in *Proc. ISSS*, 2000, pp. 149-155.

Verification of Embedded Systems using a Petri Net based Representation

Luis Alejandro Cortés, Petru Eles, and Zebo Peng
Dept. of Computer and Information Science
Linköping University, Linköping, Sweden
{luico,petel,zebpe}@ida.liu.se

Abstract

The ever increasing complexity of embedded systems consisting of hardware and software components poses a challenge in verifying their correctness. New verification methods that overcome the limitations of traditional techniques and, at the same time, are suitable for hardware/software systems are needed. In this work we formally define the semantics of PRES+, a Petri net based computational model aimed to represent embedded systems. We introduce an approach to formal verification of such systems: we make use of model checking to prove the correctness of embedded systems by determining the truth of CTL and TCTL formulas that specify required properties with respect to a PRES+ model. An ATM server illustrates the feasibility of our approach on practical applications.

1. Introduction

Modern electronic systems are typically constituted of application-specific hardware components and software running on programmable platforms. The inherent heterogeneity of this kind of systems makes them very complex and consequently difficult to verify. Moreover, the increasing demand on high-performance products has boosted the levels of sophistication of such systems.

For the levels of complexity typical to modern electronic systems, traditional validation techniques like simulation and testing are neither sufficient nor viable to verify their correctness. First, these techniques may cover just a small fraction of the system behavior. Second, long simulation times and bugs found late in prototyping phases have a negative impact on time-to-market. Formal methods are becoming a practical alternative to ensure the correctness of designs. They might overcome some of the limitations of traditional validation methods. At the same time, formal verification can give a better understanding of the system behavior, contributes to uncover ambiguities, and reveals new insights of the system.

Formal methods have been extensively used in software development [8] and hardware verification [12]. However, they are not commonplace in embedded systems design. In this paper we present an approach to verification using model checking for embedded systems represented in PRES+, a notation capable of capturing relevant information characteristic to such systems. We introduce a systematic procedure to translate PRES+ models into linear hybrid automata in order to use existing model checking tools.

Model checking is an approach to formal verification that lets the designer prove whether certain design properties hold in a given model of the system. Our approach allows to determine the truth of CTL (Computation Tree Logic) [5] and TCTL (Timed CTL) [1] formulas with respect to a PRES+ model. Thus it is possible to validate design properties including timing requirements.

The rest of this paper is organized as follows. Section 2 addresses related approaches to formal methods suitable for embedded systems. The underlying computational model that we use to represent such systems is formally defined in Section 3. Our approach to verification of embedded systems is presented in Section 4. In Section 5 we illustrate our verification method using a real-life telecom system. Finally, some conclusions are drawn in Section 6.

2. Related Work

Many models have been proposed to represent HW/SW systems. Particularly, Petri nets (PNs) have been extended to model such systems. Maciel *et. al* [13] introduce an intermediate model for hardware/software codesign, extending Petri nets to analyze certain properties used in the partitioning process. Stoy [15] presents a modeling technique where timed Petri nets with restricted transition rules are used to represent control flow in both hardware and software.

Though formal methods are not commonplace in hardware/software codesign, some coverification approaches have been proposed recently. Using the hybrid automata model, a coverification method is proposed in [10] where complex systems can be analyzed using a simplification strategy to verify individually the hardware, the software, and the interface. Balarin *et. al* [4] introduce a verification methodology based on Codesign Finite State Machines (CFSMs) in which CFSMs are translated into traditional state automata in order to check whether all possible sequences of inputs and outputs satisfy the system properties. In [9], a partitioned system is the input to the proposed coverification framework in which CTL and TCTL formulas are evaluated in order to check behavioral and timing properties. An approach to model checking of process networks

This research is sponsored by the Swedish National Board for Industrial and Technical Development (NUTEK) in the frame of the SAVE project.

is proposed in [16], where IDDs (Interval Decision Diagrams) are used to represent multi-valued functions.

Related approaches to formal verification using PNs include [17], which presents a BDD-based model checker for safe nets. An interesting approach to analysis and verification of bounded Petri nets is presented in [14], where BDDs are used to represent sets of markings.

3. PRES+

The notation we use to model embedded systems is PRES+ (Petri net based Representation for Embedded Systems). PRES+ is a slightly modified version of the model presented in [6]. It is a computational model based on Petri nets that allows to capture important features of embedded systems. Figure 1 shows a simple example that will help to illustrate the definition of this representation. In what follows we introduce the formal definition of PRES+.

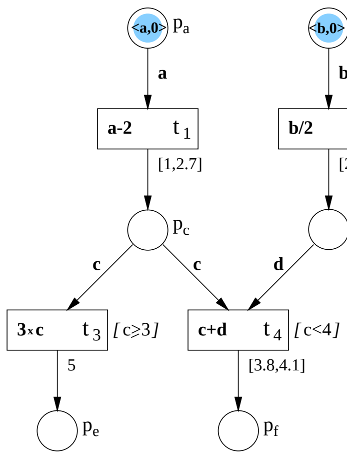

Figure 1. A PRES+ model

Definition 1. A *PRES+* model is a five-tuple $N = (P, T, I, O, M_0)$ where

$P = \{p_1, p_2, \ldots, p_m\}$ is a finite non-empty set of *places*;

$T = \{t_1, t_2, \ldots, t_n\}$ is a finite non-empty set of *transitions*;

$I \subseteq P \times T$ is a finite non-empty set of *input arcs* which define the flow relation between places and transitions;

$O \subseteq T \times P$ is a finite non-empty set of *output arcs* which define the flow relation between transitions and places;

M_0 is the initial *marking* of the net (see Definition 3). ■

Like in classical Petri nets, places are graphically represented by circles, transitions by boxes, and arcs by arrows. For the example in Figure 1, $P = \{p_a, p_b, p_c, p_d, p_e, p_f\}$ and $T = \{t_1, t_2, t_3, t_4\}$.

Definition 2. A *token* is a pair $k = \langle v, r \rangle$ where

v is the *token value*. This value may be of any type, e.g. boolean, integer, etc., or user-defined type of any complexity (for instance a structure, a set, or a record). The type of this value is referred to as *token type*;

r is the *token time*, a non-negative real number representing the time stamp of the token.

The set K denotes the set of all possible token types for a given system. ■

Definition 3. A *marking* $M : P \to \{0, 1\}$ is a function that denotes the absence or presence of tokens in the places of the net. A PRES+ net N is *safe* or *1-bounded*, that is, a place may hold at most one token for a certain marking. $M(p) = 1$ whenever the place p is *marked*, otherwise $M(p) = 0$. ■

Note that a marking M implicitly assigns one token k to each marked place. We introduce the following notation which will be useful in defining the dynamic behavior of PRES+: when a place p_i is marked, k_i denotes the token present in p_i. The token value of the token k_i is denoted v_i, and the token time of the token k_i is denoted r_i. The initial marking M_0 in Figure 1 shows p_a and p_b as places initially marked. The token $k_a = \langle a, 0 \rangle$ has a value a and a time stamp 0. For the sake of simplicity, in the examples we use the short notation w to denote the token value v_w.

Definition 4. The *type function* $\tau : P \to K$ associates a place with a token type. $\tau(p)$ denotes the token type associated with the place p. The token type is the type of value that a token may bear in that place. ■

It is worth pointing out that the token type related to a certain place is fixed, that is, it is an intrinsic property of that place and will not change during the dynamic behavior of the net. For the example in Figure 1, all places have token type *real*.

Definition 5. The *pre-set* $^{\circ}t = \{p \in P | (p, t) \in I\}$ of a transition t is the set of *input places* of t. Similarly, the *post-set* $t^{\circ} = \{p \in P | (t, p) \in O\}$ of a transition t is the set of *output places* of t.

Correspondingly, the *pre-set* $^{\circ}p$ and the *post-set* p° of a place p are given by $^{\circ}p = \{t \in T | (t, p) \in O\}$ and $p^{\circ} = \{t \in T | (p, t) \in I\}$. ■

Definition 6. All output places of a given transition have the same token type,

$$\textit{if } p, q \in t^{\circ} \Rightarrow \tau(p) = \tau(q) \quad ■$$

Definition 7. For every transition t, there exists a *transition function* f associated to t. Formally,

$$\forall t \in T \ \exists f : \tau(p_1) \times \tau(p_2) \times \ldots \times \tau(p_a) \to \tau(q)$$

where $^{\circ}t = \{p_1, p_2, \ldots, p_a\}$ and $q \in t^{\circ}$. ■

Transition functions are very important when describing the behavior of the system to be modeled. They allow systems to be modeled at different levels of granularity with transitions representing simple arithmetic operations or complex algorithms. In Figure 1 we inscribe transition functions inside transition boxes: the transition function associated to t_4, for example, is given by $f_4(c, d) = c + d$. We use inscriptions on the input arcs of a transition in order to denote the arguments of its transition function and/or its guard.

Definition 8. For every transition t, there exist a *minimum transition delay* d^- and a *maximum transition delay* d^+, which are non-negative real numbers and represent, respectively, the lower and upper limits for the execution time (delay) of the function associated to the transition. Formally,

$$\forall t \in T \;\; \exists\, d^-, d^+ \in \Re_0^+ \text{ such that } d^- \le d^+$$

with $\Re_0^+$ being the set of non-negative real numbers. ■

Referring again to Figure 1, the minimum transition delay d_1^- of t_1 is 1, and its maximum transition delay d_1^+ is 2.7 time units. Note that when $d^-=d^+=d$, we just inscribe the value d close to the respective transition, like in the case of the transition delay $d_3 = 5$.

Definition 9. The *guard G* of a transition *t* is the (necessary) condition that must be satisfied in order to enable that transition, when all its input places hold tokens. The guard

$$G : \tau(p_1) \times \tau(p_2) \times \ldots \times \tau(p_a) \to \{0, 1\}$$

of a transition *t* is a function of the token values in the places of its pre-set $°t = \{p_1, p_2, \ldots, p_a\}$. If the condition holds $G = 1$, otherwise $G = 0$. ■

For instance, in Figure 1, $c < 4$ represents the guard G_4.

Definition 10. Every transition has a *functionality*. The functionality of a transition *t* is defined in terms of:

(i) Its *transition function* f;

(ii) Its *minimum* and *maximum transition delays* d^- and d^+. ■

Unlike the classical Petri net model, each token holds a value and a time stamp. When a transition *t* is fired, the marking *M* will generally change by removing all the tokens from the pre-set $°t$ and depositing one token into each element of the post-set $t°$. These tokens, added to $t°$, have values and time stamps which depend on the previous tokens in $°t$ and the functionality of *t*.

Definition 11. A transition *t* is said to be *enabled* if all places of its pre-set are marked, its output places different from the input ones[1] are empty, and its guard is asserted. Formally, for a given marking *M*, a transition t is enabled *iff*

(i) $\forall p \in °t \;\; M(p) = 1$

(ii) $\forall q \in (t° - °t) \;\; M(q) = 0$

(iii) $G = 1$ ■

Definition 12. Every enabled transition *t* has an *enabling time et* that represents the time instant at which the transition becomes enabled. The enabling time *et* of a (enabled) transition is the maximum token time of the tokens in its input places,

$$et = max(r_1, r_2, \ldots, r_a)$$

where the pre-set of *t* is $°t = \{p_1, p_2, \ldots, p_a\}$. ■

Definition 13. The *earliest trigger time* tt^- and the *latest trigger time* tt^+ of an enabled transition are the lower and upper time bounds for the firing of the transition,

$$tt^- = et + d^-$$
$$tt^+ = et + d^+$$

An enabled transition *t* may not fire before its earliest trigger time tt^- and must fire before or at its latest trigger time tt^+, unless *t* becomes disabled by the firing of another transition. ■

Definition 14. The *firing* of an enabled transition changes a marking *M* into a new marking M^+. As a result of firing the transition *t*, with pre-set $°t = \{p_1, p_2, \ldots, p_a\}$, the following events occur:

[1] A place may be, at the same time, input and output of a transition.

(i) Tokens from its pre-set (which are not in its post-set) are removed;

$$\forall p \in (°t - t°) \;\; M^+(p) = 0$$

(ii) One token is added to each place of its post-set;

$$\forall q \in t° \;\; M^+(q) = 1$$

(iii) Each new token deposited in $t°$ has a token value, which is calculated by evaluating the transition function with the token values of tokens in $°t$ as arguments;

$$\forall q_i \in t° \quad v_i = f(v_1, v_2, \ldots, v_a)$$

(iv) Each new token added to $t°$ has a token time, that is the time instant at which the transition *t* fires;

$$\forall q_i \in t° \quad r_i = tt^* \text{ where } tt^* \in [tt^-, tt^+]$$ ■

The execution time of the function of a transition is considered in the time stamp of the new tokens. Note that, when a transition fires, all the tokens in its output places get the same token value and token time. The token time of a token represents the time at which it was "created".

When used to model embedded systems, the representation introduced above has several interesting features to be highlighted, some of them inherited from the classical Petri net model:

- Non-determinism may be naturally represented by PRES+. Non-determinism can be used as a powerful mechanism to express succinctly the behavior of certain systems and then reduce the complexity of the model.
- Parallel or concurrent activities may be easily expressed in terms of Petri nets. We recall that concurrency is present in most embedded systems.
- Since tokens carry information in our model, PRES+ overcomes the lack of expressiveness of classical Petri nets, where tokens are considered as "black dots".
- Time is a critical factor in many embedded applications. Our model captures timing aspects by associating lower and upper limits to the duration of activities related to transitions and keeping time information in token stamps.
- PRES+ has been also extended by introducing the concept of hierarchy. However, we will not further discuss this particular feature in this paper.

Summarizing, PRES+ is a model to be used in the design cycle of embedded systems. Our representation is an extension to the classical PN model that overcomes some of the drawbacks of Petri nets when modeling embedded systems: it captures explicitly timing information; PRES+ allows representations at different levels of granularity; our model is more expressive since tokens might carry information. Furthermore, the model is simple, intuitive, and can be easily handled by the designer.

4. Verification of Embedded Systems

In this section we present a verification method for systems represented using the model introduced above. The purpose of the approach presented in this paper is to reason about embedded systems using PRES+ as underlying representation. There are several types of analysis that can be performed on systems represented in PRES+. A given marking, i.e. absence or presence of tokens in places of the net, may represent the state of the system in the dynamic behavior of the net. Based on this, different properties can be studied. For instance, the designer could be interested in proving that the system eventually reaches a certain state whose marking represents the completion of a task.

The kind of analysis described above, called *reachability analysis*, is very useful but says nothing about timing aspects nor does it deal with token values. In many embedded applications, however, time is an essential factor. Moreover, in hard real-time systems, where deadlines should not be missed, it is crucial to reason quantitatively about temporal properties in order to ensure the correctness of the design. Therefore, it is needed not only to check that a certain state will eventually be reached but also to ensure that this will occur within some bound on time. In PRES+, time information is attached to tokens so that we can analyze quantitative timing properties: we may prove that a given place will be eventually marked and that its time stamp will be less than a certain time value that represents a temporal constraint. Such a study will be called *time analysis*.

A third type of analysis for systems modeled in PRES+ involves reasoning about values of tokens in marked places. This type of study is called *behavior analysis*. In this work we restrict ourselves to reachability and time analyses. In other words, we concentrate on the absence/presence of tokens in the places of the net and their time stamps. Note, however, that in some cases reachability and time analyses are influenced by token values. This aspect is discussed at the end of this section.

To verify the correctness of an embedded system, we first translate its PRES+ model into equivalent linear hybrid automata and then use existing verification tools, namely HyTech [11], to check properties expressed as CTL and TCTL formulas. CTL [5] is based on propositional logic of branching time. Formulas in CTL are composed of atomic propositions, boolean connectors and temporal operators. Temporal operators consist of forward-time operators (**G** globally, **F** in the future, **X** next time, and **U** until) preceded by a path quantifier (**A** all computation paths, and **E** some computation path). TCTL [1] is a real-time extension of CTL that allows to inscribe subscripts on the temporal operators to limit their scope in time. For instance, $\mathbf{AF}_{<\mathrm{n}}\, p$ expresses that, along all computation paths, the property p becomes true within n time units.

4.1. Hybrid Automata

In what follows we shortly describe the *hybrid automata* model. The reader is referred to [2], [3] for further reading on hybrid automata theory. Informally, a linear hybrid automaton is a finite automaton enhanced with a set of real-valued variables, where the terms involved in conditions, assignments, and invariants are required to be linear.

A hybrid automaton $H = (X, V, E, L, sync, cond, act, inv)$ consists of the following components:

- A finite set X of real-valued *variables*. A *valuation* υ is a function that assigns a real-value $\upsilon(x)$ to every variable $x \in X$.
- A finite set of *locations* or *vertices* V. A *state* $s = (v, \upsilon)$ consists of a location v and a valuation υ.
- A finite set E of *edges*. Each edge $e = (v, v')$ consists of a source location $v \in V$ and a target location $v' \in V$.
- A finite set of *synchronization labels* L and a labeling function *sync* that assigns to each edge $e \in E$ a synchronization label $l \in L$, noted as $l = sync(e)$.
- A labeling function *cond* that assigns to each edge $e = (v, v') \in E$ a *condition* $cond(e)$ that must be satisfied in order to allow the automaton H to change its location from v to v'.
- A labeling function *act* that assigns to each edge $e \in E$ a set of *activities*.
- A labeling function *inv* that assigns to each location $v \in V$ an *invariant* $inv(v)$ which allows the automaton H to stay at location v as long as its invariant $inv(v)$ is satisfied.

4.2. Translating PRES+ into Hybrid Automata

In what follows we describe the systematic procedure to translate PRES+ models into linear hybrid automata. The resulting representation consists of a collection of automata which operate and coordinate with each other through shared variables and synchronization labels. Figure 2 shows the result of translating the PRES+ model of Figure 1 into hybrid automata. We will use this example as reference to illustrate the proposed translation method.

Step 1. Define one variable in X for each place p_x of the Petri net, corresponding to the token value v_x when p_x is marked, and one variable c_i for each transition t_i, which represents a clock used to ensure the firing of the transition within the earliest-latest trigger time interval. Thus $X = \{v_1, v_2, \ldots, v_m, c_1, c_2, \ldots, c_n\}$[2]. ■

For the example in Figure 2, using the short notation w to denote v_w, $X = \{a, b, c, d, e, f, c_1, c_2, c_3, c_4\}$.

Step 2. Define the set L of synchronization labels as the set (of names) of transitions in the Petri net, that is $L = \{t_1, t_2, \ldots, t_n\}$. ■

Step 3. For every transition $t \in T$ define a hybrid automaton $\tilde{t}$ with z+1 locations $dis_0, dis_1, \ldots, dis_{z-1}, en$, where z is the number of transitions that, when fired, will deposit a token in some place in the pre-set $^\circ t$. The set of such transitions is defined by

$$pre(t) = \bigcup_{p_i \in {}^\circ t} {}^\circ p_i$$

In the case $pre(t) = \varnothing$, define an automaton with only two locations dis_0 and en. ■

Step 4a. Given the automaton $\tilde{t}$, define z edges (dis_0, dis_1), z edges (dis_1, dis_2), $\ldots$, and z edges (dis_{z-1}, en), and

[2] In the linear hybrid automata model, variables may change along the time with a constant rate: for every $c_i \in X$ the change rate must be $\dot{c}_i = 1$; for every $v_x \in X$ the change rate must be $\dot{v}_x = 0$.

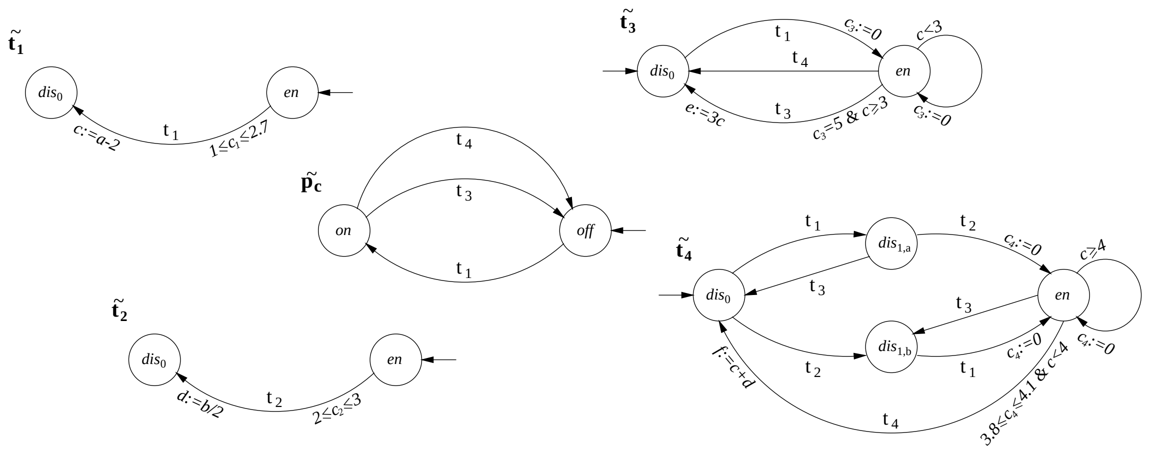

Figure 2. Equivalent hybrid automata

then assign to each group of z edges synchronization labels corresponding to the transitions in $pre(t)$. Define then one edge (en, dis_0) with synchronization label t. ■

Step 4b. If the transition t is in conflict with another transition tc (t could be disabled by the firing of tc):

Let $A = pre(t) \cap pre(tc)$ and $B = pre(t) - pre(tc)$. In the automaton $\tilde{t}$ remove all the edges except (en, dis_0). Split each one of the locations $dis_1, \ldots, dis_{z-1}$ into $dis_{1,a}, \ldots, dis_{z-1,a}$ and $dis_{1,b}, \ldots, dis_{z-1,b}$. Then define edges $(dis_{1,a}, dis_{2,a})$, $(dis_{2,a}, dis_{3,a})$, … , $(dis_{z-1,a}, en)$, $(dis_0, dis_{1,b})$, $(dis_{1,b}, dis_{2,b})$, … , $(dis_{z-2,b}, dis_{z-1,b})$ with synchronization labels corresponding to those transitions in B. Define edges $(dis_0, dis_{1,a})$, $(dis_{1,b}, dis_{2,a})$, … , $(dis_{z-1,b}, en)$ with synchronization labels corresponding to transitions in A. Define then edges $(dis_{1,a}, dis_0)$, $(dis_{2,a}, dis_{1,b})$, … , $(en, dis_{z-1,b})$ with synchronization label tc. ■

For instance, were t_4 not in conflict with t_3, the automaton $\tilde{t}_4$ would have the locations dis_0, dis_1, en and look like the one shown in Figure 3(a). However, because of such a conflict, Step 4b must be performed: location dis_1 has been split into the locations $dis_{1,a}$ and $dis_{1,b}$. Note that $pre(t_4) \cap pre(t_3) = \{t_1\}$ and $pre(t_4) - pre(t_3) = \{t_2\}$. The edges $(dis_{1,a}, en)$ and $(dis_0, dis_{1,b})$ with synchronization label t_2 are defined. Then the edges $(dis_0, dis_{1,a})$ and $(dis_{1,b}, en)$ with synchronization label t_1 are defined. Finally, the edges $(dis_{1,a}, dis_0)$ and $(en, dis_{1,b})$ with synchronization label t_3 are defined. After Step 4b, we get the automaton $\tilde{t}_4$ shown in Figure 3(b).

It is easy to observe that t_3 is in conflict with t_4. No locations, however, are split in the automaton $\tilde{t}_3$ because it has just two locations (dis_0 and en). In this case only the edge (en, dis_0) with synchronization label t_4 must be added.

Let f_i be the transition function associated to t_i, ${}^\circ t_i = \{q_1, q_2, \ldots, q_a\}$ the pre-set of t_i, and d_i^- and d_i^+ the minimum and maximum transition delays associated to t_i.

Step 5. Given the automaton $\tilde{t}_i$, assign to every edge (dis_j, en) the activity $c_i := 0$. Define the invariant of location en as $c_i \leq d_i^+$ in order to enforce the firing of t_i before or at its latest trigger time. ■

In Figure 2, the edge (dis_0, en) of automaton $\tilde{t}_3$, for example, has the activity $c_3 := 0$. c_3 is used to take into account the time since t_3 becomes enabled and ensure the firing semantics of PRES+. For the sake of clarity, location invariants are not shown in Figure 2.

Step 6. Given $\tilde{t}_i$ and its edge $e_i = (en, dis_0)$, assign to e_i the condition $d_i^- \leq c_i \leq d_i^+$. For every $p_j \in t_i^\circ$ assign to such an edge e_i the activity $v_j := f_i(v_1, v_2, \ldots, v_a)$. ■

For example, in the case of the automaton $\tilde{t}_1$ the condition $1 \leq c_1 \leq 2.7$ gives the lower and upper limits for the firing of t_1, while the activity $c := a - 2$ expresses that whenever the automaton changes from en to dis_0, i.e. t_1 fires, the value $a - 2$ is assigned to the variable c.

Step 7. If t_i is a transition with guard G_i, assign the condition G_i to the edge (en, dis_0) of the automaton $\tilde{t}_i$ (so such an edge condition becomes $d_i^- \leq c_i \leq d_i^+$ & G_i). Then add an edge (en, en) with no synchronization label, condition $\overline{G_i}$ (the complement of G_i), and activity $c_i := 0$. ■

Note the condition $3.8 \leq c_4 \leq 4.1$ & $c < 4$ in the automaton $\tilde{t}_4$ where $c < 4$ is the guard of t_4. Observe also the edge (en, en) with condition $c \geq 4$ and activity $c_4 := 0$.

Step 8. For every place $p \in P$ define a hybrid automaton $\tilde{p}$ with two locations, *on* and *off*, corresponding to the marking $M(p)$. The initial location of $\tilde{p}$ will be either *on* or *off* depending whether the place p is initially marked or not. For every transition t_j in the post-set p° of the place p, define an edge $e_j = (on, off)$ with synchronization label t_j. For every transition t_i in the pre-set ${}^\circ p$ of the place p, define an edge $e_j = (off, on)$ with synchronization label t_i. ■

Note: When, given an automaton $\tilde{p}$, a transition t_i is both input and output of the place p, define an edge $e_i = (on, on)$ and assign to it a synchronization label t_i.

In Figure 2 we only show the automaton $\tilde{p}_c$. The other automata $\tilde{p}_a$, $\tilde{p}_b$, $\tilde{p}_d$, $\tilde{p}_e$, and $\tilde{p}_f$ are similar.

The procedure that we have described above is general enough to translate *any* PRES+ model in which transition functions are linear and token types of all places are real.

Some optimizations may be performed to reduce the complexity of the resulting hybrid automata. For instance, in most cases, the automata corresponding to places are redundant and could be removed. Such optimizations are beyond the scope of this paper and therefore not discussed here.

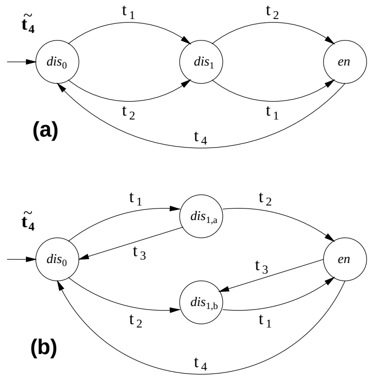

Figure 3. Illustration of Step 4

Once we have the equivalent hybrid automata, we can verify properties against the model of the system. For instance, in the simple system of Figure 1 we could check whether, for given values of *a* and *b*, there exists a reachable state in which p_e is marked. This property might be expressed as a CTL formula **EF** p_e. An interesting feature of the HyTech tool [11] is its ability to perform parametric analysis. Then, for example, we can ask the model-checker which values of *a* and *b* make a certain property true. We get that **EF** p_e holds if $a \geq 5$. If we want to check temporal properties we can express them as TCTL formulas. Thus, we could check whether p_f will be possibly marked and the time stamp of its token is less than 8.4 time units, expressing this property as $\mathbf{EF}_{<8.4}\ p_f$.

The translation procedure introduced above is valid for PRES+ models in which transition functions are linear and token types of all places are real. In this case, we could even reason about token values. Recall, however, that we want to focus on reachability and time analyses. From this perspective we can ignore transition functions if they affect neither the marking nor time stamps. This is the case of PRES+ models that bear no guards and, therefore, they can be straightforwardly verified even if their transition functions are very complex operations, because we simply ignore such functions. Those systems that include guards in their PRES+ model may also be studied if guard dependencies can be stated by linear expressions. This is the case of the system shown in Figure 1. There are many systems in which the transition functions are not linear, but their guard dependencies are, and then we can inscribe such dependencies as linear expressions and use our method for system verification.

5. Verification of an ATM Server

In this section we illustrate the verification of a practical system, modeled using PRES+. The net in Figure 4 represents an ATM-based Virtual Private Network (A-VPN) server [7] for a particular implementation. The behavior of the system can be briefly described as follows. Incoming cells are examined by *Check* to determine whether they are faulty. Fault-free cells arrive through the *UTOPIA_Rx* interface and are eventually stored in the *Shared Buffer*. If the incoming cell is faulty, it goes through the module *Faulty* and then is sent out using the *UTOPIA_Tx* interface without processing. The module *Address Lookup* checks the *Lookup Memory* and, for each non-defective input cell, a compressed form of the Virtual Channel (VC) identifier in the cell header is computed. With this compressed form of the VC identifier, the module *Traffic* checks its internal tables and decides whether to accept the incoming cell or discard it in order to avoid congestion. If the cell is accepted, *Traffic* gives instructions to *Queue Manager* indicating where to store the incoming cell in the buffer. *Traffic* also indicates to *Queue Manager* the cell (stored in *Shared Buffer*) to be output. *Supervisor* is the module in charge of updating internal tables of *Traffic* and the *Lookup Memory*. The selected outgoing cell is emitted through the module *UTOPIA_Tx*. The specification of the system includes a time constraint given by the rate (155 Mbit/s) of the application: one input cell and one output cell must be processed every 2.7 µs.

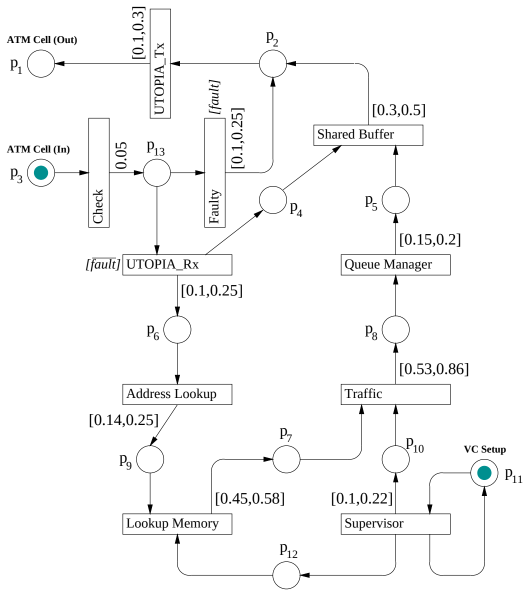

Figure 4. PRES+ model of an A-VPN server

To verify the correctness of the A-VPN server, we must prove that for all possible conditions the system will eventually complete its functionality, and that such a functionality will eventually fit within a cell time-slot. The completion of the task of the A-VPN server, modeled by the net in Figure 4, is represented by the state (marking) in which the place p_1 is marked. Then we must prove that for all computation paths, p_1 will eventually get a token and its time stamp will be less than 2.7 μs. These conditions might be straightforwardly specified using CTL and TCTL formulas, namely $\mathbf{AF}\ p_1$ and $\mathbf{AF}_{<2.7}\ p_1$. Notice that the first formula is a necessary condition for the second one. Using the translation procedure described above and the HyTech tool [11], we found out that the CTL formula $\mathbf{AF}\ p_1$ holds while the TCTL formula $\mathbf{AF}_{<2.7}\ p_1$ does not. Moreover, we have checked the formula $\mathbf{EF}_{<2.7}\ p_1$ that turned out to be true, which means that it is possible to get a token in p_1 with a time stamp less than 2.7 μs. However, recall that $\mathbf{AF}_{<2.7}\ p_1$ does not hold and therefore this implementation does not fulfill the system specification because it is not guaranteed that the time constraint will be satisfied.

We must consider an alternative solution. To do so, suppose we want to modify *Traffic*, keeping its functional behavior but seeking superior performance: we want to explore the allowed interval of delays for *Traffic* in order to fulfill the constraints. We can define the minimum and maximum transition delays of *Traffic* as parameters d^- and d^+, and then perform parametric analysis to find out the values for which $\mathbf{AF}_{<2.7}\ p_1$ is satisfied. We get that if $d^+ < 0.57$ and, by definition, $d^- \leq d^+$ then the property $\mathbf{AF}_{<2.7}\ p_1$ holds. This indicates that the worst case execution time of the function associated to *Traffic* must be less than 0.57 μs to fulfill the system specification.

Running the HyTech tool on a Sun Ultra 10 workstation, both the verification of the TCTL formula $\mathbf{AF}_{<2.7}\ p_1$ for the model given in Figure 4, and the parametric analysis described in the paragraph above take roughly 1 second.

The example of the ATM server described above has shown that our approach is not only suitable to verify the correctness of embedded systems but also that this technique can be a useful tool during design space exploration. Information, like the one obtained through parametric analysis, can guide the designer when exploring design alternatives. Thus, at the same time that we check the correctness of designs, we get valuable information that serves as guideline along the design process.

6. Conclusions

We have formally defined PRES+, a Petri net based model aimed to represent embedded systems. The model is simple, intuitive, and can be easily handled by the designer. It is a computational model with extensions to capture important characteristics of embedded systems.

We presented a practical approach to verification of embedded systems represented by PRES+. We use symbolic model checking to prove the correctness of such systems in respect to reachability and time, specifying design properties as CTL and TCTL formulas. Thus verification with timing properties is possible.

In order to use existing verification tools, we introduced a systematic procedure to translate PRES+ models into linear hybrid automata. This method can be automated in a relatively simple manner. An ATM server has been studied to illustrate the applicability of our verification approach to practical systems.

The approach presented in this work is not only appropriate to verify the correctness of embedded systems, but may also be a useful tool for design space exploration.

References

[1] R. Alur, C. Courcoubetis and D. L. Dill, "Model Checking for Real-Time Systems," in *Proc. Symposium on Logic in Computer Science*, 1990, pp. 414-425.

[2] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine, "The Algorithmic Analysis of Hybrid Systems," in *Theoretical Computer Science*, vol. 138, pp. 3-34, February 1995.

[3] R. Alur, T. A. Henzinger, and P.-H. Ho, "Automatic Symbolic Verification of Embedded Systems," in *IEEE Trans. Software Engineering*, vol. 22, pp. 181-201, March 1996.

[4] F. Balarin, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli, "Formal Verification of Embedded Systems based on CFSM Networks," in *Proc. DAC*, 1996, pp. 568-571.

[5] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," in *ACM Trans. on Programming Languages and Systems*, vol. 8, pp. 244-263, April 1986.

[6] L. A. Cortés, P. Eles, and Z. Peng, "A Petri Net based Model for Heterogeneous Embedded Systems," in *Proc. NORCHIP Conference*, 1999, pp. 248-255.

[7] E. Filippi, L. Lavagno, L. Licciardi, A. Montanaro, M. Paolini, R. Passerone, M. Sgroi, and A. Sangiovanni-Vincentelli, "Intellectual Property Re-use in Embedded System Co-design: an Industrial Case Study," in *Proc. ISSS*, 1998, pp. 37-42.

[8] J. D. Gannon, J. M. Purtilo, and M. V. Zelkowitz, *Software Specification: A Comparison of Formal Methods*. Norwood, NJ: Ablex Publishing, 1994.

[9] E. H. A. Garcez and W. Rosenstiel, "CVF - Coverification Framework," in *Proc. Brazilian Symposium on Integrated Circuit Design*, 1998, pp. 103-106.

[10] P.-A. Hsiung, "Hardware-Software Coverification of Concurrent Embedded Real-Time Systems," in *Proc. Euromicro RTS*, 1999, pp. 216-223.

[11] HyTech: The HYbrid TECHnology Tool, `http://www-cad.eecs.berkeley.edu/~tah/HyTech/`

[12] C. Kern and M. R. Greenstreet, "Formal Verification in Hardware Design: A Survey," in *ACM Trans. on Design Automation of Electronic Systems*, vol. 4, pp. 123-193, April 1999.

[13] P. Maciel, E. Barros, and W. Rosenstiel, "A Petri Net Model for Hardware/Software Codesign," in *Design Automation for Embedded Systems*, vol. 4, pp. 243-310, Oct. 1999.

[14] E. Pastor, O. Roig, J. Cortadella, and R. M. Badia, "Petri Net Analysis Using Boolean Manipulation," in *Application and Theory of Petri Nets 1994*, R. Valette, Ed. *LNCS 815*, Berlin: Springer-Verlag, 1994, pp. 416-435.

[15] E. Stoy and Z. Peng, "An Integrated Modelling Technique for Hardware/Software Systems," in *Proc. ISCAS*, 1994, pp. 399-402.

[16] K. Strehl and L. Thiele, "Symbolic Model Checking of Process Networks Using Interval Diagrams Techniques," in *Proc. ICCAD*, 1998, pp. 686-692.

[17] G. Wimmel, "A BDD-based Model Checker for the PEP Tool," Major Individual Project Report, Dept. of Computing Science, University of Newcastle, May 1997.