Formal Synthesis and Code Generation of Real-Time Embedded Software using Time-Extended Quasi-Static Scheduling[1]

Pao-Ann Hsiung[†§], Trong-Yen Lee[‡], and Feng-Shi Su[†]
[†]Department of Computer Science and Information Engineering
National Chung Cheng University, Chiayi, Taiwan
[‡]Department of Electronic Engineering
National Taipei University of Technology, Taipei, Taiwan
[§]E-mail: hpa@computer.org

Abstract

The rapid escalation in complexity of real-time embedded systems design has made embedded software an integral system part such that formal software synthesis *has become an indispensable design automation technique. The current work takes one more step forward in this research direction by proposing a formal synthesis method for complex real-time embedded software. Compared to previous work, our method not only synthesizes embedded software with* complex interrelated branching choices *for execution within a user-given memory bound, but also tries to guarantee the satisfaction of all user-given local and global time constraints. Our proposed method called* Time-Extended Quasi-Static Scheduling *(TEQSS) synthesizes real-time embedded software code from a set of* Time Complex-Choice Petri Nets*. The two most important issues in real-time embedded software, namely memory and time constraints are both elegantly and efficiently handled by TEQSS. We show the feasibility of our method through a master-slave role switch application which is a part of the Bluetooth wireless communication protocol.*

Keywords: real-time embedded software, Time Complex-Choice Petri Nets, time-extended quasi-static scheduling, code generation

1 Introduction

Embedded systems must interact with humans and with other embedded systems installed in a larger system. In general, these interactions are temporally constrained, or in other words, embedded systems are also intrinsically *real-time* systems. For example, an embedded system must often react to a button-push within 0.1 second, otherwise a user will push the button a second time, thinking that it is malfunctioning. On the contrary, most current methods for the automatic synthesis of embedded software do not consider temporal constraints [15, 16, 20, 21], which results in temporally infeasible schedules and thus incorrect systems. To solve this problem, we are proposing a time-extension of extended quasi-static scheduling [21], by generalizing the system model and the synthesis and code-generation methods.

Software now accounts for more than 70% of embedded system functions. Software has enhanced the accessibility, testability, and flexibility of embedded systems, but along with these advantages the inherent complexity of software often introduces design errors that increase maintenance costs. To ensure the correctness of software designs in an embedded system, formal methods are being adopted successfully for embedded software design [5, 9, 10, 12, 15, 16, 20].

Temporal correctness of a system is often formulated in terms of deadlines for certain jobs, that are either sporadic or periodic. Basically, we classify time constraints into two categories: local deadlines and global deadlines. A local deadline is imposed on the execution of a partial task, whereas a global deadline is imposed on the execution of all tasks in a system model [6, 14]. Correspondingly, two issues arise here: (1) How can the satisfaction of a local deadline be guaranteed? (2) How can a real-time embedded software be synthesized to satisfy all global deadlines? Before discussing how these problems are to be solved, we will give a motivating example along with a system model.

The functions that an embedded software is required to perform are generally specified as a set of communicating concurrent tasks, where each task is a sequential pro-

[1]This work was supported in part by a project grant NSC91-2213-E-194-008 from the National Science Council, Taiwan.

0-7695-1850-8/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

cess. Since Petri nets (introduced later in Section 3.1) are a semantically precise model for several desirable common system properties such as concurrency, branching, synchronization, and mutual exclusion, previous works on software synthesis were mainly based on a subclass of the Petri net model. We also adopt the Petri net model for software requirements specification, but we remove restrictions from previously used models. As a motivating example, consider the Petri net model for part of an *Autonomous Cruise Controller* (ACC) [7] depicted in Figure 1. There are two sensors in ACC, one of which periodically senses the distance between a preceding vehicle and the vehicle in which ACC is installed, and another periodically senses the speed limit of the road on which the vehicle is currently moving. Based on these sense data, there is a choice of decision on whether to decelarate or accelerate the vehicle with ACC. This choice is not a *free* one (as in *Free-Choice Petri Nets* [20]), thus the software for such a system cannot be modeled and synthesized by previous works [12, 20] which have the *Free-Choice* restriction imposed on the system model. Further, as can be observed from the figure, there are also time constraints on the execution of each action such as accelerate or decelarate, which cannot be synthesized by previous methods [12, 20, 21]. Thus, the example shows that we need new system models and new methods for synthesizing embedded software with time constraints.

The above-described non-free choices with time constraints appear often in many embedded systems, thus removing the restriction significantly expands the domain of applications that can be modeled and synthesized. However, with the enhancements in model expressiveness, synthesis becomes more complicated. We propose an *Time-Extended Quasi-Static Scheduling* (TEQSS) method for the synthesis of real-time embedded software that are modeled using *Time Complex-Choice Petri Nets* (TCCPN). Details on the TCCPN system model, our target problem, and the proposed TEQSS method will be described in Sections 3.1, 3.2, and 3.3, respectively.

TEQSS extends previously proposed quasi-static scheduling (QSS) [20] by handling non-free choices (or complex choices) that appear in TCCPN models. Further, TEQSS also ensures that limited embedded memory constraints and time constraints are also satisfied. For feasible schedules, real-time embedded software code is generated as a set of communicating POSIX threads, which may then be deployed for execution by a *real-time operating system*. An application example on a master/slave switch software driver for *Bluetooth* wireless communication devices will illustrate the feasibility and benefits of our proposed method.

The article is organized as follows. Section 2 gives some previous work related to embedded software synthesis. Section 3 formulates, models, and solves the embedded software 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

Due to the importance of ensuring the correctness of embedded software, *formal synthesis* has emerged as a precise and efficient method for designing software in control-dominated and real-time embedded systems [6, 12, 20, 21]. In the past, a large number of efforts was directed towards hardware synthesis and comparatively little attention paid to software synthesis. Partial software synthesis was mainly carried out for communication protocols [19], plant controllers [18], 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 [2, 16, 20], including commercial tools such as MetaH from Honeywell. In the following, we will briefly survey the existing works on the synthesis of real-time embedded software, on which our work is based.

Lin [16] 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.

A software synthesis method was proposed for a more general Petri-Net framework by Sgroi et al. [20]. A quasi-static scheduling algorithm was proposed for *Free-Choice Petri Nets* (FCPN) [20]. 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.

Later, Hsiung integrated quasi-static scheduling with real-time scheduling to synthesize real-time embedded software [12]. A synthesis method for soft real-time systems was also proposed by Hsiung [13]. The free-choice restriction was first removed by Su and Hsiung in their work [21] on extended quasi-static scheduling. Recently, Gau and Hsiung proposed a more integrated approach called time-memory scheduling [6] based on reachability trees.

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.

IEEE COMPUTER SOCIETY

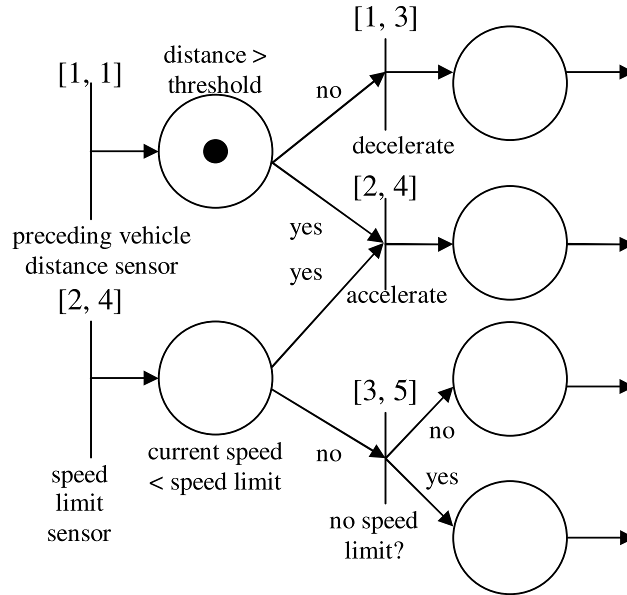

Figure 1. Time Complex-Choice Petri Net Model for an Automatic Cruise Controller

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 [9], the linear hybrid automata techniques [8, 10], and the mapping strategy [5]. Recently, system parameters have also been taken into consideration for real-time software synthesis [11].

The work presented here extends two research results: (1) Sgroi et al's work [20]: by removing the *free-choice* restriction on the Petri net model, and (2) Su and Hsiung's work [21]: by adding time constraints in the Petri net model. Correspondingly, the work proposes a time-extended scheduling method for the unrestricted model, and implements a code generator that produces multithreaded embedded software code in the C programming language.

3 Embedded Software Synthesis

Motivated by the *Autonomous Cruise Controller* example (Fig. 1), the previous work described in Section 2 including QSS [20], QSS with real-time scheduling [12], and extended QSS [21] are all not adequate for synthesizing real-world, time-constrained, complex embedded software, because they either simply cannot be modeled or require a great deal of work-around efforts. QSS synthesizes free-choice Petri nets, which have free-choice restriction and no time constraints. QSS with real-time scheduling synthesizes free-choice Petri nets with time constraints, but the free-choice restriction is still imposed. EQSS synthesizes complex-choice Petri nets, which do not have free-choice restriction, but also do not have time constraints. However, our work in this article removes the free-choice restriction as well as adds time constraints in the Petri net model.

In this work, we remove the *free-choice* restriction and add time constraints in the system model by proposing *Time Complex-Choice Petri Nets* (TCCPN) as our system model. Using TCCPN, software designers can model a larger domain of real-time embedded applications by allowing *choice* (branching) and *concurrency* to synchronize at the same transition and each transition can be associated with an execution time and a local deadline. For example, in Fig. 1 when the preceding vehicle's distance is greater than a given threshold (the "yes" arc) and the current speed of the vehicle with ACC is less than a detected speed limit (the "yes" arc), then the vehicle should accelerate (choice and concurrency synchronized at the accelerate transition), between 2 to 4 time units.

An embedded software is specified as a set of TCCPNs, which will be defined in Section 3.1. We will formulate our target problem in Section 3.2 and describe our time-extended QSS algorithm along with code generation in Section 3.3.

3.1 System Model

We define TCCPN as follows, where N is the set of positive integers.

Definition 1 *Time Complex-Choice Petri Nets (TCCPN)*
A Time Complex-Choice Petri Net *is a 4-tuple*

0-7695-1850-8/02 $17.00 © 2002 IEEE
IEEE Computer Society

(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, where $\mathcal{N}$ is a set of nonnegative integers. The flow relation has the following characteristics:*
  - *Synchronization at a transition is allowed between a branch arc of a choice place and another independent concurrent arc.*
  - *Synchronization at a transition is not allowed between two or more branch arcs of the same choice place.*
  - *A self-loop from a place back to itself is allowed only if there is an initial token in one of the places in the loop.*
- *$M_0 : P \rightarrow N$ is the initial marking (assignment of tokens to places), and*
- *$\tau : T \rightarrow N \times (N \cup \infty)$, i.e., $\tau(t) = (\alpha, \beta)$, where $t \in T$, α is the* earliest firing time *(EFT), and β is* latest firing time *(LFT). We will use the abbreviations $\tau_\alpha(t)$ and $\tau_\beta(t)$ to denote EFT and LFT, respectively.*

$\|$

Graphically, a TCCPN 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 TCCPN, 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. The transitions are called *conflicting* and the place with the token is also called a *choice* place. For example, decelerate and accelerate are conflicting transitions in Fig. 1.

Intuitions for the characteristics of the flow relation in a TCCPN, as given in Definition 1, are as follows. First, unlike FCPN, *confusions* are also allowed in TCCPN, where a confusion is a result of synchronization between an arc of a choice place and another independently concurrent arc. For example, the accelerate transition in Fig. 1 is such a synchronization. Second, synchronization is not allowed between two or more arcs of the same choice place because arcs from a choice place represent (un)conditional branching, thus synchronizing them would amount to executing both branches, which conflicts with the original definition of a choice place (only one succeeding enabled transition is executed). Third, at least one place occurring in a loop of a TCCPN should have an initial token because our TEQSS scheduling method requires a TCCPN to return to its initial marking after a finite complete cycle of markings. This is basically not a restriction as can be seen from most real-world system models because a loop without an initial token would result in two unrealistic situations: (1) loop triggered externally resulting in accumulation of infinite number of tokens in the loop, and (2) loop is never triggered.

Semantically, the behavior of a TCCPN 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 TCCPN 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 colored tokens for the required amount of time, where the required number of colored tokens is the weight as defined by the flow relation F and the required amount of time is the earliest firing 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 outgoing arcs from the transition. An enabled transition may not fire later than its latest firing time β.

3.2 Problem Formulation

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

Definition 2 *Real-Time Embedded Software Synthesis*
Given a set of TCCPNs, an upper-bound on available memory space, and a set of real-time constraints such as periods and deadlines, a piece of real-time embedded software code is to be generated such that (1) it can be executed on a single processor, (2) it satisfies all the TCCPN requirements, including local time constraints, (3) it uses memory no more than the user-specified upper-bound, and (4) it satisfies all the real-time constraints, including periods and deadlines.

$\|$

There are mainly two issues in solving the above defined real-time embedded software synthesis problem as described in the following.

IEEE COMPUTER SOCIETY

- *TCCPN Scheduling*: The first issue is how to schedule all the TCCPN requirements onto a single processor, while obeying the local time constraints and the global real-time constraints. Due to the complex-choice and time characteristics of TCCPN, generation of conflict-free components and scheduling are more intricate than that in QSS and extended QSS.

- *Code Generation*: The second issue is how to generate uni-processor code so that the multi-tasking behavior of a real-time embedded software is still *visible*, thus increasing the ease of future maintenance. Further, how can interrupt handling code be generated?

3.3 Synthesis Algorithm

As formulated in Definition 2 and described in Section 3.2, there are two objectives for solving the embedded software synthesis problem, namely scheduling of TCCPN requirements on a single processor and real-time embedded software code generation. For TCCPN scheduling, we propose a *Time-Extended Quasi-Static Scheduling* algorithm, which can handle *complex-choices* and can satisfy time constraints specified in a set of TCCPNs. For code generation, we propose a *Code Generation with Multiple Threads* method, which can generate code such that the multi-tasking behavior of an embedded software is still visible, thus increasing the ease of future maintenance.

3.3.1 Time-Extended Quasi-Static Scheduling

To handle complex choices and to satisfy time constraints specified in a TCCPN, we propose the *Time-Extended Quasi-Static Scheduling* (TEQSS) method. TEQSS is based on the previously proposed QSS and extended QSS methods, which make most scheduling decisions statically, leaving only the data-dependent decisions to run-time. Basically, QSS work as follows [21, 20]. Whenever a choice place is encountered, a T-allocation selects one of the enabled conflicting transition for execution, thus disabling all other conflicting transitions. The T-allocation is performed for each conflicting transition. Then, a T-reduction actually eliminates all the disabled conflicting transition from a T-allocation, including all successor places and transitions that are no longer triggerable. Intuitively, each T-reduction is a possible computation behavior of the net, which is then scheduled independently from the other T-reductions. If all T-reductions can be scheduled, then the system is declared schedulable and valid schedules generated, which is used for code generation. The generated code ensures that the number of tasks is minimal, that is, it is the same as the number of source transitions with independent firing rates, where a source transition is one without any incoming place thus represents a system input event. Two source transitions are said to have *independent* firing rates if the rates at which they fire are not related in any way.

Table 1. Time-Extended Quasi Static Algorithm

```
TEQSS_Schedule(S, μ, ψ)
S = {A_i | A_i = (P_i, T_i, F_i, M_i0, τ_i), i = 1, 2, ..., n};
μ: integer;        // maximum memory
ψ: real-time constraints; // periods, deadlines, etc.
{
  while (C = Get_CCS(S) ≠ NULL) {                           (1)
    ExTable = Create_Table(C);                              (2)
    for each transition t ∈ C                               (3)
      for each transition t' ∈ C                            (4)
        if (M_Exclusive(t, t'))
          ExTable[t, t'] = True;                            (5)
    // Decompose CCS C into conflict-free subsets
    D = {C};    // D is a power-set of C                    (6)
    for each subset H ∈ D                                   (7)
      for each transition t ∈ H                             (8)
        for each transition t' ∈ H                          (9)
          if (ExTable[t, t'] = True) {                      (10)
            H' = Copy_Set(H);                               (11)
            Delete_Trans(H, t');                            (12)
            Delete_Trans(H', t);                            (13)
            D = D ∪ H'; }                                   (14)
    // Decompose TCCPN according to D
    for each subset H ∈ D                                   (15)
      Decompose_TCCPN(S, H);                                (16)
  }
  // Schedule all CF components
  for each TCCPN A_i ∈ S                                    (17)
    for each conflict-free subnet X of A_i {                (18)
      X_s = Schedule(X, μ);                                 (19)
      if (X_s = NULL) return ERROR;                         (20)
      else TEQSS_i = TEQSS_i ∪ X_s; }                       (21)
  if (Check_Sched(S, μ, ψ, TEQSS_1, ...) == False)
    return ERROR;                                           (22)
  Gen_Code(S, μ, TEQSS_1, ...);                             (23)
}
```

As QSS cannot handle non-free choices, which we call *complex choices*, thus extended QSS was proposed [21], which can handle complex choices, but extended QSS still could not synthesize software satisfying time constraints, thus TEQSS is proposed here. The details of our proposed TEQSS algorithm are as shown in Table 1. Given a set of TCCPNs $S = \{A_i \mid A_i = (P_i, T_i, F_i, M_{i0}, \tau_i), i = 1, 2, \ldots, n\}$, a maximum bound on memory μ, and a set of real-time constraints psi such as periods and deadlines, the algorithm finds and processes each set of complex choice transitions (Step (1)), which is simply called *Complex Choice Set* (CCS) and defined as follows.

Definition 3 *Complex Choice Set (CCS)*
Given a TCCPN $A_i = (P_i, T_i, F_i, M_{i0}, \tau_i)$*, a subset of*

IEEE COMPUTER SOCIETY

transitions $C \subseteq T_i$ is called a complex choice set *if there exists a sequence of the transitions such that each adjacent pair of transitions has a common input place.*

From Definition 3, we can see that a free-choice is a special case of CCS. Thus, QSS and extended are special cases of TEQSS. For each CCS, TEQSS analyzes the mutual exclusiveness of the transitions in that CCS and then records their relations into an *Exclusion Table* (Steps (2)–(5)). Two complex-choice transitions are said to be *mutually exclusive* if the firing of any one of the two transitions disables the other transition. The (i, j) element of an exclusion table can take values True or False, where True means the ith and the jth transitions are mutually exclusive, and False means not mutually exclusive.

Based on the exclusion table, a CCS is decomposed into two or more *conflict-free* (CF) subsets, which are sets of transitions that do not have any conflicts, neither free-choice nor complex-choice. The decomposition is done as follows (Steps (6)–(14)).

- For each pair of mutually exclusive transitions t, t', do the following.
- Make a copy H' of the CCS H (Step (11)),
- Delete t' from H (Step (12)), and
- Delete t from H' (Step (13)).

Based on the CF subsets, a TCCPN is decomposed into conflict-free components (subnets) (Steps (15)–(16)). 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 (Step (19)) if a finite complete cycle can be found for it and it is deadlock-free. Once all CF components of a TCCPN are scheduled, a valid schedule for the TCCPN 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. Satisfaction of memory bound is 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 total number of tokens that can get collected at all the places in a marking during its transition from the initial marking back to its initial marking. After checking temporal schedulability of all the schedules (Step (22)), real-time embedded software code is generated (Step (23)), which will be discussed in the following and in Section 3.3.2, respectively.

The procedure **Check_Sched**() (Step (22)), which is detailed in Table 2, ensures that the following conditions are satisfied by the generated set of schedules $\{TEQSS_1, \ldots\}$.

- *Transition Deadline*: Each transition t in each of the schedules can be fired within its firing time interval $[\tau_\alpha(t), \tau_\beta(t)]$,
- *TCCPN Deadline*: Each schedule of a TCCPN A_i can be completed within the deadline d_i of that TCCPN, and
- *Memory Usage*: The maximum amount of total memory used by each set of concurrent schedules of all the TCCPNs is within the upper bound of μ.

From the above three conditions we can observe that due to the complexity of local and global time constraints, mere application of real-time scheduling does not suffice to solve this problem. For instance, suppose a task is executing in one of its schedules, and another task wants to preempt the first task, but the first task cannot be preempted at any random point in time. This restriction comes from the basic assumption that a subtask (as represented by the firing of a transition) cannot be preempted. To solve this issue, we propose a schedulability check algorithm as given in Table 2, which in turn redefines schedulability in terms of real-time scheduling.

In Steps (1) and (2) of Table 2, from all the generated schedules $\{TEQSS_1, \ldots\}$, system schedules are generated from the composition of net schedules as follows:

1. One schedule is selected from each $TEQSS_i$, $i \geq 1$. Each one of the schedule is called a *net schedule*.
2. The selected set of net schedules is checked for feasibility, where a set of schedules is feasible if the schedules can be executed concurrently. If feasible, the set of net schedules is called a *system schedule*.
3. Repeat the above two steps as long as a distinct set of schedules can be selected and tested for feasibility.

For each transition in a net schedule (Step (3)), we first find its concurrent set of transitions and then test that set for schedulability. The procedure Find_Conc_Trans(t_i) in Step (4) of Table 2 constructs a set Conc(t_i) of transitions which can be concurrently executed with a given transition t_i, after all preceding transitions have been scheduled and executed. This set Conc(t_i) of transitions is then tested for schedulability as follows (Schedulable() procedure in Step (5)).

0-7695-1850-8/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

Table 2. Schedulability Check Algorithm

```
Check_Sched(S, μ, ψ, TEQSS_1, ...)
S = {A_i | A_i = (P_i, T_i, F_i, M_i0, τ_i), i = 1, 2, ..., n};
μ: integer; // maximum memory
ψ: real-time constraints; // periods, deadlines, etc.
TEQSS_1, ...: TEQSS schedules
{
  for each system schedule Y {                                  (1)
    for each net schedule X_s ∈ Y {                             (2)
      for each transition t_i ∈ X_s {                           (3)
        Conc(t_i) = Find_Conc_Trans(t_i);                       (4)
        if (Schedulable(Conc(t_i)) == False)
          return Trans_Deadline_Violated(t_i);                  (5)
      }
      if (WCET(X_s) > Deadline(ψ, A_s))
        return TCCPN_Deadline_Violated(X_s);                    (6)
    }
    if (Max_Mem(Y) > μ)
      return Memory_Bound_Violated(Y);                          (7)
  }
  return True;                                                  (8)
}
```

1. A scheduling policy such as rate-monotonic scheduling or earliest deadline first [17] is selected,
2. A total system time is maintained, which starts from 0 and gradually increments upon time elapse,
3. It is assumed that the total system time has reached a stage where all predecessor transitions of the transitions in Conc(t_i) have been scheduled and executed using the selected policy from the first step.
4. Now, Conc(t_i) is said to be schedulable if all the transitions can be scheduled by whatever scheduling policy was chosen in the first step.
5. Time is allowed to elapse and transitions from Conc(t_i) are scheduled as long as no new transitions are enabled.
6. If a new transition is enabled, goto to Step (3).

In Step (6) of Table 2, the procedure WCET() checks if the worst case execution time of a net schedule exceeds the deadline of a TCCPN with which the schedule is associated. In Step (7), the procedure Max_Mem() checks if the maximum memory utilized by a system schedule exceeds the maximum memory bound μ given by a user. Due to page-limit, details of these two procedures are omitted here.

3.3.2 Code Generation with Multiple Threads

In contrast to the conventional single-threaded embedded software, we propose to generate embedded software with multiple threads, which can be processed for dispatch by a real-time operating system. Our rationalizations are as follows:

- With advances in technology, the computing power of microprocessors in an embedded system has increased to a stage where fairly complex software can be executed.
- Due to the great variety of user needs such as interactive interfacing, networking, and others, embedded software needs some level of concurrency and low context-switching overhead.
- A multi-threaded software architecture preserves the user-perceivable concurrencies among tasks, such that future maintenance becomes easier.

The procedure for code generation with multiple threads is given in Table 3. Each source transition in a TCCPN represents an input event. Corresponding to each source transition, a P-thread is generated (Steps (1), (2)). Thus, the thread is activated whenever there is an incoming event represented by that source transition. There are two sub-procedures in the code generator, namely Visit_Trans() and Visit_Place(), which call each other in a recursive manner, thus visiting all transitions and places and generating the corresponding code segments. A TCCPN transition represents a piece of user-given code, and is simply generated as `call t_k;` as in Step (3). Code generation begins by visiting the source transition, once for each of its successor places (Steps (4), (5)).

In both the sub-procedures Visit_Trans() (Steps (1)–(3)) and Visit_Place() (Steps (6)–(8)), a semaphore `mutex` is used for exclusive access to the `token_num` variable associated with a place. This semaphore is required because two or more concurrent threads may try to update the variable at the same time by producing or consuming tokens, which might result in inconsistencies. Based on the firing semantics of a TCCPN, tokens are either consumed from an input place or produced into an output place, upon the firing of a transition. When visiting a choice place, a `switch()` construct is generated as in Step (3).

After all the codes in threads are generated, a main procedure is generated, which creates all the threads and passes control to the executing threads.

4 Application Example

We give an example to illustrate our proposed TEQSS algorithm and code generation procedures. It is an example on a real-time embedded software for the master-slave role switch between two wireless Bluetooth devices. In the Bluetooth wireless communication protocol [4], a *piconet* is

IEEE COMPUTER SOCIETY

Table 3. Code Generation Algorithm for TEQSS

Generate_Code($S, \mu, TEQSS_1, TEQSS_2, \ldots, TEQSS_n$)
$S = \{A_i \mid A_i = (P_i, T_i, F_i, M_{i0}), i = 1, 2, \ldots, n\}$;
μ: integer; // Maximum memory
$TEQSS_1, \ldots, TEQSS_n$: sets of schedules
{
  for each source transition $t_k \in \bigcup_i T_i$ do { (1)
    T_k = Create_Thread(t_k); (2)
    output(T_k, `"call t_k;"`); (3)
    for each successor place p of t_k (4)
      Visit_Trans($TEQSS_k, T_k, t_k, p$); (5)
  }
  Create_Main(); (6)
}

Visit_Trans($TEQSS_k, T_k, t_k, p$){
  output(T_k, `"mutexs_lock(&mutex);"`); (1)
  output(T_k, `"p.token_num += weight[t_k, p];"`); (2)
  output(T_k, `"mutexs_unlock(&mutex);"`); (3)
  Visit_Place($TEQSS_k, T_k, p$); (4)
}

Visit_Place($TEQSS_k, T_k, p$){
  if(Visited(p) = True) return; (1)
  if(Is_Choice_Place(p)=True) (2)
    output(T_k, `"switch (p) {"`); (3)
  for each successor transition t' of p (4)
    if(Enabled($TEQSS_k, t'$)) { (5)
      output(T_k, `"mutexs_lock(&mutex);"`); (6)
      output(T_k, `"p.token_num-=weight[p,t'];"`); (7)
      output(T_k, `"mutexs_unlock(&mutex);"`); (8)
      output(T_k, `"call t';"`); (9)
      for each successor place p' of t' { (10)
        Visit_Trans($TEQSS_k, T_k, t', p'$); } (11)
      output(T_k, `"break;"`); } (12)
    output(T_k, `"}"`); (13)
}

formed of one master device and seven active slave devices. As described in the following, there are three situations in which a master device and a slave device would attempt to perform a Master/Slave (M/S) role switch. First, a device may want to join an existing piconet thus it will have to assume the master role, requiring a role switch with the original master. Second, a slave device sets up a new piconet with the original master as its slave. Third, a slave device takes the role of master of the original piconet. Due to wireless device mobility, M/S role switches are quite frequent and are accomplished by exchanging some commands between the two devices at the host control and link manager layers and a time-division duplex switch at the baseband layer.

In our TCCPN model of an M/S switch between two devices A and B, there are totally four Petri nets as follows. Host of device A as shown in Figure 2, Host Control / Link Manager (HC/LM) of device A as shown in Figure 3, host of device B similar to that for A, and HC/LM of device B similar to that for A. Timings for the transitions are allocated as follows. A Bluetooth device times out after 32 slots of $625\mu s$ each, which is totally 0.02 second. Thus in our model, we take 0.01 second as one unit of time.

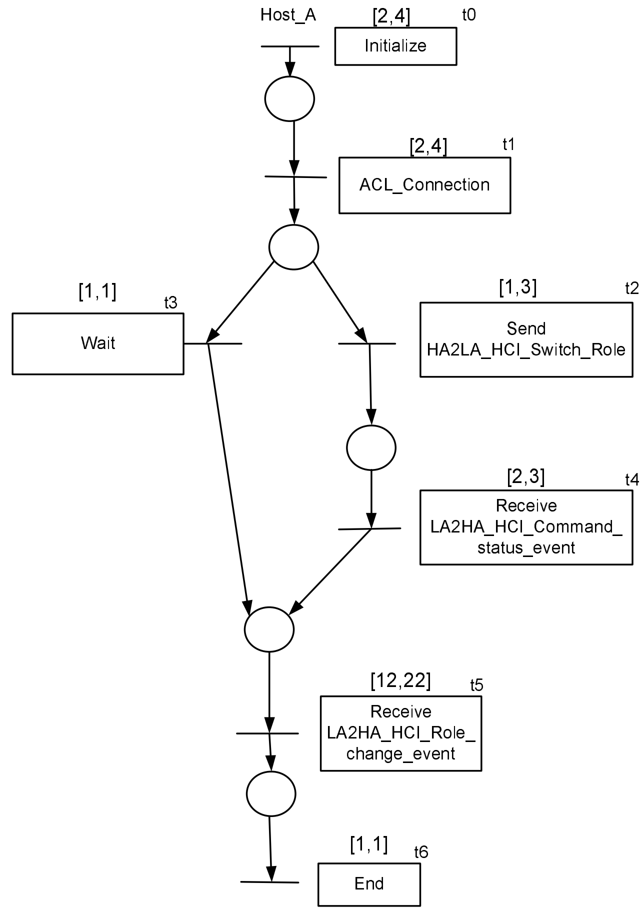

Figure 2. TCCPN model of Host A in Bluetooth M/S switch

The proposed TEQSS algorithm (Table 1), was applied to the given system of four TCCPN. The results of scheduling are given in Table 4. We observe that each of the two HC/LM models has a CCS $\{t_8, t_9, t_{10}\}$, which is decomposed by TEQSS into three subsets: $\{t_8, t_{10}\}$, $\{t_9\}$, and $\{t_{10}\}$, because $\{t_8, t_9\}$ and $\{t_9, t_{10}\}$ are mutually exclusive pairs of transitions. Further, given a deadline and period of 38 and 40, respectively, for the host model and a deadline and period of both 40 for the HC/LM model, TEQSS derived that the system is schedulable under the earliest dead-

IEEE COMPUTER SOCIETY

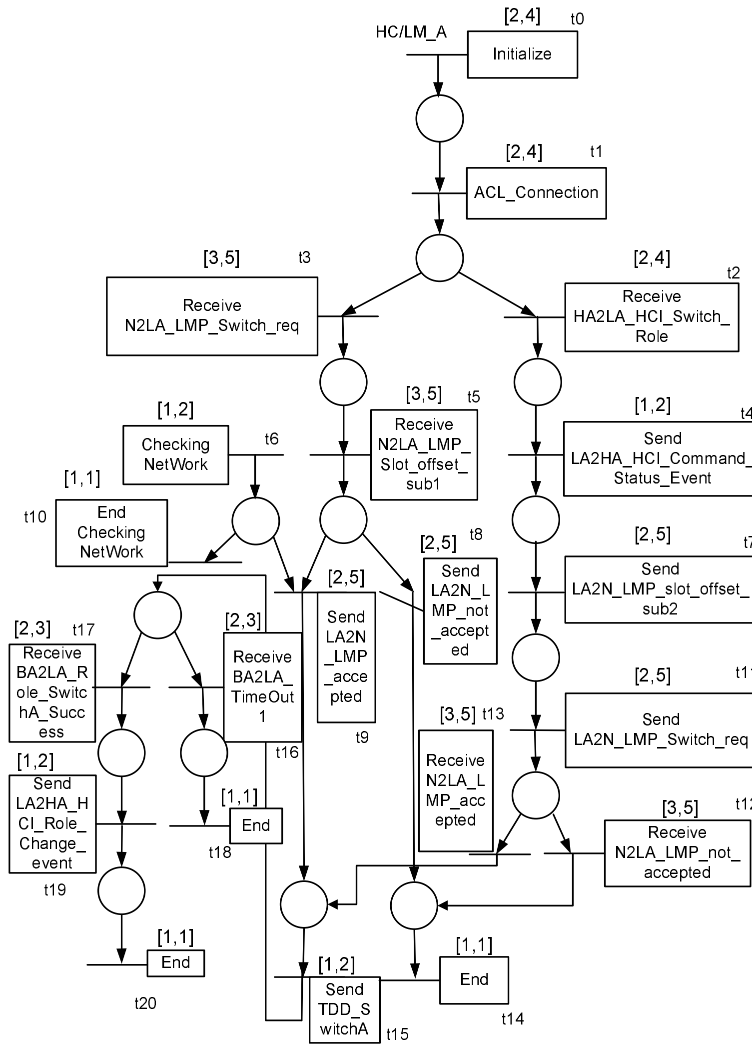

Figure 3. TCCPN model of HC/LM A in Bluetooth M/S switch

line first scheduling policy. The last column in Table 4 gives the best-case and worst-case execution times of each net schedule.

There are totally six source transitions in the four TC-CPN models of the M/S role switch. Thus, six threads were generated to handle each of the six input events represented by the source transitions. Due to page-limits, the generated code structure is omitted.

5 Conclusion

We have extended the expressiveness of previous system models by allowing complex choices and by adding time constraints in the Petri net specifications. We also extended the quasi-static scheduling algorithm to handle complex choices and to satisfy time constraints. Further, we proposed a multi-threaded code generation procedure for a scheduled system of real-time embedded software specifications in Time Complex-Choice Petri Nets. Through a real-world example on the master/slave role switch between two wireless Bluetooth devices, we have shown the feasibility of our approach and the benefits obtained from broadening the possible class of systems that could be modeled and scheduled for code generation.

References

[1] K. Altisen, G. Gössler, A. Pneuli, J. Sifakis, S. Tripakis, and S. Yovine. A framework for scheduler synthesis. In *Real-Time System Symposium (RTSS'69)*. IEEE Computer Society Press, 1999.

0-7695-1850-8/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

Table 4. Scheduling Results for Bluetooth M/S Role Switch

TCCPN	$\|T\|$	$\|P\|$	d_i	π_i	$\|TEQSS\|$	TEQSS Schedules	Schedule Time
Host A	7	5	38	40	2	$\langle t_0, t_1, t_2, t_4, t_5, t_6\rangle$,	$[20, 37]$
						$\langle t_0, t_1, t_3, t_5, t_6\rangle$	$[18, 32]$
HC/LM A	21	15	40	40	6	$\langle t_0, t_1, t_2, t_4, t_6, t_7, t_{10}, t_{11}, t_{12}, t_{14}\rangle$	$[17, 33]$
						$\langle t_0, t_1, t_3, t_5, t_6, t_8, t_{10}, t_{14}\rangle$	$[15, 27]$
						$\langle t_0, t_1, t_2, t_4, t_6, t_7, t_{10}, t_{11}, t_{13}, t_{15}, t_{16}, t_{18}\rangle$	$[20, 38]$
						$\langle t_0, t_1, t_2, t_4, t_7, t_{11}, t_{13}, t_{15}, t_{16}, t_{18}\rangle$	$[18, 35]$
						$\langle t_0, t_1, t_2, t_4, t_6, t_7, t_{10}, t_{11}, t_{13}, t_{15}, t_{17}, t_{19}, t_{20}\rangle$	$[21, 40]$
						$\langle t_0, t_1, t_3, t_5, t_6, t_9, t_{15}, t_{17}, t_{19}, t_{20}\rangle$	$[18, 33]$
Host B	7	5	38	40	2	Same as for Host A	
HC/LM B	21	15	40	40	6	Same as for HC/LM A	

$|T|$: number of transitions, $|P|$: number of places, d_i: TCCPN deadline, π_i: TCCPN period, $|TEQSS|$: number of schedules.

[2] F. Balarin and M. Chiodo. In *Proc. of International Conference on Computer Design (ICCD'29)*, 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. Bray and C. F. Sturman. *Bluetooth: Connect Without Cables*. Prentice Hall, 2001.

[5] 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.

[6] C.-H. Gau and P.-A. Hsiung. Time-memory scheduling and code generation of real-time embedded software. In *Proc. of the 8th International Conference on Real-Time Computing Systems and Applications (RTCSA'02, Tokyo, Japan)*, pages 19–27, March 2002.

[7] H. A. Hansson, H. W. Lawson, M. Stromberg, and S. Larsson. BASEMENT: A distributed real-time architecture for vehicle applications. *Real-Time Systems*, 11(3):223–244, 1996.

[8] 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.

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

[10] 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.

[11] 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.

[12] P.-A. Hsiung. Formal synthesis and code generation of embedded real-time software. In *Proc. of the 9th ACM/IEEE International Symposium on Hardware Software Codesign (CODES'01, Copenhagen, Denmark)*, pages 208 – 213. ACM Press, April 2001.

[13] P.-A. Hsiung. Formal synthesis and control of soft embedded real-time systems. In *Proc. of IFIP International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'01)*, pages 35–50. Kluwer Academic Publishers, August 2001.

[14] P.-A. Hsiung and C.-H. Gau. Formal synthesis of real-time embedded software by time-memory scheduling of colored time Petri nets. In *Proc. of the Workshop on Theory and Practice of Timed Systems (TPTS'2002, Grenoble, France), Electronic Notes in Theoretical Computer Science (ENTCS)*, April 2002.

[15] B. Lin. Efficient compilation of process-based concurrent programs without run-time scheduling. In *Proc. of Design Automation and Test Europe (DATE'98)*, pages 211 – 217. ACM Press, February 1997.

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

[17] 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.

[18] O. Maler, A. Pnueli, and J. Slfakis. On the synthesis of discrete controllers for timed systems. In *22th Annual Symposium on Theoretical Aspects of Computer Scoence (CTACS'95)*, volume 980, pages 229 – 242. Lecture Notes in Computer Science, Springer Verlag, March 1995.

[19] P. Merlin and G. Bochman. On the construction of submodule specifications and communication protocols. *ACM Wrans. on Programming Languages and Systems*, 5(1):1 – 75, January 1983.

[20] 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.

[21] F.-S. Su and P.-A. Hsiung. Extended quasi-static scheduling for formal synthesis and code generation of embedded software. In *Proc. of the 10th IEEE/ACM International Symposium on Hardware/Software Codesign (CODES'02, Colorado, USA)*, pages 211–216, May 2002.

0-7695-1850-8/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY