SAVE Project Report, Dept. of Computer and Information
Science, Linköping University, Linköping, December 2001.

Modeling and Verification of Embedded Systems using Petri Net based Methods: Application to an Industrial Case

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

Abstract

Embedded systems are becoming increasingly common in objects that we use in our everyday life. Embedded systems are typically characterized by their dedicated function and real-time behavior. Many of them must fulfill strict requirements in terms of reliability and correctness. Designing systems with such features, combined with high levels of complexity and tight time-to-market constraints, is a challenging task. In order to devise systems with such features, a formal design methodology is necessary to carry out systematically the different tasks along the design flow. The SAVE project [SAV] aims at the development of a formal approach to specification, implementation, and verification of heterogeneous electronic systems. We have developed techniques for modeling and verifying embedded systems. This document reports the main results that have been obtained within the frame of SAVE in the fields of modeling and verification. An industrial system is used as study case in order to demonstrate the feasibility of the approach on practical applications.

1. Introduction

Embedded systems are becoming pervasive in the electronics industry. Applications include automotive and aircraft controllers, cellular phones, network switches, household appliances, medical devices, and consumer electronics, among others. Embedded systems are part of larger systems and typically interact continuously with their environment. Embedded systems generally include both software and hardware elements, that is, programmable processors and hardware components like application specific integrated circuits (ASICs) and field programmable gate arrays (FPGAs). Besides their heterogeneity, embedded systems are characterized by their dedicated function, real-time behavior, and high requirements on reliability and correctness [Cam96]. Designing systems with such characteristics is a difficult task. Moreover, the ever increasing complexity of embedded systems combined with tight time-to-market windows poses interesting challenges for the designers.

An essential issue of any systematic methodology aiming at designing embedded systems is the underlying model of computation. The design process must be based on a model with precise mathematical meaning so that the different tasks from specification to implementation can be carried out systematically [Edw97]. A sound representation allows capturing unambiguously the functionality of the system, verifying the correctness of the system with respect to its desired properties, reasoning formally about the refinements and steps in the synthesis

process, and using CAD tools in order to assist the designer [Sgr00]. Therefore, the use of a formal representation in embedded systems design is a must.

Correctness plays a key role in many embedded applications. As we become more dependent on computer systems, the cost of a failure can be extremely high, in terms of loss of both human lives and money. In safety-critical systems, for instance, reliability and safety are the most important criteria. Traditional validation techniques like simulation and testing are not sufficient to verify the correctness of such systems. Formal verification is becoming a practical way to ensure the correctness of designs by complementing simulation and testing.

As pointed out above, the model of computation is the backbone of a design flow. In this report we present a modeling formalism that can capture relevant characteristics of embedded systems at different levels of granularity.

Since correctness is becoming increasingly important in embedded systems, we also present an approach to the problem of formal verification of embedded systems represented in our modeling formalism.

2. The Design Representation

In order to devise embedded systems the design process must be based upon a sound model of computation that captures important features of such systems. The notation we use to model embedded systems is an extension to Petri nets, called PRES+ (Petri Net based Representation for Embedded Systems). This section presents the formal definition of PRES+.

2.1 Basic Definitions

Definition 2.1. 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 2.4). ▪

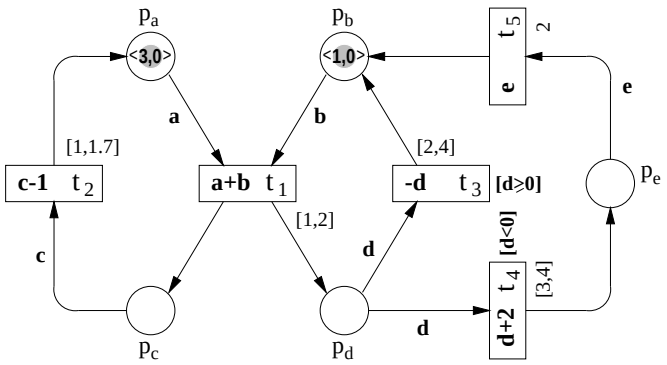

Figure 1: A PRES+ model

We use the example of Figure 1 in order to illustrate the definitions of the model presented in this chapter. Like in classical Petri nets, places are graphically represented by circles, transitions by boxes, and arcs by arrows. For this example, $P=\{p_a, p_b, p_c, p_d, p_e\}$ and $T=\{t_1, t_2, t_3, t_4, t_5\}$.

Definition 4.2. A *token* is a pair $k=\langle v, r\rangle$ where

v is the *token value*. 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. ▪

A token 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). A token type is defined by the set of possible values that the token may take. Thus K is a set of sets.

For the initial marking of the net shown in Figure 1, for instance, in place p_a there is a token k_a with token value $v_a=3$ and token time $r_a=0$.

Definition 2.3. The *type function* $\tau : P \to K$ associates every place $p \in P$ with a token type. $\tau(p)$ denotes the set of possible values that tokens may bear in p. The set of possible tokens in place p is given by $E_p=\{\langle v, r\rangle | v \in \tau(p) \wedge r \in \mathbb{R}_0^+\}$. $E=\bigcup_{p \in P} E_p$ denotes the set of all tokens. ▪

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 given in Figure 1, $\tau(p)=\mathbb{Z}$ for all $p \in P$, i.e. all places have token type *integer*. Thus the set of all possible tokens in the system is $E=\{\langle v, r\rangle | v \in \mathbb{Z} \wedge r \in \mathbb{R}_0^+\}$.

Definition 2.4. A *marking* M is an assignment of tokens to places of the net. The marking of a place $p \in P$, denoted $M(p)$, can be represented as a multi-set[1] over E_p. For a particular marking M, a place p is said to be *marked* iff $M(p) \neq \varnothing$. ▪

The initial marking M_0 in the net of Figure 1 shows p_a and p_b as the only places initially marked: $M_0(p_a)=\{\langle 3, 0\rangle\}$ and $M_0(p_b)=\{\langle 1, 0\rangle\}$, whereas $M_0(p_c)=M_0(p_d)=M_0(p_e)=\varnothing$.

Definition 2.5. The *pre-set* ${}^\circ t=\{p \in P | (p, t) \in I\}$ of a transition $t \in T$ is the set of *input places* of t. Similarly, the *post-set* $t^\circ=\{p \in P | (t, p) \in O\}$ of a transition $t \in T$ is the set of *output places* of t. The *pre-set* ${}^\circ p$ and the *post-set* p° of a place $p \in P$ are given by ${}^\circ p=\{t \in T | (t, p) \in O\}$ and $p^\circ=\{t \in T | (p, t) \in I\}$ respectively. ▪

Definition 2.6. All output places of a given transition have the same token type, that is, $p, q \in t^\circ \Rightarrow \tau(p)=\tau(q)$ ▪

2.2 Description of Functionality

Definition 2.7. For every transition $t \in T$, there exists a *transition function* f associated to t. Formally, for all $t \in T$ there 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 functionality 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_1, for example, is given by $f_1(a, b)=a+b$. We use inscriptions on the input arcs of a transition in order to denote the arguments of its transition function.

Definition 2.8. For every transition $t \in T$, 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, for all $t \in T$ there exist $d^-, d^+ \in \mathbb{R}_0^+$ such that $d^- \leq d^+$. ▪

Referring again to Figure 1, the minimum transition delay of t_2 is $d_2^-=1$, and its maximum

1. A *multi-set* or *bag* is a collection of elements over some domain in which, unlike a set, multiple occurrences of the same element are allowed. For example, $\{a, b, b, b\}$ is a multi-set over $\{a, b, c\}$.

transition delay is $d_2^+=1.7$ 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 $d_5=2$.

Definition 2.9. A transition $t \in T$ may have a *guard* G associated to it. The guard of a transition t is a predicate $G : \tau(p_1) \times \tau(p_2) \times \ldots \times \tau(p_a) \rightarrow \{0, 1\}$ where ${}^\circ t=\{p_1, p_2, \ldots, p_a\}$. ■

Note that the guard of a transition t is a function of the token values in places of its pre-set ${}^\circ t$. For instance, in Figure 1, $d < 0$ represents the guard G_4.

2.3 Dynamic Behavior

Definition 2.10. A transition $t \in T$ is *bound*, for a given marking M, iff all its input places are marked. A *binding* b of a bound transition t, with pre-set ${}^\circ t=\{p_1, p_2, \ldots, p_a\}$, is an ordered tuple of tokens $b=(k_1, k_2, \ldots, k_a)$ where $k_i \in M(p_i)$ for all $p_i \in {}^\circ t$. ■

Observe that, for a particular marking M, a transition may have different bindings. The existence of a binding is a necessary condition for the enabling of a transition. For the initial marking of the net shown in Figure 1, t_1 has a binding $b=(\langle 3, 0\rangle, \langle 1, 0\rangle)$. Since t_1 has no guard, it is enabled for the initial marking (as formally stated in Definition 2.11).

We introduce the following notation which will be useful in the coming definitions. Given the binding $b=(k_1, k_2, \ldots, k_a)$, the token value of the token k_i is denoted v_i, and the token time of k_i is denoted r_i.

Definition 2.11. A bound transition $t \in T$ with guard G is *enabled*, for a binding $b=(k_1, k_2, \ldots, k_a)$, iff $G(v_1, v_2, \ldots, v_a)=1$. A transition $t \in T$ with no guard is *enabled* if t is bound. ■

Definition 2.12. The *enabling time* et of an enabled transition $t \in T$, for a binding $b=(k_1, k_2, \ldots, k_a)$, is the time instant at which t becomes enabled. et is given by the maximum token time of the tokens in the binding b, that is, $et=max(r_1, r_2, \ldots, r_a)$. ■

Definition 2.13. The *earliest trigger time* $tt^-=et + d^-$ and the *latest trigger time* $tt^+=et + d^+$ of an enabled transition $t \in T$, for a binding $b=(k_1, k_2, \ldots, k_a)$, are the lower and upper time limits for the firing of t. An enabled transition $t \in T$ 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 2.14. The *firing* of an enabled transition $t \in T$, for a binding $b=(k_1, k_2, \ldots, k_a)$, changes a marking M into a new marking M^+. As a result of firing the transition t, the following occurs:

(i) Tokens from its pre-set ${}^\circ t$ are removed, that is, $M^+(p_i)=M(p_i) - \{k_i\}$ for all $p_i \in {}^\circ t$;

(ii) One new token $k=\langle v, r\rangle$ is added to each place of its post-set t°, that is, $M^+(p)=M(p) + \{k\}$ for all $p \in t^\circ$. The token value of k is calculated by evaluating the transition function f with token values of tokens in the binding b as arguments, that is, $v=f(v_1, v_2, \ldots, v_a)$. The token time of k is the instant at which the transition t fires, that is, $r=tt^*$ where $tt^* \in [tt^-, tt^+]$;

(iii) The marking of places different from input and output places of t remain unchanged, that is, $M^+(p)=M(p)$ for all $p \in P - {}^\circ t - t^\circ$. ■

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 instant at which it was "created".

In Figure 1, transition t_1 is the only one initially enabled (binding $(\langle 3, 0\rangle, \langle 1, 0\rangle)$) so that its enabling time is 0. Therefore, t_1 may not fire before 1 time units and must fire before or at 2

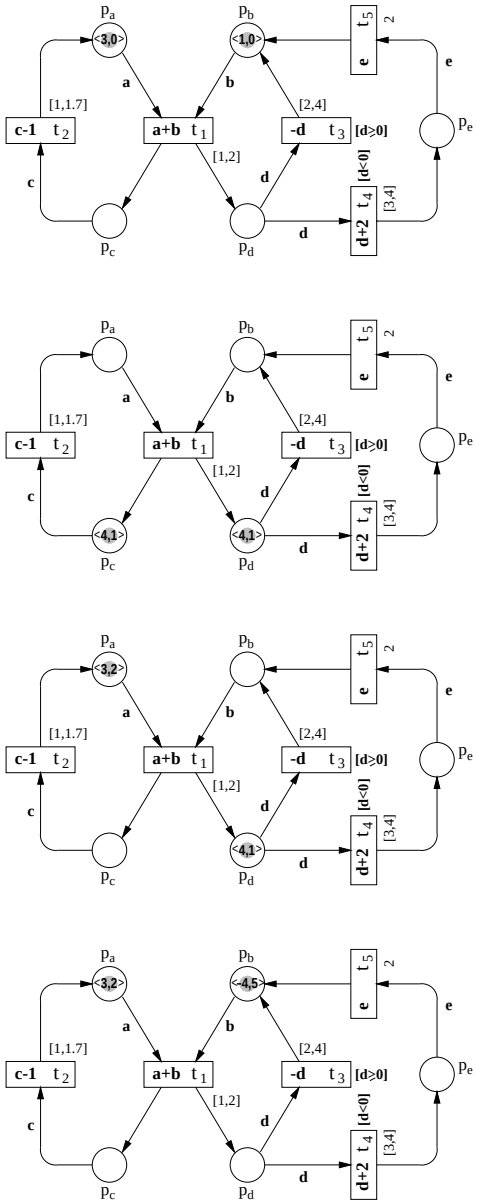

Figure 2: Illustration of the dynamic behavior of a PRES+ model

time units. Let us assume that t_1 fires at 1 time units: tokens $\langle 3, 0\rangle$ and $\langle 1, 0\rangle$ are removed from p_a and p_b respectively, and a new token $\langle 4, 1\rangle$ is added to both p_c and p_d. At this moment, only t_2 and t_3 are enabled (t_4 is bound but not enabled because its guard is not satisfied for the binding ($\langle 4, 1\rangle$)). Note that transition t_2 has to fire strictly before t_3: according to the firing rules, t_2 must fire no earlier than 2 and no later than 2.7 time units, while t_3 is restricted to fire in the interval $[3, 5]$. Figure 2 illustrates a possible behavior of the PRES+ model.

2.4 Summary

To sum up, when used to model embedded systems, PRES+ has several interesting features to be highlighted, some of them inherited from the classical Petri net model:

- PRES+ allows representations at different levels of granularity.
- 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.
- 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 thus reduce the complexity of the model.
- Sequential as well as concurrent activities may be easily expressed in terms of Petri nets. Recall that concurrency is present in most embedded systems.
- Both control and data information might be captured by a unified design representation.
- PRES+ has been also extended by introducing the concept of hierarchy (see Section 3).
- Furthermore, the model is simple, intuitive, and can be easily handled by the designer.

We have developed a software tool, called SimPRES, that allows PRES+ models to be simulated. It has a graphical interface that lets the designer construct, modify, and simulate systems represented in PRES+. A screen shot of the SimPRES tool is shown in Figure 3. Such a tool is of great help for the designer because it allows visualizing the model of the system under design and running it, so that an animation of the dynamic behavior of the net is possible. SimPRES supports full graphical edition of the model of the system and provides methods to store/recover the net in/from a file.

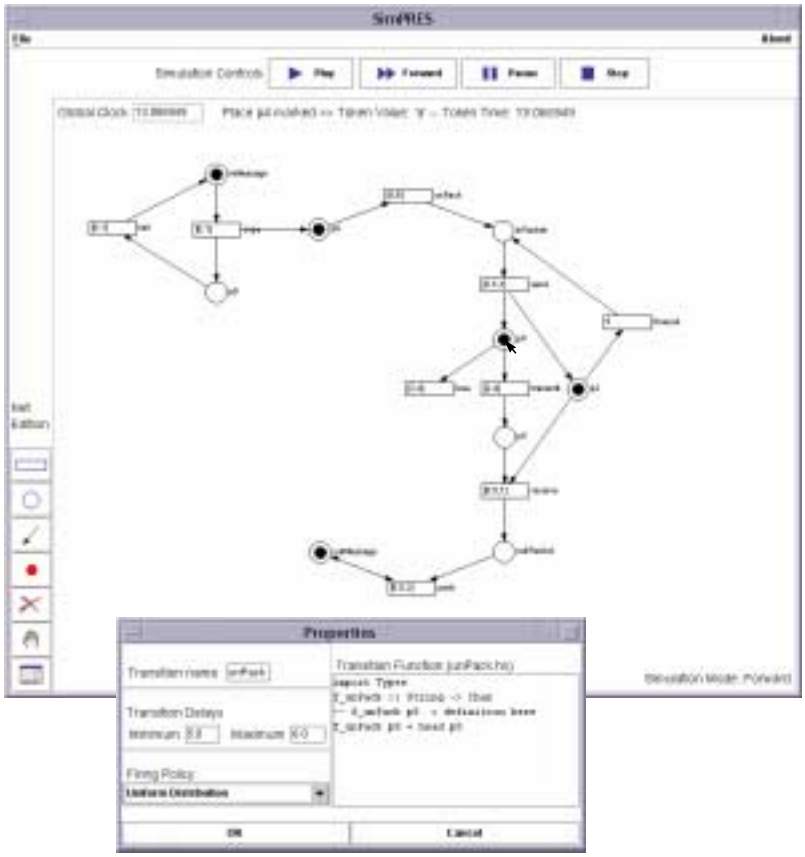

Figure 3: SimPRES: a simulator for PRES+ models

3. Notions of Equivalence and Hierarchy for PRES+

Several notions of equivalence for embedded systems represented in PRES+ are defined in this section. Such notions constitute the foundations of a framework to compare PRES+ models.

In this section we also extend PRES+ by introducing the concept of hierarchy. Hierarchy is a

convenient way to structure the system so modeling can done in a comprehensible form. Without hierarchical composition, it is difficult to specify and understand large systems.

3.1 Notions of Equivalence

The synthesis process requires a number of refinement steps starting from the initial system model until a more detailed representation is achieved. Such steps correspond to transformations in the system model so that design decisions are included in the representation.

The validity of a transformation depends on the concept of equivalence in which it is contrived. When we claim that two systems are equivalent, it is very important to understand the meaning of equivalence. Two equivalent systems are not necessarily the same but have properties that are common to both systems. Thus a clear notion of equivalence allows comparing systems and pointing out the properties in terms of which the systems are equivalent.

The following three definitions are basic concepts to be used when defining the four notions of equivalence for systems modeled in PRES+.

Definition 3.1. A marking M^+ is *immediately reachable* from M if there exists a transition $t \in T$ whose firing changes M into M^+. ▪

Definition 3.2. The *reachability set* $R(N)$ of a net N is the set of all markings reachable from M_0 and is defined by:
(i) $M_0 \in R(N)$;
(ii) If $M \in R(N)$ and M^+ is immediately reachable from M, then $M^+ \in R(N)$. ▪

Definition 3.3. A place $p \in P$ is said to be an *in-port* iff $(t, p) \notin O$ for all $t \in T$, that is, there is no transition t for which p is output place. Similarly, a place $p \in P$ is said to be an *out-port* iff $(p, t) \notin I$ for all $t \in T$, that is, there is no transition t for which p is input place. ▪

The set of in-ports is denoted inP while the set of out-ports is denoted $outP$.

Before formally presenting the notions of equivalence, we first give an intuitive idea of them. Such notions rely on the concepts of in-ports and out-ports: the initial condition to establish an equivalence relation between two nets N_1 and N_2 is that both have the same number of in-ports as well as out-ports. In this way, it is possible to define a one-to-one correspondence between in-ports and out-ports of the nets. Thus we can assume the same initial marking in corresponding in-ports and then check the tokens obtained in the out-ports after some transition firings in the nets. It is like an external observer putting in the same data in both nets and obtaining output information. If such an external observer can not distinguish between N_1 and N_2, based on the output data that he gets, then N_1 and N_2 are "equivalent". As defined later, such a concept is called *total-equivalence*. We also define weaker concepts of equivalence in which the external observer may actually distinguish between N_1 and N_2, but still there is some commonality in the data obtained in corresponding out-ports, namely number of tokens, token values, or token times.

We introduce the following notation to be used in the coming definitions: for a given marking M, $m(p)$ denotes the number of tokens in place p, i.e. $m(p)=|M(p)|$.

Definition 3.4. Two nets N_1 and N_2 are *cardinality-equivalent* or *N-equivalent* iff:
(i) There exist such bijections $f_{in} : inP_1 \to inP_2$ and $f_{out} : outP_1 \to outP_2$ that define one-to-one correspondences between in(out)-ports of N_1 and N_2;
(ii) The initial markings $M_{1,0}$ and $M_{2,0}$ satisfy
$M_{1,0}(p)=M_{2,0}(f_{in}(p)) \neq \varnothing$ for all $p \in inP_1$,
$M_{1,0}(q)=M_{2,0}(f_{out}(q))=\varnothing$ for all $q \in outP_1$;
(iii) For every $M_1 \in R(N_1)$ such that
$m_1(p)=0$ for all $p \in inP_1$,

$m_1(s)=m_{1,0}(s)$ for all $s \in P_1 - inP_1 - outP_1$

there exists $M_2 \in R(N_2)$ such that

$m_2(p)=0$ for all $p \in inP_2$,

$m_2(s)=m_{2,0}(s)$ for all $s \in P_2 - inP_2 - outP_2$,

$m_2(f_{out}(q))=m_1(q)$ for all $q \in outP_1$

and vice versa. ▪

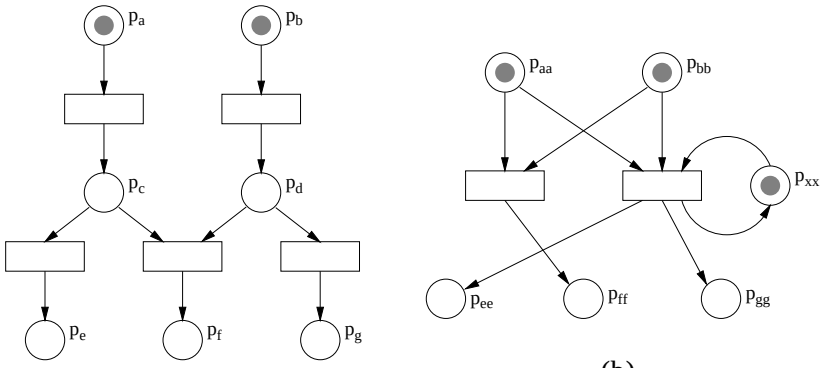

Figure 4: N-equivalent nets

The above definition expresses that if the same tokens are put in corresponding places of two N-equivalent nets, then the same number of tokens will be obtained in corresponding out-ports. Let us consider the nets N_1 and N_2 shown in Figures 4(a) and 4(b) respectively, in which we have abstracted away information not relevant for the current discussion like transition delays and token values. For such nets we have $inP_1=\{p_a, p_b\}$, $outP_1=\{p_e, p_f, p_g\}$, $inP_2=\{p_{aa}, p_{bb}\}$, $outP_2=\{p_{ee}, p_{ff}, p_{gg}\}$, and f_{in} and f_{out} are defined by $f_{in}(p_a)=p_{aa}$, $f_{in}(p_b)=p_{bb}$, $f_{out}(p_e)=p_{ee}$, $f_{out}(p_f)=p_{ff}$, and $f_{out}(p_g)=p_{gg}$. Let us assume that $M_{1,0}$ and $M_{2,0}$ satisfy condition (ii) in Definition 3.4. A simple reachability analysis shows there exist two cases in which the first part of condition (iii) in Definition 3.4. is satisfied: a) $m_1^{i}(p)=1$ if $p \in \{p_f\}$, and $m_1^{i}(p)=0$ otherwise; b) $m_1^{ii}(p)=1$ if $p \in \{p_e, p_g\}$, and $m_1^{ii}(p)=0$ otherwise. For each of these cases there exist a marking satisfying the second part of condition (iii) in Definition 3.4, respectively: a) $m_2^{i}(p)=1$ if $p \in \{p_{ff}, p_{xx}\}$, and $m_2^{i}(p)=0$ otherwise; b) $m_2^{ii}(p)=1$ if $p \in \{p_{ee}, p_{gg}, p_{xx}\}$, and $m_2^{ii}(p)=0$ otherwise. Hence N_1 and N_2 are N-equivalent.

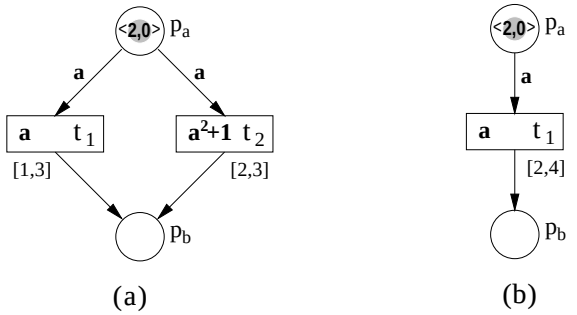

Figure 5: N-equivalent nets with different behavior

Before defining the concepts of *function-equivalence* and *time-equivalence*, let us study the simple nets N_1 and N_2 shown in Figures 5(a) and 5(b) respectively. It is straightforward to see that N_1 and N_2 fulfill the conditions established in Definition 3.4 and therefore are N-equivalent. However, note that N_1 may produce tokens with different values in its output: when t_1

fires the token in p_b will be $k_b=\langle 2, r_b^{i}\rangle$ with $r_b^{i} \in [1,3]$, but when t_2 fires the token in p_b will be $k_b=\langle 5, r_b^{ii}\rangle$ with $r_b^{ii} \in [2,3]$. The reason of this behavior is the non-determinism of N_1. On the other hand, when the only out-port of N_2 is marked, the corresponding token value will always be $v_b=2$.

As shown in the example of Figure 5, even if two nets are N-equivalent the tokens in their outputs may be different. For instance, there is no marking $M_2 \in R(N_2)$ in which the out-port has a token with value $v_b=5$, whereas it does exist $M_1 \in R(N_1)$ in which the out-port is marked and $v_b=5$. Thus the external observer could distinguish between N_1 and N_2 because of different token values—moreover different token times—in their out-ports when marked.

Definition 3.5. Two nets N_1 and N_2 are *function-equivalent* or *F-equivalent* iff:
(i) N_1 and N_2 are N-equivalent;
(ii) Let M_1 and M_2 be markings satisfying condition (iii) in Definition 3.4. For every $\langle v_1, r_1\rangle \in M_1(q)$, where $q \in outP_1$, there exists $\langle v_2, r_2\rangle \in M_2(f_{out}(q))$ such that $v_1=v_2$, and vice versa. ▪

Definition 3.6. Two nets N_1 and N_2 are *time-equivalent* or *T-equivalent* iff:
(i) N_1 and N_2 are N-equivalent;
(ii) Let M_1 and M_2 be markings satisfying condition (iii) in Definition 3.4. For every $\langle v_1, r_1\rangle \in M_1(q)$, where $q \in outP_1$, there exists $\langle v_2, r_2\rangle \in M_2(f_{out}(q))$ such that $r_1=r_2$, and vice versa. ▪

Two nets are F-equivalent if, besides being N-equivalent, the tokens obtained in corresponding out-ports have the same token value. Similarly, if tokens obtained in corresponding out-ports have the same token time, the nets are T-equivalent.

Definition 3.7. Two nets N_1 and N_2 are *total-equivalent* or *§-equivalent* iff:
(i) N_1 and N_2 are F-equivalent;
(ii) N_1 and N_2 are T-equivalent. ▪

Figure 6 shows the relation between the different concepts of equivalence introduced above. The graph captures the dependence between the notions of equivalence. Thus, for instance, N-equivalence is necessary for T-equivalence and also for F-equivalence. Similarly, §-equivalence implies all other equivalences. §-equivalence is the strongest notion of equivalence defined in this work.

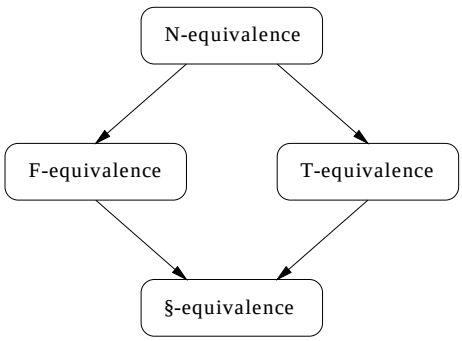

Figure 6: Relation between the notions of equivalence

3.2 Hierarchical PRES+ Model

Embedded systems require sound models along their design cycle. PRES+ supports systems modeled at different levels of granularity with transitions representing simple arithmetic operations or complex algorithms. However, in order to handle efficiently the modeling of large systems, a mechanism of hierarchical composition is needed so that the model may be con-

structed in a structured manner, composing simple units fully understandable by the designer.

Hierarchical modeling can be conveniently applied along the design process of embedded systems. Sometimes the specification or requirements may not be complete or thoroughly understood. In a top-down approach, a designer may define the interface to each component and then gradually refine those components. On the other hand, a system may be constructed reusing existing elements such as IP blocks in a bottom-up approach. A hierarchical PRES+ model can be devised bottom-up, top-down, or by mixing both approaches.

A flat representation of a real-life embedded system can be too big and complex to handle and understand. The concept of hierarchy allows systems to be modeled in a structured way. Thus the system may be broken down into a set of comprehensible nets structured in a hierarchy. Each one of these nets may represent a sub-block of the current design. Such a sub-block can be a pre-designed IP component as well as a design alternative corresponding to a subsystem of the system under design.

In this section we formalize the concept of hierarchy for PRES+ models. Some trivial examples are used in order to illustrate the definitions.

Definition 3.8. A transition $t \in T$ is an *in-transition* of $N=(P, T, I, O, M_0)$ iff $\bigcup_{p \in inP} p^\circ = \{t\}$. In a similar manner, a transition $t \in T$ is an *out-transition* of N iff $\bigcup_{p \in outP} {}^\circ p = \{t\}$. ▪

Note that the existence of non-empty sets *inP* and *outP* is a necessary condition for the existence of in- and out-transitions. For the net N_1 shown in Figure 7, $inP_1=\{p_a, p_b\}$, $outP_1=\{p_d\}$, and t_{in} and t_{out} are in-transition and out-transition respectively.

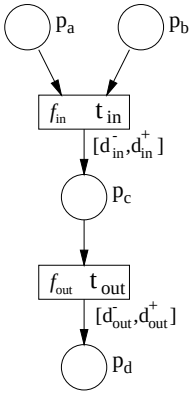

Figure 7: A simple subnet N_1

Definition 3.9. An *abstract PRES+* model is a six-tuple $H=(P, T, \Lambda, I, O, M_0)$ 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 set of transitions;
$\Lambda=\{\boldsymbol{S}_1, \boldsymbol{S}_2, \ldots, \boldsymbol{S}_l\}$ is a finite set of *super-transitions*;
$I \subseteq P \times (\Lambda \cup T)$ is a finite set of input arcs;
$O \subseteq (\Lambda \cup T) \times P$ is a finite set of output arcs;
M_0 is the initial marking. ▪

Observe that a (non-abstract) PRES+ net is a particular case of an abstract PRES+ net with $\Lambda=\varnothing$. Figure 8 illustrates a hierarchical PRES+ net. Super-transitions are represented by thick-line boxes.

Definition 3.10. The *pre-set* ${}^\circ\boldsymbol{S}$ and *post-set* $\boldsymbol{S}^\circ$ of a super-transition $\boldsymbol{S} \in \Lambda$ are given by ${}^\circ\boldsymbol{S}=\{p \in P | (p, \boldsymbol{S}) \in I\}$ and $\boldsymbol{S}^\circ=\{p \in P | (\boldsymbol{S}, p) \in O\}$ respectively. ▪

Similar to transitions, the pre(post)-set of a super-transition $\boldsymbol{S} \in \Lambda$ is the set of input(output) places of $\boldsymbol{S}$.

Definition 3.11. For every super-transition $\boldsymbol{S} \in \Lambda$ there exists a *high-level function* $\boldsymbol{g} : \tau(p_1) \times \tau(p_2) \times \ldots \times \tau(p_a) \rightarrow \tau(q)$ associated to $\boldsymbol{S}$, where $^\circ\boldsymbol{S}=\{p_1, p_2, \ldots, p_a\}$ and $q \in \boldsymbol{S}^\circ$. ■

Recall that $\tau(p)$ denotes the *type* associated with the place $p \in P$, i.e. the type of value that a token may bear in that place. Observe the usefulness of high-level functions associated to super-transitions in, for instance, a top-down approach: for a certain component of the system, the designer may define its interface and a high-level description of its functionality through a super-transition, and in a later design phase refine the component. In current design methodologies it is also very common to reuse predefined elements such as IP blocks. In such cases, the internal structure of the component is unknown to the designer and therefore the block is best modeled by a super-transition and its high-level function.

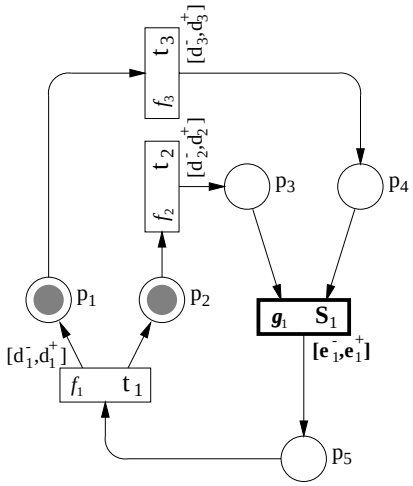

Figure 8: A hierarchical PRES+ model

Definition 3.12. For every super-transition $\boldsymbol{S} \in \Lambda$ there exist a *minimum estimated delay* $\boldsymbol{e}^-$ and a *maximum estimated delay* $\boldsymbol{e}^+$, where $\boldsymbol{e}^- \leq \boldsymbol{e}^+$ are non-negative real numbers that represent the estimated lower and upper limits for the execution time of the high-level function associated to $\boldsymbol{S}$. ■

Definition 3.13. A super-transition may not be in *conflict* with other transitions or super-transitions, that is:
(i) $^\circ\boldsymbol{S}_1 \cap {}^\circ\boldsymbol{S}_2=\varnothing$ and $\boldsymbol{S}_1^\circ \cap \boldsymbol{S}_2^\circ=\varnothing$ for all $\boldsymbol{S}_1, \boldsymbol{S}_2 \in \Lambda$ such that $\boldsymbol{S}_1 \neq \boldsymbol{S}_2$;
(ii) $^\circ\boldsymbol{S} \cap {}^\circ t=\varnothing$ and $\boldsymbol{S}^\circ \cap t^\circ=\varnothing$ for all $\boldsymbol{S} \in \Lambda$, $t \in T$. ■

In other words, a super-transition may not "share" input places with other transitions/ super-transitions, nor output places. In what follows, the input and output places of a super-transition will be called *surrounding* places.

Definition 3.14. A super-transition $\boldsymbol{S}_i \in \Lambda$ together with its surrounding places in the hierarchical net $H=(P, T, \Lambda, I, O, M_0)$ is a *semi-abstraction* of the (hierarchical) subnet $N_i=(P_i, T_i, \Lambda_i, I_i, O_i, M_{i,0})$ (or conversely, N_i is a *semi-refinement* of $\boldsymbol{S}_i$ and its surrounding places) iff:
(i) There exists a unique in-transition $t_{in} \in T_i$;
(ii) There exists a unique out-transition $t_{out} \in T_i$;
(iii) There exists a bijection $h_{in} : {}^\circ\boldsymbol{S}_i \rightarrow inP_i$ that maps the input places of $\boldsymbol{S}_i$ onto the in-ports of N_i;
(iv) There exists a bijection $h_{out} : \boldsymbol{S}_i^\circ \rightarrow outP_i$ that maps the output places of $\boldsymbol{S}_i$ onto the out-ports of N_i;
(v) $M_0(p)=M_{i,0}(h_{in}(p))$ and $\tau(p)=\tau(h_{in}(p))$ for all $p \in {}^\circ\boldsymbol{S}_i$;

(vi) $M_0(p)=M_{i,0}(h_{out}(p))$ and $\tau(p)=\tau(h_{out}(p))$ for all $p \in \boldsymbol{S}_i^{\circ}$;
(vii) t is disabled in the initial marking $M_{i,0}$ for all $t \in (T_i\text{-}t_{in})$. ■

A subnet may, in turn, contain super-transitions. It is straightforward to prove that the net N_1 of Figure 7 is indeed a semi-refinement of $\boldsymbol{S}_1$ in the hierarchical net of Figure 8.

If a net N_i is the semi-refinement of some super-transition $\boldsymbol{S}_i$, it is possible to *characterize* N_i in terms of both function and time by putting tokens in its in-ports and then observing the value and time stamp of tokens in its out-ports after a certain firing sequence. If the time stamp of all tokens deposited in the in-ports of N_i is zero, the token time of tokens obtained in the out-ports is called the *execution time* of N_i. For example, the net N_1 shown in Figure 7 can be characterized by putting tokens $k_a=\langle v_a, 0\rangle$ and $k_b=\langle v_b, 0\rangle$ in its in-ports and observing the token $k_d=\langle v_d, r_d\rangle$ after firing t_{in} and t_{out}. Thus the execution time of N_1 is equal to the token time r_d, bounded in this case by $d_{in}^- + d_{out}^- \le r_d \le d_{in}^+ + d_{out}^+$. Note the token value v_d is given by $v_d=f_{out}(f_{in}(v_a, v_b))$, where f_{in} and f_{out} are the transition functions of t_{in} and t_{out} respectively.

The definition of semi-abstraction/refinement is just "syntactic sugar" that allows a complex design to be constructed in a structured way by composing simpler entities. We have not defined, so far, a semantic relation between the functionality of super-transitions and their refinements. Below we define the concepts of *strong* and *weak refinement* of a super-transition.

Definition 3.15. A subnet $N_i=(P_i, T_i, \Lambda_i, I_i, O_i, M_{i,0})$ is a *strong refinement* of the super-transition $\boldsymbol{S}_i \in \Lambda$ together with its surrounding places in the hierarchical net $H=(P, T, \Lambda, I, O, M_0)$ (or $\boldsymbol{S}_i$ and its surrounding places is a *strong abstraction* of N_i) iff:
(i) N_i is a semi-refinement of $\boldsymbol{S}_i$;
(ii) N_i "implements" $\boldsymbol{S}_i$, that is, N_i is *function-equivalent* to $\boldsymbol{S}_i$ and its surrounding places;
(iii) The minimum estimated delay $\boldsymbol{e}_i^-$ of $\boldsymbol{S}_i$ is equal to the lower bound of the execution time of N_i;
(iv) The maximum estimated delay $\boldsymbol{e}_i^+$ of $\boldsymbol{S}_i$ is equal to the upper bound of the execution time of N_i. ■

The subnet N_1 shown in Figure 7 is a semi-refinement of $\boldsymbol{S}_1$ in the hierarchical net of Figure 8. N_1 is a strong refinement of the super-transition $\boldsymbol{S}_1$ if, in addition: (a) $\boldsymbol{g}_1=f_{out} \circ f_{in}$; (b) $\boldsymbol{e}_i^-=d_{in}^-+d_{out}^-$; (c) $\boldsymbol{e}_i^+=d_{in}^++d_{out}^+$ (Definitions 3.15(ii), 3.15(iii), and 3.15(iv) respectively).

Observe that the concept of strong refinement requires the super-transition and its strong refinement to have the very same time limits. Such a concept could have limited practical use since the high-level description and the implementation perform the same function but typically have different timings and therefore their bounds for the execution time do not coincide. We relax the requirement of exact correspondence of lower and upper bounds on time; this yields to a weaker notion of refinement, yet more practical.

Definition 3.16. A subnet $N_i=(P_i, T_i, \Lambda_i, I_i, O_i, M_{i,0})$ is a *weak refinement* of the super-transition $\boldsymbol{S}_i \in \Lambda$ together with its surrounding places in the hierarchical net $H=(P, T, \Lambda, I, O, M_0)$ (or $\boldsymbol{S}_i$ and its surrounding places is a *weak abstraction* of N_i) iff:
(i) N_i is a semi-refinement of $\boldsymbol{S}_i$;
(ii) N_i "implements" $\boldsymbol{S}_i$, that is, N_i is *function-equivalent* to $\boldsymbol{S}_i$ and its surrounding places;
(iii) The minimum estimated delay $\boldsymbol{e}_i^-$ of $\boldsymbol{S}_i$ is less than or equal to the lower bound of the execution time of N_i;
(iv) The maximum estimated delay $\boldsymbol{e}_i^+$ of $\boldsymbol{S}_i$ is greater than or equal to the upper bound of the execution time of N_i. ■

In the sequel whenever we refer to *refinement* it will mean *weak refinement*.

Given a hierarchical PRES+ net $H=(P, T, \Lambda, I, O, M_0)$ and refinements of its super-transitions, it is possible to construct an equivalent non-hierarchical net. For the sake of clarity, in the following discussion we will consider nets with a single super-transition, nonetheless these

concepts can be easily extended to the general case.

Definition 3.17. Let us consider the net $H=(P, T, \Lambda, I, O, M_0)$ where $\Lambda=\{\boldsymbol{S}_1\}$, and let the sub-net $N_1=(P_1, T_1, \Lambda_1, I_1, O_1, M_{1,0})$ be a refinement of $\boldsymbol{S}_1$ and its surrounding places. Let $t_{in}, t_{out} \in T_1$ be unique in-transition and out-transition respectively. Let inP_1 and $outP_1$ be respectively the sets of in-ports and out-ports of N_1. The equivalent net $H'=(P', T', \Lambda', I', O', M_0')$, *one level lower*, is defined as follows:

(i) $\Lambda'=\Lambda_1$;

(ii) $P'=P \cup (P_1\text{-}inP_1\text{-}outP_1)$;

(iii) $T'=T \cup T_1$;

(iv) $(p, \boldsymbol{S}) \in I'$ if $(p, \boldsymbol{S}) \in I_1$;
$(p, t) \in I'$ if $(p, t) \in I$, or $(p, t) \in I_1$ and $p \notin inP_1$;
$(p, t_{in}) \in I'$ if $(p, \boldsymbol{S}_1) \in I$;

(v) $(\boldsymbol{S}, p) \in O'$ if $(\boldsymbol{S}, p) \in O_1$;
$(t, p) \in O'$ if $(t, p) \in O$, or $(t, p) \in O_1$ and $p \notin outP_1$;
$(t_{out}, p) \in O'$ if $(\boldsymbol{S}_1, p) \in O$;

(vi) $M_0'(p)=M_0(p)$ for all $p \in P$;
$M_0'(p)=M_{1,0}(p)$ for all $p \in P_1\text{-}inP_1\text{-}outP_1$. ■

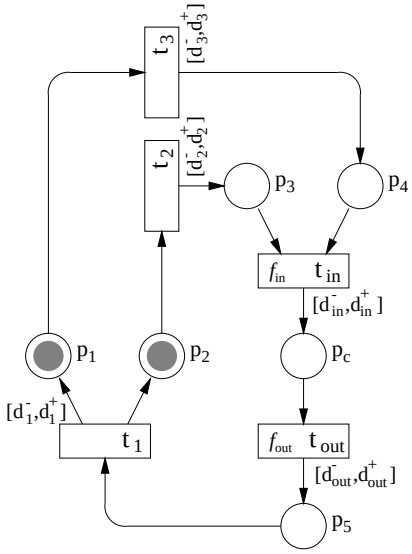

Figure 9: A non-hierarchical PRES+ model

Given the hierarchical net of Figure 8 and being N_1 (Figure 7) a refinement of $\boldsymbol{S}_1$, we can construct the equivalent non-hierarchical net as illustrated in Figure 9.

3.2.1 Hierarchical Modeling of a GMDFα

In this section we model a GMDFα (Generalized Multi-Delay frequency-domain Filter) [Fre97] using PRES+. GMDFα has been used in acoustic echo cancellation for improving the quality of hand-free phone and teleconference applications. The GMDFα algorithm is a frequency-domain block adaptive algorithm: a block of input data is processed at one time, producing a block of output data. The impulse response of length L is segmented into K smaller blocks of size N ($K=L/N$), thus leading to better performance. R new samples are processed at each iteration and the filter is adapted α times per block ($R=N/\alpha$).

The filter inputs are the signal X and its echo E, and the output is the reduced or cancelled echo E'. In Figure 10 we show the hierarchical PRES+ model of a GMDFα. The transition t_1

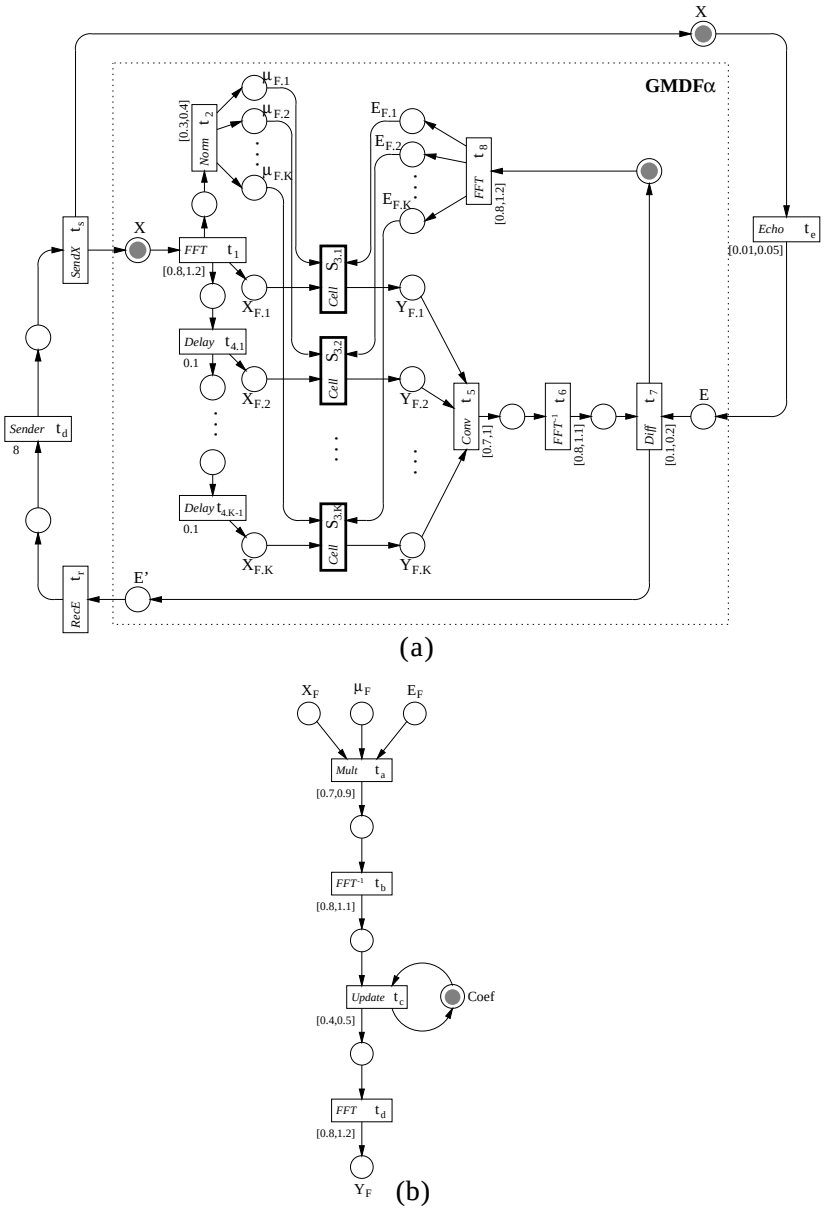

Figure 10: GMDFα modeled using PRES+

transforms the input signal X into the frequency domain by a FFT (Fast Fourier Transform). t_2 corresponds to the normalization block. In each one of the basic cells $\boldsymbol{S}_{3.i}$ the filter coefficients are updated. Transitions $t_{4.i}$ serve as delay blocks. t_5 computes the estimated echo in the frequency domain by a convolution product and then it is converted into the time domain by t_6. The difference between the estimated echo and the actual one (signal E) is calculated by t_7 and output as E'. Such a cancelled echo is also transformed into the frequency domain by t_8 to be used in the next iteration when updating the filter coefficients. InFigure 10 we also model the environment with which the GMDFα interacts: t_e models the echoing of signal X, t_s and t_r represent, respectively, the sending of the signal and the reception of the cancelled echo, and t_d is the entity that emits X.

The refinement of the basic cells $\boldsymbol{S}_{3.i}$ is shown inFigure 10(b) where the filter coefficients are computed and thus the filter is adapted by using FFT^{-1} and FFT operations. It is worth noticing that instances of the same subnet (Figure 10(b)) are used as refinements of the different cells $\boldsymbol{S}_{3.i}$. Transition delays in Figure 10 are given in milliseconds.

4. Formal Verification of Embedded Systems

As the complexity of electronic systems increases, the likelihood of subtle errors becomes much greater. A way to cope, to a certain extent, with the issue of correctness is the use of mathematically-based techniques, known as *formal methods*.

Correctness plays a key role in embedded systems. For the levels of complexity typical to modern electronic systems, traditional validation techniques like simulation and testing are not enough to verify the correctness of such systems. First, these methods may cover just a small fraction of the system behavior. Second, bugs found late in prototyping phases have a negative impact on time-to-market. Third, as more applications become dependent on computer systems, a failure may lead to catastrophic situations, e.g. in safety-critical systems like transportation, defense, and medical applications.

In this section we introduce our approach to formal verification of real-time embedded systems represented in PRES+.

4.1 Analyses of PRES+ Models

There are several types of analysis that can be performed on systems represented in PRES+. The absence or presence of tokens in places of the net may represent the state of the system at a certain moment in the dynamic behavior of the net. Based on this, different properties can be studied. For instance, two places marked simultaneously could represent a dangerous situation that must be avoided. This sort of safety requirement might be formally proved by checking that such dangerous state is never reached. Also, the designer could be interested in proving that the system eventually reaches a certain state, in which the presence of tokens in a particular place represents the completion of a task. This kind of analysis, absence/presence of tokens in places of the net, is termed *reachability analysis*.

The type of analysis described above is 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 eventually be marked and that its time stamp will be less than a certain time value that represents a temporal constraint. Such a study is called *time analysis*.

A third type of analysis for systems modeled in PRES+ involves reasoning about values of tokens in marked places. Such kind of study is called *functionality analysis*. In this report 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.

4.2 Our Approach to Formal Verification

Model checking is one of the well-established approaches to formal verification: a number of desired properties (called in this context *specification*) are checked against a given model of the system. The two inputs to the model checking problem are the system model and the properties that such a system must satisfy, usually expressed as temporal logic formulas.

The purpose of our verification approach is to formally reason about real-time embedded systems represented in PRES+. For verification purposes, we restrict ourselves to *safe* PRES+ nets, that is, every place $p \in P$ holds at most one token for every marking M reachable from

M_0. Otherwise, the formal analysis would become more cumbersome. This is a trade-off between expressiveness and analysis power.

Our approach allows determining the truth of formulas expressed in CTL [Cla86] and TCTL (Timed CTL) [Alu90] with respect to a (safe) PRES+ model. 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). For instance, **AF** p holds if for every possible path there exists at least one state in which p is satisfied, that is, p will eventually happen. TCTL is a real-time extension of CTL that allows inscribing subscripts on the temporal operators to limit their scope in time. For instance, $\mathbf{AF}_{<n}\ P$ expresses that, along all computation paths, the property P becomes true within n time units. In our approach the atomic propositions of CTL/TCTL correspond to the absence/presence of tokens in places in the net. Thus the atomic proposition p holds iff $p \in P$ is marked.

In order to verify the correctness of a real-time embedded system, we propose a systematic procedure to translate PRES+ into timed automata so that it is possible to make use of existing model checking tools, namely HyTech [HyT], KRONOS [Kro], and UPPAAL [Upp]. Figure 11 summarizes our general approach to formal verification of embedded systems using model checking. The system is described by a PRES+ model and the properties it must satisfy are expressed by CTL/TCTL formulas. The model checker automatically verifies whether the required properties hold in the model of the system. In case the CTL/TCTL formulas are not satisfied, diagnostic information is generated. Given enough resources, the procedure will terminate with a *yes/no* answer. However, due to the huge state space of practical systems, it might be the case that it is not feasible to obtain an answer at all, even though in theory the procedure will "always" terminate (probably after many years and enough memory). That case corresponds to *???* in Figure 11.

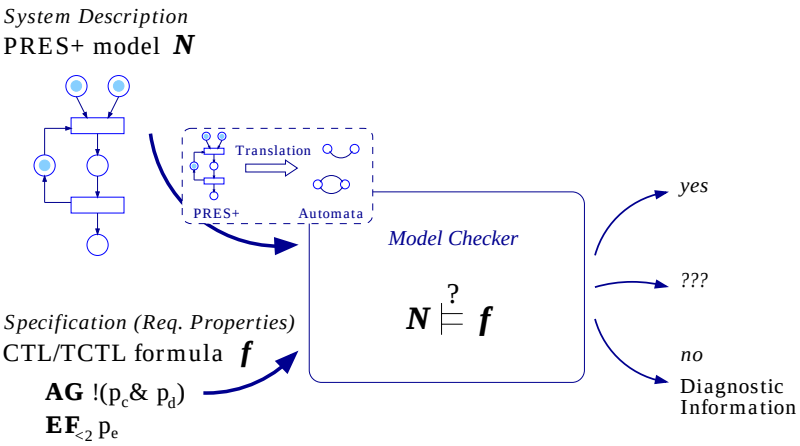

Figure 11: Model checking

The verification of hierarchical PRES+ models is done by constructing the equivalent non-hierarchical net as stated in Definition 3.17, and then using the translation procedure discussed in the next section. Note that obtaining the non-hierarchical PRES+ model can be done automatically so that the designer is not concerned with flattening the net: he just inputs a hierarchical PRES+ model as well as the properties he is interested in.

4.3 Translating PRES+ into Timed Automata

A timed automaton is a finite automaton augmented with a finite set of real-valued clocks [Alu99]. Timed automata can be thought as a collection of automata which operate and coordinate with each other through shared variables and synchronization labels. There is a set of real-valued variables, named *clocks*, all of which change along the time with the same constant rate. There might be conditions over clocks that express timing constraints.

An extended timed automata model can be expressed as a tuple $\overrightarrow{M} = (L, L_0, E, \Sigma, \sigma, X, V, \Phi, \upsilon, R, A, I)$, where

L is a finite set of *locations*;

$L_0 \subseteq L$ is a set of *initial locations*;

$E \subseteq L \times L$ is a set of *edges*;

Σ is a finite set of *labels*;

$\sigma : E \to \Sigma$ is a mapping that labels each edge in E with some label in Σ;

X is a finite set of real-valued *clocks*;

V is a finite set of *variables*;

Φ is a mapping that assigns to each edge $e=(l, l')$ a *clock condition* $\Phi(e)$ over X that must be satisfied in order to allow the automaton to change its location from l to l';

υ is a mapping that assigns to each edge $e=(l, l')$ a *variable condition* $\upsilon(e)$ over V that must be satisfied in order to allow the automaton to change its location from l to l';

$R : E \to 2^X$ is a *reset function* that gives the clocks to be reset on each edge;

A is the *activity mapping* that assigns to each edge e a set of *activities* $A(e)$;

I is a mapping that assigns to each location l an invariant $I(l)$ which allows the automaton to stay at location l as long as its invariant is satisfied.

In order to use existing model checking tools, we first translate the PRES+ model into timed automata. In the procedure presented in this chapter, the resulting model will consist of one automaton and one clock for each transition in the Petri net. We use the PRES+ model shown in Figure 12 in order to illustrate the translation procedure. Figure 13 shows the resulting timed automata.

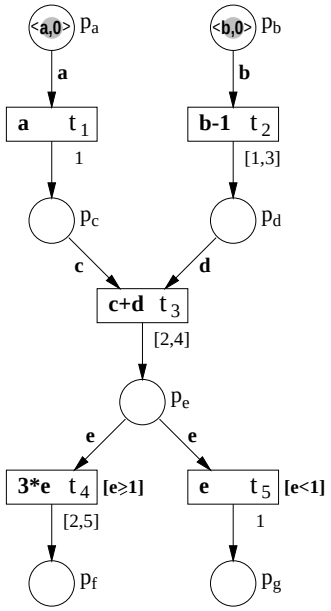

Figure 12: PRES+ model to be translated into automata

Step 4.1. Define one clock c_i in X for each transition t_i of the Petri net. Define one variable in

V for each place p_x of the Petri net, corresponding to the token value v_x when p_x is marked. ▪

The clock c_i is used to ensure the firing of the transition t_i within its earliest-latest trigger time interval. For the example in Figure 12, using the short notation w to denote v_w, $X=\{c_1, c_2, c_3, c_4, c_5\}$, $V=\{a, b, c, d, e, f, g\}$.

Step 4.2. Define the set Σ of labels as the set of transitions in the Petri net. ▪

Step 4.3. For every transition t_i in the Petri net, define an automaton $\vec{t}_i$ with z+1 locations $s_1, s_2, \ldots, s_z, en$, where z is the number of transitions that, when fired, will deposit a token in some place of the pre-set ${}^\circ t_i$. The set of such transitions is defined by $pr(t_i)= \bigcup_{p \in {}^\circ t_i} {}^\circ p$. In the case $pr(t_i)=\varnothing$, define an automaton with only two locations s_1 and en. ▪

The resulting model consists of five automata. The automaton $\vec{t}_3$, for instance, has three locations.

Step 4.4. Given the automaton $\vec{t}_i$, corresponding to transition t_i:
a) Transition t_i is not in conflict. Let $z=|pr(t_i)|$. Define z edges (s_1, s_2), z edges (s_2, s_3), ..., and z edges (s_z, en). Then assign, to each group of z edges, synchronization labels corresponding to the transitions in $pr(t_i)$. Define then one edge (en, s_1) with synchronization label t_i;
b) Transition t_i is in conflict with another transition tc. Let $A=pr(t_i) \cap pr(tc)$, $B=pr(t_i)-pr(tc)$, $x=|A|$, $y=|B|$, and $z=|pr(t_i)|$. Split each one of the locations $s_2, \ldots, s_z$ into $s_{2,a}, \ldots, s_{z,a}$ and $s_{2,b}, \ldots, s_{z,b}$. Then define y edges $(s_{2,a}, s_{3,a})$, y edges $(s_{3,a}, s_{4,a})$, ..., y edges $(s_{z,a}, en)$, y edges $(s_1, s_{2,b})$, ..., and y edges $(s_{z-1,b}, s_{z,b})$, each group with synchronization labels corresponding to those transitions in B. Define x edges $(s_1, s_{2,a})$, x edges $(s_{2,b}, s_{3,a})$ ..., and x edges $(s_{z,b}, en)$, each group with synchronization labels corresponding to transitions in A. Define then one edge $(s_{2,a}, s_1)$, one edge $(s_{3,a}, s_{2,b})$, ..., and one edge $(en, s_{z,b})$, each with synchronization label tc. Finally, define one edge (en, s_1) with synchronization label t_i. ▪

For example, transition t_3 in the model of Figure 12 is not in conflict and, therefore, case a) applies. Since $pr(t_3)=\{t_1, t_2\}$, for the automaton $\vec{t}_3$ there are two edges (s_1, s_2), and two edges (s_2, en) with labels t_1 and t_2 as shown in Figure 13. The edge (en, s_1) has label t_3.

On the other hand, t_4 is in conflict with t_5 and case b) applies. Since $pr(t_4)=pr(t_5)=\{t_3\}$ the automaton $\vec{t}_4$ still has two locations as shown in Figure 13. If transition t_5 did not exist, the automaton $\vec{t}_4$ would not have the edge (en, s_1) with synchronization label t_5.

In the following, let f_i be the transition function associated to t_i, ${}^\circ t_i$ the pre-set of t_i, and d_i^- and d_i^+ the minimum and maximum transition delays associated to t_i.

Step 4.5. Given the automaton $\vec{t}_i$, for every edge $e_k=(s_z, en)$ define $R(e_k)=\{c_i\}$. For any other edge e in $\vec{t}_i$ define $R(e)=\varnothing$. Define the invariant of location en as $c_i \le d_i^+$ in order to enforce the firing of t_i before or at its latest trigger time. ▪

This means that in all edges (s_z, en) the clock c_i will be reset. In Figure 13, the assignment c_i:=0 represents the reset of c_i. The two edges (s_2, en) of automaton $\vec{t}_3$, for example, have c_3:=0 inscribed on them. c_3 is used to take into account the time since t_3 becomes enabled and ensure the firing semantics of PRES+.

Step 4.6. Given $\vec{t}_i$ and its edge $e=(en, s_1)$ with synchronization label t_i, assign to e the clock condition $d_i^- \le c_i \le d_i^+$. For every $p_j \in t_i^\circ$ assign to such an edge e the activity v_j:=f_i. ▪

For example, in the case of the automaton $\vec{t}_2$ the condition $1 \le c_2 \le 3$ gives the lower and upper limits for the firing of t_2, while the activity d:=b-1 expresses that whenever the automaton $\vec{t}_2$ changes from en to s_1, i.e. t_2 fires, the value b-1 is assigned to the variable d.

Step 4.7. Given the automaton $\vec{t}_i$, if the transition t_i has guard G_i, assign the variable condition G_i to the edge (en, s_1) with synchronization label t_i. Then add an edge $e=(en, en)$ with no synchronization label, condition $\overline{G_i}$ (the complement of G_i), and $R(e)=\{c_i\}$. ▪

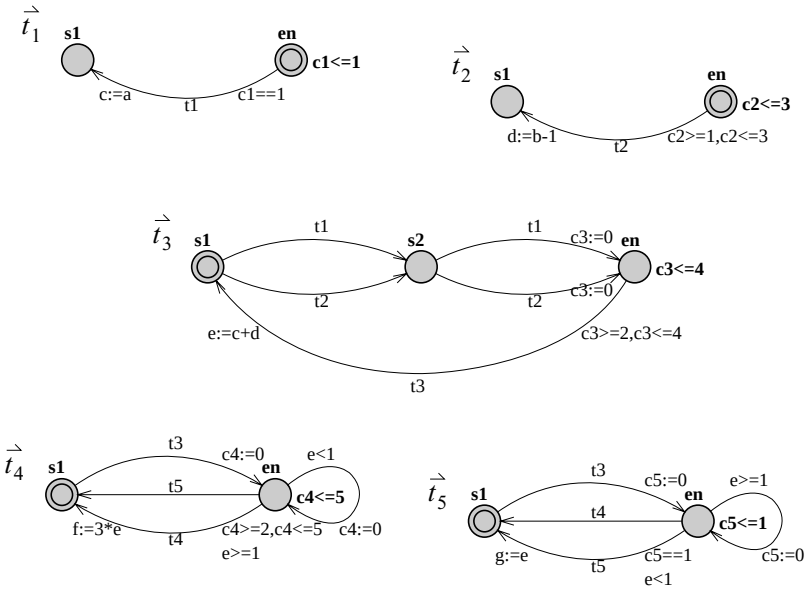

Figure 13: Timed Automata equivalent to the PRES+ model of Figure 12

Note the condition $e < 1$ assigned to the edge (en, s_1) in the automaton $\vec{t}_5$, where $e < 1$ represents the guard of t_5. Observe also the edge (en, en) with condition $e \geq 1$ and c_5:=0.

Step 4.8. If the transition t_i is enabled in the initial marking, make the location *en* the initial location of $\vec{t}_i$. Otherwise, if there are k places initially marked in the pre-set $^\circ t_i$ of the transition t_i ($0 \leq k < |^\circ t_i|$ so that t_i is not enabled), make s_{k+1} the initial location of $\vec{t}_i$. ▪

In our example, *en* is the initial location of $\vec{t}_1$ because the transition t_1 is enabled in the initial marking of the net. Since no place in $^\circ t_3$ is initially marked, the automaton $\vec{t}_3$ has s_1 as initial location.

Once we have the equivalent timed automata, we can verify properties against the model of the system. For instance, in the simple system of Figure 12 we could check whether, for given values of a and b, there exists a reachable state in which p_f is marked. This property can be expressed as a CTL formula **EF** p_f. If we want to check temporal properties we can express them as TCTL formulas. Thus, we could check whether p_g will possibly be marked and the time stamp of its token be less than 5 time units, expressing this property as $\mathbf{EF}_{<5}\ p_g$.

Some of the model checking tools, namely HyTech [HyT], are capable of performing parametric analyses. Then, for the example shown in Figure 12, we can ask the model-checker which values of a and b make a certain property hold in the system model. For instance, we obtain that **EF** p_g holds if $a+b < 2$.

Due to the nature of the model checking tools that we use, the translation procedure introduced above is applicable for PRES+ models in which transition functions are expressed using arithmetic operations and token types of all places are rational. 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 absence/presence of tokens nor time stamps. This is the case of PRES+ models that bear no guards and, therefore, they can straightforwardly be 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 12. There are many sys-

tems 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. Reduction of Verification Complexity by using Transformations

The advantages of transformations in the verification of embedded systems are addressed in this section. We have introduced an approach to the formal verification of systems modeled in PRES+. The verification efficiency can be improved considerably by using a transformational approach. The model that we use to represent embedded systems supports a transformation based concept which is of great benefit in the formal verification process.

For the sake of reducing the verification effort, we first transform the system model into a simpler one, still semantically equivalent, and then verify the simplified model. If a given model is modified using correctness-preserving transformations and then the resulting one is proved correct with respect to its specification, the initial model is guaranteed to be correct by construction and no intermediate steps need to be verified. This simple observation allows us to reduce significantly the complexity of verification.

5.1 Transformations

As it was argued in Section 3, the concept of hierarchy makes it possible to model systems in a structured way. Thus, using the notion of abstraction/refinement, the system may be broken down into a set of comprehensible nets.

Transformations performed on large and flat systems are, in general, difficult to handle. Hierarchical modeling permits a structural representation of the system in such a way that the composing (sub)nets are simple enough to be transformed efficiently.

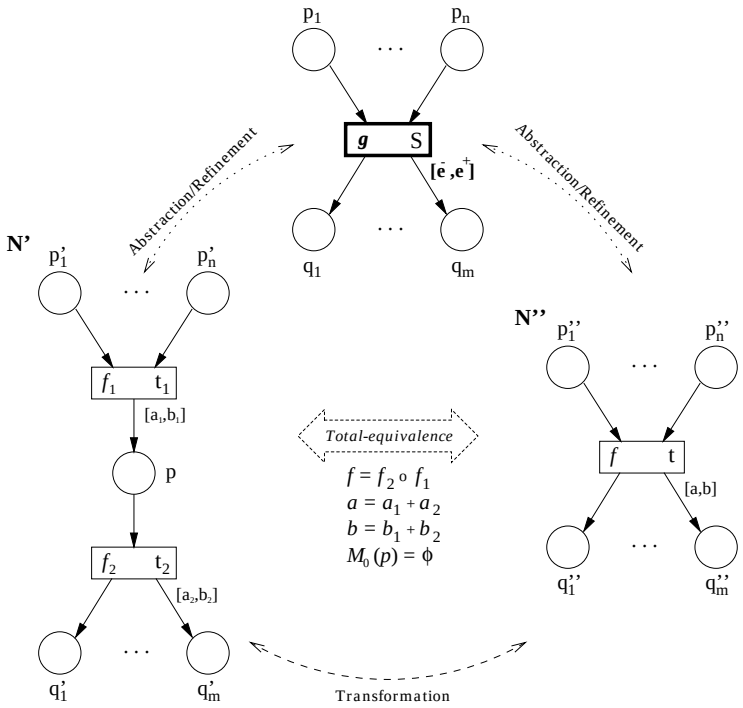

Figure 14: Transformation rule TR1

We can define a set of transformation rules that make it possible to transform only a part of

the system model. A simple but useful transformation is shown in Figure 14. It is not difficult to formally prove that N' and N'' are total-equivalent, provided that the conditions given in Figure 14 are satisfied. It is interesting to observe that if the net N' is a refinement of a certain super-transition $\boldsymbol{S} \in \Lambda$ in the hierarchical net $H=(P, T, \Lambda, I, O, M_0)$ and N' is transformed into N'' (so that N' and N'' are total-equivalent), then N'' is also a refinement of $\boldsymbol{S}$ and may be used instead of N'. Such a transformation does not change the overall system at all. First, having tokens with the same token value and time in corresponding in-ports of N' and N'' will lead to a marking with the very same token value and time in corresponding out-ports, so that the external observer (i.e. the rest of the net H) can not distinguish between N' and N''. Second, once tokens are put in the in-ports of the subnets, there is nothing that externally "disturbs" the behavior of the subnets N' and N'' (for example a transition in conflict with the in-transition that could take away tokens from the in-ports) because, by definition, super-transitions may not be in conflict. Thus the overall behavior is the same using either N' or N''. Such a transformation rule could be used, therefore, to simplify PRES+ models and accordingly reduce the complexity of the verification process.

It is worth clarifying the concept of transformation in the context of verification. Along the design flow, the system model is refined to include different design decisions, like architecture selection, partitioning, and scheduling. Such refinements is what we call *vertical transformations*. On the other hand, at certain stage of the design flow, the system model can be transformed into another one that preserves certain properties under consideration and, at the same time, makes easier the verification process. These are called *horizontal transformations*.

Horizontal transformations are a mathematical tool to deal with the verification complexity. By simplifying the representation to be model-checked, the verification cost is reduced in a significant manner. In this report, we concentrate on horizontal transformations.

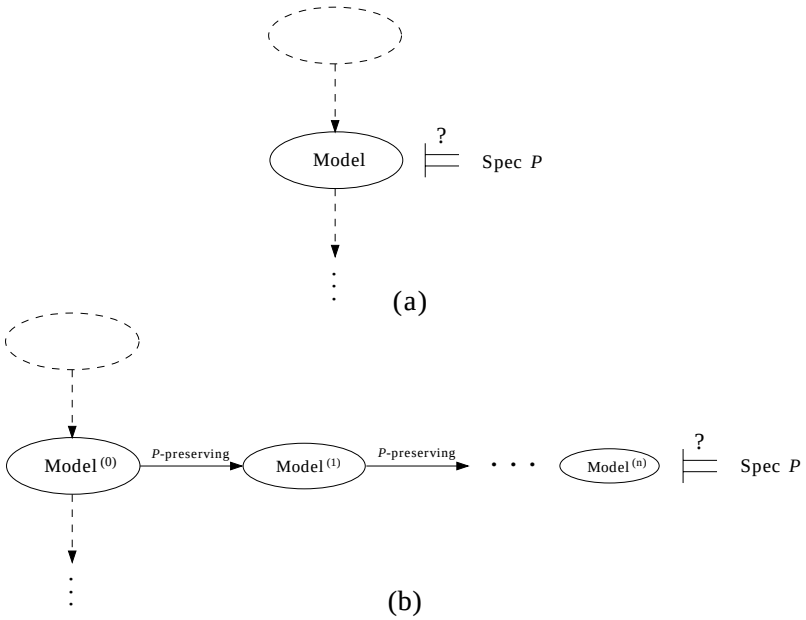

Figure 15: Using transformations in order to reduce verification cost

Figure 15(a) depicts how the system model, at a given phase of the design flow, is verified. The model together with the required properties P are input to the model checking tool to find out whether the model conforms its desired properties. It is possible to do better by trying to apply horizontal transformations in order to get a simpler model, yet semantically equivalent with respect to the properties P. Our transformational approach to verification is illustrated

in Figure 15(b). If the transformations are *P*-preserving, only the simplest model is verified and there is no need to model-check intermediate steps, thus saving time in the verification process.

We may take advantage of transformations to reduce the complexity of verification. The idea is to simplify the system model using transformations from a library. In the case of total-equivalence transformations, since an external observer could not distinguish between two total-equivalent nets (for the same tokens in corresponding in-ports, the observer would get in both cases the very same tokens in corresponding out-ports), the global system properties are preserved in terms of reachability, time, and functionality. Therefore such transformations are *correctness-preserving*: if a property P holds in a net that contains a subnet N', it does in another in which N' has been transformed into a total-equivalent subnet N''; if P does not hold in the first net, it does not in the second either.

If the system model does not have guards, we can ignore transition functions as reachability and time analyses (which are the focus of our verification approach) will not be affected by token values. In such a case, we can use time-equivalence transformations to obtain a simpler model, as they preserve properties related to absence/presence of tokens in the net as well as time stamps of tokens.

5.2 Verification of the GMDFα

In this section we verify the GMDFα (Generalized Multi-Delay frequency-domain Filter) modeled using PRES+ in Section 3.2.1. We illustrate the benefits of using transformations in the verification of the filter.

We consider two cases of a GMDFα of length 1024: a) with an overlapping factor of 4, we have the following parameters: L=1024, α=4, K=4, N=256, and R=64; b) with an overlapping factor of 2, we have the following parameters: L=1024, α=2, K=8, N=128, and R=64. Having a sampling rate of 8 kHz, the maximum execution time for one iteration is in both cases 8 ms (64 new samples must be processed at each iteration). The completion of one iteration is determined by the marking of the place E'.

We want to prove that the system will eventually complete its functionality. According to the time constraint of the system, it is not sufficient to finish the filtering iteration but also to do so with a bound on time (8 ms). This aspect of the specification is captured by the TCTL formula $\mathbf{AF}_{<8}\ E'$. At this point, our task is to verify that the model of the GMDFα shown in Figure 10 satisfies the formula $\mathbf{AF}_{<8}\ E'$.

A straightforward way could be flattening the system model and applying directly the verification technique discussed in Section 4. However, a wiser approach would be trying to first simplify the system model by transforming it into an equivalent one, through transformations from a library. Such transformations are a mathematical tool that allows a significant improvement in the verification efficiency. The improvement is possible because of the following observation: the smaller the model is, the lower the verification cost becomes, in terms of both time and memory. Therefore we try to reduce the model aiming at obtaining a simpler one, still semantically equivalent from the point of view of reachability and time analyses, so that correctness is preserved.

We start by using the transformation rule illustrated in Figure 14 on the refinement of the basic cell, so that we obtain the subnet of Figure 16(b). Note that in this transformation step, no time is spent on-line in proving the transformation itself because transformations are proved off-line (once in a lifetime) and stored in a library. Since the subnets of Figures 16(a) and 16(b) are total-equivalent, the functionality of the entire GMDFα, so far, remains unchanged. We may also use time-equivalence transformations because the PRES+ model of the GMDFα has no guards. Using simple time-equivalence transformations, it is possible to

obtain a simpler representation of the basic cell as shown in Figure 16(c). We continue until the basic cell refinement is further simplified into the single-transition net of Figure 16(d). Finally we check the specification against the simplest model of the system, that is, the one in which the refinement of the basic cells $\boldsymbol{S}_{3.i}$ is the net shown in Figure 16(d). We have verified the formula $\mathbf{AF}_{<8}\ E'$ and the model of the GMDFα indeed satisfies its specification for both K=4 and K=8. The verification times using UPPAAL [Upp] on a Sun Ultra 10 workstation are shown in the last row of Table 1.

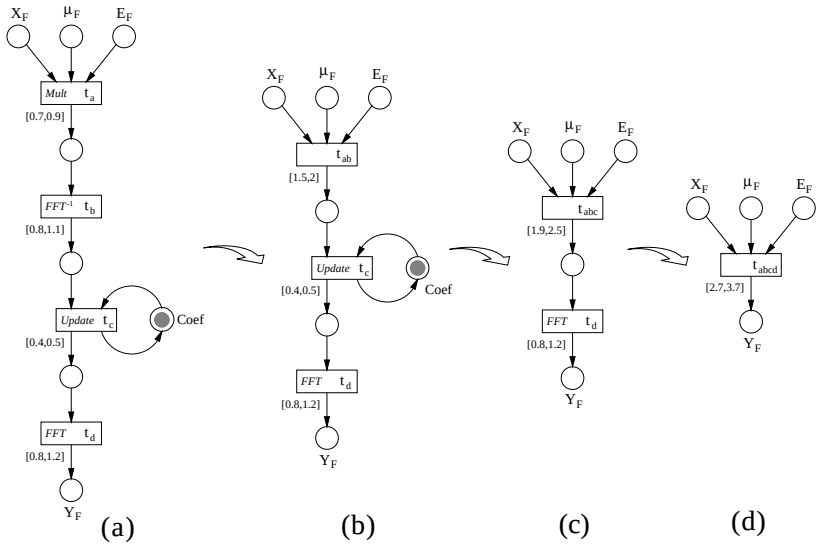

Figure 16: Transformations of the GMDFα basic cell

Table 1: Verification times of the GMDFα

Refinement of the basic cell	**Verification time [s]**	
	α=4, *K*=4	**α=2, *K*=8**
Fig. 16(a)	108	NA*
Fig. 16(b)	61	8177
Fig. 16(c)	9	1368
Fig. 16(d)	1	9

*. Not available: out of time

Since the transformations used along the simplification of the GMDFα model are correctness-preserving, the initial model of Figure 10 is correct, i.e. satisfies the system specification, and therefore need not be verified. However, in order to illustrate the verification cost (time) at different stages, we have verified the intermediate steps (models in which the refinements of the basic cells $\boldsymbol{S}_{3.i}$ are given by the nets shown in Figures 16(b) and 16(c)) as well as the initial model. The results are shown in Table 1. Recall, however, that this is not needed as long as the transformation rules are preserve the correctness in terms of reachability and time analyses. Observe how much effort is saved when the basic cells $\boldsymbol{S}_{3.i}$ are refined by the simplest net compared to the original model.

Thus verification is carried out at low cost (short time) by first using correctness-preserving transformations aiming at simplifying the system representation. If the simpler model is correct (its specification holds), the initial one is guaranteed to be correct and intermediate steps

need not be verified.

6. Reduction of Verification Time by Clustering Transitions

Our approach to verification allows reasoning formally about real-time embedded systems represented in PRES+. We have proposed in Section 4 a systematic procedure to translate PRES+ into timed automata in order to make use of existing model checking tools. Such a procedure can be improved by exploiting the structure of the net and, in particular, extracting the sequential behavior of the system.

In this section we present a clustering algorithm that extracts the sequential behavior of the Petri net. Then we propose a translation procedure where we obtain one automaton for each cluster (sequential part of the net). In this manner we improve significantly the procedure to translate PRES+ models into timed automata presented in Section 4 and consequently the efficiency of the verification process. The example of the GMDFα is revisited in this chapter in order to illustrate the reduction in verification time when the structure of the net is considered.

6.1 Clustering

The approach proposed in Section 4 translates PRES+ models into a collection of timed automata which operate and coordinate with each other through shared variables and synchronization labels. One automaton with one clock variable is obtained for each transition. The main problem of such an approach is that the complexity of model checking of timed automata is exponential in the number of clocks.

In order to reduce the number of automata/clocks resulting from the translation of PRES+ models into timed automata, we propose an algorithm that extracts the sequential behavior of the Petri net by *clustering* transitions. Intuitively, each *cluster* consists of a sequence of transitions where the firing of one of them *enables* the next one. The output of the algorithm is a set of clusters, each representing a sequential part of the net. Then we obtain the timed automata, with one automaton and one clock per cluster (instead of one automaton and one clock per transition of the PRES+ model).

Definition 6.1. A *cluster* is an ordered tuple of distinct transitions denoted $C=(t_1, \ldots, t_n)$, such that t_{i+1} becomes enabled iff t_i fires, for $1 \le i < n$. We say that t_1 and t_n are, respectively, the *head* and the *tail* of C. ▪

In Figure 17, a possible cluster is $C=(t_1, t_3, t_5)$ with head t_1 and tail t_5.

Definition 6.2. The *cluster set* S_C of a cluster $C=(t_1, \ldots, t_n)$ is the set of transitions that are components of C, that is $S_C=\{t_1, \ldots, t_n\}$. ▪

We explicitly make a distinction between *cluster* and *cluster set* because in the former case the order of the components is relevant whereas the order of elements in a set is immaterial. The objective of our clustering algorithm is to find a set of clusters such that their cluster sets form a partition of T (the set of transitions of the Petri net). In other words, we aim at finding a number of clusters such that each transition $t \in T$ is in one and only one cluster.

Definition 6.3. The *anterior set* of a transition $t \in T$, denoted $ant(t)$, is the set of those transitions that when fired will deposit a token in some place in the pre-set $^\circ t$, that is, $ant(t)= \bigcup_{p_i \in {}^\circ t} {}^\circ p_i$. The *posterior set* of a transition $t \in T$, denoted $post(t)$, is the set of transitions that will get a token in some place of their pre-set when t is fired, that is, $post(t) = \bigcup_{p_i \in t^\circ} p_i^\circ$. ▪

Definition 6.4. The *anterior set* $ant(C)$ of a cluster $C=(t_1, \ldots, t_n)$ is the anterior set of its head

t_1, that is, $ant(C)=ant(t_1)$. The *posterior set* $post(C)$ of a cluster $C=(t_1, \ldots, t_n)$ is the posterior set of its tail t_n, that is, $post(C)=post(t_n)$. ■

Consider, for example, the cluster $C=(t_{10}, t_1, t_3)$ in the net shown in Figure 17. Its anterior and posterior sets are, respectively, $ant(C)=\{t_9\}$ and $post(C)=\{t_5, t_6\}$.

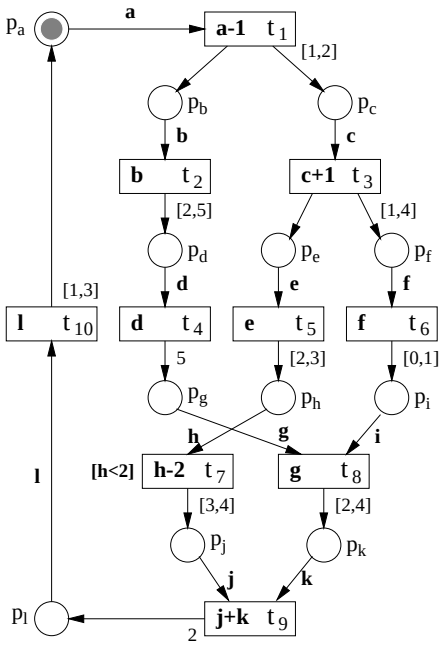

Figure 17: PRES+ model to be clustered

The clustering algorithm we propose tries to add a new head or tail to an existing cluster C. We keep a list of "free" transitions *freeT*, i.e. transitions not allocated yet to any cluster. Let $C=(t_h, \ldots, t_t)$ be a cluster with head t_h and tail t_t and let *freeT* be the set of free transitions. We may add a *new tail* t_{nt} to the cluster C if $ant(t_{nt})-\{t_{nt}\}=\{t_t\}$ and $t_{nt} \in freeT$. We may add a *new head* t_{nh} to C if $t_{nh} \in freeT$ and $ant(C)-\{t_h\}=\{t_{nh}\}$. Consider the example given in Figure 17. Assume this time $C=(t_9, t_{10}, t_1)$ and $freeT=T-S_C=\{t_2, t_3, t_4, t_5, t_6, t_7, t_8\}$. Since $t_2, t_3 \in freeT$ and also $ant(t_2)=ant(t_3)=\{t_1\}$, both t_2 and t_3 fulfill the requirements for new tail stated above, but only one of them can be added as new tail to the cluster. In our algorithm this choice is made arbitrarily. If, for instance, t_3 is added to the cluster we obtain $C=(t_9, t_{10}, t_1, t_3)$ and $freeT=\{t_2, t_4, t_5, t_6, t_7, t_8\}$. Note that t_3 was removed from *freeT*. It is not hard to see that there is no transition to be added as new head of the cluster.

Our clustering algorithm starts by selecting arbitrarily a transition t from the free list. A a new cluster C is formed so that t is initially both head and tail of C, and t is removed from *freeT*. The next step is to examine only those transitions in $post(C)$ that are also in *freeT* and check whether they may be a new tail of C. If so, the cluster is enhanced by adding a new tail. We repeat the process until no new tail may be added to the cluster. Then, in a similar fashion, we try to enhance the cluster by adding a new head and repeat until there is no new head candidate in the free list. When the cluster can no longer be enhanced, we select another transition from *freeT*, form a new cluster, and repeat the process until all transitions have been allocated to a cluster. The clustering algorithm is shown in Figure 18.

By applying our clustering algorithm on the system shown in Figure 17, we obtain the following clusters: $C_1=(t_9, t_{10}, t_1, t_2, t_4)$, $C_2=(t_3, t_5, t_7)$, $C_3=(t_6)$, $C_4=(t_8)$. Note that the output of the algorithm is not unique since there might be new-tail transitions chosen arbitrarily. We could also have got, for instance, $C_1'=(t_9, t_{10}, t_1, t_3, t_6)$, $C_2'=(t_2, t_4)$, $C_3'=(t_5, t_7)$, $C_4'=(t_8)$. How-

ever, in either case, the number of clusters is the same. Recall that we will obtain one automaton and one clock per cluster.

```
clustering(safePN N)
    set freeT := T
    while freeT ≠ ∅ do
        with an arbitrary t ∈ freeT do
            new cluster C=(t)
            set freeT := freeT-{t}
            set newhead := true
            set newtail := true
            // try to add a new tail t_nt
            while newtail do
                set newtail := false
                with an arbitrary t_nt ∈ post(C) ∩ freeT
                 such that ant(t_nt)-{t_nt}={t_t} do
                    add t_nt to C
                    set freeT := freeT-{t_nt}
                    set newtail := true
                endwith
            endwhile
            // try to add a new head t_nh
            while newhead do
                set newhead := false
                with t_nh ∈ ant(C) ∩ freeT such
                 that ant(C)-{t_h}={t_nh} do
                    add t_nh to C
                    set freeT := freeT-{t_nh}
                    set newhead := true
                endwith
            endwhile
        endwith
    endwhile
endclustering
```

Figure 18: Clustering algorithm

A simple analysis shows that the proposed algorithm has a (worst-case) time complexity $O(n^2)$, where n is the number of transitions in the net. We have applied the clustering algorithm to three different examples that can be scaled up. It is not our intention to discuss them here but rather use these examples in order to illustrate the performance of the algorithm in terms of execution time. Figure 19 shows the execution times of the clustering algorithm, running on a Sun Ultra 10 workstation, for the three cases studied.

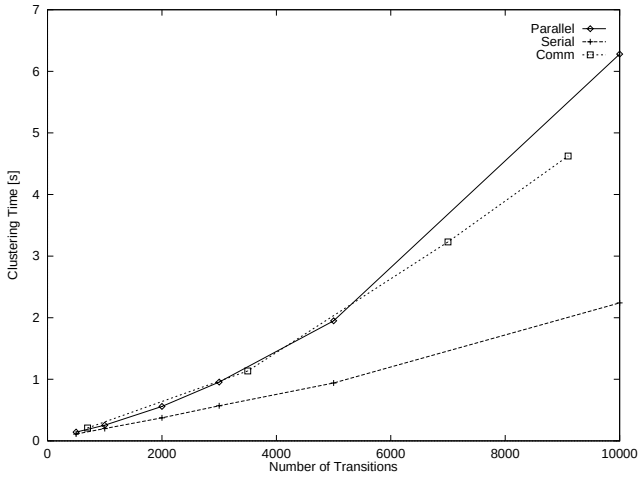

Figure 19: Performance of the clustering algorithm

6.2 Improved Translation Procedure

As discussed previously, in order to verify the correctness of a real-time embedded system represented in PRES+, we translate the system model into timed automata so that model checking tools can be used. In what follows we describe the systematic procedure to translate PRES+ models into timed automata after clustering has been performed. The resulting model will consist of one automaton and one clock per cluster. We use the example of Figure 17 in order to illustrate the translation procedure.

Step 6.1. Define one clock in X for each cluster. Define one variable in V for each place p_x of the Petri net, corresponding to the token value v_x when p_x is marked. ▪

Step 6.2. Define the set Σ of synchronization labels as the set of transitions in the Petri net. ▪

Steps 6.3 through 6.9 must be performed for each one of the clusters obtained by using the clustering algorithm. Consider a cluster $C=(t_1, \ldots, t_n)$ with head t_1 and tail t_n. For $t_i \in S_C$ (t_i denotes the i-th transition in cluster C), let f_i be the transition function associated to t_i, and let d_i^- and d_i^+ be the minimum and maximum transition delays associated to t_i. Let G_i be the guard associated to the transition t_i. Let v_x be the value of the token in the place p_x when marked. The timed automaton corresponding to the cluster C will be denoted $\overrightarrow{C}$. The clock corresponding to $\overrightarrow{C}$ is denoted c. For the sake of clarity, we first present the translation steps for the simplest case: we initially assume that $(post(C)\text{-}\{t_n\}) \cap S_C = \varnothing$ and that t_i is not in conflict for all $t_i \in S_C$. Recall that a transition is in conflict if it can be disabled by the firing of a different transition. Later we will discuss the general case where these assumptions do not hold.

Step 6.3. Define $m+n$ locations $a_1, \ldots, a_m, b_1, \ldots, b_n$, where $m=|ant(C)\text{-}\{t_1\}|$ and $n=|S_C|$. These are the locations of $\overrightarrow{C}$. Define m edges (a_j, a_{j+1}), for $j=1, \ldots, m\text{-}1$, with synchronization labels corresponding to the transitions in $ant(C)\text{-}\{t_1\}$. Define also m edges (a_m, b_1) with synchronization labels corresponding to the transitions in $ant(C)\text{-}\{t_1\}$. Then define one edge (b_i, b_{i+1}), for $i=1, \ldots, n\text{-}1$, with synchronization label t_i. Define one edge (b_n, a_1) with synchronization label t_n. ▪

Consider the cluster $C_1=(t_9, t_{10}, t_1, t_2, t_4)$ for the model given in Figure 17. We have $n=5$ for this cluster. Since $ant(C_1)=\{t_7, t_8\}$ we have $m=2$. Therefore, the automaton $\overrightarrow{C_1}$ corresponding to the cluster C_1 has 7 locations $a_1, a_2, b_9, b_{10}, b_1, b_2, b_4$ and its edges are as shown in Figure 20. Note that b_k corresponds to the location in which transition t_k is bound (or enabled if t_k has no guard). The change of location, for example, from b_1 to b_2 corresponds to the firing of transition t_1.

Step 6.4. For every edge $e_j=(a_m, b_1)$ and every edge $e_i=(b_i, b_{i+1})$, $1 \le i < n$, define $R(e_j)=R(e_i)=\{c\}$. For any other edge e in $\overrightarrow{C}$, define $R(e)=\varnothing$. ▪

This means that on all edges but (a_j, a_{j+1}), $1 \le j < m$, and (b_n, a_1) the clock c will be reset. In Figure 20, the assignment $c_k := 0$ represents the reset of clock c_k.

Step 6.5. For every location b_i, $1 \le i \le n$, define its location invariant as $c \le d_i^+$. ▪

This enforces the firing of t_i before or at its latest trigger time.

Step 6.6. To every edge with synchronization label t_i, where $t_i \in S_C$, assign the clock condition $d_i^- \le c \le d_i^+$. ▪

In Figure 20, for example, the edge (b_2, b_4) (with synchronization label t_2) of the automaton $\overrightarrow{C_1}$ has a clock condition $2 \le c_1 \le 5$ where 2 and 5 are the minimum and maximum transition delays of t_2.

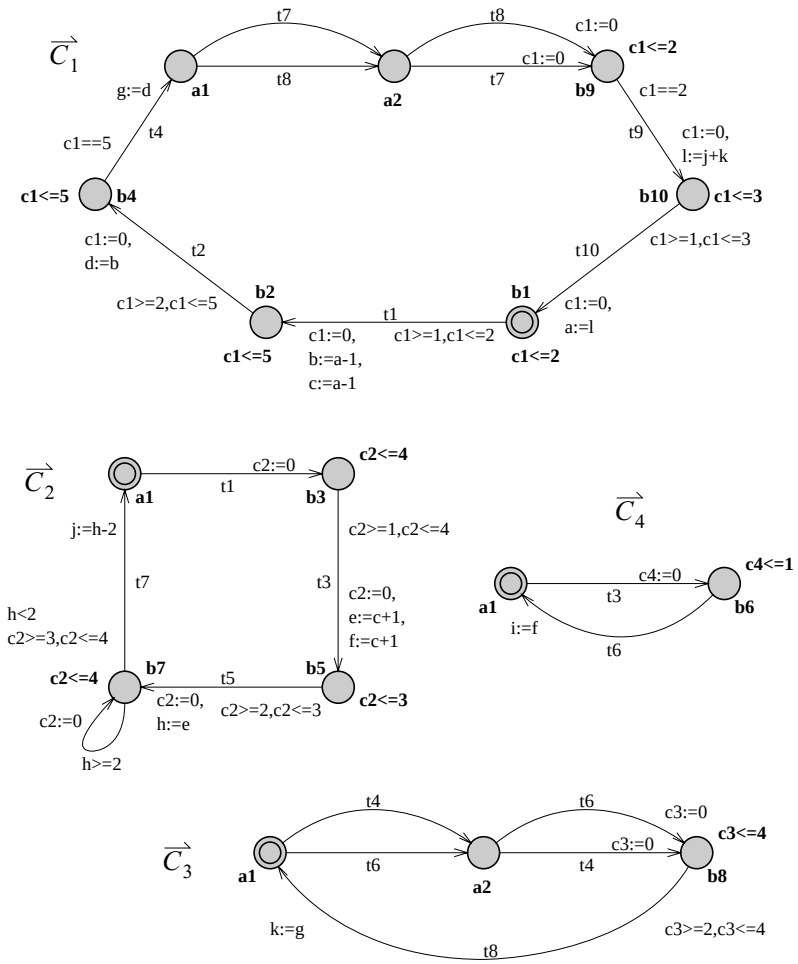

Figure 20: Automata equivalent to the model of Figure 17

Step 6.7. For every edge with synchronization label t_i, where $t_i \in S_C$, and for every $p_j \in t_i^\circ$ assign to such an edge the activities $v_j := f_i$. ▪

For instance, the activities assigned to the edge (b_1, b_2) with synchronization label t_1 in the automaton $\overrightarrow{C_1}$ are b := a-1 and c := a-1, where a-1 is the transition function of t_1.

Step 6.8. If the transition $t_i \in S_C$ has a guard G_i, assign the variable condition G_i to the edge with synchronization label t_i. Then add an edge $e=(b_i, b_i)$ with no synchronization label, variable condition $\overline{G_i}$ (the complement of G_i), and $R(e)=\{c\}$. ▪

Note the variable condition $h < 2$ on (b_7, a_1) and the edge (b_7, b_7) in the automaton $\overrightarrow{C_2}$. This is due to the guard $h < 2$ of transition t_7.

Step 6.9. If the transition $t_i \in S_C$ is enabled in the initial marking, make the location b_i the initial location of $\overrightarrow{C}$. Otherwise, if there are k places initially marked in the pre-set $^\circ t_1$ of the head t_1 ($0 \le k < m$ so that t_1 is not enabled), make a_{k+1} the initial location of $\overrightarrow{C}$. ▪

In our example, b_1 is the initial location of $\overrightarrow{C_1}$ because the transition $t_1 \in S_{C_1}$ is enabled in the initial marking of the net. The automaton $\overrightarrow{C_2}$ has a_1 as initial location because none of the transitions of the cluster $\overrightarrow{C_2}$ is initially enabled.

Observe that one and only one of the transitions of a given cluster will be enabled at a time. If two transitions in a cluster were enabled simultaneously, that would imply that the (underlying untimed) Petri net is not safe. The translation procedure we propose here is correct as

long as the untimed Petri net is safe.

We have assumed, so far, that t_i is not in conflict, for all $t_i \in S_C$, and $(post(C)\text{-}\{t_n\}) \cap S_C = \varnothing$. Now we discuss the cases in which these assumptions do not hold:

a) In case that $post(C)\text{-}\{t_n\} = \{t_1\}$ (the posterior set of the cluster tail is the singleton containing the cluster head) the automaton $\overrightarrow{C}$ will have n locations $b_1, \ldots, b_n$, where $n=|S_C|$, but no a_i locations. There will be additionally one edge (b_n, b_1) with synchronization label t_n and clock condition, variable condition, clock reset, and activities similar to the other edges (b_i, b_{i+1});

b) If one of the transitions $t_i \in S_C$ is in conflict with another transition t_c, just add to the automaton $\overrightarrow{C}$ one edge (b_i, a_1) with synchronization label t_c.

6.3 Revisiting the GMDFα

In Section 3.2.1 we have modeled a Generalized Multi-Delay frequency-domain Filter (GMDFα). In Section 5.2 such an application has been verified by transforming the system model and using the "naive" translation procedure described in Section 4.3.

In this section we revisit the verification of the GMDFα and compare it with the results shown previously in Section 5.2. We also consider here the two cases of a GMDFα of length 1024: a) with an overlapping factor $\alpha=4$, $K=4$; b) with an overlapping factor $\alpha=2$, $K=8$. Recall that having a sampling rate of 8 kHz, the maximum execution time for one iteration is in both cases 8 ms. What we want to prove is that the filter eventually completes its functionality and it does so within a bound on time (8 time units). This is captured by the TCTL formula $\mathbf{AF}_{<8}\ E'$. As seen in Figure 10, K affects directly the dimension of the model and, therefore, the complexity of verification.

Table 2: Verification of the GMDFα

GMDFα *L*=1024	**Verification time [s]**			
	Naive	**Transformations**	**Clustering**	**Transf. + Clustering**
α=4, *K*=4	108	1	2	<1
α=2, *K*=8	NA*	9	540	1

*. Not available: out of time

We have used UPPAAL, running on a Sun Ultra 10 workstation, in order to model-check the formula $\mathbf{AF}_{<8}\ E'$ against the model of the filter. For both cases ($K=4$ and $K=8$), $\mathbf{AF}_{<8}\ E'$ indeed holds (this fact was known beforehand from Section 5.2). The results are shown in Table 2. The second column corresponds to the verification time using the approach described in Section 4 (naive translation of PRES+ into timed automata). The third column in Table 2 shows the results of verification when using the approach discussed in Section 5 (transformation of the model into a semantically equivalent and simpler one in order to reduce complexity, followed by naive translation into timed automata). The verification time for the GMDFα using the clustering method presented in this section is shown in the fourth column of Table 2. These results include the execution time of the clustering algorithm. By combining the transformational approach with the clustering one, it is possible to further improve the efficiency of the verification process as shown in the last column of Table 2.

7. Experimental Results

In this section we illustrate the verification approach on a scalable example, comparing the technique based on a naive translation from PRES+ into automata discussed in Section 4, the transformational approach presented in Section 5, and the one formulated in Section 6 where the structure of the net is exploited to achieve higher efficiency.

The example that we use represents a number *n* of processes arranged in a ring configuration. The model for one such process is illustrated in Figure 21. Each one of the *n* processes in the system has a bounded response requirement, namely whenever the process starts it must strictly finish within a time limit, in this case 25 time units. Referring to Figure 21, the start of one such process is denoted by the marking of p_{start} while the marking of p_{end} denotes the end of the process. This requirement is expressed by the TCTL formula $\mathbf{AG}(p_{start} \Rightarrow \mathbf{AF}_{<25}\ p_{end})$.

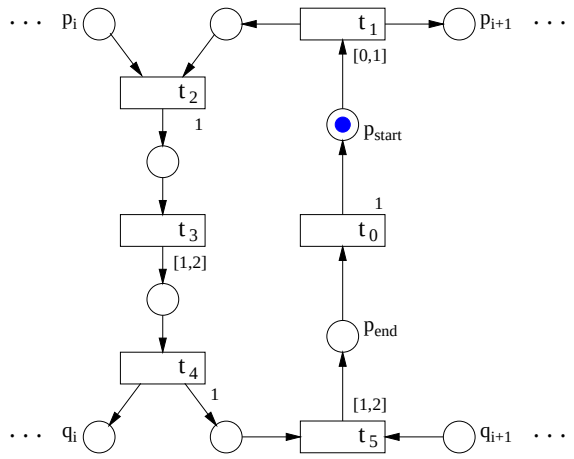

Figure 21: PRES+ model for one ring-process

We have used UPPAAL in order to model-check the timing requirements of the processes in the ring-configuration example. The results are summarized in Table 3.

Table 3: Verification of the ring-configuration example

Number of Processes (*n*)	**Verification time [s]**			
	Naive	**Transfor-mations**	**Clustering**	**Transf. + Clustering**
2	1	<1	<1	<1
3	29	5	2	1
4	704	85	31	17
5	8700	1275	453	205
6	NA*	13260	5771	2295
7†	NA*	NA*	NA*	16200

*. Not available: out of time
†. Specification does not hold

The second column of Table 3 shows the verification time using the naive translation procedure of Section 4. The third column corresponds to the transformational approach discussed in Section 5. The fourth column of Table 3 shows the verification time of the method based on transition clustering (Section 6). The results of combining the transformation-based technique

with clustering are shown in the last column. We have plotted all these experimental results in Figure 22.

Observe that for n=7 the bounded response requirement expressed by the formula $\mathbf{AG}(p_{start} \Rightarrow \mathbf{AF}_{<25}\ p_{end})$ is not satisfied, a fact which, at first glance, is not obvious at all. An informal explanation is that since transition delays are given in terms of intervals, one process may take longer to execute than another; thus different processes can execute "out of phase" and this phase difference may be accumulated depending on the number of processes, causing one such process to take eventually longer than 25 time units (for n=7). It is also worth mentioning that, although the model has relatively few transitions and places, this example is very complex because of its large (untimed) state space which is due to the high degree of parallelism.

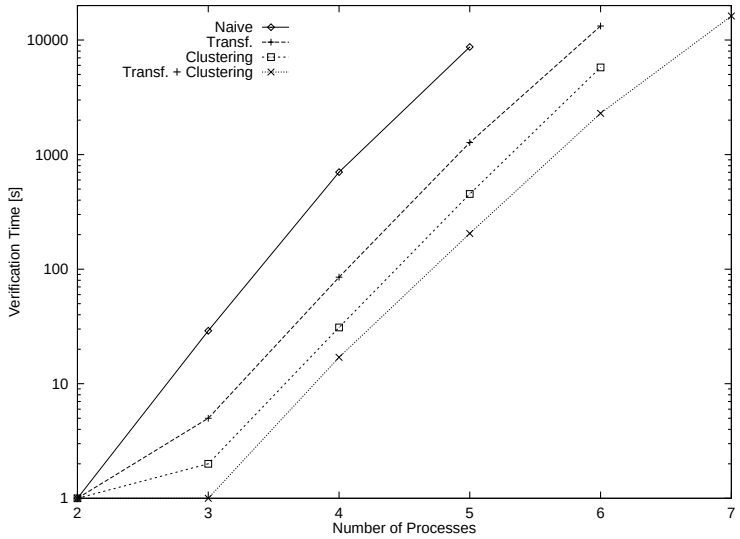

Figure 22: Verification of ring-configuration processes

8. Radar Jammer: The Industrial Case

The example that we describe in this section corresponds to a real-life application used in the military industry [Lin01]. The function of such a system is to deceive a radar apparatus by jamming signals.

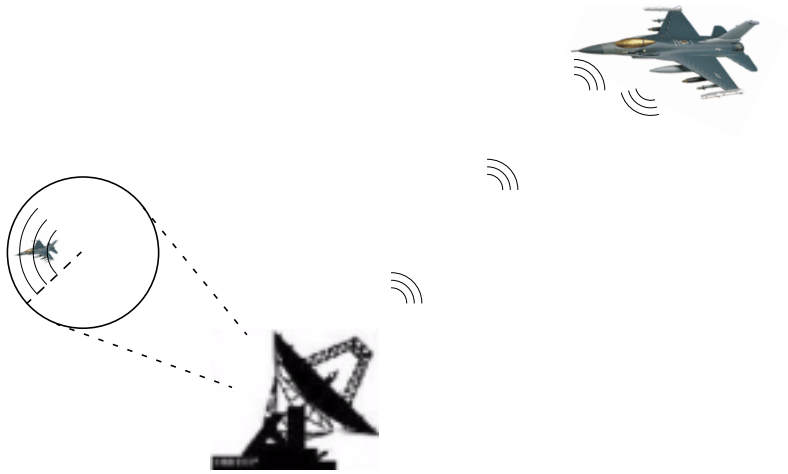

Figure 23: Radar jammer and its environment

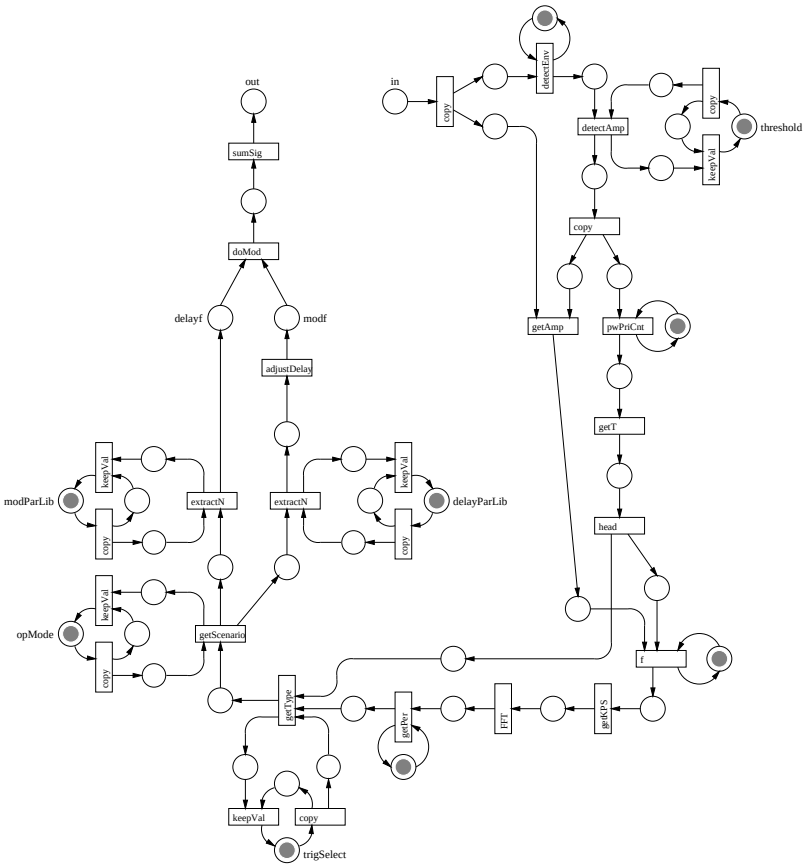

Figure 24: PRES+ model of a jammer

The jammer is a subsystem placed on an object (target), typically an aircraft, moving in the area observed by a radar. The radar sends out pulses and some of them are reflected back to the radar by the objects in the area. When a radar receives pulses, it might determine the distance and direction of the object, and even its velocity and the type of object. The distance is calculated by measuring the time the pulse has travelled from its emission until it returns to the radar. By rotating the radar antenna lobe, it is possible to find the direction returning maximum energy, that is, the direction of the object. The velocity of the object is found out based on the doppler shift of the returning pulse. The type of object can be determined by comparing the shape of the returning pulse with a library of radar signatures for different objects.

The basic function of the jammer is to deceive a radar scanning the area in which the object is moving. The jammer receives a radar pulse, modifies it, and then sends it back to the radar after a certain delay. Based on input parameters, the jammer can create pulses that contain specific doppler and signature information as well as the desired space and time data. Thus the radar will see a false target. A view of the jammer and its environment is shown in Figure 23.

The jammer example has been used as a test case for the SAVE design methodology. The system is described using Haskell as specification language. The Haskell description is based on *skeletons*, which are higher-order functions used to model elementary processes.

The radar jammer has been specified in Haskell using a number of skeletons. Based on a basic procedure to translate Haskell descriptions (using skeletons) into PRES+ [Cor01] and assisted by a software tool developed by our research group, we may get the PRES+ model of the jammer from its Haskell description. The obtained model contains no timing information which can later be annotated as transition delays. The PRES+ model of the radar jammer, obtained from its Haskell description, is shown in Figure 24.

We briefly discuss the structure of the PRES+ model of the jammer. We do not intend to pro-

vide here a detailed description of each one of the transitions of the model of the radar jammer given in Figure 24 but rather present an intuitive idea about it. When a pulse arrives, it is initially detected and some of its characteristics are calculated by processing the samples taken from the pulse. Such processing is performed by the initial transitions, e.g. *detectEnv*, *detectAmp*, ..., *getPer*, *getType*, and based on internal parameters like *threshold* and *trigSelect*. Different scenarios are handled by the middle transitions, e.g. *getScenario*, *extractN*, and *adjustDelay*. The final transitions *doMod* and *sumSig* are the ones that actually alter the pulse to be returned to the radar.

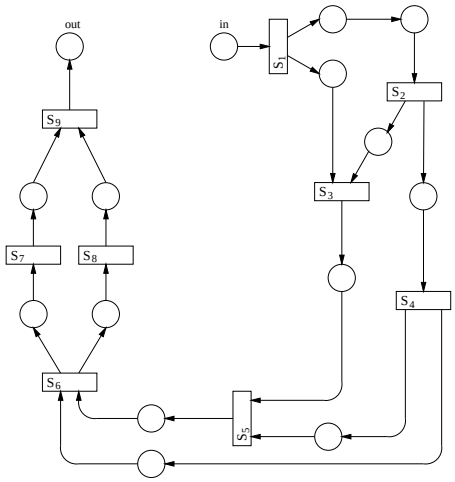

Figure 25: Higher-level abstraction of the radar jammer

Using the concept of hierarchy, it is possible to obtain a higher-level view of the radar jammer represented in PRES+ as depicted in Figure 25. The super-transitions abstract parts of the model given in Figure 24. For example, super-transition S_5 corresponds to the abstraction of the subnet shown in Figure 26. Such a subnet (Figure 26) can easily be identified as a portion of the model depicted in Figure 24.

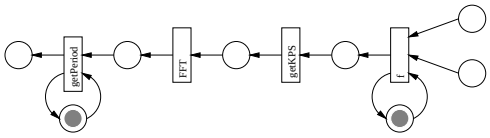

Figure 26: Refinement of S_5 in the model of Figure 25

Also, many of the transitions presented in the model of Figure 24 could be refined (for example, during the design process). To illustrate this, we show how transition *doMod*, for instance, can be refined according to our concept of hierarchy. Its refinement is presented in Figure 27. In this form, hierarchy can conveniently be used to structure the design in a comprehensible manner.

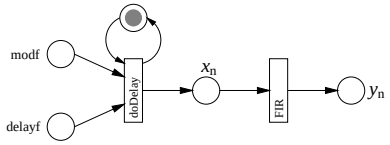

Figure 27: Refinement corresponding to transition *doMod* in the model of Figure 24

We aim at verifying a pipe-lined version of the jammer where the stages correspond precisely to the super-transitions of the model shown in Figure 25. In order to represent a pipe-lined structure it is necessary to add a number of places and arcs to the model as follows. For every place $p \in P$ such that $(t_a, p) \in O$, $(p, t_b) \in I$, and $t_a \neq t_b$: a) add a place p' initially marked; b) add an input arc (p', t_a); c) add an output arc (t_b, p'). In this way, all places but *in* and *out* will hold at most one token, and still several of them might be marked simultaneously, representing the progress of activities along the pipeline.

The model of the pipe-lined jammer is shown in Figure 28. The minimum and maximum transition delays are given in ns. The timing information is discussed later in this section. We have included in this model a few more places and transitions that represent the environment. The input to the jammer is a radar pulse (actually, a number of samples taken from it). Transition *sample* will fire n times (where n is the number of samples), every PW/n (where PW is the pulse width), depositing the samples in the place *inSig* which are later buffered in the place *in*. In this form, we model the input of the incoming radar pulse. A token in *inSig* means that the input is being sampled.

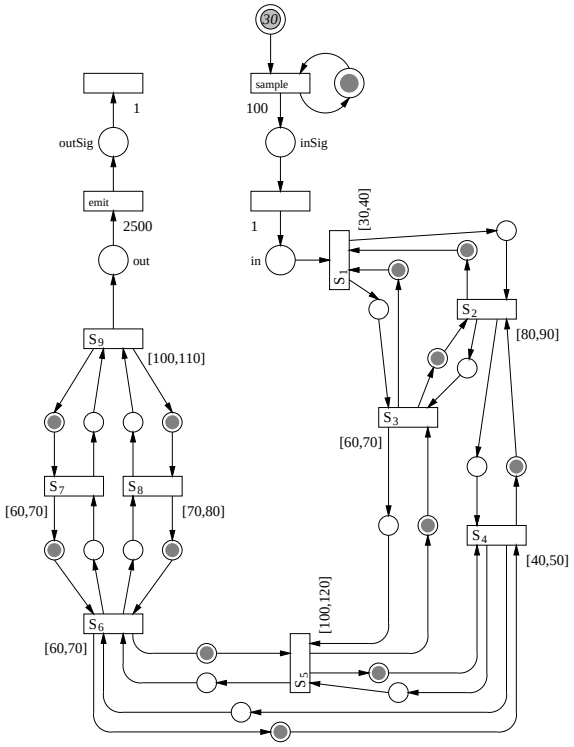

Figure 28: Pipe-lined model of the jammer

Regarding the emission of the pulse produced by the jammer, the data obtained is buffered in place *out* before being transmitted. After some delay, it is sent out by transition *emit* so that the marking of place *outSig* represents a part of the outgoing pulse being transmitted back to the radar.

We have applied our verification technique to the PRES+ model of the jammer shown in Figure 28. We have performed what we call “time budget verification”. At this point, we have no accurate estimates of the execution time of the function associated to each one of the transitions of the model. However, we do know the constraints of the system. The idea is to assign values to the minimum and maximum transition delays based on the designer’s experience. Having such values, we perform verification of the required properties. If such properties are satisfied, the transition intervals constitute the time budget for the different functions to be

implemented.

The time budget information can be used by the designer to guide the design process. It is possible that some of the intended implementations of a certain function do not fit in the time budget obtained previously. Then, it is necessary to modify the timing information of the model based on more accurate data and verify again the desired properties. Thus the process is repeated so that the designer gets valuable information from the very early stages of the design flow.

There are two properties that are important for the jammer. The first is that there cannot be output while sampling the input. The second requirement is that the whole outgoing pulse must be transmitted before another pulse arrives. The minimum Pulse Repetition Interval (PRI), i.e. the separation in time of two consecutive incoming pulses, for our system is 10 μs, so this is the value we will use for verifying the second property. For a PRI of 10 μs, the Pulse Width (PW) can vary from 100 ns up to 3 μs. Therefore, we will consider the most critical case, that is, when the pulse width is 3 μs. We assume that the number of samples is n=30 (so that the delay of transition *sample* in Figure 28 is 100 ns).

The properties described above can be expressed, respectively, by the formulas $\mathbf{AG}\ \neg(inSig \wedge outSig)$ and $\neg\mathbf{EF}_{>10000}\ outSig$. The first formula states that the places *inSig* and *outSig* are never marked at the same time, while the second says that there is no computation path for which *outSig* is marked after 10000 ns.

In order to verify the model of the jammer shown in Figure 28, we have translated it into timed automata. We have used the systematic translation procedure for the part of the net that is safe. The rest of the model (for example, transitions *sample* and *emit*) has been translated in an *ad hoc* manner. We have verified that the required properties are indeed satisfied in the model of Figure 28. Using UPPAAL, the verification of $\mathbf{AG}\ \neg(inSig \wedge outSig)$ takes 115 s while the verification of the formula $\neg\mathbf{EF}_{>10000}\ outSig$ takes 384 s.

The radar jammer is a realistic example that has illustrated how our modeling and verification approach can be used for practical applications. The concept of hierarchy has proved to be very convenient to handle this example in an understandable way. The verified requirements are very interesting as not only they impose an upper bound for the completion of the activities but also a lower one, since the emission and sampling of pulses cannot overlap. Though there are few transitions in the model, the state space is very large because of the pipeline. Despite the large space, the verification of the two studied properties takes relatively short time.

9. Conclusions

This document has presented an approach to modeling and formal verification of embedded systems. The research addressed in this report has been performed within the frame of the SAVE Project.

The design flow must be based upon an unambiguous formalism that can represent relevant characteristics of the system and capture design decisions. A sound model of computation supports a precise representation of the system, the use of formal methods to verify its correctness, and the automation of different tasks along the design process.

We have presented a formal model of computation for embedded systems. PRES+ is a model based on Petri nets with a well-defined semantics. It has been extended in order to capture essential characteristics of real-time embedded systems: tokens carry information and transitions perform transformation of data when fired; timing is explicitly included by associating lower and upper limits to the duration of activities related to transitions; both sequential and concurrent activities may be easily expressed; PRES+ supports the concept of hierarchy.

A military industry application has been studied in order to demonstrate the applicability of

our modeling technique to different systems.

We have proposed an approach to the formal verification of systems represented in PRES+. We make use of model checking to prove whether certain properties, expressed as CTL and TCTL formulas, hold with respect to the system model. We have introduced a systematic procedure to translate PRES+ models into timed automata so that it is possible to use existing model checking tools.

Two strategies have been addressed in order to reduce the complexity of the verification process. First, we apply transformations to the initial system model, aiming at simplifying it, still preserving the properties under consideration. This is a transformational approach that tries to reduce the model, and therefore improve the efficiency of verification, by using correctness-preserving transformations. Thus if the simpler model is correct, the initial one is guaranteed to be correct.

Second, we have shown that verification complexity can further be reduced by improving the translation procedure from PRES+ into automata. We proposed an algorithm that extracts the sequential behavior of the net by clustering transitions. Thus we obtain one automaton with one clock per cluster, instead of one automaton with one clock per transition. Moreover, experimental results have shown that by combining the clustering strategy and the transformational approach the efficiency of verification is improved considerably.

10. References

[Alu90] 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.

[Alu99] R. Alur, "Timed Automata," in *Computer-Aided Verification*, D. Peled and N. Halbwachs, Eds. *LNCS 1633*, Berlin: Springer-Verlag, 1999, pp. 8-22.

[Cam96] R. Camposano and J. Wilberg, "Embedded System Design," in *Design Automation for Embedded Systems*, vol. 1, pp. 5-50, Jan. 1996.

[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.

[Cor01] L. A. Cortés, P. Eles, and Z. Peng, "From Haskell to PRES+: Basic Translation Procedures," SAVE Project Report, Dept. of Computer and Information Science, Linköping University, Linköping, April 2001.

[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.

[Fre97] L. Freund, M. Israel, F. Rousseau, J. M. Bergé, M. Auguin, C. Belleudy, and G. Gogniat, "A Codesign Experiment in Acoustic Echo Cancellation: GMDFα," in *ACM Trans. on Design Automation of Electronic Systems*, vol. 4, pp. 365-383, Oct. 1997.

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

[Kro] KRONOS, `http://www-verimag.imag.fr/TEMPORISE/kronos/`

[Lin01] P. Lind and S. Kvist, "Jammer Model Description," Technical Report, Saab Bofors Dynamics AB, April 2001.

[SAV] SAVE Project, `http://www.ida.liu.se/~eslab/SAVE.html`

[Sgr00] M. Sgroi, L. Lavagno, and A. Sangiovanni-Vincentelli, “Formal Models for Embedded System Design,” in *IEEE Design & Test of Computers*, vol. 17, pp. 14-27, April-June 2000.

[Upp] UPPAAL, `http://www.uppaal.com/`