SAVE Project Report, Dept. of Computer and Information
Science, Linköping University, Sweden, January 2000.

Verification Methodology for Heterogeneous Hardware/Software Systems

Luis Alejandro Cortés, Petru Eles and Zebo Peng
Department of Computer and Information Science
Linköping University
S-581 83 Linköping, Sweden

Abstract

Modern electronic systems are constituted by heterogeneous elements, e.g. hardware/software, and are typically embedded. The complexity of this kind of systems is such, that traditional validation techniques, like simulation and testing, are not enough to verify the correctness of these systems. In consequence, new formal verification techniques that overcome the limitations of traditional validation methods and are suitable for hardware/software systems are needed. Formal methods require the system to be represented by a formal computational model with clear semantics. We present a Petri net based representation, called PRES, which is able to capture information relevant to embedded systems. This report also explores an approach to formal verification of embedded systems in which the underlying representation is PRES. We use symbolic model checking to prove the correctness of such systems, specifying properties in CTL and verifying whether they hold under all possible situations. This coverification method permits to reason formally about design properties as well as timing requirements. This work has been done in the frame of the SAVE project, which aims to study the specification and verification of heterogeneous electronic systems.

1. Introduction

In the coming years, most objects of common use will be based on electronic systems. The electronic market demands high-performance and low-cost products, and—for safety-critical applications—reliable components. Thus, the chip industry has faced the challenges of increasing the system complexity and reducing design times. In order to reduce time-to-market, designers tend to use programmable processors. When using programmable components, new specifications of evolving products can easily be adapted to previous designs. Though nowadays programmable processors are powerful, certain applications demand special features, in terms of performance, power consumption, and correctness, among others, so that some systems require the design of specific hardware. Still for those hardware components there exists a trend to re-use elements like intellectual property (IP) blocks.

Most modern electronic systems consist of dedicated hardware elements and software running on specific platforms. Such systems are obviously heterogeneous, i.e. are composed of elements with inherent distinct properties. For instance, they may contain microprocessors

(MPUs), digital signal processors (DSPs) and microcontrollers (MCUs), as well as application specific integrated circuits (ASICs) and field programmable gate arrays (FPGAs). At the same time, such systems are typically embedded, that is, they are part of larger systems and interact continuously with their environment. Hardware/software codesign, i.e. the concurrent design of mixed hardware/software systems, exploits the advantages of HW and SW working closely for a given task as long as they are considered as a whole instead of independent entities. Although benefits of hardware and software working together are evident, the design of such complex systems, involving both HW and SW components, is a non-trivial task.

We advocate design cycles based on formal models so that the synthesis of a design from specification to implementation can be carried out systematically. In order to devise systems that meet the performance, cost and reliability goals, the design process should be founded upon a clear representation that allows to accomplish the design cycle, based on formal notation. Modeling is an essential issue of any systematic design methodology. In this work, we propose a model suited to embedded systems. The model, called Petri net based Representation for Embedded Systems (PRES), is an extension to classical Petri nets. It explicitly captures time information, allows representations at different levels of granularity, and supports hierarchical decomposition. Another important feature of this model is its expressiveness since the tokens might carry information. Concurrency and sequential behavior are also captured by PRES. The above characteristics are very important when modeling embedded systems. As any other Petri net based model, PRES is inherently asynchronous.

The inherent heterogeneity of embedded systems makes them very complex and 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 the correctness of digital designs. 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. Formal verification can give a better understanding of the system behavior, help to uncover ambiguities and reveal new insights of the system.

Formal methods have been extensively used in software development [Gan94] and hardware verification as well [Ker99]. However, they are not commonplace in embedded systems design. There is a lack of techniques for formal verification of hardware/software systems. In this paper we also present an approach to verification of embedded systems using symbolic model checking, based on PRES. With this approach it is possible to validate properties of the system as well as timing requirements. Design properties are specified as CTL (Computation Tree Logic) formulas and the model checker determines whether they are satisfied.

2. HW/SW Codesign Representation Models

Many computational models have been proposed in the literature to represent digital systems. These models encompass a broad range of styles, characteristics and application domains. Particularly in the field of hardware/software codesign, a variety of models has been developed and used for system representation. Many different models coexist in the scenario of HW/SW codesign. Their features largely differ even though they all are computational models intended

for heterogeneous hardware/software systems. These computational representations have usually distinct characteristics and support diverse kinds of applications.

Many of the computational models used for hardware/software systems are based on extensions to finite-state machines, Petri nets, discrete-event systems, data-flow graphs, the so-called synchronous/reactive models, communicating processes, among others. We addressed several representative computational models used to capture hardware/software systems [Cor99]. The key aspects of these models are discussed and a comparison of their most relevant features is presented. Edwards *et al*. [Edw97] evaluate the properties of several representations employed for the design of embedded systems, based on the tagged-signal model [Lee96], a framework where computational models can be compared. Similarly, Lavagno *et al*. [Lav98] review and study some models of computation for embedded system design using the same framework.

2.1. Modeling Embedded Systems using Petri Nets

Due to their intrinsic characteristics and particular extensions to the conventional model, PNs might be an interesting representation for embedded systems. We address in this section some known approaches to the modeling of such systems using Petri nets in the frame of hardware/software codesign.

Stoy [Sto95] presents a modeling technique for hardware/software systems, based on an extended timed Petri nets notation [Pen94]. This Petri net representation is founded on a parallel model with data-control notation and provides timing information. The model consists of two different but closely related parts: control unit and computational/data part. In this approach, timed Petri nets with restricted transition rules are used to represent control flow in both hardware and software. This representation allows to capture hardware and software in a consistent way, so it can be utilized during the synthesis process taking advantage of the efficient movement of functionality between hardware and software domains.

Maciel and Barros [Mac96] use timed Petri nets as intermediate format for the partitioning process: an occam description constitutes the input of the design cycle and is translated into the proposed representation. Timed PNs, in this approach, are associated with dataflow augmented with time information. Timing analysis guides the hardware/software partitioning process. The definition of sub-nets permits handling hierarchies through special places called ports.

A combination of time Petri nets and predicate/transition nets augmented with object-oriented concepts is utilized by Esser *et al.* [Ess98]. Tokens carry data and transitions have associated functions, condition guards, and time constraints. Hierarchical constructions are allowed, providing the capability to represent various levels of granularity.

3. Extensions to Petri Nets

Petri nets (PN) have been widely used for system modeling in many fields of science over three decades. They are a well-understood graphical and mathematical tool. Powerful formal theories, defining its structure and firing rules, have been developed around this model. We do

not address here the basic concepts of PNs, but instead we concentrate in this subsection on the main extensions and modifications—concerning our purposes—proposed along the years. In our discussion we assume that the reader has a basic knowledge of PNs. [Mur89], [Pet81], [Zur94] are suggested for further reading on PN theory and their applications.

Two important intrinsic features of Petri nets are their concurrency and asynchronous nature. These features together with the generality of PNs and their flexibility have stimulated their applications in different areas. However, several drawbacks of the classical PN model have been pointed out along the years.

A major weakness of PNs is the so-called state explosion problem. Petri nets tend to become large even for relatively small systems. The lack of hierarchical decomposition makes it difficult to specify and understand complex systems using the conventional model. To overcome this disadvantage, the classical PN model has been extended introducing the concept of hierarchy [Zub96], [Dit95]. Single elements (transitions and places) may represent a more detailed structure. Thus, different levels of abstraction can be used to model high-complexity systems so that refinements may provide several degrees of granularity.

The conventional PN model lacks the notion of time. This concept is not given in its original definition. However in many embedded applications time is a critical factor. Several extensions have been proposed in order to capture timing aspects. Such are, for example: timed Petri nets [Ram74], time Petri nets [Mer76], and timed place-transition nets [Sif80]. In the first approach, timed PNs, an execution time is associated with each transition, representing the finite duration of a firing. Unlike classical Petri nets, the transition is not instantaneous and the firing rule is modified to make a transition to be fired as soon as it is enabled. In time PNs two values of time are associated with each transition: the minimum and maximum time (starting from the moment the transition has been enabled) in which the transition has to fire, unless it is disabled by the firing of another transition. These limits represent the interval in which the transition may fire. The firing of a transition in a time Petri net is instantaneous. Finally, in timed place-transition nets, unlike the former cases, the time information is associated to places instead of transitions. The time parameter of each place has the meaning of a delay, so that a token must remain in the place a certain interval of time before it may be removed. In these time-extended models, time associated to elements is deterministically given. Other approaches, e.g. stochastic Petri nets [Mol82], consider timing information which is probabilistically associated to transitions.

There is another disadvantage regarding classical Petri nets. This model lacks expressiveness for formulating computations as long as tokens are considered “black dots”. No value is transferred by communications, limiting the modeling power. Allowing tokens to carry information makes it possible to obtain more succinct and precise representations suitable for practical applications. The extensions that include this new dimension to PNs are encompassed in the called high-level Petri nets [Jen91]. High-level PNs, in a broad sense, include predicate/transition nets and coloured Petri nets. The former introduce the concept of individuals with changing properties and relations [Gen81]. Places (predicates) represent variable properties or relations of individuals, and transitions depict types of changes of those properties. Graphically, places and transitions are labeled with identifiers which define the net characteristics. Coloured Petri nets have been introduced in [Jen92] and a strong mathematical theory has been built up around this representation. Transitions describe actions and tokens may carry data values. The arcs between transitions/places and places/transitions have attached expres-

sions that describe the behavior of the net. A transition is enabled if there are enough tokens in its input places and, additionally, these tokens match the arc expressions. Coloured PNs permit hierarchical constructions which, together with the characteristic of valued tokens, make the model powerful in terms of compactness and expressiveness. Although time is not explicitly defined in the model, computer tools developed around coloured Petri nets allow tokens to have time stamps during simulation, in addition to its value.

4. PRES: Petri Net based Representation for Embedded Systems

In the following we present PRES, a Petri net based model, aimed to represent embedded systems. As mentioned before, it can be used to model a system at different levels of detail using the feature of hierarchical decomposition. The model also includes an explicit notion of time, which is essential for the design of embedded systems. In PRES tokens hold information and transitions, when fired, perform transformation of data. As typical of Petri nets, our model is innately asynchronous which means that there is no global clock mechanism for firing transitions. Concurrency and sequential behavior are naturally represented in PRES.

4.1. Basic Definitions

Definition 1. A *Petri Net based Representation for Embedded Systems* 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).
Defined in this way, this structure is an *ordinary Petri net*, which means that there exist no multiple arcs, if any, from a place p_i to a transition t_j (or from a transition t_i to a place p_j). Additionally, P and T must be disjoint, i.e. $P \cap T = \varnothing$.
Properties, characteristics, and behavior of PRES will be introduced and defined in detail in what follows.

Definition 2. A *token* is a pair $k = \langle v_k, r_k \rangle$ where
v_k 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, a record);
r_k is the *token time*, a finite positive real number representing the time stamp of the token.
Let K be the set of all possible token types for a given system.

Definition 3. A *marking* is a function $M : P \to \{0, 1\}$ that denotes the absence or presence of tokens in the places of the net.
For our purposes, we will only consider *bounded* Petri nets, i.e. nets where the number of tokens in each place does not exceed a finite number. Specifically, we aim to use structures which are *safe* or *1-bounded*. Since the intended Petri net in this model must be safe, this function M might also express the number of tokens in each place. We will say that a place p is

marked if $M(p) = 1$. 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* is marked, k^p denotes the token present in *p*. Thus, the token value of the token in a marked place *p* will be v_{k^p}, and the token time of the token in *p* will be r_{k^p}.

Definition 4. The *type function* $\tau : P \rightarrow K$ associates a place with a token type. Thus, we will call $\tau(p)$ the token type associated with the place *p*.
It is worth to point 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.

4.2. System Description

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

Definition 6. For every place in the post-set t° of a transition *t*, there exists an *output function* associated to *t*. Let us consider the transition *t* with its pre-set $^\circ t$ and post-set t°. Formally,

$$\forall p_j \in t^\circ \ \exists f_j : \tau(q_1) \times \tau(q_2) \times \ldots \times \tau(q_a) \rightarrow \tau(p_j)$$

with $^\circ t = \{q_1, q_2, \ldots, q_a\}$ and $t^\circ = \{p_1, p_2, \ldots, p_b\}$.
Output 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.

Definition 7. For every output function associated to a transition *t*, there exists a *function delay* *fd*, a positive real number, which represents the execution time (delay) of that function. Formally,

$$\forall f_j \ \exists \ fd_j \in \Re^+$$

with $\Re^+$ the set of positive reals. If no function delay is explicitly defined, it will be assumed 0.

Definition 8. The *guard* G_t of a transition *t* is the set of boolean *conditions* that must be satisfied in order to enable that transition, when all its input places hold tokens. A *condition* of a transition

$$cond_i : \tau(q_1) \times \tau(q_2) \times \ldots \times \tau(q_a) \rightarrow \{0, 1\}$$

is function of the token values in the places of the pre-set of *t* ($^\circ t = \{q_1, q_2, \ldots, q_a\}$).
The guard G_t of *t* is the conjunction of all conditions of that transition.There is no restriction in the number of conditions for a certain transition. If *all* conditions are satisfied $G_t = 1$, otherwise $G_t = 0$. If no guard is explicitly defined, it will be assumed constantly asserted.

Definition 9. Every transition has a *functionality.* The functionality of a transition *t* is defined in terms of:
(i) Its *output functions*;
(ii) Its *function delays*;
(iii) Its *guard*.
Intuitively, this functionality describes the “behavior” of the transition when it is fired. Unlike the classical Petri net model, each token holds a value and a time tag. When a transition *t* is fired the marking *M* will generally change by removing all the tokens from the pre-set of *t* and

depositing one token into each element of the post-set of *t*. These tokens, added to t°, have values and time stamps which depend on the previous tokens in ${}^\circ t$ and the functionality of *t*.

4.3. Dynamic Behavior

Definition 10. 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 $t \in T$ is *enabled iff* (if and only if)

$$[\forall q_i \in {}^\circ t \;\; M(q_i) = 1] \wedge [\forall p_j \in \{t^\circ - {}^\circ t\} \;\; M(p_j) = 0] \wedge [G_t = 1]$$

If the transition *t* is enabled, we will note it as *t**. Then, the subset of enabled transitions, for certain marking *M*, will be $T^* = \{t \in T | t^*\}$.

Definition 11. Every enabled transition *t** has a *trigger time tt** that represents the time instant at which the transition may fire. Each token in the pre-set of an enabled transition has, in general, a different token time. From the point of view of time, the transition could not fire before the tokens are ready. The concept of trigger time is needed to describe how token times are handled when the transition is fired. The trigger time of an enabled transition is the maximum token time of the tokens in its input places,

$$tt^* = max(r_{k^{q_1}}, r_{k^{q_2}}, \ldots, r_{k^{q_a}})$$

where the pre-set of *t** is ${}^\circ t = \{q_1, q_2, \ldots, q_a\}$.
Note that this trigger time varies during the execution of the net and, if the transition is not enabled, it does not make sense.

Definition 12. The *firing* of an enabled transition changes a marking *M* into a new marking M^+. As a result of firing the transition *t* (with ${}^\circ t = \{q_1, q_2, \ldots, q_a\}$ and $t^\circ = \{p_1, p_2, \ldots, p_b\}$), the following events occur:
(i) Tokens from its pre-set are removed;

$$\forall q_i \in {}^\circ t \;\; M^+(q_i) = 0$$

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

$$\forall p_j \in t^\circ \;\; M^+(p_j) = 1$$

(iii) Each new token deposited in t° has a token value, which is calculated evaluating the respective output function with the token values of tokens in ${}^\circ t$ as arguments;

$$\forall p_j \in t^\circ \quad v_{k^{p_j}} = f_j(v_{k^{q_1}}, v_{k^{q_2}}, \ldots, v_{k^{q_a}})$$

(iv) Each new token added to t° has a token time, which is the sum of the respective function delay and the trigger time of the transition;

$$\forall p_j \in t^\circ \quad r_{k^{p_j}} = fd_j + tt^*$$

Note that only enabled transitions may fire. The execution time of the functionality of that transition is considered in the time tag of the new tokens.

4.4. Hierarchy

Definition 13. A *Hierarchical PRES Structure* is a seven-tuple $HN = (\Lambda, P, T, I, O, \rho, M_0)$ where
$\Lambda = \{HN_1, HN_2, \ldots, HN_l\}$ is a finite set of *subnets* with
$HN_i = (\Lambda_{HN_i}, P_{HN_i}, T_{HN_i}, I_{HN_i}, O_{HN_i}, \rho_{HN_i}, M_{HN_i,0})$;

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

$P = \{p_1, p_2, \ldots, p_m\}$ is a finite non-empty set of places;
$T = \{t_1, t_2, \ldots, t_n\}$ is a finite set of transitions;
$I \subseteq P \times T$ is a finite set of input arcs;
$O \subseteq T \times P$ is a finite set of output arcs;
$\rho \subseteq \{P_{HN_1} \cup P_{HN_2} \cup \ldots \cup P_{HN_l}\} \times P$ is a *place assignment relation* which maps *some* of the places of the subnets onto places of the hierarchical (higher level) net;
M_0 is the initial marking.

4.5. A Simple Example

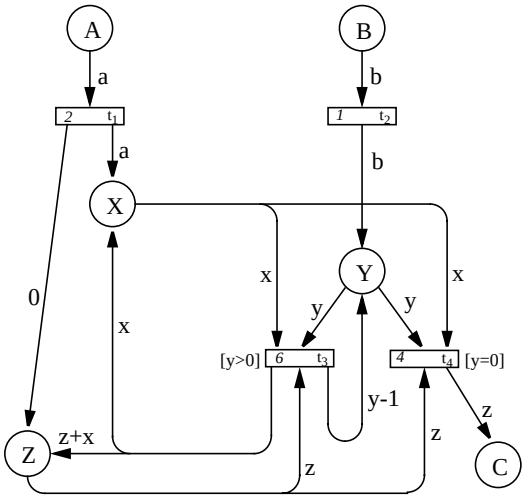
This example will not show all the power of the model and its capabilities but rather explain the different concepts and definitions aforementioned. The purpose of this very simple net is to illustrate the semantics of our model. The net represents a multiplier which takes two positive integers and produces as output the result of multiplying those numbers. It implements a simple algorithm of iterative sums.

The PRES model of this multiplier is shown in Figure 1. We also show the C description corresponding to this algorithm. Like in classical Petri nets, places are graphically represented by circles, transitions by boxes, and arcs by arrows. For this example $P = \{A, B, X, Y, Z, C\}$ and $T = \{t_1, t_2, t_3, t_4\}$. In this particular example, we consider that the function delays for a given transition are the same, so we can call it transition time *rt* and it is inscribed to the left of transition boxes. For instance, $rt_3 = 6$ time units. We have borrowed notation from Coloured Petri nets [Jen92] to graphically express output functions and guards. We use inscriptions on the arcs: given a transition, its output functions (inscribed on output arcs) are captured as expressions in terms of the variables written on its input arcs. Guards are enclosed in square brackets and are also functions of the variables on input arcs.

```
int mult(int a,int b)
{
   int x,y,z;
   x=a;
   y=b;
   z=0;
   while (y>0) {
      z=z+x;
      y=y-1;
   }
   return z;
}


c=mult(a,b);
```

(a)



(b)

Figure 1. Multiplier: (a) algorithm; (b) PRES.

Figure 2 shows the behavior of the net for an initial marking M_0 ($M_0(A) = M_0(B) = 1$). Marked places are shaded and enabled transitions are highlighted using thicker lines. Token information is also shown in marked places. When several transitions are enabled simultaneously, the

one that has minimum trigger time will fire in the next step. If the trigger time of two or more enabled transitions is the same, any of them may fire (one at each step). Let us assume, for the initial marking, $k^A = (5, 0)$ and $k^B = (2, 0)$. It means that the token in place *A* has a value 5 and a time stamp 0. Initially, transitions t_1 and t_2 are enabled and both have trigger time $tt_1^* = tt_2^* = 0$. Then either t_1 or t_2 may fire.

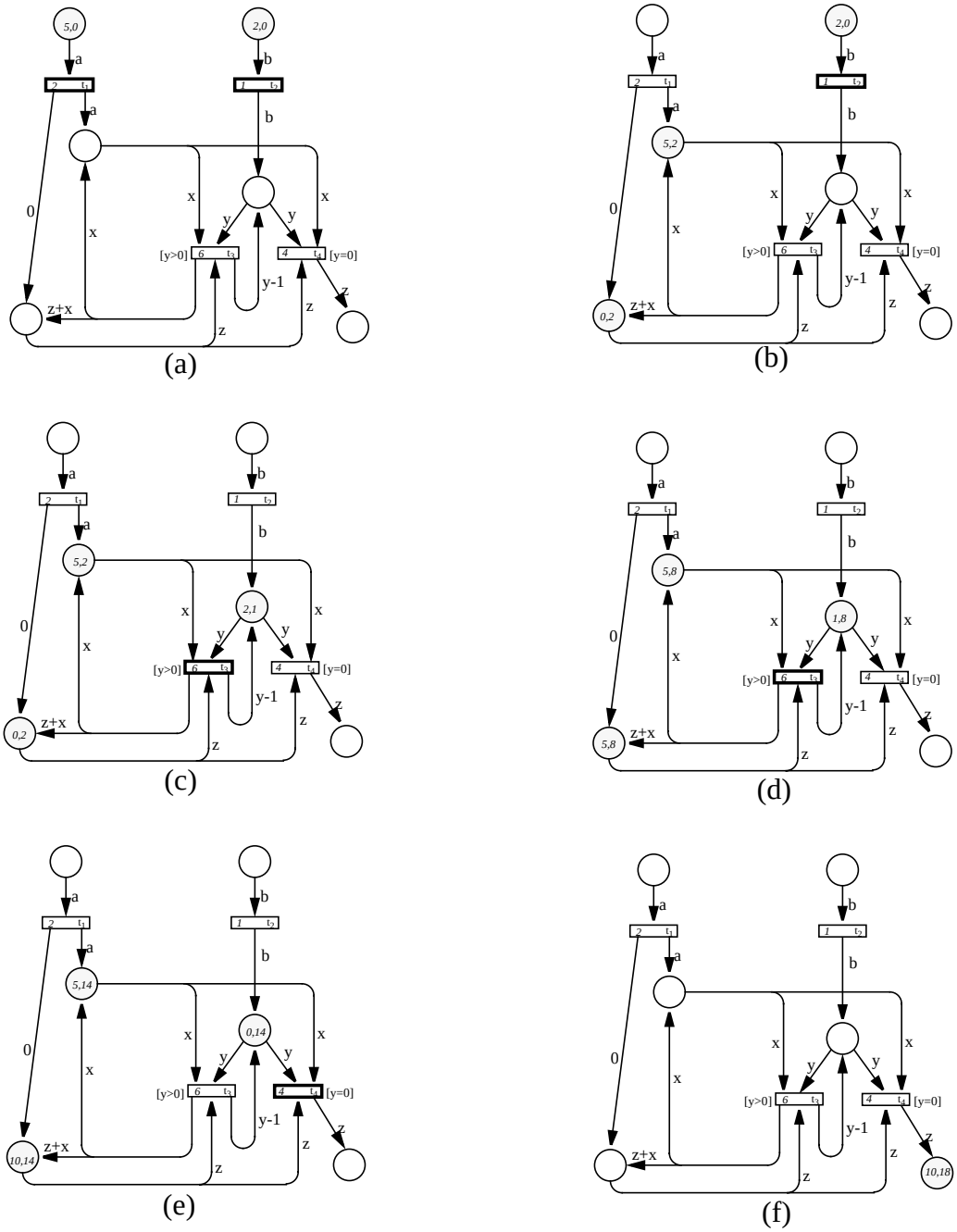


Figure 2. Dynamic Behavior of the Multiplier.

Firing t_1 produces the marking shown in Fig. 2(b), where $k^X = (5, 2)$, $k^Z = (0, 2)$ and $k^B = (2, 0)$. Value and time of the new tokens are calculated following Definition 12. It is easy to see that, for this particular system, firing transitions with equal trigger time in any order does not affect the final result.

Fig. 2(c) illustrates some interesting aspects of PRES. Even if each place in the pre-set of t_4 ($^\circ t_4 = \{X, Y, Z\}$) has a token, the transition is not enabled because its guard is not asserted. For the marking in this figure, t_3 is the only enabled transition so it will be fired in the next step. Looking at the token time of tokens in $^\circ t_3$, we note that they have different time stamps ($r_{k^X} = r_{k^Z} = 2$, $r_{k^Y} = 1$). Hence, t_3 may not fire before $tt_3^* = max(r_{k^X}, r_{k^Y}, r_{k^Z}) = 2$.

After t_3 fires the marking changes into the one shown in Fig. 2(d). Let us analyze, for instance, the new token in Z, $k^Z = (5, 8)$. The arc (t_3, Z) has the inscription "z+x", so that the token value in Z is calculated adding the previous token values of X and Z ($v_{k^Z} = v_{k^Z} + v_{k^X} = 0 + 5 = 5$). The token time in Z is done by $r_{k^Z} = rt_3 + tt_3^* = 6 + 2 = 8$.

Finally, Fig. 2(f) shows the output result of the multiplication (10) and the token time shows the total time needed for the operation (18 time units). The net is not live in this configuration because it is not possible to ultimately fire any transition. If this multiplier is part of a larger system, the token in place C will likely be consumed and new tokens will be added to A and B, allowing the net to perform its function again.

5. Formal Verification of Embedded Systems

5.1. Related Work

The increasing complexity of embedded systems poses a challenge in verifying their correctness. Some verification approaches, suitable to hardware/software systems, have been proposed recently. Alur *et. al* [Alu96] present a model checking procedure based on the Hybrid Automata model: given a system represented as communicating machines with real-valued variables, the method shows if an ICTL-formula (Integrator Computation Tree Logic), specifying system requirements, holds for all possible states of the automaton. Using the same model, Hybrid Automata, another coverification method is proposed in [Hsi99], where complex systems can be analyzed using a simplification strategy to verify individually the hardware, the software and the interface. Balarin *et. al* [Bal96] introduce a verification methodology based on Codesign Finite State Machines (CFSMs), in which CFSMs are translated into traditional state automata. This technique checks if all possible sequences of inputs and outputs of the system satisfy the desired properties. To do so, those sequences that meet the requirements constitute the language of another automaton, reducing the problem to verify language containment between automata. In [Gar98], a partitioned system, described using a Pascal-like language, 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 symbolic model checking of process networks and related models is proposed in [Str98], where IDDs (Interval Decision Diagrams) are used to represent multi-valued functions.

On the other hand, related work in the area of Petri nets (PNs) includes [Wim97], which presents a BDD-based model checker for safe nets. Although the approach is intended to verify Petri nets in general, with no particular interest in embedded systems and without dealing with time information, it studies different forms of describing PNs using the SMV system [SMV], developed at Carnegie Mellon University. An interesting approach used for analysis and verification of bounded Petri nets is presented in [Pas94]. Using the efficiency of BDDs to represent sets of markings and reduction rules to transform PNs, this technique can be used for reachability analyses and verification of some properties of PNs with large state spaces.

5.2. Symbolic Model Checking

The above approaches reveal a big interest for model checking in the community of hardware/ software codesign. In consequence, this section will consider the basic ideas of model checking and BDDs as efficient structures to represent symbolically transition relations and sets of states.

Model checking is an approach to formal verification that lets the designer determine if the model of a system satisfies certain required properties. The model checker proves whether those properties hold. Clarke *et. al* [Cla86] introduced a model checking algorithm for formulas specified in temporal logic CTL (Computation Tree Logic). CTL is based on propositional logic of branching time, that is, a logic where time may split into more than one possible future using a discrete model of 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). Thus, formulas may describe properties of computation paths over labeled state-transition structures. This algorithm, however, requires the entire state transition graph to be constructed, causing a serious state explosion problem.

One way to overcome the state explosion is to represent symbolically the transition relation instead of explicit enumeration. A compact and efficient form of representing boolean formulas and transition relations is using ordered binary decision diagrams (BDDs). BDDs are canonical representations that make boolean manipulations much simpler computationally [Bry92]. Symbolic model checking [Bur94] makes use of BDDs to represent sets of states and the transition relation, and the algorithm employs fixed-point techniques that manipulate sets using their characteristic functions encoded as BDDs. Therefore, it is possible to reason about designs with large state spaces without constructing the state graph of the system.

SMV [SMV] is one of the available tools that uses the BDD-based symbolic model checking algorithm. This model checker has an input language that allows to describe systems using boolean, scalar or fixed-array data types, and boolean and basic scalar operations. CTL formulas to be checked, also specified in the SMV language, may express safety, fairness, liveness, and deadlock-freedom, among other properties.

6. The Coverification Methodology

The coverification methodology will be explained in reference to a medical monitoring system in order to illustrate the different aspects of this technique. This system has been modeled using PRES and will be formally verified using this methodology.

6.1. Patient Monitoring Application

Figure 3 shows a net represents a patient monitoring system as introduced in [Dro93] and studied in [Cam94]. The patient monitor measures physiological phenomena and analyzes this information. If the system detects abnormal conditions on the patient, it activates aural and visual alarms. The patient condition information is displayed and recorded as well. The functionality of the system can be captured as a set of processes. The *acquire* process reads infor-

mation from the sensors. Usually this information contains spurious data that must be debugged. *filter* processes such data and eliminates false information received from the sensors. Once the information has been filtered, processes which detect anomalous conditions on blood pressure, heart rate, or temperature may start, depending on the data available. For instance, a possible anomaly in the blood pressure will make the process *blood* activate, in order to study the data. If, after analyzing the information, an irregular condition of the patient is encountered, the process *alarm* will be executed and an audio signal (process *audio*) will be triggered. The information resulted from the *filter* process is displayed on a screen and recorded by the processes *display* and *recorder*, respectively. The specification of the patient monitoring system includes a timing constraint which states that data from sensors must be sampled every 15 ms and that acquisition of new information requires the system to finish its functionality before the next execution.

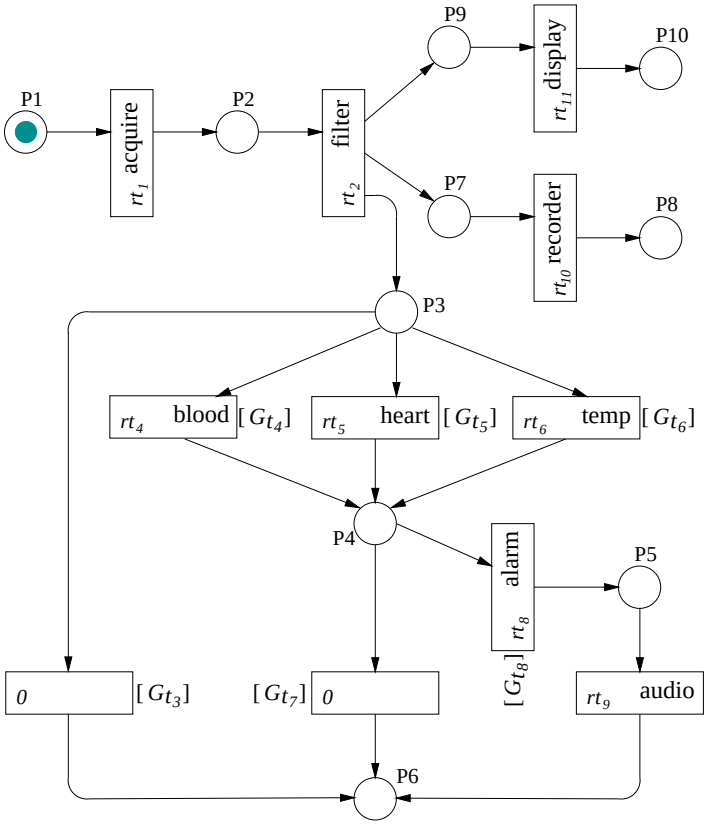


Figure 3. Medical Monitoring System.

The medical monitoring system is modeled in PRES as shown in Figure 3, where the operations performed in the process are captured by transitions and the data dependence between them is captured by the structure of the net. Transitions have been named after the processes. The marking M_0, for the model of the monitoring system, shows *P1* as the only place initially marked. As it has been mentioned above, output functions are very important when describing the behavior of the system to be modeled. For instance, in Figure 3, there are three output functions associated to transition *filter*, which define token values of new tokens in *P3*, *P7* and *P9*, when *filter* is fired (executed). These three functions represent what has been earlier called the process *filter*. In this particular example, we assume that all function delays corresponding to output functions associated to a given transition are identical. This time is captured as “tran-

sition time" and is inscribed on the respective transition box. Thus, rt_6 represents the execution time of the functions associated to transition *temp*.

In Figure 3, for example, G_{t_4} represents the condition that must be fulfilled to execute the process *blood*. If no guard is explicitly defined for a given transition, that transition will be enabled whenever its input places are marked. There are two transitions that have no name attached: we have introduced them in order to model the situation in which no abnormal condition is detected. The associated execution time is zero because there are no activities to be performed in this case.

6.2. Methodology

The coverification method presented here is based on the Petri net based model introduced in Section 4. The purpose of the approach presented in this work 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 this model. A given marking, i.e. absence or presence of tokens in places of the net, may represent the state of the system in a certain moment in the dynamic behavior of the net. Based on this, different properties can be studied. For instance, in a landing gear controller, the door must not close while the plane is landing, under any circumstance. This sort of safety requirement might be formally proven, checking that the places which represent such states are never marked simultaneously. Sometimes, 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 can not be missed, it is crucial to reason quantitatively about temporal properties to assure 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, for example, prove that a given place will always be marked in the future and that its time stamp, for any possible condition, will be less than a certain time value that represents a temporal constraint. Such study will be called *time analysis*.

A third type of analysis using PRES involves reasoning about values of tokens in marked places. This type of *behavior analysis* is not part of the coverification method proposed here. In this work we address just 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, but we do not deal with the values of those tokens. We assume that output functions (see Section 4) are correctly defined.

As it has been mentioned above, in a PRES model a place may hold at most one token for a certain marking. Thus, it is possible to encode a marking—or a set of markings—as a boolean function where the variables correspond to places of the net. Boolean functions can be straightforwardly represented by BDDs. Firing a transition in a Petri net changes the marking into a new one, which is a variation in the state of the system. It is possible to build the BDD that represents the transition relation of the system and then compute efficiently the reachable

states using BDDs [Bry92], [Hu97]. With such a BDD-based representation we can formally verify properties, specified in CTL, using symbolic model checking [Bur94] and accomplish reachability analyses. In our experiments, we use the SMV tool (a BDD-based symbolic model checker) [SMV] and its input language to describe and verify systems modeled in PRES.

A program in SMV describes both the system and the specification (properties to verify). The system is described as a collection of "modules". Each module may contain variables, its initial state, and assignments of variables for the next state. A "process" is an instance of a module, in such a way that the model checker executes a step by choosing non-deterministically a process and then executing all assignment statements of that process in parallel.

To translate a PRES model into the SMV input language, we declare in the `main` module a boolean as well as an integer variable for each place of the net. The boolean variable represents absence/presence of tokens in that place, while the integer one represents the time stamp of the token when the particular place is marked. We instantiate each transition as a process that has as parameters its input and output places as well as time stamps of tokens in those places. In the `main` module we also define the initial marking of the net assigning initial values to the variables that represent places and to time stamps of tokens in initially marked places.
We describe each transition of the Petri net as a module that adds/removes tokens (changes the marking) when it is executed (fires). Figure 5 illustrates the description of the *blood* process corresponding to the implementation shown in Figure 4(b). When a transition fires, it changes the marking of the system removing tokens from its input places and adding new tokens to output places. This is captured using `next` assignments for input/output places of the transition. Thus, if the transition is enabled (`enabled := P3 & Pa & !P4`), execution of *blood* will assign boolean values to *P3*, *Pa* and *P4* according to the transition firing rules, that is, `1` to output places and `0` to input places.

As stated in the definition of PRES, time tags of new tokens are calculated as the sum of the trigger time and the respective function delay, e.g. `(trigger_time + func_delay_a) mod 28`. In this case, we are using integer addition "modulo 28" because integer variables in SMV must be bounded when they are defined. The bound of variables for time stamps, in the example of Figure 4(b), is `27`. This upper bound can be determined summing all transition times when there no loops in the net, and it represents the maximum time stamp that a token may have in the net (this is, of course, assuming that time stamps of all tokens in the initial marking are zero). The larger the value of this bound, the longer is the computation time needed to verify timing properties. The complexity of this problem grows exponentially in the size of the bound for time stamps. This becomes a limitation of our approach when large time stamps are needed to characterize the timing aspects of a system. Every transition described as an SMV process must include the declaration `FAIRNESS running` to ensure that it is executed infinitely often and the system progresses.

In PRES each token has two components: a value and a time tag. Time tags, according to the definition of the model, can be positive reals. However, the reader might have noted, that in order to use the SMV system we need to restrict the time to discrete (integer) values. Another issue is that, since we are not dealing with token values, only certain kind of systems that include guards in their models may be analyzed using this approach. On the other hand, models in which transitions bear no guard may be straightly studied. For some models that include guards, as the one in Figure 3, those guards are "complementary". For this example, G_{t_3}, G_{t_4},

G_{t_5} and G_{t_6} are complementary because whenever *P3* is marked, just one of these guards will be asserted. Similarly, G_{t_7} and G_{t_8} are complementary. For PRES models with complementary guards, those guards can be ignored without affecting the reachability and time analyses. In that case, the model will exhibit non-determinism when firing transitions whose guards have been dropped.

6.3. Verification of the Medical Monitoring System

In this section we show the verification of the medical monitoring application described above for two possible implementations of the system. This practically illustrates a transformational design space exploration methodology based on formal verification. We consider first, in Figure 4(a), an implementation using a single programmable processor. Note that values have been assigned to transition times. These times are the estimated worst case execution times of the respective functions on the selected processor. For example, the transition time for *heart* is 4 ms. The reader can also notice that, by reason of the considerations explained in the previous section, the guards have been ignored. The place *Pa* (initially marked) models the processor: this place is both input and output of all transitions, which captures processes mapped onto that processor. We use lines with no arrowheads to indicate this bidirectional flow relation between transitions and *Pa*. Place *P1* models the data read from the sensors, *P8* and *P10* the information to be recorded and displayed, respectively, and *P6* is the indicator which shows that the analysis has been performed. *P1*, *P6*, *P8* and *P10* are the places through which the system interacts with its environment. Initially, *P1* and *Pa* are marked and time stamps for tokens in *P1* and *Pa* are $r_{k^{P1}} = 0$ and $r_{k^{Pa}} = 0$. If the system operates properly, a new token will be added to *P1* after the patient monitor finishes its functionality.

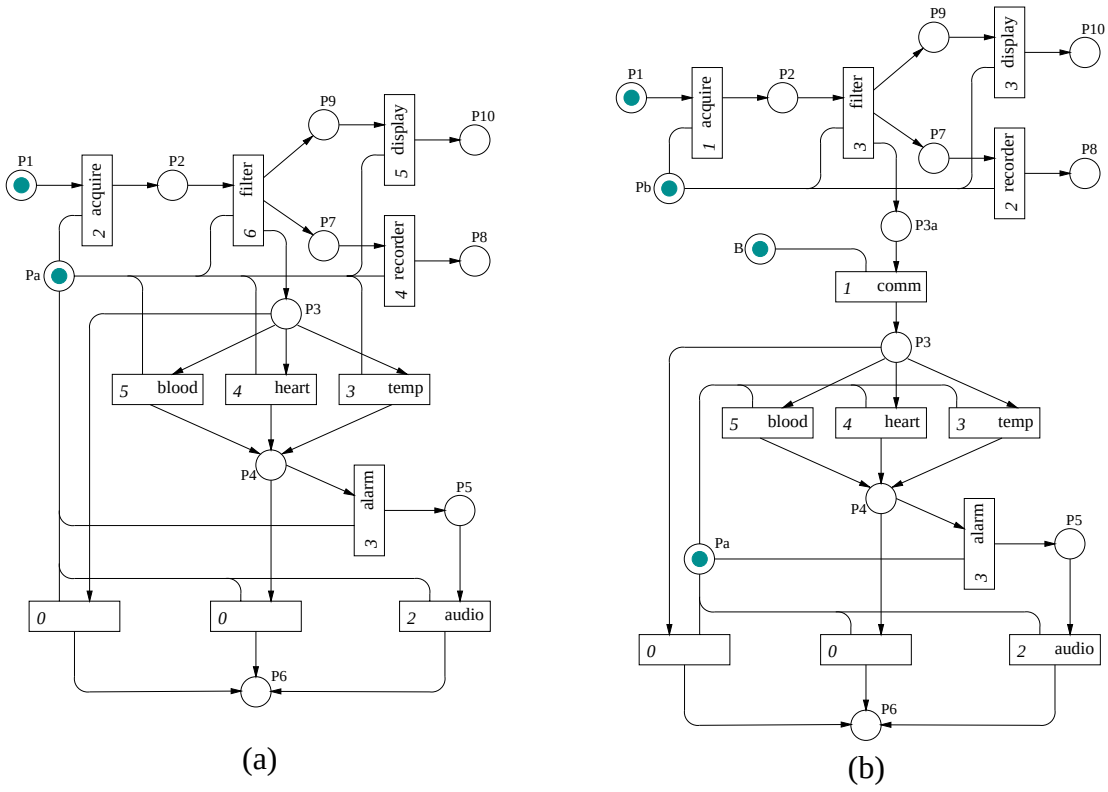


Figure 4. Different Implementations of the Patient Monitoring System.

We have to verify first that the system, under any circumstance, will complete its functionality, that is, eventually *P6*, *P8* and *P10* will be marked. Using the SMV tool, this property can be expressed as a CTL formula preceded by the by the keyword SPEC:

```
SPEC AF (P6 & P8 & P10)
```

which reads "always eventually (P6 & P8 & P10)", i.e. the state in which *P6*, *P8* and *P10* are simultaneously marked is inevitable. This formula holds for both representations in Figure 4. The second property, that concerns our design, is the constraint of a maximum delay of 15 ms. We have to formally verify that when *P6*, *P8* and *P10* are marked (which has been shown to be true) time stamps of tokens in these places are less than or equal 15. We may express this constraint in our description as three CTL formulas:

```
SPEC AF (P6 & (time_P6<=15))
SPEC AF (P8 & (time_P8<=15))
SPEC AF (P10 & (time_P10<=15))
```

All three formulas above turn out to be false, for the model in Figure 4(a), and SMV gives counter-examples. As the implementation in Figure 4(a) does not meet the time constraint, we consider an alternative architecture. Figure 4(b) models the patient monitoring system implemented using one programmable processor (represented by *Pa*) and one hardware component (*Pb)*. Processes *acquire*, *filter*, *recorder* and *display* are mapped onto *Pb* while the other processes onto *Pa*. A new transition (*comm*) has been introduced in the model to consider the cost of inter-processor communication. This process *comm* is the only one that utilizes bus resources (place *B*). Note that execution times of processes mapped onto *Pb* have changed with respect to the previous design alternative. For the model in Figure 4(b) we have formally verified, using symbolic model checking through the SMV system, that the properties mentioned above do hold for all possible situations. This implementation has superior performance because of parallelism and lower execution times for the hardware.

```
MODULE blood(P3,time_P3,Pa,time_Pa,P4,time_P4)
ASSIGN
   next(P3) := case
      enabled : 0;
      1 : P3;
   esac;
   next(Pa) := case
      enabled : 1;
      1 : Pa;
   esac;
   next(P4) := case
      enabled : 1;
      1 : P4;
   esac;
   next(time_Pa) := case
      enabled : (trigger_time + func_delay_a) mod 28;
      1 : time_Pa;
   esac;
   next(time_P4) := case
      enabled : (trigger_time + func_delay_4) mod 28;
      1 : time_P4;
   esac;
DEFINE
   func_delay_a := 5;
   func_delay_4 := 5;
   trigger_time := case
      (time_P3 >= time_Pa) : time_P3;
      (time_Pa >= time_P3) : time_Pa;
   esac;
   enabled := P3 & Pa & !P4;
FAIRNESS running
```

Figure 5. Description of Transition blood using SMV.

7. Conclusions

We have presented PRES, a Petri net based model with extensions to capture important features of embedded systems. The model is simple, intuitive and can be easily handled by the designer. We introduced an approach to formal verification of embedded systems using symbolic model checking with PRES as underlying computational model. Thus, coverification is possible dealing with timing properties.

It has been also shown how to translate PRES models into the input formalism of a model checker. A patient monitoring system has been studied to illustrate the applicability of the coverification approach to practical systems. Transformations during design space exploration can be smoothly captured and properties to be checked can derived directly from the model in an easy manner as well.

One of the main contribution lies in modeling embedded systems in such a way that the representation is adequate to be analyzed using formal methods. The model that we use is a Petri net based notation in which tokens bear both value and time tag. We address a coverification method that allows to reason formally about the presence/absence of tokens in places of the net and their time stamps, but we do not deal with their token values. This is a problem worth for further research.

References

[Alu96] 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.

[Bal96] 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.

[Bry92] R. E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams," in *ACM Computing Surveys*, vol. 24, pp. 293-318, Sept. 1992.

[Bur94] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill, "Symbolic Model Checking for Sequential Circuit Verification," in *IEEE Trans. CAD of Integrated Circuits and Systems*, vol. 13, pp. 401-424, April 1994.

[Cam94] S. Campos, E. M. Clarke, W. Marrero, and M. Minea, "Timing Analysis of Industrial Real-Time Systems," in *Proc. Workshop on Industrial-Strength Formal Specification Techniques*, 1995, pp. 97-107.

[Cla86] 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.

[Cor99] L. A. Cortés, P. Eles, and Z. Peng, "A Survey on Hardware/Software Codesign Representation Models," SAVE Project Report, Dept. of Computer and Information Science, Linköping University, Linköping, 1999.

[Dit95] G. Dittrich, "Modeling of Complex Systems Using Hierarchically Represented Petri Nets," in *Proc. Intl. Conference on Systems, Man and Cybernetics*, 1995, pp. 2694-2699.

[Dro93] P. J. Drongowski, "Software architecture in realtime systems," in *Proc. Workshop on Real-Time Applications*, 1993, pp. 198-203.

[Edw97] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, "Design of Embedded Systems: Formal Models, Validation, and Synthesis," in *Proc. IEEE*, vol. 85, pp. 366-390, March 1997.

[Ess98] R. Esser, J. Teich, and L. Thiele, "CodeSign: An embedded system design environment," in *IEE Proc. Computers and Digital Techniques*, vol. 145, pp. 171-180, May 1998.

[Gan94] J. D. Gannon, J. M. Purtilo, and M. V. Zelkowitz, *Software Specification: A Comparison of Formal Methods*. Norwood, NJ: Ablex Publishing, 1994.

[Gar98] E. H. A. Garcez and W. Rosenstiel, "CVF - Coverification Framework," in *Proc. Brazilian Symposium on Integrated Circuit Design*, 1998, pp. 103-106.

[Gen81] H. J. Genrich and K. Lautenbach, “System modelling with high-level Petri nets,” in *Theoretical Computer Science*, vol. 13, pp. 109-136, Jan. 1981.
[Hsi99] P.-A. Hsiung, “Hardware-Software Coverification of Concurrent Embedded Real-Time Systems,” in *Proc. Euromicro RTS*, 1999, pp. 216-223.
[Hu97] A. J. Hu, “Formal Hardware Verification with BDDs: An Introduction,” in *Proc. Pacific Rim Conference on Communications, Computers and Signal Processing*, 1997, pp. 677-682.
[Jen91] K. Jensen and G. Rozenberg, Eds. *High-level Petri Nets*. Berlin: Springer-Verlag, 1991.
[Jen92] K. Jensen, *Coloured Petri Nets*. Berlin: Springer-Verlag, 1992.
[Ker99] 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.
[Lav98] L. Lavagno, A. Sangiovanni-Vincentelli, and E. Sentovich, “Models of Computation for Embedded System Design,” in *NATO ASI Proc. on System Synthesis*, 1998, pp. 1-57.
[Lee96] E. A. Lee and A. Sangiovanni-Vincentelli, “Comparing Models of Computation,” in *Proc. ICCAD*, 1996, pp. 234-241.
[Mac96] P. Maciel and E. Barros, “Capturing Time Constraints by Using Petri Nets in the Context of Hardware/Software Codesign,” in *Proc. Intl. Workshop on Rapid System Prototyping*, 1996, pp. 36-41.
[Mer76] P. M. Merlin and D. J. Farber, “Recoverability of Communication Protocols—Implications of a Theoretical Study,” in *IEEE Trans. Communications*, vol. COM-24, pp. 1036-1042, Sept. 1976.
[Mol82] M. K. Molloy, “Performance Analysis Using Stochastic Petri Nets,” in *IEEE Trans. Computers*, vol. C-31, pp. 913-917, Sept. 1982.
[Mur89] T. Murata, “Petri Nets: Analysis and Applications,” in *Proc. IEEE*, vol. 77, pp. 541-580, April 1989.
[Pas94] 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.
[Pen94] Z. Peng and K. Kuchcinski, “Automated Transformation of Algorithms into Register-Transfer Level Implementations,” in *IEEE Trans. CAD of Integrated Circuits and Systems*, vol. 13, pp. 150-166, Feb. 1994.
[Pet81] J. Peterson, *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1981.
[Ram74] C. Ramchandani, “Analysis of asynchronous concurrent systems by timed Petri nets,” Project MAC, Technical Report 120, Massachusetts Institute of Technology, Cambridge, Feb. 1974.
[Sif80] J. Sifakis, “Performance Evaluation of Systems using Nets,” in *Net Theory and Applications*, W. Brauer, Ed. *LNCS 84*, Berlin: Springer-Verlag, 1980, pp. 307-319.
[SMV] The SMV System, `http://www.cs.cmu.edu/~modelcheck/smv.html`
[Sto95] E. Stoy, “A Petri Net Based Unified Representation for Hardware/Software Co-Design,” Licentiate Thesis, Dept. of Computer and Information Science, Linköping University, Linköping, 1995.
[Str98] K. Strehl and L. Thiele, “Symbolic Model Checking of Process Networks Using Interval Diagrams Techniques,” in *Proc. ICCAD*, 1998, pp. 686-692.
[Wim97] G. Wimmel, “A BDD-based Model Checker for the PEP Tool,” Major Individual Project Report, Dept. of Computing Science, University of Newcastle, Newcastle, May 1997.
[Zub96] W. M. Zuberek and I. Bluemke, “Hierarchies of Place/Transitions Refinements in Petri Nets,” in *Proc. Conference on Emerging on Technologies and Factory Automation*, 1996, pp. 355-360.
[Zur94] R. Zurawski and M. Zhou, “Petri Nets and Industrial Applications: A Tutorial,” in *IEEE Trans. Industrial Electronics*, vol. 41, pp. 567-583, Dec. 1994.