Formal Synthesis and Code Generation of Embedded Real-Time Software

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

ABSTRACT

Due to rapidly increasing system complexity, shortening time-to-market, and growing demand for hard real-time systems, formal methods are becoming indispensable in the synthesis of embedded systems, which must satisfy stringent temporal, memory, and environment constraints. There is a general lack of practical formal methods that can synthesize complex embedded real-time software (ERTS). In this work, a formal method based on *Time Free-Choice Petri Nets* (TFCPN) is proposed for ERTS synthesis. The synthesis method employs quasi-static data scheduling for satisfying limited embedded memory requirements and uses dynamic real-time scheduling for satisfying hard real-time constraints. Software code is then generated from a set of quasi-statically and dynamically scheduled TFCPNs. Finally, an application example is given to illustrate the feasibility of the proposed TFCPN-based formal method for ERTS synthesis.

Keywords

Embedded real-time software, Petri Nets, scheduling, code generation

1. INTRODUCTION

Recently, there has been a proliferation of *embedded real-time systems* in the form of home appliances, internet appliances, personal assistants, wearable computers, telecommunication gadgets, and transportation facilities among numerous others. In the near future, we will see a continuing escalation of system complexity, shortening of time-to-market, and growing demands for hard real-time. All these factors, coupled with the need to satisfy stringent temporal, memory, and environment constraints, have propelled the requirement of practical formal methods for the efficient synthesis of such systems, which usually have both embedded *hardware* and embedded *software*. In contrast to the maturity of hardware design methodologies [10], software design techniques are still relatively immature and sparse. Thus, there is a need for practical formal synthesis techniques targeted at *embedded real-time software* (ERTS).

In light of the above-mentioned need, a formal synthesis method based on *Time Free-Choice Petri Nets* (TFCPN) is proposed, which employs quasi-static data scheduling for satisfying limited embedded memory restrictions and uses dynamic real-time scheduling for satisfying hard real-time constraints. Software code is then generated from a set of scheduled TFCPNs. An application example will illustrate the feasibility and benefits of our proposed method.

An embedded real-time system is a computation unit, installed in a larger environment system, such that it helps the environment accomplish some dedicated set of tasks. Some examples include avionics flight control, vehicle cruise control, washing machine fuzzy control, and network-enabling devices in home appliances such as embedded web servers. In general, an embedded system has both hardware and software parts. Hardware is fabricated as one or more ASICs, ASIPs, or PLDs. Software is executed on one or more microprocessors, with or without an operating system. *Embedded real-time software* (ERTS) is a piece of program code that must satisfy real-time constraints such as response time, deadlines, and periods. ERTS communicates with the embedded hardware either through an interface or through direct connections.

Two main issues involved in the design of ERTS are:

- *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.
- *Real-Time Constraints*: A processor may have to execute several concurrent tasks with precedence and temporal constraints. Thus, an ERTS is generally composed of several concurrent, computation, real-time tasks.

In solution to the above two issues, a synthesis method for ERTS must generate program code that can be executed in a bounded amount of memory, while satisfying all given real-time constraints. The proposed solutions to the above two issues are as follows:

- *Quasi-Static Data Scheduling*: The bounded memory execution issue can be solved by *quasi-static data scheduling* (QSDS), which guarantees that, for all possible outcomes in a non-deterministic data-dependent execution choice, the memory utilized for computation is always bounded and the execution of the software is periodic, that is it always returns to its initial status.
- *Dynamic Real-Time Scheduling*: The real-time constraints issue can be solved by *dynamic real-time scheduling* (DRTS), which guarantees that a set of concurrent real-time software tasks can be executed on a processor, while satisfying all precendence and temporal constraints.

CODES 01 Copenhagen Denmark
Copyright ACM 2001 1-58113-364-2/01/04...$5.00

This article is organized as follows. Section 2 gives some previous work related to ERTS synthesis. Section 3 formulates, models, and solves the ERTS synthesis problem. Section 4 illustrates the proposed problem solution through an application example. Section 5 concludes the article giving some future work.

2. PREVIOUS WORK

Currently, *software synthesis* is a hot topic of research in the field of hardware-software codesign of embedded systems [6]. Previously, a large effort was directed towards hardware synthesis and comparatively little attention paid to software synthesis. Partial software synthesis was mainly carried out for communication protocols [14], plant controllers [13], and real-time schedulers [1] because they generally exhibited regular behaviors. Only recently has there been some work on automatically generating software code for embedded systems [11, 16, 17, 2]. Except for MetaH from Honeywell, no other automatic software synthesis method is available for *concurrent embedded real-time software*. In the following, we will briefly survey the existing works on the synthesis of non real-time software, on which our work is based.

Lin [11] 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 [17] 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. [16]. A quasi-static scheduling algorithm was proposed for *Free-Choice Petri Nets* (FCPN) [16]. 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. [2] proposed a software synthesis procedure for reactive embedded systems in the *Codesign Finite State Machine* (CFSM) [3] framework with the POLIS hardware-software codesign tool [3]. This work cannot be easily extended to other more general frameworks.

Besides synthesis of software, there are also some recent work on the verification of software in an embedded system such as the *Schedule-Verify-Map* method [7], the linear hybrid automata techniques [5, 8], and the mapping strategy [4]. Recently, system parameters have also been taken into consideration for real-time software synthesis [9].

3. EMBEDDED REAL-TIME SOFTWARE SYNTHESIS

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

An embedded real-time software is specified as a set of *Time Free-Choice Petri Nets* (TFCPN), which are time extensions of Free-Choice Petri Nets (FCPN) [16]. 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* software. Hence, we propose a time extension of FCPN, just as *Time Petri Nets* (TPN) are a time extension of standard Petri Nets, which was proposed by Merlin and Farber [15].

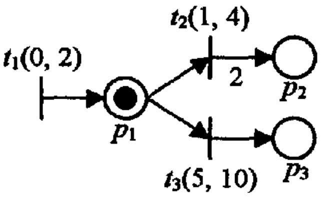

Figure 1: A Time Free-Choice Petri Net

In the rest of this section, we first define TFCPN, its properties, and explain why TFCPN are used for modeling ERTS. Then, the problem formulation is given. Finally, our proposed synthesis algorithm is described, along with code generation.

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). ||

Graphically, a TFCPN can be depicted as shown 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.

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 need not necessarily fire. But 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.

ERTS has both data-dependent executions, as well as, time-dependent specifications. Both of these characteristics are well-captured by TFCPN. TFCPN can distinguish clearly between *concurrency* and *choice*, 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 ERTS.

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

A user specifies the requirements for the design an embedded real-time software by a set of TFCPNs. The problem we are trying to solve here is to find a construction method by which a set of TFCPNs can be made feasible to execute as a software code, running under given limited memory space and satisfying all given real-time constraints. The following is a formal definition of the ERTS synthesis problem.

DEFINITION 2. : ERTS Synthesis
Given a set of TFCPNs, an upper-bound on memory use, and a set of real-time constraints, a software code is to be generated such that (1) it can be executed on a single processor, (2) it uses memory less than or equal to the upper-bound, and (3) it satisfies all the real-time constraints. ||

3.3 Synthesis Algorithm

As introduced in Section 1 and formulated in Definition 2, there are two objectives for our ERTS synthesis algorithm, namely bounded memory execution and real-time constraints satisfaction. The algorithm proposed here is thus 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 set of periods $E = \{\pi_i \mid \pi_i \in \mathcal{N}, i = 1, 2, \ldots, n\}$, where π_i is the period of A_i, a software code is generated after the following two phases:

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. The CF components are not distinct decompositions as a transition may occur in more than one component. 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 it is deadlock-free. Once all CF components of a TFCPN are scheduled, a valid quasi-static data schedule for the TFCPN 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. Some details of this procedure can be found in [16]. Satisfaction of memory bound can be checked by observing if the memory space represented by the maximum number of tokens in any place 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 at a place during its transition from the initial marking back to its initial marking.

Table 1: Embedded Real-Time Software Synthesis Algorithm

```
ERTS_Synth($S, \mu, E$)
$S = \{A_i \mid A_i = (P_i, T_i, F_i, M_{i0}, \tau_i), i = 1, 2, \ldots, n\}$;
integer $\mu$;     // Maximum memory
$E = \{\pi_i \mid \pi_i \in \mathcal{N}, i = 1, 2, \ldots, n\}$; {
  // Quasi-Static Data Scheduling (QSDS)
  for each $A_i$ in $S$ {
    $B_i$ = CF_generate($A_i$); // $B_i$: set of CF components
    for each CF component $A_{ij}$ in $B_i$ {
      $QSS_{ij}$ = quasi_static_schedule($A_{ij}, \mu$);
      if $QSS_{ij}$ = NULL {
        print "QSDS failed for $A_{ij}$";
        return $QSDS\_Error$; }
      else $QSS_i = QSS_i \cup \{QSS_{ij}\}$; }}
  // Dynamic Real-Time Scheduling (DRTS)
  $RTS$ = real_time_schedule($QSS_1, \ldots, QSS_n$,
    $B_1, B_2, \ldots, B_n, E$);
  if $RTS$ = NULL {
    print "DRTS failed for $S$";
    return $DRTS\_Error$; }
  else generate_code($S, QSS_1, \ldots, QSS_n, RTS$);
  return $Synthesized$;
}
```

2. *Dynamic Real-Time Scheduling* (DRTS): The basic concept here is to find if all the TFCPNs can be scheduled for execution along a single time axis (because we are considering only single processor systems). From QSDS, each CF component has a corresponding finite complete cycle, thus the execution time interval for this firing sequence can be calculated by summing up all the EFT and LFT values, respectively, of each transition in the sequence. Among all the execution time intervals of CF components belonging to the same TFCPN, the maximum LFT is selected as the worst-case execution time of that TFCPN. Then, a real-time scheduling algorithm such as *Rate-Monotonic* or *Earliest-Deadline First* is employed to scheduled all the TFCPNs with their worst-case execution times and periods from the set E.

After data and real-time scheduling, the set of TFCPNs is translated into software programs by a *code generation procedure* as shown in Table 2. A *real-time process* is created for each TFCPN. In each process, a *task* is created for each transition with indepen-

Table 2: Code Generation Algorithm

```
generate_code($S, QSS_1, QSS_2, \ldots, QSS_n, RTS$)
$S = \{A_i \mid A_i = (P_i, T_i, F_i, M_{i0}, \tau_i), i = 1, 2, \ldots, n\}$;
set of finite complete cycles $QSS_i, i = 1, \ldots, n$;
a finite periodic real-time schedule $RTS = \langle A_{i_1}, A_{i_2}, \ldots \rangle$; {
    for $i = 1, \ldots, n$ {
        $D_i$ = create_process($QSS_i$);
        for $j = 1, \ldots, IFR(A_i)$ {
            $d_{ij}$ = create_task($QSS_i$);
            generate_task_code($d_{ij}$);
            add_task($d_{ij}, D_i$); } }
    create_main();
    output “for(i=0, i<length(RTS); i++) { ”;
    for $k = 1, \ldots, |RTS|$
            output_code($D_{i_k}$);
    output “}”;
}
```

$IFR(A_i)$: # transitions in A_i with *independent firing rates*

Table 3: Task Code Generation Algorithm

```
generate_task_code($d_{ij}$)
$d_{ij}$: $j$th task with independent firing rate in $A_i$, where
$d_{ij} = \{d_{ijk} \mid d_{ijk} = \langle t_{k0}, t_{k1}, \ldots, t_{ku_k} \rangle, k \geq 0\}$ {
    output $t_0$;    // $t_0$: source transition
    for each ICF sub-component $d_{ijk}$ in $d_{ij}$ {
        for $l = 1, \ldots, u_k$ {
            if $t_l$ is visited continue;
            if $t_l$ is a conflicting transition in $T_i$ {
                if $p$ = in_place($t_l$) is not yet visited
                    output “switch(p) {”;
                else output “break;”;
                output “case tl:  call tl;”;
                for all $p'$ = out_place($t_l$)
                    output “count(p')+=F(t(l),p');”
                $times\_visited_p$ + +; }
            if NumFire($t_l$) < NumFire($t_{l-1}$) {
                output “if(count(p)>=F(p,tl) {”;
                output “call tl;”
                output “count(p)-=
                        NumFire(t(l-1));}”; }
            if NumFire($t_l$) > NumFire($t_{l-1}$) {
                output “while(count(p)
                        >=F(p,tl)){ call tl;”;
                for all $p$ = in_place($t_l$)
                    output “count(p)-=F(p,tl);”;
                output “}”; }
            if NumFire($t_l$) = NumFire($t_{l-1}$) {
                output “count(p) -= F(p,tl);”;
                output “call tl;”;
                output “count(p') += F(tl,p');”; }
            if $times\_visited_p$ = num_choice($p$)
                output “} ”; } }
}
```

dent firing rate. Here, a transition is said to have an independent firing rate if it is a source transition and its firing does not depend on any tokens being in any place. This method of task code generation optimizes (minimizes) the number of tasks in a process because the degree of concurrency in a process is equal to the number of independently firing transitions [16]. The transitions that constitute a task can be either a subset of a single CF component or a union of two or more subsets of different CF components.

Table 3 shows code generation for a task. A switch-case structure is generated whenever a conflicting transition is encountered, such that each choice of the conflict is represented by a case statement. Each case in the structure is constructed by scanning parts of the task from different CF components. In the case of multi-rate TFCPN, the following three cases hold, where NumFire(t) is the number of times a transition t fires in a given QSDS schedule:

1. NumFire(t_l) < NumFire(t_{l-1}): a transition t_{l-1} may fire several times for tokens to accumulate in an output place such that some succeeding transition t_l that needs more than one token is enabled for firing. A count(p) variable is used to keep track of tokens accumulated at place p.
2. NumFire(t_l) > NumFire(t_{l-1}): after a transition t_{l-1} fires once, there may be more than enough tokens in one of its output places such that a succeeding transition t_l may have to fire several times to consume the generated tokens. A count(p) variable is used to keep track of tokens left unconsumed at place p.
3. NumFire(t_l) = NumFire(t_{l-1}): since both successive transitions have the same rate, a direct output of the transition computation code is performed.

After all task codes are generated for each process. A *main()* procedure is generated by constructing a schedule-loop for the real-time schedules generated during dynamic real-time scheduling.

4. APPLICATION EXAMPLE

A 2-process example is given in this section to illustrate the proposed ERTS synthesis algorithm, including code generation. Figure 2 shows two TFCPN (F_1 and F_2) and the associated firing intervals, which constitute the ERTS requirements. Our goal is to generate feasible scheduled code from the requirements.

According to our proposed algorithm (Table 1), we apply quasi-static data scheduling and dynamic real-time scheduling 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. 3) are derived, which are then individually scheduled, resulting in the following two schedules, with their associated execution time intervals.

$$v_{11} = (t_{11}t_{12}t_{11}t_{12}t_{14}), \qquad 11 \leq \tau(v_{11}) \leq 22 \qquad (1)$$

$$v_{12} = (t_{11}t_{13}t_{15}t_{15}), \qquad 13 \leq \tau(v_{12}) \leq 26 \qquad (2)$$

There are two sets of valid schedules for this TFCPN:

$$\Sigma_1 = \{v_{11}, v_{12}\} \qquad (3)$$

$$\Sigma_2 = \{v_{12}, (t_{11}t_{12}v_{12}^{k}t_{11}t_{12}t_{14}, \forall k \in \mathcal{N} \cup \{\infty\})\} \qquad (4)$$

QSDS for F_2: Since t_2 and t_3 are conflicting transitions, two CF components (R_{21} and R_{22} in Fig. 4) are derived, which are then individually scheduled, resulting in the following two schedules, with their associated execution time intervals.

$$v_{21} = (t_{21}t_{22}t_{24}t_{24}t_{26}t_{26}t_{26}t_{26}t_{28}t_{29}t_{26}), 31 \leq \tau(v_{21}) \leq 68 \quad (5)$$

$$v_{22} = (t_{21}t_{23}t_{25}t_{27}t_{27}t_{28}t_{29}t_{26}), 15 \leq \tau(v_{22}) \leq 36 \quad (6)$$

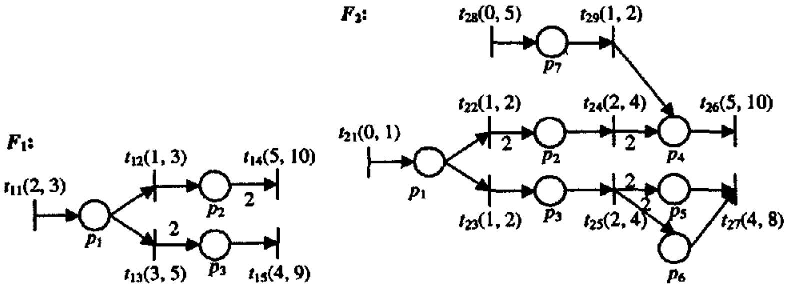

Figure 2: Application Example $S = (F_1, F_2)$

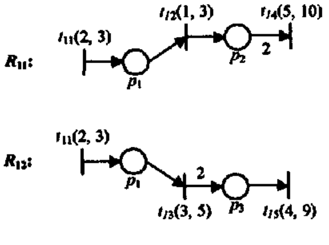

Figure 3: Conflict-Free Components for F_1

Table 4: Dynamic Real-Time Scheduling for the Example

Task	Priority	π_i	$\tau_{max}(\Sigma_1)$	$\tau_{max}(\Sigma_2)$
T_1	1	100	26	48
T_2	2	110	68	68
Schedulable			Yes	No
Algorithms			RM, EDF[12]	

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

$$\Sigma_3 = \{v_{21}, v_{22}\} \tag{7}$$

DRTS for S: As shown in Table 4, when we used Σ_1 as the set of valid QSDS schedules for F_1 and applied the rate-monotonic scheduling algorithm to S, we found that though the total utilization (0.87818) is above the Liu and Layland's bound of $2(2^{1/2} - 1 = 0.828$, yet S is rate-monotonic schedulable. If instead of Σ_1, we used Σ_2 as the set of QSDS schedules for F_1, the system was not schedulable as the utilization is above 1. This example shows how the synthesis of an ERTS depends on both QSDS and DRTS.

Code Generation for S: After performing QSDS for each TFCPN and DRTS for the full system, embedded real-time software code is generated for the system S. Applying our code generation algorithm (Table 2), the generated code for *task*$_{11}$() of F_1 is shown in Table 5. Since there is only one source transition in F_1, there is only one task in the process for this TFCPN. In the case of F_2, there are two source transitions with independent firing rates, hence there are two tasks, namely *task*$_{21}$ and *task*$_{22}$, the codes of which are given in Tables 6 and 7. Thus, in total there are three concurrent tasks in the two process code for system S. It must be noted that calling the transitions in the code, in fact, represents a sequence of computations as modeled by the transition. The *main*() program is generated according to the DRTS schedules. A non-preemptive version is given in Table 8. Preemption can be added.

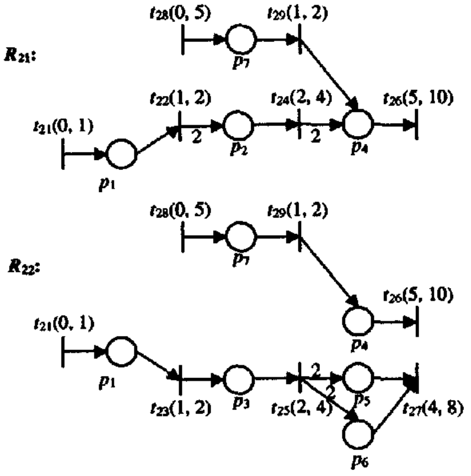

Figure 4: Conflict-Free Components for F_2

5. CONCLUSION

The formal automatic synthesis of *Embedded Real-Time Software* (ERTS) was proposed through an algorithm along with code generation that minimizes the number of tasks in a concurrent system. Two phases of scheduling, namely *Quasi-Static Data Scheduling* (QSDS) and *Dynamic Real-Time Scheduling* (DRTS), clearly distinguish between data and time scheduling, which respectively tries to satisfy the limited memory and processor requirements of an embedded system. When an ERTS is transferred to a faster processor, as long as preemptive scheduling is used, there is no need of re-scheduling the software. When an ERTS is installed in two embedded systems with different memory space sizes but same computation power then we need only perform QSDS twice. The same holds for DRTS in the case of different processing power.

6. REFERENCES

[1] K. Altisen, G. Gobler, A. Pneuli, J. Sifakis, S. Tripakis, and

Table 5: ERTS Code for *task*$_{11}$

```
task$_{11}$() {    // $d_{11} = \langle t_{11}, t_{12}, t_{14}, t_{13}, t_{15} \rangle$
   call $t_{11}$;
   switch($p_1$) {
      case $t_{12}$:   call $t_{12}$; count($p_2$) += 1;
                   if(count($p_2$) $\geq$ 2)
                      { call $t_{14}$; count($p_2$)−= 2;}
                   break;
      case $t_{13}$:   call $t_{13}$; count($p_3$) += 2;
                   while($count(p_3) \geq 1$)
                      { call $t_{15}$; count($p_3$) −= 1;}
                   break; }
}
```

Table 6: ERTS Code for *task*$_{21}$

```
task$_{21}$() {    // $d_{21} = \langle t_{21}, t_{22}, t_{24}, t_{26}, t_{23}, t_{25}, t_{27} \rangle$
   call $t_{21}$;
   switch($p_1$) {
      case $t_{22}$:   call $t_{22}$;   count($p_2$) += 2;
                   while(count($p_2$) $\geq$ 1) {
                     count($p_4$) += 2; call $t_{24}$; count($p_2$) −= 1;}
                   while(count($p_4$) $\geq$ 1) {
                      call $t_{26}$;   count($p_4$) −= 1; }
                   break;
      case $t_{23}$:   call $t_{23}$; count($p_3$) += 1; count($p_3$) −= 1;
                   call $t_{25}$; count($p_5$) += 2; count($p_6$) += 2;
                   while(count($p_5$) $\geq$ 1 $\wedge$ count($p_6$) $\geq$ 1) {
                    call $t_{27}$; count($p_5$) −= 1; count($p_6$) −= 1;}
                   break;}
}
```

Table 7: ERTS Code for *task*$_{22}$

```
task$_{22}$() {    // $d_{22} = \langle t_{28}, t_{29}, t_{26} \rangle$
   call $t_{28}$; call $t_{29}$; call $t_{26}$;
}
```

Table 8: ERTS Code for *main()*

```
main() {
   $k_{11} = k_{21} = k_{22} = 0$; // iteration numbers
   while true {
      if(now() − $\alpha_{11} \geq k_{11} \times 100$) { task$_{11}$(); $k_{11}$ + +; }
      if(now() − $\alpha_{21} \geq k_{21} \times 110$) { task$_{21}$(); $k_{21}$ + +; }
      if(now() − $\alpha_{22} \geq k_{22} \times 110$) { task$_{22}$(); $k_{22}$ + +; } }
}
```

now(): current time; $\alpha_{11}, \alpha_{21}, \alpha_{22}$: release offset time

S. Yovine. A framework for scheduler synthesis. In *Real-Time System Symposium (RTSS'99)*. IEEE Computer Society Press, 1999.

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

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

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

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

[6] 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.

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

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

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

[10] P.-A. Hsiung. POSE: A parallel object-oriented synthesis environment. *ACM Transactions on Design Automation of Electronic Systems*, 6(1):to appear, January 2001.

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

[12] C. Liu and J. Laylang. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[13] 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.

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

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

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

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