preserving transformations for timed transition models,” *IEEE Transactions on Automatic Control*, vol. 40, no. 7, pp. 1167–1179, 1995.

[4] E. M. Clarke and J. M. Wing, “Formal methods: State of the art and future directions,” *ACM Computing Surveys*, vol. 28, no. 4, pp. 626–643, 1996.

[5] C. Heitmeier and D. Mandrioli, Eds., *Formal Methods for Real-Time Computing*. John Wiley & Sons, 1996.

[6] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, “Symbolic model checking for real-time systems,” *Information and Computation*, vol. 111, no. 2, pp. 193–244, 1994.

[7] L. Mangeruca, M. Baleani, A. Ferrari, and A. Sangiovanni-Vincentelli, “Semantics-preserving design of embedded control software from synchronous model,” *IEEE Transactions on Software Engineering*, vol. 33, no. 8, pp. 497–509, 2007.

[8] S. Cigoli, P. Leblanc, S. Malaponti, D. Mandrioli, M. Mazzucchelli, A. Morzenti, and P. Spoletini, “An experiment in applying UML 2.0 to the development of an industrial critical application,” in *Proceedings of CSDUML’03*, 2003.

[9] B. Berthomieu and M. Diaz, “Modeling and verification of time dependent systems using time Petri nets,” *IEEE Transactions on Software Engineering*, vol. 17, no. 3, pp. 259–273, 1991.

[10] P. Merlin and D. Farber, “Recoverability of communication protocols – implications of a theoretical study,” *IEEE Transactions on Communications*, pp. 1036–1043, 1976.

[11] W. Reisig, *Petri Nets: An Introduction*, ser. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1985.

[12] M. Tanelli, G. Osorio, M. di Bernardo, S. Savaresi, and A. Astolfi, “Limit cycles analysis in hybrid anti-lock braking systems,” in *Proceedings of the 46th Conference on Decision and Control, CDC 2007, New Orleans, Louisiana, USA*, 2007, pp. 3865–3870.