FORMAL SYNTHESIS AND CONTROL OF SOFT EMBEDDED REAL-TIME SYSTEMS

Pao-Ann Hsiung
Department of Computer Science and Information Engineering
National Chung Cheng University, Chiayi–621, Taiwan, ROC
hpa@computer.org

Abstract Due to rapidly increasing system complexity, ever-shortening time-to-market, and growing demands for soft real-time, formal methods are becoming indispensable in the synthesis of embedded real-time systems. In this work, a formal method based on *Time Free-Choice Petri Nets* (TFCPN) is proposed for synthesizing and controlling *Soft Embedded Real-Time Systems* (SERTS). Technically, the proposed method employs quasi-static data scheduling for satisfying limited embedded memory requirements and controls firing interval bounds for satisfying soft real-time constraints. An application example is given to illustrate the feasibility of the formal method, which can also be used for code generation.

Keywords: embedded real-time systems, formal synthesis, control, quasi-static scheduling, Time Free-Choice Petri nets

1. INTRODUCTION

With the proliferation of *Soft Embedded Real-Time Systems* (SERTS) into our daily lives in the form of home appliances, internet appliances, personal assistants, wearable computers, telecommunication gadgets, and transportation facilities, we are now faced with a growing escalation of system complexity, ever-shortening time-to-market, and growing demands for soft real-time applications. All these factors have propelled the need for practical formal methods that can be used to synthesize and control SERTS. A large portion of formal methods is mainly devoted to hard real-time system analysis. In contrast, we show how formal methods can also be applied to *soft* real-time systems that have flexible ranges of acceptable behaviors.

Our target systems are soft real-time systems such as multimedia servers, communication networks, telecommunication devices, home electric appliances, and information appliances. These systems can tolerate deadline misses up to a certain threshold value. Not every deadline needs to be met, only most of the deadlines need be met. Further, deadlines themselves can be specified as

a time interval (α, β) such that if a system task completes execution no earlier than α and no later than β, then the task does not miss its deadline, where α and β are integers representing some points in the time-line.

Informally, our target problem is to synthesize an embedded real-time system starting from an initial set of loose specifications into a final set of strict specifications such that the final specification satisfies all user-given real-time constraints such as response times, deadline, and periods. A specification ψ_1 is said to be looser than another specification ψ_2 if all the behaviors given by ψ_2 are implied by ψ_1. In plain terms, our solution is to restrict loose specifications into stricter ones such that given constraints are met.

The two main issues involved in the design of SERTS are as follows:

- *Bounded Memory Execution*: A processor cannot have infinite amount of memory space for the execution of any software process. This fact is even more emphasized in an embedded system, which generally has only a few hundreds of kilobytes memory installed. Thus, a SERTS must utilize as less memory as possible.
- *Soft Real-Time Constraints*: A processor may have to execute several concurrent tasks with precedence and temporal constraints. Thus, a SERTS is generally composed of several soft concurrent real-time tasks.

Our formal model is based on the recently proposed *Time Free-Choice Petri Nets* (TFCPN) [15], which is a sub-class of Time Petri Nets. In solution to the above two issues, our proposed method consists of the following two phases:

- *Quasi-Static Data Scheduling* (QSDS): This scheduling phase ensures that embedded real-time applications do not require an unbounded amount of memory for execution, since embedded real-time systems have limited amount of embedded memory,
- *Firing-Interval Bound Synthesis* (FIBS): This synthesis phase ensures that an embedded real-time system meets all soft real-time constraints, which are generally modeled as action or firing time intervals. This phase is also called *Controller Synthesis* since controllers can be synthesized for soft real-time systems using this method.

Software code can also be generated for soft real-time systems by applying our synthesis method to a recently proposed code generation scheme, which was for hard real-time systems [15]. Due to page-limits, we will not delve into this part of the work in this article.

This article is organized as follows. Section 2 gives some previous work related to SERTS synthesis. Section 3 will formulate, model, and solve the SERTS synthesis problem. Section 4 will illustrate the proposed method through an application example. Section 5 will conclude the article with some research directions for future work.

2. PREVIOUS WORK

Currently, *synthesis of soft real-time systems* is a hot topic of research in the field of hardware-software codesign of embedded systems [11]. Previously, a large effort was directed towards synthesis of hard real-time systems, especially in the application of formal methods. Synthesis was mainly carried out for communication protocols [19], plant controllers [4, 18, 5], and real-time schedulers [25, 1] because they generally exhibited regular behaviors. Only recently has there been some work on automatically generating code for embedded systems [17, 16, 23, 26, 6]. In the following, we will briefly survey the existing works on the synthesis of non real-time software and controller synthesis, on which our work is based.

Lin [16, 17] proposed an algorithm that generates a software program from a concurrent process specification through intermediate Petri-Net representation. This approach is based on the assumption that the Petri-Nets are safe, *i.e.*, buffers can store at most one data unit, which implies that it is always schedulable. The proposed method applies *quasi-static scheduling* to a set of safe Petri-Nets to produce a set of corresponding state machines, which are then mapped syntactically to the final software code. Later, Zhu and Lin [26] proposed a compositional version of the synthesis method that reduced the generated code size and was thus more efficient.

A software synthesis method was proposed for a more general Petri-Net framework by Sgroi et al. [23]. A quasi-static scheduling algorithm was proposed for *Free-Choice Petri Nets* (FCPN) [23]. A necessary and sufficient condition was given for a FCPN to be schedulable. Schedulability was first tested for a FCPN and then a valid schedule generated by decomposing a FCPN into a set of *Conflict-Free* (CF) components which were then individually and statically scheduled. Code was finally generated from the valid schedule.

Balarin et al. [6] proposed a software synthesis procedure for reactive embedded systems in the *Codesign Finite State Machine* (CFSM) [7] framework with the POLIS hardware-software codesign tool [7]. This work cannot be easily extended to other more general frameworks.

Besides synthesis, there are also some recent work on the verification of software in an embedded system such as the *Schedule-Verify-Map* method [12], linear hybrid automata techniques [10, 13], and mapping strategy [8].

Controller synthesis for plants (also called supervisor synthesis) was mainly performed in the discrete time domain, with a large portion of classical work done by Ramadge and Wonham [21, 22]. Around 1994, when timed automata was proposed as a dense-time model for real-time systems [3], controller synthesis was extended to dense real-time systems [4, 18, 25] as well as to hybrid systems [24]. Recently, the same technique was further extended to multimedia scheduler synthesis [1]. Given a dense real-time system modeled by timed

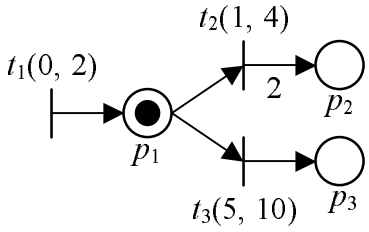

Figure 1. **A Time Free-Choice Petri Net**

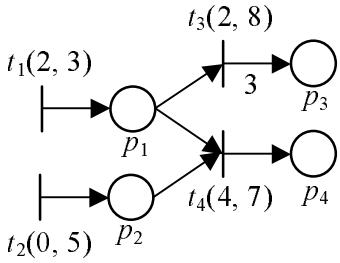

Figure 2. **Not A TFCPN (Confusion)**

automata and a (temporal) property given as a formula in *Timed Computation Tree Logic* (TCTL) [2, 9], a controller is synthesized such that it restricts the behavior of the system for satisfying the property. This is the *controller synthesis problem*. Recently, system parameters have also been taken into consideration for real-time controller synthesis [14].

3. FORMAL SYNTHESIS AND CONTROL

A formal synthesis method for soft embedded real-time systems is presented in this section. Its basic features are that the synthesized system executes in *bounded memory* and satisfies all user-given *soft real-time constraints*. Before going into details, the system model and related terminologies are presented.

A soft embedded real-time system is specified as a set of *Time Free-Choice Petri Nets* (TFCPN), which are time extensions of Free-Choice Petri Nets (FCPN) [23]. As mentioned in Section 2, FCPN was used for the quasi-static scheduling of embedded real-time software. But, there was no concept of time in the FCPN model, which makes it an inconvincing model for *real-time* systems. FCPN was recently extended to include time just as Merlin and Farber's *Time Petri Nets* (TPN) [20] are time extension of standard Petri Nets. The extended version called TFCPN was introduced in [15] and is presented here.

In the rest of this section, we first define TFCPN, give its properties, and explain why TFCPN are used for modeling SERTS. Then, the problem to be solved is formulated. Finally, our proposed synthesis algorithm is described.

3.1 System Model

Definition 1 *:* **Time Free-Choice Petri Nets** (TFCPN)
A *Time Free-Choice Petri Net* is a 5-tuple (P, T, F, M_0, τ), where: P is a finite set of places, T is a finite set of transitions, $P \cup T \neq \emptyset$, and $P \cap T = \emptyset$, $F : (P \times T) \cup (T \times P) \rightarrow \mathcal{N}$ is a weighted flow relation between places and transitions, represented by arcs, such that every arc from a place is either a unique outgoing arc or a unique incoming arc to a transition (this is called

Free-Choice), where $\mathcal{N}$ is a set of nonnegative integers, $M_0 : P \rightarrow N$ is the initial marking (assignment of tokens to places), and $\tau : T \rightarrow Q^* \times (Q^* \cup \infty)$, i.e., $\tau(t) = (\alpha, \beta)$, where $t \in T$, α is the *earliest firing time* (EFT), and β is *latest firing time* (LFT); together they are called *firing interval bounds* (FIB).

Graphically, a TFCPN can be depicted as in Fig. 1, where circles represent places, vertical bars represent transitions, arrows represent arcs, black dots represent tokens, and integers labeled over arcs represent the weights as defined by F. Here, $F(x, y) > 0$ implies there is an arc from x to y with a weight of $F(x, y)$, where x and y can be a place or a transition. *Conflicts* are allowed in a TFCPN, where a conflict occurs when there is a token in a place with more than one outgoing arc such that only one enabled transition can fire, thus consuming the token and disabling all other transitions. For example, t_2 and t_3 are conflicting transitions in Fig. 1. But, *confusions* are not allowed in TFCPN, where a confusion is a result of coexistence of concurrency and conflict. An example of confusion is given in Fig. 2. Transitions t_1 and t_2 are concurrent and t_3 and t_4 are in conflict.

By disallowing confusions, a system modeled by TFCPN can be easily analyzed and synthesized because conflicts can be resolved through net decomposition of a TFCPN into conflict-free components. Though the free-choice restriction disables a designer from describing systems that have coexisting concurrency and conflicts (i.e. synchronization with conflict as in Fig. 2), yet the net decomposition approach can be extended to general Petri nets, which will be part of our future research.

Semantically, the behavior of a TFCPN is given by a sequence of *markings*, where a marking is an assignment of tokens to places. Formally, a marking is a vector $M = \langle m_1, m_2, \ldots, m_{|P|} \rangle$, where m_i is the non-negative number of tokens in place $p_i \in P$. Starting from an initial marking M_0, a TFCPN may transit to another marking through the firing of an enabled transition and re-assignment of tokens. A transition is said to be *enabled* when all its input places have the required number of tokens for the required amount of time, where the required number of tokens is the weight as defined by the flow relation F and the required amount of time is the earliest starting time α as defined by τ. An enabled transition upon firing, the required number of tokens are removed from all the input places and the specified number of tokens are placed in the output places, where the specified number of tokens is that specified by the flow relation F on the connecting arcs. An enabled transition may not fire later than the latest firing time β defined by τ.

SERTS have both data-dependent executions as well as time-dependent specifications. Both of these characteristics are well-captured by TFCPN. TFCPN can distinguish clearly between *choice* and *concurrency*, hence they are good models of data-dependent and concurrent computations. Further, TFCPN can

also distinguish clearly between data-dependent and time-dependent choices, thus TFCPN are well-defined models for our target SERTS.

Some properties of Petri Nets (PN) can be defined as follows. *Reachability*: a marking M' is reachable from a marking M if there exists a firing sequence σ starting at marking M and finishing at M'. *Boundedness*: a PN is said to be k-bounded if the number of tokens in every place of a reachable marking does not exceed a finite number k. A safe PN is one that is 1-bounded. *Deadlock-free*: a PN is deadlock-free if there is at least one enabled transition in every reachable marking. *Liveness*: a PN is live if for every reachable marking and every transition t it is possible to reach a marking that enables t.

3.2 Problem Formulation

In multimedia presentations, network computing, distance learning, and other soft real-time systems, the real-time behavior can be *controlled*, that is, restricted such that the system satisfies some pre-defined specification. For example, if the tolerable network lag in some kind of network computing is pre-specified as 10 seconds, then the behavior of the network computing environment could be controlled such that under all circumstances a maximum of 10 seconds network lag is encountered during computation.

To model the above soft real-time behavior, we define a new simplified linear temporal logic, which a controller is supposed to enforce in a SERTS.

Definition 2 *:* **Timed Reachability Specification**
A *Timed Reachability Specification* (TRS) for a TFCPN $A = (P, T, F, M_0, \tau)$ has the following syntax:

$$\phi ::= \exists\Diamond_{\sim c}\vec{p} \mid \forall\Box_{\sim c}\vec{p} \mid \phi_1 \wedge \phi_2 \tag{1}$$

where $\sim \in \{<, \leq, =, \geq, >\}$, $\vec{p}$ is a non-negative integer vector of $|P|$ elements, and ϕ_1 and ϕ_2 are TRS formulae. ‖

Semantically, $\exists\Diamond_{\sim c}\vec{p}$ means eventually and obeying the timing restriction $\sim c$ there exists a TFCPN marking M such that $M = \vec{p}$, where $\vec{p}$ is a *token assignment* represented by a non-negative integer vector of $|P|$ elements such that each element represents the amount of tokens that must reside in the corresponding place. This definition is the same as a marking, but we do not call it a marking because $\vec{p}$ might not be reachable from the initial marking. Further, $\forall\Box_{\sim c}\vec{p}$ means for all reachable markings M, while obeying the timing restriction $\sim c$, $M = \vec{p}$. Thus, a TRS gives a linear temporal condition that a TFCPN must satisfy. Since we consider a single microprocessor (executing software) in our soft embedded real-time systems, linear temporal logic in the above TRS form (Equation 1) is sufficient for expressing all reachability properties such as safeness, deadlines, boundedness, deadlock-free, and starvation.

Table 1. Soft Embedded Real-Time System Synthesis Algorithm

SERTS_Synthesize(S, μ, ϕ)
set of TFCPN $S = \{A_i \mid A_i = (P_i, T_i, F_i, M_{i0}, \tau_i), i = 1, 2, \ldots, n\}$;
integer μ; // Maximum memory
TRS ϕ; // Specification
{
// *Quasi-Static Data Scheduling* (QSDS)
for each A_i in S { (1)
$B_i =$ **CF_Generate**(A_i); // B_i: set of CF components (2)
for each CF component A_{ij} in B_i { (3)
$QSS_{ij} =$ **Quasi_Static_Schedule**(A_{ij}, μ); // QSS: schedules (4)
if QSS_{ij} = NULL { (5)
print "QSDS failed for A_{ij}"; (6)
return $QSDS\_Error$; } (7)
else $QSS_i = QSS_i \cup \{QSS_{ij}\}$; } } (8)
// *Firing Interval Bound Synthesis* (FIBS)
if **Controller_Synthesize**$(S, QSS_1, \ldots, QSS_n, \phi) = NULL$
return $FIBS\_Error$; (9)
else return $Synthesized$; (10)
}

Other properties which are not as important for SERTS such as liveness cannot be specified using TRS.

Given a system model TFCPN (Definition 1) and a specification logic (Definition 2), we are now ready to formulate our problem as follows.

Definition 3 *:* **Soft Embedded Real-Time System Synthesis**
Given a system modeled by a set of TFCPN $S = \{A_i \mid A_i = (P_i, T_i, F_i, M_{i0}, \tau_i), i = 1, 2, \ldots, n\}$ and a specification ϕ in TRS, the system description S is to be synthesized by scheduling and by modifying firing interval bounds such that S is made to satisfy ϕ. ||

3.3 Synthesis Algorithm

As introduced in Section 1 and formulated in Definition 3, there are two objectives for our SERTS synthesis algorithm, namely bounded memory execution and soft real-time constraints satisfaction. Thus, the algorithm **SERTS_Synthesize()** proposed in Table 1 is intuitively divided into two phases corresponding to the two objectives.

As shown in Table 1, given a set of TFCPNs $S = \{A_i \mid A_i = (P_i, T_i, F_i, M_{i0}, \tau_i), i = 1, 2, \ldots, n\}$, a maximum bound on memory μ, and a TRS ϕ, a system is synthesized upon completion of the following two phases.

3.3.1 Quasi-Static Data Scheduling (QSDS). The basic concept here is to employ net decomposition such that firing choices that exists in a TFCPN are segregated into individual *Conflict-Free* (CF) components. This is done by a procedure **CF_Generate()** as in Step (2) for each A_i, which results in a set B_i of CF components corresponding to A_i. The CF components are not distinct decompositions as a transition may occur in more than one component. As in Step (4), each CF component of each TFCPN is quasi-static scheduled, that is, starting from an initial marking for each component, a *finite complete cycle* is constructed, where a finite complete cycle is a sequence of transition firings that returns the net to its initial marking. A CF component is said to be *schedulable* if a finite complete cycle can be found for it and if it is deadlock-free. Once all CF components of a TFCPN are scheduled, a *valid quasi-static data schedule* QSS_i for the TFCPN A_i can be generated as a set of the finite complete cycles. The reason why this set is a valid schedule is that since each component always returns to its initial marking, no tokens can get collected at any place. Details of this procedure can be found in [23].

We have extended the quasi-static scheduling approach given in [23] to consider timing constraints on transition firings during the scheduling process. A quasi-static schedule is said to be feasible only if all transition firing intervals are satisfied. Satisfaction of memory bound can be checked by observing if the memory space represented by the maximum number of tokens in any marking does not exceed the bound. Here, each token represents some amount of buffer space (i.e., memory) required after a computation (transition firing). Hence, the total amount of actual memory required is the memory space represented by the maximum number of tokens that can get collected in any marking, which results from the transition firings in a quasi-static data schedule.

3.3.2 Firing Interval Bound Synthesis (FIBS). This phase consists of a procedure **Controller_Synthesize()** as in Step (9) of Table 1, which synthesizes a controller for system S with quasi-static schedules QSS_1, $\ldots$, QSS_n to satisfy a TRS ϕ.

Some embedded soft real-time systems, such as multimedia and networks, can tolerate *latencies* that occur due to network lags, inferior display technologies, weak processing power, and limited memory bandwidth. In order to control such systems, normally a *controller* is needed to ensure quality of service (QOS), predictability, and reliability. The two main issues involved in the design of a controller for embedded soft real-time systems are as follows:

- *Synchronization Wait*: A software task, upon completion of its scheduled jobs, may have to wait for a period of time to synchronize with another software task or with the hardware.
- *Real-Time Specification*: In order to satisfy some given real-time specification, such as deadlines, a software task must finish execution of its scheduled jobs earlier than system-permitted deadlines.

Solving the above two issues, a synthesis method must generate a controller that ensures all synchronizations and real-time specifications are met. In our proposed method, the above two issues are solved as follows. Here, each software task T is associated with a time interval (α, β), where α is the earliest start time of T and β is the latest finish time of T.

- *Postpone Release Time*: For synchronization to be feasible and for predictable behavior, a software task that needs to wait for some other tasks, should have its earliest start time α changed into $\alpha + \delta_w$, where $\delta_w > 0$ is the amount of wait time required.
- *Advance Finish Time*: For satisfaction of real-time specifications, the deadline of a software task is advanced from β to $\beta - \delta_h$, where $\delta_h > 0$ is the difference in the user-specified and system-permitted deadlines.

As shown in Table 2, a solution to FIBS is proposed as an algorithm **Controller_Synthesize()**, which consists of three nested for-loops spanning over each TFCPN (Step (1)), over each schedule of a TFCPN (Step (2)), and over each transition in a schedule (Step (3)). *Firing Interval Bound Synthesis* or *Controller Synthesis* mainly restricts some transition firing interval $\tau(t) = (\alpha, \beta)$ into a smaller interval (α', β'), where $\alpha' \geq \alpha$ and $\beta' \leq \beta$, such that a given TRS formula is satisfied. In the above case, (α, β) is said to be *less restricted* than (α', β').

The conditions given in Step (3) of Table 2 specifies that we consider only each prefix $\vec{t} = \langle t_0, t_1, \ldots, t_k \rangle$ of a schedule v_{ij} that leads to a possible token assignment specified in some component of ϕ. A transition in in_trans(p) is an incoming transition of place p and the function token$_{\phi_i}(p)$ gives the number of tokens at place p specified in the ith component ϕ_i of ϕ. First, as in Step (4) the aggregate delay interval τ is calculated for a schedule prefix $\vec{t}$ by summing up all the EFT α_i and all the LFT β_i of transitions t_i in $\vec{t}$. Then, a full set of new interval bounds (New_IBS$_i$) is constructed by procedure **IBS_Synthesize()** in Step (5), with details in Table 3.

Corresponding to the two kinds of path-formulae in a TRS ϕ, there are two ways for incorporating the new set of interval bounds in S.

1 $\phi = \exists \Diamond_{\sim c} \vec{p_i}$:
A variable Min_IBS$_i$ keeps track of the set of minimally restricted transition firing intervals of A_i for satisfying ϕ (Steps (6) and (7)). A solution

Table 2. Controller Synthesis Algorithm for TFCPN/TRS

Controller_Synthesize$(S, QSS_1, \ldots, QSS_n, \phi)$
set of TFCPN $S = \{A_i \mid A_i = (P_i, T_i, F_i, M_{i0}, \tau_i), i = 1, 2, \ldots, n\}$;
TRS $\phi = M_1\vec{p_1} \wedge M_2\vec{p_2} \wedge \ldots, M_n\vec{p_n},$ where $\phi_i = M_i\vec{p_i}$,
$M_i \in \{\exists\Diamond_{\sim c}, \forall\Box_{\sim c}\}, \vec{p_i} = \langle x_1, x_2, \ldots, x_{|P_i|}\rangle, x_i \in \mathcal{N}_{\geq 0}$;
{
  for $i = 1, \ldots, n$ { (1)
    for each schedule $v_{ij} \in QSS_i$ { (2)
      for each $t_k \in v_{ij}$, $t_k \in$ in_trans(p) and token$_{\phi_i}(p) > 0, p \in P_i$ { (3)
        $\tau = (\Sigma_{i=0}^{k}\alpha_i, \Sigma_{i=0}^{k}\beta_i)$; // $\langle t_0, t_1, \ldots, t_k\rangle$ is a prefix of v_{ij} (4)
        New_IBS$_i$ = **IBS_Synthesize**$(v_{ij}, t_k, \tau, \phi_i)$; (5)
      if $M_i = \exists\Diamond_{\sim c}$ and New_IBS$_i >$ Min_IBS$_i$ (6)
        Min_IBS$_i$ = New_IBS$_i$; (7)
      if $M_i = \forall\Box_{\sim c}$ Old_IBS$_i$ = Old_IBS$_i \cap$ New_IBS$_i$; } } (8)
    if $M_i = \exists\Diamond_{\sim c}$ and Min_IBS$_i \neq NULL$
      IBS_assign(Min_IBS$_i$); // modify τ (9)
    else if $M_i = \forall\Box_{\sim c}$ and Old_IBS$_i \neq NULL$
      IBS_assign(Old_IBS$_i$); // modify τ (10)
    else return NULL; } (11)
  return τ; (12)
}

consisting of a set of *minimally restricted* intervals is sought because such a solution contains the *maximal behavior* of the original system S that satisfies specification ϕ.

2 $\phi = \forall\Box_{\sim c}\vec{p_i}$:
A variable Old_IBS$_i$ records the intersection of all sets of restricted transition firing intervals of A_i for satisfying ϕ (Step (8)). A set intersection is performed by individual intersections of each pair of intervals $\tau_1 = (\alpha_1, \beta_1)$ and $\tau_2 = (\alpha_2, \beta_2)$, that is, $\tau_1 \cap \tau_2 = (\alpha', \beta')$, where $\alpha' = \max\{\alpha_1, \alpha_2\}$ and $\beta' = \min\{\beta_1, \beta_2\}$.

IBS_assign() in Steps (9) and (10) assigns the final set of interval bounds to the system S.

IBS_Synthesize() in Table 3 synthesizes (modifies) the firing interval bounds for a sequence of transition firings, which is a prefix of a schedule $v_{ij} = \langle t_0, t_1, \ldots, t_k, \ldots\rangle$, such that the modified system satisfies both ϕ and the aggregate delay interval τ. The switch-case statement in Step (1) to Step (7) first decides what is the least restriction on $\tau = (\alpha, \beta)$ by calculating $\tau' = (\alpha', \beta')$ such that ϕ is satisfied. Then, depending on whether the calculated restriction

Table 3. Synthesis of Interval Bounds Set

IBS_Synthesize(v_{ij}, t, τ, ϕ)
schedule $v_{ij} = \langle t_{i0}, \ldots, t_{ik}, \ldots \rangle$; // $\tau(t_{ij}) = (\alpha_{ij}, \beta_{ij})$
transition $t = t_k \in v_{ij}$;
FIB $\tau = (\alpha, \beta)$;
TRS $\phi = M\vec{p}, M \in \{\exists\Diamond_{\sim c}, \forall\Box_{\sim c}\}, \vec{p} = \langle x_1, \ldots, x_{|P_i|} \rangle$;
{
switch "$\sim$" { (1)
case $<$: if $(c \leq \beta)$ $\tau' = (\alpha, c-1)$; break; (2)
case $\leq$: if $(c \leq \beta)$ $\tau' = (\alpha, c)$; break; (3)
case $=$: $\tau' = (c, c)$; break; (4)
case $\geq$: if $(c \geq \alpha)$ $\tau' = (c, \beta)$; break; (5)
case $>$: if $(c \geq \alpha)$ $\tau' = (c+1, \beta)$; break; (6)
} // let $\tau' = (\alpha', \beta')$ (7)
if $\alpha' > \alpha$ { (8)
for $j = k, \ldots, 0$ do { (9)
α_{ij} += min$\{\alpha' - \alpha, \beta_{ij} - \alpha_{ij}\}$; (10)
if $(\alpha' - \alpha \leq \beta_{ij} - \alpha_{ij})$ break; (11)
else $\alpha' -= \beta_{ij} - \alpha_{ij}$; (12)
if $j = 0$ return *Unsynthesizable*; } } (13)
if $\beta' < \beta$ do { (14)
for $j = k, \ldots, 0$ do { (15)
β_{ij} -= min$\{\beta - \beta', \beta_{ij} - \alpha_{ij}\}$; (16)
if $(\beta - \beta' \leq \beta_{ij} - \alpha_{ij})$ break; (17)
else $\beta' += \beta_{ij} - \alpha_{ij}$; (18)
if $j = 0$ return *Unsynthesizable*; } } (19)
return $\{(\alpha_{ij}, \beta_{ij}) : j = 0, \ldots, k\}$; (20)
}

is on EFT (Step (8)) or LFT (Step (14)), there is a loop for modifying the firing interval bounds $(\alpha_{ij}, \beta_{ij})$ of transitions starting from the kth one. If even after all transitions have firing intervals modified and ϕ is still not satisfied then an error is returned (Steps (13) and (19)). Otherwise, the set of modified firing intervals is returned (Step (20)).

After applying the controller synthesis algorithm (Table 2) to a system S, some transition firing intervals of the TFCPNs in S are restricted into smaller intervals such that the restricted (controlled) system S' satisfies a given specification TRS ϕ and there is no other lesser restricted system S'' that can satisfy ϕ. An example will be given in Section 4.

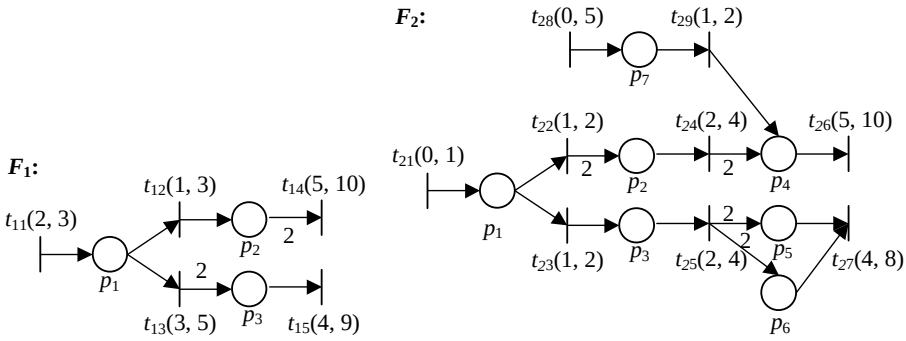

Figure 3. **Application Example** $S = (F_1, F_2)$

4. APPLICATION EXAMPLE

A 2-process system example is given in this section to illustrate the proposed SERTS synthesis algorithm. It consists of two TFCPN (F_1 and F_2) as shown in Fig. 3 and a *Timed Reachability Specification* (TRS) formula as below:

$$\phi : \exists \Diamond_{\leq 7} \langle 002 \rangle \wedge \exists \Diamond_{\geq 30} \langle 0000001 \rangle \tag{2}$$

According to our proposed algorithm (Table 1), we apply quasi-static data scheduling and controller synthesis to the given system.

QSDS for F_1: Since t_{12} and t_{13} are conflicting transitions, two CF components (R_{11} and R_{12} in Fig. 4) are derived, which are then individually scheduled, resulting in the following two schedules, with their associated execution time intervals.

$$v_{11} = (t_{11}t_{12}t_{11}t_{12}t_{14}), \qquad 11 \leq \tau(v_{11}) \leq 22 \tag{3}$$
$$v_{12} = (t_{11}t_{13}t_{15}t_{15}), \qquad 13 \leq \tau(v_{12}) \leq 26 \tag{4}$$

There can be two sets of valid schedules for this TFCPN as given below.

$$\Sigma_1 = \{(t_{11}t_{12}t_{11}t_{12}t_{14}), (t_{11}t_{13}t_{15}t_{15})\} \tag{5}$$
$$\Sigma_2 = \{(t_{11}t_{13}t_{15}t_{15}), (t_{11}t_{12}(t_{11}t_{13}t_{15}t_{15})^k t_{11}t_{12}t_{14}, \quad k \in \mathcal{N})\} \tag{6}$$

QSDS for F_2: Since t_{22} and t_{23} are conflicting transitions, two CF components (R_{21} and R_{22} in Fig. 5) are derived, which are then individually scheduled, resulting in the following two schedules, with their associated execution time

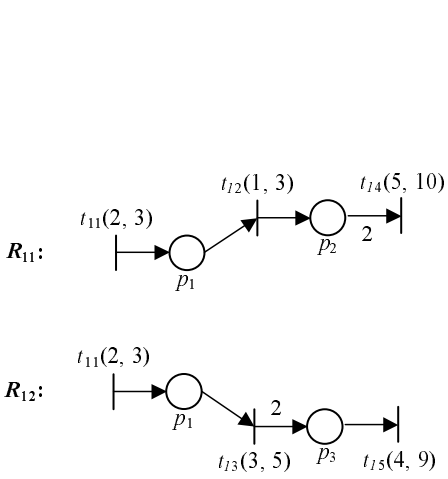

Figure 4. **Conflict-Free Components of F_1**

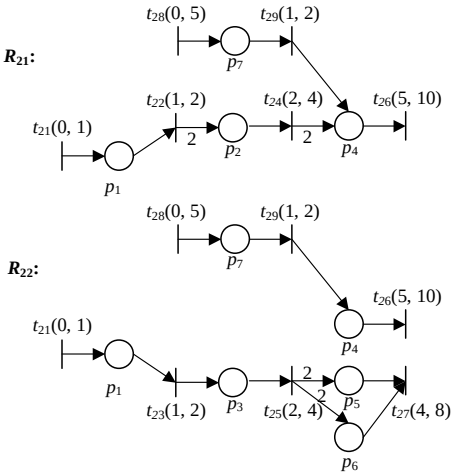

Figure 5. **Conflict-Free Components of F_2**

intervals.

$$v_{21} = (t_{21}t_{22}t_{24}t_{24}t_{26}t_{26}t_{26}t_{28}t_{29}t_{26}), \qquad 31 \leq \tau(v_{21}) \leq 68 \tag{7}$$

$$v_{22} = (t_{21}t_{23}t_{25}t_{27}t_{27}t_{28}t_{29}t_{26}), \qquad 15 \leq \tau(v_{22}) \leq 36 \tag{8}$$

The set of valid schedules for this TFCPN is as given below.

$$\Sigma_3 = \{(t_{21}t_{22}t_{24}t_{24}t_{26}t_{26}t_{26}t_{28}t_{29}t_{26}), (t_{21}t_{23}t_{25}t_{27}t_{27}t_{28}t_{29}t_{26})\} \tag{9}$$

Controller Synthesis: In Equation (2), the first conjunct in ϕ corresponds to F_1 and specifies that the TFCPN F_1 reaches a marking within less than or equal to 7 time units such that there are no tokens in places p_1 and p_2 and there are two tokens in p_3. The second conjunct corresponds to F_2 and specifies that the TFCPN F_2 reaches a marking after 30 time units, inclusive, such that there are no tokens in any of the first six places ($p_1, \ldots, p_6$) and there is one token in place p_7. Applying the controller synthesis algorithm from Table 2, we have the following results.

FIBS for F_1: First, consider the conjunct in ϕ that corresponds to F_1, that is, $\exists\Diamond_{\leq 7}\langle 002\rangle$. Since there is only one schedule ($v_{12} = (t_{11}t_{13}t_{15}t_{15})$) in Σ_1 (Equation (4)) that results in p_3 having tokens. We calculate the time required by the prefix of the schedule that leads to 2 tokens in p_3 as follows:

$$\begin{aligned} 2+3 &\leq \tau(t_{11}) + \tau(t_{13}) \leq 3+5 \\ 5 &\leq \tau(t_{11}) + \tau(t_{13}) \leq 8 \end{aligned}$$

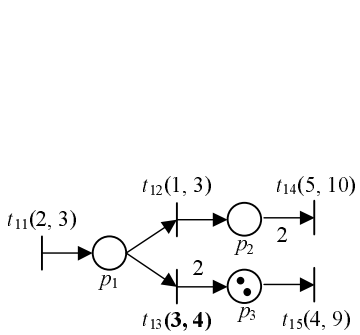

Figure 6. **Controlled TFCPN F_1**

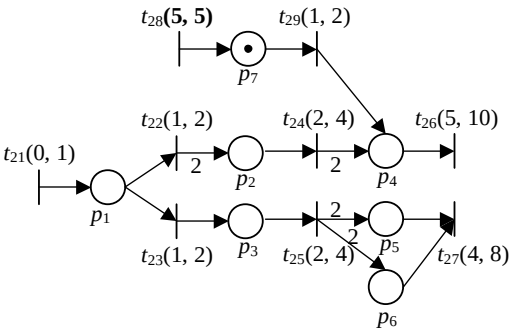

Figure 7. **Controlled TFCPN F_2**

Thus applying the IBS synthesis algorithm from Table 3, for the time spent on the schedule prefix ($t_{11}t_{13}$) to satisfy (≤ 7) constraint, the firing interval of t_{13} is modified as follows.

$$\tau(t_{13}) = (3, 4) \tag{10}$$

FIBS for F_2: For TFCPN F_2, we must consider both the schedules v_{21} and v_{22} from Equations (7) and (8) because both the schedules have prefixes that lead to a token in place p_7. First, the aggregate delay interval is calculated for a prefix of v_{21} as follows.

$$25 \leq \tau(t_{21}t_{22}t_{24}t_{24}t_{26}t_{26}t_{26}t_{26}t_{28}) \leq 56$$

Thus, to satisfy the constraint of ≥ 30, the firing interval of t_8 is modified as follows.

$$\tau(t_{28}) = (5, 5) \tag{11}$$

When we consider a prefix of schedule v_{22}, as shown below it is impossible to modify any firing interval of transitions in T_2 to satisfy the ≥ 30 constraint because the maximum firing delay is only 28.

$$11 \leq \tau(t_{21}t_{23}t_{25}t_{27}t_{27}t_{28}) \leq 28$$

After modifying the firing intervals of transitions t_3 and t_8, we get the two controlled TFCPN as illustrated in Figs. 6 and 7.

As a last note on controller synthesis for this example, let us suppose the TRS specification is changed to the following.

$$\phi' : \exists\Diamond_{\leq 7}\langle 002\rangle \wedge \exists\Diamond_{\geq 60}\langle 0000001\rangle \tag{12}$$

Then, there is no modification of any firing interval of any transition that can make the system S to satisfy ϕ'. In this case, the controller is *unsynthesizable*.

5. CONCLUSION

Instead of ad-hoc, trial-and-error methods that engineers use in developing *Soft Embedded Real-Time Systems* (SERTS), it has been proposed in this work how SERTS can be developed by a formal automatic synthesis method. To satisfy the limited memory space and processor power requirements of a soft real-time embedded system, two phases, namely *Quasi-Static Data Scheduling* (QSDS) and *Firing Interval Bound Synthesis* (FIBS) are performed before code generation. Engineers will benefit from our work when he/she applies the proposed method to automatically and formally synthesize a system specification modeled as a set of TFCPNs. Future research directions include the extension of system models (TFCPN) to more general ones such that a larger domain of system can be synthesized.

REFERENCES

[1] K. Altisen, G. Gobler, A. Pneuli, J. Sifakis, S. Tripakis, and S. Yovine. A framework for scheduler synthesis. In *Proceedings of the Real-Time System Symposium (RTSS'99)*. IEEE Computer Society Press, 1999.

[2] R. Alur, C. Courcoubetis, N. Halbwachs, and D. Dill. Modeling checking for real-time systems. In *Proceedings of the IEEE International Conference on Logics in Computer Science (LICS'90)*, 1990.

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

[4] E. Asarin, O. Maler, and A. Pneuli. Symbolic controller synthesis for discrete and timed systems. In P. Antsaklis, W. Kohn, A. Nerode, and S. Sastry, editors, *Hybrid Systems II*, volume 999, pages 1 – 20. Lecture Notes in Computer Science, Springer Verlag, 1995.

[5] E. Asarin, O. Maler, A. Pneuli, and J. Sifakis. Controller synthesis for timed automata. In *Proceedings of System Structure and Control*. IFAC, Elsevier, July 1998.

[6] F. Balarin and M. Chiodo. Software synthesis for complex reactive embedded systems. In *Proceedings of International Conference on Computer Design (ICCD'99)*, pages 634 – 639. IEEE CS Press, October 1999.

[7] F. Balarin and et al. *Hardware-software Co-design of Embedded Systems: the POLIS approach*. Kluwer Academic Publishers, 1997.

[8] J.-M. Fu, T.-Y. Lee, P.-A. Hsiung, and S.-J. Chen. Hardware-software timing coverification of distributed embedded systems. *IEICE Transactions on Information and Systems*, E83-D(9):1731–1740, September 2000.

[9] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. In *Proceedings of the IEEE International Conference on Logics in Computer Science (LICS'92)*, 1992.

[10] P.-A. Hsiung. Timing coverification of concurrent embedded real-time systems. In *Proceedings of the 7th IEEE/ACM International Workshop on Hardware Software Codesign (CODES'99, Rome, Italy)*, pages 110 – 114. ACM Press, May 1999.

[11] P.-A. Hsiung. CMAPS: A cosynthesis methodology for application-oriented parallel systems. *ACM Transactions on Design Automation of Electronic Systems*, 5(1):51–81, January 2000.

[12] P.-A. Hsiung. Embedded software verification in hardware-software codesign. *Journal of Systems Architecture — the Euromicro Journal*, 46(15):1435–1450, December 2000.

[13] P.-A. Hsiung. Hardware-software timing coverification of concurrent embedded real-time systems. *IEE Proceedings — Computers and Digital Techniques*, 147(2):81–90, March 2000.

[14] P.-A. Hsiung. Synthesis of parametric embedded real-time systems. In *Proceedings of the International Computer Symposium (ICS'00), Workshop on Computer Architecture (ISBN 957-02-7308-9)*, pages 144–151, December 2000.

[15] P.-A. Hsiung. Formal synthesis and code generation of embedded real-time software. In *International Symposium on Hardware/Software Codesign (CODES'01, Copenhagen, Denmark)*, pages 208–213. ACM Press, New York, USA, April 2001.

[16] B. Lin. Efficient compilation of process-based concurrent programs without run-time scheduling. In *Proceedings of Design Automation and Test Europe (DATE'98)*, pages 211 – 217. ACM Press, February 1998.

[17] B. Lin. Software synthesis of process-based concurrent programs. In *Proceedings of IEEE/ACM Design Automation Conference (DAC'98)*, pages 502 – 505. ACM Press, June 1998.

[18] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *12th Annual Symposium on Theoretical Aspects of Computer Science (STACS'95)*, volume 900, pages 229 – 242. Lecture Notes in Computer Science, Springer Verlag, March 1995.

[19] P. Merlin and G.V. Bochman. On the construction of submodule specifications and communication protocols. *ACM Transactions on Programming Languages and Systems*, 5(1):1 – 25, January 1983.

[20] P. Merlin and D. Farber. Recoverability of communication protocols – implication of a theoretical study. *IEEE Transactions on Communications*, September 1976.

[21] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal of Control and Optimization*, 25:206 – 230, 1987.

[22] P.J. Ramadge and W.M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77:81 – 98, 1989.

[23] M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli. Synthesis of embedded software using free-choice petri nets. In *Proceedings IEEE/ACM Design Automation Conference (DAC'99)*. ACM Press, June 1999.

[24] H. Wong-Toi. The synthesis of controllers for linear hybrid automata. In *Proceedings of the International Conference CDC'97*, 1997.

[25] H. Wong-Toi and G. Hoffman. The control of dense real-time discrete event systems. Technical Report STAN-CS-92-1411, Stanford University, 1992.

[26] X. Zhu and B. Lin. Compositional software synthesis of communicating processes. In *Proceedings of International Conference on Computer Design (ICCD'99)*, pages 646 – 651. IEEE CS Press, October 1999.