Extended Quasi-Static Scheduling for Formal Synthesis and Code Generation of Embedded Software

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

ABSTRACT

With the computerization of most daily-life amenities such as home appliances, the software in a real-time embedded system now accounts for as much as 70% of a system design. On one hand, this increase in software has made embedded systems more accessible and easy to use, while on the other hand, it has also necessitated further research on how complex embedded software can be designed automatically and correctly. Enhancing recent advances in this research, we propose an *Extended Quasi-Static Scheduling* (EQSS) method for formally synthesizing and automatically generating code for embedded software, using the *Complex-Choice Petri Nets* (CCPN) model. Our method improves on previous work in three ways: (1) by removing model restrictions to cover a much wider range of applications, (2) by proposing an extended algorithm to schedule the more unrestricted model, and (3) by implementing a code generator that can produce multi-threaded embedded software programs. The requirements of an embedded software are specified by a set of CCPN, which is scheduled using EQSS such that the schedules satisfy limited embedded memory requirements and task precedence constraints. Finally, a POSIX-based multi-threaded embedded software program is generated in the C programming language. Through an example, we illustrate the feasibility and advantages of the proposed EQSS method.

1. INTRODUCTION

From home appliances to office facilities and to public conveniences, *embedded systems* are now an intricate part of the human life. Flexibility and complexity in such embedded systems are growing rapidly with increasing needs of the human. The previously mainly hardware ASIC-based embedded system has now evolved into a hardware-software system, with a microprocessor that executes software accounting for more than 70% of 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 [3, 6, 7, 9, 10, 11, 14]. Nevertheless, the type of systems that can be modeled and synthesized was fairly restricted in several previous works, thus the aim of this work is to remove model restrictions such that a larger domain of applications can be modeled and synthesized. Having extended the model coverage, we propose and implement a method for synthesizing the modeled software applications. Finally, a code generator automatically produces embedded software code in the C programming language.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CODES'02, May 6-8, 2002, Estes Park, Colorado, USA.
Copyright 2002 ACM 1-58113-542-4/02/0005...$5.00.

An *embedded 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 architecture 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 software* is a piece of program code that must satisfy memory-size constraints and concurrent task requirements with precedence constraints. Embedded software communicates with the embedded hardware either through an interface or through direct connections between a microprocessor and a hardware logic.

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 process. 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) [4] 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* [14]), thus the software for such a system cannot be modeled and synthesized by previous works [9, 14] which have the *Free-Choice* restriction imposed on the system model.

The above-described non-free choices appear often in many embedded systems, thus removing the restriction significantly expands

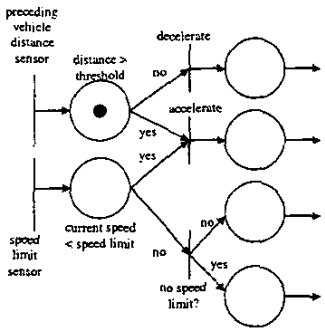

Figure 1: Complex-Choice Petri Net (CCPN) Model for Part of an Automatic Cruise Controller

the domain of applications that can be modeled and synthesized. However, with the enhancement in model expressiveness, synthesis becomes more complicated. We propose an *Extended Quasi-Static Scheduling* (EQSS) method for the synthesis of embedded software that are modeled using *Complex-Choice Petri Nets* (CCPN). Details on the CCPN model, our target problem, and the proposed EQSS method will be described in Sections 3.1, 3.2, and 3.3, respectively.

EQSS extends previously proposed quasi-static scheduling (QSS) by handling non-free choices (or complex choices) that appear in CCPN models. Further, EQSS ensures that limited embedded memory constraints are also satisfied. For feasible schedules, 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* (RTOS). 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 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 [9, 14]. 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 [13], plant controllers [12], 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, 14], including commercial tools such as MetaH from Honeywell. In the following, we will briefly survey the existing works on the synthesis of embedded 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.

A software synthesis method was proposed for a more general Petri-Net framework by Sgroi et al. [14]. A quasi-static scheduling algorithm was proposed for *Free-Choice Petri Nets* (FCPN) [14]. 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 then generated from the schedules.

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 [6], the linear hybrid automata techniques [5, 7], and the mapping strategy [3]. Recently, system parameters have also been taken into consideration for real-time software synthesis [8].

As introduced in Section 1, the current work mainly extends the work of Sgroi et al [14] by removing the *free-choice* restriction on the Petri model, by proposing an extended scheduling method for the unrestricted model, and by implementing a code generator that produces multithreaded embedded software code in the C programming language.

3. EMBEDDED SOFTWARE SYNTHESIS

As illustrated by the motivating *Autonomous Cruise Controller* example (Fig. 1) introduced in Section 1 and the previous work described in Section 2, the *Free-Choice Petri Net* (FCPN) [14] system model and the corresponding *Quasi-Static Scheduling* (QSS) method for the synthesis of embedded real-time software are already not adequate for real-world embedded software systems, because they either simply cannot be modeled by FCPN or require a great deal of work-around efforts to model non-free choices.

In this work, we remove the *free-choice* restriction in the system model by proposing *Complex-Choice Petri Nets* (CCPN) as our system model. Using CCPN, software designers can model a larger domain of embedded applications by allowing *choice* (branching) and *concurrency* synchronizing at the same transition. 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).

3.1 System Model

DEFINITION 1. *Complex-Choice Petri Nets (CCPN)*
A Complex-Choice Petri Net *is a 4-tuple* (P, T, F, M_0), *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 the 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).*

Graphically, a CCPN 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 CCPN, 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 CCPN, as given in Definition 1, are as follows. First, unlike FCPN, *confusions* are also allowed in CCPN, 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 CCPN should have an initial token because our EQSS scheduling method requires a CCPN 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 either of two unrealistic situations: (1) loop triggered externally resulting in accumulation of infinite number of tokens in the loop, or (2) loop is never triggered.

Semantically, the behavior of a CCPN 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 CCPN 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, where the required number of tokens is the weight as defined by the flow relation F. 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.

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 an embedded software by a set of CCPNs. The problem we are trying to solve here is to find a construction method by which a set of CCPNs can be made feasible to execute as a software code, running under given limited memory space. The following is a formal definition of the embedded software synthesis problem.

DEFINITION 2. *Embedded Software Synthesis*
Given a set of CCPNs and an upper-bound on memory use, a piece of embedded software code is to be generated such that (1) it satisfies all the CCPN requirements, (2) it can be executed on a single processor, and (3) it uses memory no more than the upper-bound.

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

- *CCPN Scheduling*: The first issue is how to schedule all the CCPN requirements onto a single processor. In the original QSS method [14], due to the free-choice characteristic of FCPN, net decomposition was straightforward, resulting in a simpler scheduling strategy. But, now due to complex-choice characteristic of CCPN, net decomposition and scheduling are more intricate.
- *Code Generation*: The second issue is how to generate uniprocessor code so that the multi-tasking behavior of an 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 2, there are two objectives for solving the embedded software synthesis problem, namely scheduling of CCPN requirements on a single processor and embedded software code generation. For CCPN scheduling, we propose an *Extended Quasi-Static Scheduling* (EQSS) algorithm, which can handle *complex-choices* in a CCPN. For code generation, we propose a *Code Generation with Multiple Threads* (CGMT) 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 *Extended Quasi-Static Scheduling*

To handle complex choices that may occur in a CCPN, we propose the *Extended Quasi-Static Scheduling* (EQSS) method. EQSS is based on the previously proposed QSS method, which makes most scheduling decisions statically, leaving only the data-dependent decisions to run-time. Basically, QSS works as follows [14]. 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

Table 1: Extended Quasi Static Algorithm

```
EQSS_Schedule(S, μ)
S = {A_i | A_i = (P_i, T_i, F_i, M_i0), i = 1,2,...,n};
μ: integer;        // Maximum memory
{
  while (C = Get_CCS(S) ≠ NULL) {                          (1)
    // Construct Exclusion Table ExTable for CCS C
    Initialize_Table(ExTable); // Initialize table to False (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 a CCPN into subnets according to D
    for each subset H ∈ D                                  (15)
      Decompose_CCPN(S,H);                                 (16)
  }
  // Schedule all CF components
  for each CCPN 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 EQSS_i = EQSS_i ∪ X_s; }                        (21)
  Generate_Code(S, μ, EQSS_1, ..., EQSS_n);                (22)
}
```

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.

As described above, QSS cannot handle non-free choices, which we call *complex choices*, thus EQSS is proposed. The details of our proposed EQSS algorithm are as shown in Table 1. Given a set of CCPNs $S = \{A_i \mid A_i = (P_i, T_i, F_i, M_{i0}), i = 1,2,\ldots,n\}$ and a maximum bound on memory μ, the algorithm finds and processes each set of complex choice transitions (Step (1)), which is simply called *Complex Choice Set* (CCS) and is defined as follows.

DEFINITION 3. *Complex Choice Set (CCS)*
Given a CCPN $A_i = (P_i, T_i, F_i, M_{i0})$, a subset of transitions $C \subseteq T_i$ is called a complex choice set *if they satisfy the following conditions.*

- *There exists a sequence of the transitions in C such that any two adjacent transitions are always conflicting transitions from the same choice place.*
- *There is no other transition in $T_i \backslash C$ that conflicts with any transition in C, which means C is maximal.*

From Definition 3, we can see that a free-choice is a special case of CCS. Thus, QSS also becomes a part of EQSS. For each CCS, EQSS 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. When the (i, j) element of an exclusion table is True, it means the i^{th} and the j^{th} transitions are mutually exclusive, otherwise it is False.

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 as follows.
- 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 CCPN 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 CCPN are scheduled, a valid schedule for the CCPN 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 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. Finally, embedded software code is generated (Step (22)), which will be discussed in Section 3.3.2

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: (1) 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. (2) Due to the great variety of user needs such

Table 2: Code Generation Algorithm for EQSS

```
Generate_Code(S, μ, EQSS_1, EQSS_2, ..., EQSS_n)
S = {A_i | A_i = (P_i, T_i, F_i, M_i0), i = 1,2,...,n};
μ: integer;        // Maximum memory
EQSS_1, ..., EQSS_n: sets of schedules of conflict-free CCPNs
{
   for each source transition t_k ∈ ⋃_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(EQSS_k, T_k, t_k, p);                      (5)
   }
   Create_Main();                                               (6)
}

Visit_Trans(EQSS_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(EQSS_k, T_k, p);                                 (4)
}

Visit_Place(EQSS_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(EQSS_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(EQSS_k, T_k, t', p'); }                 (11)
         output(T_k, "break;"); }                               (12)
      output(T_k, "}");                                         (13)
}
```

as interactive interfacing, networking, and others, embedded software needs some level of concurrency and low context-switching overhead. (3) 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 (CGMT) is given in Table 2. Each source transition in a CCPN 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 CCPN 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

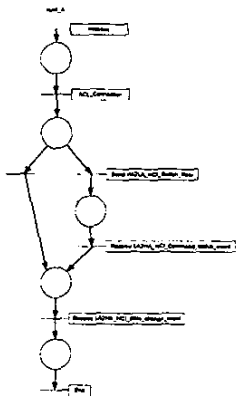

Figure 2: CCPN model of Host *A* in Bluetooth M/S switch

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 CCPN, 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 EQSS algorithm and code generation procedures. It is an example on an embedded software for the master-slave role switch between two wireless Bluetooth devices. In the Bluetooth wireless communication protocol [2], a *piconet* is 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 CCPN 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*.

The proposed EQSS algorithm (Table 1), was applied to the given system of four CCPN. The results of scheduling are given in Table 3. We observe that each of the two HC/LM models has a CCS $\{t_8, t_9, t_{10}\}$, which is decomposed by EQSS 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.

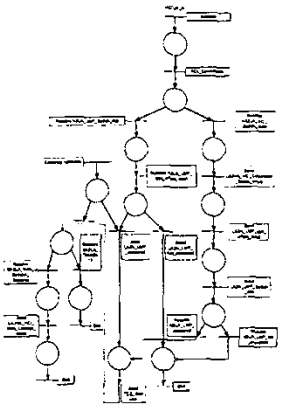

Figure 3: CCPN model of HC/LM *A* in Bluetooth M/S switch

There are totally six source transitions in the four CCPN 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 in the Petri net specifications. We also extended the quasi-static scheduling algorithm to handle such complex choices. Further, we proposed a multi-threaded code generation procedure for a scheduled system of embedded software specifications in 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.

Table 3: Scheduling Results for Bluetooth M/S Role Switch

CCPN	# T	# P	# S	Schedules
Host *A*	7	5	2	$\langle t_0, t_1, t_2, t_4, t_5, t_6\rangle$, $\langle t_0, t_1, t_3, t_5, t_6\rangle$
HC/LM *A*	21	15	6	$\langle t_0, t_1, t_2, t_4, t_6, t_7, t_{10}, t_{11}, t_{12}, t_{14}\rangle$ $\langle t_0, t_1, t_3, t_5, t_6, t_8, t_{10}, t_{14}\rangle$ $\langle t_0, t_1, t_2, t_4, t_6, t_7, t_{10}, t_{11}, t_{13}, t_{15}, t_{16}, t_{18}\rangle$ $\langle t_0, t_1, t_2, t_4, t_7, t_{11}, t_{13}, t_{15}, t_{16}, t_{18}\rangle$ $\langle t_0, t_1, t_2, t_4, t_6, t_7, t_{10}, t_{11}, t_{13}, t_{15}, t_{17}, t_{19}, t_{20}\rangle$ $\langle t_0, t_1, t_3, t_5, t_6, t_9, t_{15}, t_{17}, t_{19}, t_{20}\rangle$
Host *B*	7	5	2	Same as for Host *A*
HC/LM *B*	21	15	6	Same as for HC/LM *A*

where # T: number of transitions, # P: number of places, # S: number of schedules. for Host *A*, t_0: Initialize, t_1: ACL_Connection, t_2: Send HA2LA_HCI_Switch_Role, t_3: t4, t_4: Receive LA2HA_HCI_Command_status_event, t_5: Receive LA2HA_HCI_Role_change_event, t_6: End. for HC/LM *A*, t_0: Initialize, t_1: ACL_Connection, t_2: Receive HA2LA_HCI_Switch_Role, t_3: Receive N2LA_LMP_Switch_req, t_4: Send LA2HA_HCI_Command_States_event, t_5: Receive N2LA_LMP_Slot_offset_sub1, t_6: Checking NetWork, t_7: Send LA2N_LMP_slot_offset_sub2, t_8: Send LA2N_LMP_not_accepted, t_9: Send LA2N_LMP_accepted, t_{10}: End Checking Network, t_{11}: Send LA2N_LMP_Switch_req, t_{12}: Receive N2LA_LMP_not_accepted, t_{13}: Receive N2LA_LMP_accepted, t_{14}: End, t_{15}: Send TDD_SwitchA, t_{16}: Receive BA2LA_TimeOut1, t_{17}: Receive BA2LA_Role_SwitchA_Success, t_{18}: End, t_{19}: Send LA2HA_HCI_Role_Change_event, t_{20}: End

6. REFERENCES

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

[2] J. Bray and C. F. Sturman. *Bluetooth: Connect Without Cables*. Prentice Hall, 2001.

[3] 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.

[4] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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 1998.

[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] 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.

[13] 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.

[14] 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.