Synthesis of Real-Time Embedded Software with Local and Global Deadlines[1]

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

ABSTRACT

Current methods cannot synthesize real-time embedded software applications when the global deadline of a task is shorter than the total of all local deadlines along a critical path in the task. This creates unnecessary modeling limitations which directly affect the types of systems synthesizable. We propose a *quasi-dynamic scheduling* algorithm for simultaneously guaranteeing both local and global deadlines, while satisfying all precedence constraints among subtasks and among tasks. Through this scheduling procedure, we are able to formally synthesize real-time embedded software from a network of *Real-Time Petri Nets* specification. Application examples, including a driver for the Master/Slave role switch in Bluetooth wireless communication devices, are given to illustrate the feasibility of the scheduling algorithm.

Categories and Subject Descriptors

C.3 [**Special Purpose and Application-Based Systems**]: Real-time and embedded systems; D.2.2 [**Software Engineering**]: Design tools and techniques—*Petri nets*

General Terms

Design, Algorithms

Keywords

real-time embedded software, Real-Time Petri Nets, quasi-dynamic scheduling, software synthesis, code generation

1. INTRODUCTION

A real-time embedded system task is composed of some constituent subtasks, each of which has its own *local deadline*, while the task itself has a *global deadline*. Current scheduling algorithms do not explicitly consider such multilevel deadlines leading to the necessity for work-around efforts. We propose a scheduling algorithm to resolve this issue and show how it can be used for synthesizing real-time embedded software specifications into actual program code.

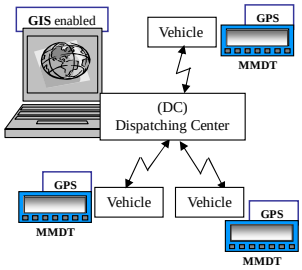

Figure 1: Modular Mobile Dispatching System

As a motivating example depicted in Fig. 1, consider the *Modular Mobile Dispatching System* (MMDS) [16], which consists of a GPS receiver, a GIS database, a GSM communication module, and other I/O peripherals for dispatching of vehicles through a call center. Besides the local deadlines on each GPS, GIS, and GSM task, there is also a global deadline on each scenario which is composed of several tasks with precedence and concurrency relationships. A typical scenario would be that of a vehicle driver encountering an emergency situation, in which the driver uses MMDS and expects to get help within 4 minutes from the time a call is made from the vehicle to the call center. Within this time span, MMDS must get GPS location information, transmit it to the call center through GSM communication, the call center must plot the driver's location on a digital map using GIS, locate the nearest help on the map, dispatch help (such as an ambulance) to the location by notifying the target help through GSM, while providing navigation guidelines through an active GIS database.

There are several issues involved in such a typical real-time scenario, as detailed in the following.

- How to determine which subtasks are concurrently enabled at any point of execution?
- How to check if each subtask completes execution within its local deadline, while satisfying all precedence constraints among the subtasks?
- How to check if each task completes execution within its global deadline?
- How to obtain an optimal schedule of all system tasks such that shortest execution time is guaranteed, if one exists?

[1]This work was supported by project grants NSC91-2213-E-194-008 and NSC92-2213-E-194-003 from the National Science Council, Taiwan.

CODES+ISSS'03, October 1–3, 2003, Newport Beach, California, USA.

- How to estimate the amount of memory space required for the execution of a real-time embedded software system?

Corresponding to each of the above issues, we propose a set of solutions in the form of a scheduling method called *Quasi-Dynamic Scheduling* (QDS), which incorporates the respective solutions as briefly described in the following.

- *Concurrently Enabled Group*: We maintain a group of concurrently enabled subtasks, while the system's behavior is statically simulated to satisfy all precedence relationships.
- *Tentative Schedulability Check*: Since the group of concurrently enabled subtasks changes dynamically with system execution, its schedulability can be checked only tentatively for the current group.
- *Global System Timer*: A global system timer is maintained that keeps count of the current total amount of processor time taken by the execution of all tasks.
- *Pruned Reachability Tree*: Because schedulability checks are only tentative for a group of subtasks, a reachability tree is created so that an optimal schedule can be found. The tree is pruned for efficiency without affecting optimality.
- *Maximum Memory Estimation*: Using various memory estimation techniques, both static and dynamic memory space allocations are statically counted, including memory spaces for both local and global variables.

Basically, quasi-dynamic scheduling is a combination of quasi-static scheduling and dynamic scheduling. Data dependent branch executions are statically decomposed into different behavior configurations and quasi-statically scheduled [17]. For each quasi-statically decomposed behavior configuration, dynamic scheduling is employed to satisfy all local deadlines of each subtask, all precedence constraints among subtasks, and all global deadlines of each task.

To illustrate the importance of this research result, consider how existing scheduling approaches must be applied to a system with both local and global deadlines. In this case, there is a need for work-around methods such as making global deadline the sum of all local deadlines in a critical path of the task. The user is burdened with the responsibility of analyzing a task and finding the critical path, a non-trivial task in some cases, apriori to scheduling. Further, this work-around method only works if the global deadline is not smaller than the sum of all local deadlines in a critical path of a task, because otherwise it would amount to restraining each local deadline, thus making an otherwise schedulable system unschedulable. In summary, the work presented here is not only a flexibility enhancement to current scheduling methods, but also a necessary effort in checking schedulability for real systems.

This article is organized as follows. In Section 2, we delve on some previous work in quasi-static scheduling and real-time scheduling related to the synthesis of real-time embedded software. In Section 3, we formulate our target problem to be solved, our system model, and give an illustrative example. In Section 4, we present our quasi-dynamic scheduling algorithm and how it is applied to the running example. Section 6 concludes the article giving some future work.

2. PREVIOUS WORK

Since our target is formally synthesizing real-time embedded software, we will only discuss scheduling algorithms that have been used for this purpose.

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 [11, 9, 17, 18]. Partial software synthesis was mainly carried out for communication protocols [15], plant controllers [14], 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, 13, 17], 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.

Previous methods for the automatic synthesis of embedded software mostly do not consider temporal constraints [13, 17, 18], which results in temporally infeasible schedules and thus incorrect systems. Some recently proposed methods [9, 12] explicitly take time into consideration while scheduling, but have not solved the multilevel deadlines issue. Details of each method are given in the rest of this section.

Lin [13] 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. [17]. A quasi-static scheduling (QSS) algorithm was proposed for *Free-Choice Petri Nets* (FCPN) [17]. 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 [9]. A synthesis method for soft real-time systems was also proposed by Hsiung [10]. The free-choice restriction was first removed by Su and Hsiung in their work [18] on extended quasi-static scheduling (EQSS). Recently, Gau and Hsiung proposed a more integrated approach called time-memory scheduling [11] based on reachability trees.

A recently proposed *timed quasi-static scheduling* (TQSS) method [12] extends two previous works: (1) the QSS [17] method by handling non-free choices (or complex choices) that appear in system models, and (2) the EQSS [18] by adding time constraints in the system model. Further, TQSS 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*.

Balarin et al. [2] proposed a software synthesis procedure for reactive embedded systems in the *Codesign Finite State Machine* (CFSM) framework with the POLIS hardware-software codesign tool. 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 [6], the linear hybrid automata techniques [5, 7], and the mapping strategy [4]. Recently, system parameters have also been taken into consideration for real-time software synthesis [8].

3. SOFTWARE SYNTHESIS

Our target is the formal synthesis of real-time embedded software, with local and global deadlines, using scheduling techniques. A system is specified as a set of concurrent tasks, where each task is composed of a set of subtasks, with precedence relationships. Time constraints are classified into two categories: *local* deadlines and *global* deadlines. A local deadline is imposed on the execution of a subtask, whereas a global deadline is imposed on the execution of a task in a system model [11].

Previous work on software synthesis were mainly based on a subclass of the Petri net model (introduced later in Section 3.1). We also adopt the Petri net model for software requirements specification, but we associate explicit semantics to the firing time intervals, which will be explained when the system model *Real-Time Petri Net* (RTPN) is defined. Just like *Time Complex-Choice Petri Nets* (TCCPN) used in [12], RTPN places no free-choice restriction on the model expressivity and adds timing constraints on each transition, which represents a subtask. Thus, a wider domain of applications can be precisely modeled by RTPN. Details on the RTPN system model, our target problem, and an illustrative example will be described in Sections 3.1, 3.2, and 3.3, respectively.

3.1 System Model

The requirements for real-time embedded software are modeled by a set of RTPNs. Graphically, an RTPN can be depicted as shown in Fig. 2, 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. A place with more than one outgoing transition is called a *choice* place and the transitions are said to be *conflicting*. For example, p_0 is a choice place and t_1 and t_2 are conflicting transitions in Fig. 2. We define RTPN as follows, where N is the set of positive integers.

DEFINITION 1. *Real-Time Petri Nets (RTPN)*
A Real-Time 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, $P \cup T \neq \emptyset$, $P \cap T = \emptyset$, and some of the transitions are* source *transitions, which fire periodically. $F : (P \times T) \cup (T \times P) \rightarrow N$ is a weighted flow relation between places and transitions, represented by arcs, such that: (a) Synchronization at a transition is allowed between a branch arc from a choice place and another arc, and (b) 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). $\tau : T \rightarrow N \times (N \cup \infty)$, i.e., $\tau(t) = (\alpha,\beta)$, where $t \in T$, α is the* worst case execution time *(WCET) of t, and β is the* deadline *for t, which will be denoted as $\tau_\alpha(t)$ and $\tau_\beta(t)$, respectively.* ‖

3.2 Problem Formulation

A user specifies the requirements for a real-time embedded software by a set of RTPNs. The problem we are trying to solve here is to find a construction method by which a set of RTPNs can be made feasible to execute on a single processor as a piece of software code, running under given finite 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 RTPNs, an upper-bound on available memory space, and a set of real-time constraints such as periods and deadlines for each RTPN, a piece of real-time embedded software code is to be generated such that: (a) it can be executed on a single processor, (b) it satisfies all the RTPN requirements, including precedence constraints and local deadlines, (c) it satisfies all global real-time constraints, including RTPN (task) periods and deadlines, and (d) it uses memory no more than the user-given upper-bound. ‖

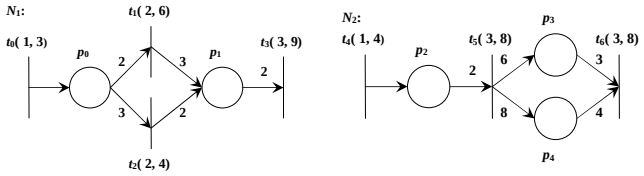

Figure 2: Illustration Example

As described in Section 1, there are five issues involved in solving this problem and the solutions to these issues are integrated into a quasi-dynamic scheduling method, which will be presented in Section 4. Due to page-limit, we leave out the code generation part of software synthesis, which has a multi-threaded architecture based on PThreads [18].

3.3 Illustration Example

This is a simple toy example to illustrate how our proposed scheduling method works. The RTPN model for this example is shown in Fig. 2, which consists of two nets $N_1 = (P_1,T_1,F_1,M_{01},\tau_1)$ and $N_2 = (P_2,T_2,F_2,M_{02},\tau_2)$, where $P_1 = \{p_0,p_1\}$, $P_2\{p_2,p_3,p_4\}$, $T_1 = \{t_0,t_1,t_2,t_3\}$, $T_2 = \{t_4,t_5,t_6\}$, the flow relations F_1, F_2, and the firing intervals τ_1, τ_2 are obvious from the numbers on the arcs and transitions, respectively. The initial markings M_{01}, M_{02} are all empty.

4. QUASI-DYNAMIC SCHEDULING

To solve the several issues raised in Section 1 for synthesizing real-time embedded software, a *Quasi-Dynamic Scheduling* (QDS) method is proposed. QDS employs both quasi-static and dynamic scheduling techniques. Details of the QDS algorithm are presented in Tables 1, 2, 3. Rather than going into the details of each step of the algorithms, we present the main ideas as follows.

- Data dependent branch executions are statically decomposed into different behavior configurations and quasi-statically scheduled using EQSS [17, 18]. (Step 1 of Table 1)
- For each quasi-statically decomposed behavior configuration, dynamic scheduling is employed to satisfy the local deadline of each subtask, all precedence constraints among subtasks, and the global deadline of each task as follows.
  - A *set G of concurrently enabled transitions* is used to represent the state of the system (corresponding to an RTPN marking in the model). A global system clock (*STime*) and a global memory usage (*SMem*) are used to record the absolute global time and memory for each group, respectively.
  - To find a feasible optimal schedule, a reachability tree is constructed in a depth-first search manner (Step 15 of Table 2), where each node represents a marking with a corresponding set of concurrently enabled transitions and each edge represents the firing of a selected transition. Exhaustive construction of the tree is avoided by pruning it under appropriate conditions as described in the following. It can be proved that optimality of the feasible solution is not affected by such pruning conditions.

Table 1: Quasi Dynamic Scheduling

```
QDS(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: global real-time constraints; // periods, deadlines, etc.
{
    m = EQSS(S, μ, H); // m = |H|, H: EQSS schedules [18]      (1)
    for(j = 0; j < m; j++) {                                   (2)
        G = initial_group(H, j);                               (3)
        if(schedule_tree(H, G, S, ψ, μ)) output(H, j);         (4)
        else return Unschedulable_Error;                       (5)
    }
}
```

Table 2: Schedule Tree Construction in QDS

```
schedule_tree(H, G, S, ψ, μ)
H: set of EQSS schedules;
G: group of concurrently enabled transitions;
S: set of RTPN;
ψ: global real-time constraints; // periods, deadlines, etc.
μ: integer;        // maximum memory
{
    if(choose_schedulable(G, G') == False) return False;       (1)
    for each transition t ∈ G' {                               (3)
        STime = t → exec + G → STime;                          (4)
        if (STime > deadline(ψ)) continue;                     (5)
        SMem = t → mem + G → SMem;                             (6)
        if (SMem > μ) continue;                                (7)
        G'' = copy(G);                                         (8)
        G'' → STime = STime; G'' → SMem = SMem;                (9)
        fire_trans(t);                                         (10)
        if (last_firing(t)) G'' = G''\{t};                     (11)
        for each transition t' ∈ successor(t, S)               (12)
            G'' = G'' ∪ {t'};   // add newly enabled transitions (13)
        if(G'' == NULL) return True;   // end of schedule      (14)
        if(schedule_tree(H, G'', S, ψ, μ)) return True;        (15)
    }
    return False;                                              (16)
}
```

  - ∗ *Negative Laxity*: There is not enough time left for at least one of the enabled transitions to execute until completion. (Steps 4, 5 of Table 3)
  - ∗ *Local Deadline Violation Forecast*: After a simulation-based analysis of the group of enabled transitions, if it is found that none of the transitions can be executed last in the group, then that group of transitions is not schedulable. (Steps 6–11 of Table 3)
  - ∗ *Global Deadline Violation*: The system clock has exceeded the global deadline of at least one of the RTPN. (Steps 4, 5 of Table 2)
  - ∗ *Memory Bound Violation*: The memory usage has exceeded a user-given upper bound. (Steps 6, 7 of Table 2)
- For each node in the tree, not all successor nodes are generated. Some nodes are not generated under various conditions as described in the following. (Steps 12–24 of Table 3)
  - ∗ If there is at most only one urgent transition, with execution time ($\tau_\alpha(t)$) same as its remaining time ($\rho(t)$) (i.e., $\tau_\alpha(t) = \rho(t)$: zero laxity), then only one successor node is generated.
  - ∗ All transitions whose execution can be deferred such that even if they are the last ones to execute among the currently enabled transitions, they will still satisfy their respective deadlines, then their corresponding nodes are not generated. This heuristic is applied provided there is some node to be generated.

Some advantageous features of QDS are as follows.

- *No need of system-wide WCET analysis*: After quasi-dynamic scheduling, we have total execution time for each system schedule, which may be smaller than the sum of worst-case execution times of all the transitions in that schedule.
- *Optimal schedules*: QDS always generates a set of optimal schedules because all feasible schedules are explored using the reachability tree. Due to page limits, the optimality of generated schedules and the time/space complexity of the QDS algorithm are given without proofs in this Section.
- *Efficient scheduling*: QDS uses several different heuristics to avoid searching exhaustively in the solution space and these heuristics are proven to be helpful, but harmless, that is, they do not eliminate any optimal schedule.
- *Multi-objective optimizations*: Since both time and memory constraints are considered during scheduling, QDS allows a user to easily optimize the resulting schedules in terms of either shortest schedule time or smallest memory usage. Trade-offs are inevitable between these two objectives, and QDS leaves such trade-off analysis to the user.
- *All issues solved*: All the issues presented in Section 1 are solved by QDS.

The following lemma and theorem show the correctness of the QDS algorithm.

LEMMA 1. *RTPN transitions that are eliminated from successor node generation by the* **choose_schedulable** *algorithm given in Table 3 cannot be in a feasible schedule.*

THEOREM 1. *The schedule generated by the QDS algorithm given in Table 1 is optimal in terms of shortest execution time.*

Instead of the exponential complexity in terms of the total number of transitions in a system configuration, the QDS algorithm induces a multiplicative factor reduction through the careful selection of successor nodes to generate (Table 3). Details are omitted.

Limitations of QDS are as follows.

- *Transition parameters*: Worst case execution time of each transition and local deadlines must be user given or derived from some analysis of the software code represented by a transition.
- *Interrupt handling*: QDS must be extended to handle interrupts. This part of the work is still ongoing and the basic idea is to include the set of allowable interrupts to the parameters of each transition and to consider the worst-case of interrupts arriving during the execution of each transition.
- *Periods and deadlines*: Currently, in QDS it is assumed that all RTPN have the same periods and deadlines. This restriction can be easily removed by scheduling a time slot that spans the least common multiple of all periods.

To illustrate how QDS works, we use the running illustration example given in Fig. 2. First of all, EQSS is applied to the two

Table 3: Selection of Schedulable Transitions in QDS

```
choose_schedulable(G, G')
G: set of concurrently enabled transitions
G': empty set of transitions
{
    G' = copy(G); G'_old = NULL;                                   (1)
    while(True) {                                                  (2)
        G_last = G_tmp = NULL;                                     (3)
        for each transition t ∈ G' {                               (4)
            if(t → remain < t → exec) return False;                (5)
            Gtime += t → exec; }                                   (6)
        for each transition t ∈ G' {                               (7)
            if(t → remain >= Gtime) G_last = G_last ∪ {t};         (8)
            else G' = G'\{t};     }                                (9)
        if (empty(G_last)) {                                       (10)
            if empty(G'_old) return False;                         (11)
            else return True; }                                    (12)
        else if (empty(G')) {                                      (13)
            G' = G_last; return True; }                            (14)
        else {                                                     (15)
            Gtime = 0;                                             (15)
            for each transition t ∈ G' Gtime += t → exec;           (16)
            for each transition t ∈ G_last {                       (17)
                Gtime' = Gtime + t → exec;                         (18)
                for each transition t' ∈ G'                        (19)
                    if (t' → remain >= Gtime') {                   (20)
                        G_tmp = G_tmp ∪ {t}; break; }              (21)
            }
            G' = G' ∪ G_tmp;                                       (22)
            if (equal(G', G'_old)) return True;                    (23)
            G'_old = G';     } // end else                         (24)
    } // end of while
}
```

RTPN. The resulting conflict-free components and corresponding schedule for each of those components are given in Fig. 3. There are totally three such components: R_{11} and R_{12} for N_1 and R_{21} for N_2. But, the EQSS schedule for each component has some degree of choices in the repeated firings, for example in the schedule for R_{11}, $\langle t_0^4, t_1^2, t_3^3\rangle$, it can also be scheduled as $\langle t_0^2, t_1, t_3^2, t_0^2, t_1, t_3\rangle$, where the exponents represent number of firings. QDS explores this degree of choices for satisfying the local deadlines and global deadlines of each system configuration, where a system configuration is a combination of one conflict-free component from each RTPN. Thus, there are totally two system configurations for this example, namely $\{R_{11}, R_{21}\}$ and $\{R_{12}, R_{21}\}$.

On applying QDS to this example, we found that it is indeed schedulable and satisfies all local and global deadlines. Though there are two reachability trees for the two system configurations, we present only one of them for illustration. The reachability tree for $\{R_{12}, R_{21}\}$ is presented in a tabular form in Table 4. The first column is the index of the nodes in the tree and the last column gives the child nodes of the corresponding node from the first column. G is the group of concurrently enabled transitions in the marking represented by that node. α is the execution time of each transition. ρ is the time left before the deadline β is reached and it is calculated from the time the transition was enabled. *STime* and *SMem* are the current global records of system time and memory, respectively. $G' \subseteq G$ is the subset transitions that are chosen in the generation of successor nodes. The 8th column consists of the actual transitions that are fired and thus also gives the schedule that is generated by QDS. At the end of Table 4, it is found that the system configuration is schedulable. The total time and memory used for the schedule $\langle t_0^3, t_4, t_2, t_4, t_3, t_5, t_6^2\rangle$ are 19 time units and 14 memory units, respectively. Similarly, when QDS is applied to the other system configuration $\{R_{11}, R_{21}\}$, it is schedulable and the total time and memory used are 28 time units and 18 memory units, respectively, for the schedule $\langle t_0^3, t_4, t_0, t_1, t_4, t_1, t_3, t_5, (t_3, t_6)^2\rangle$.

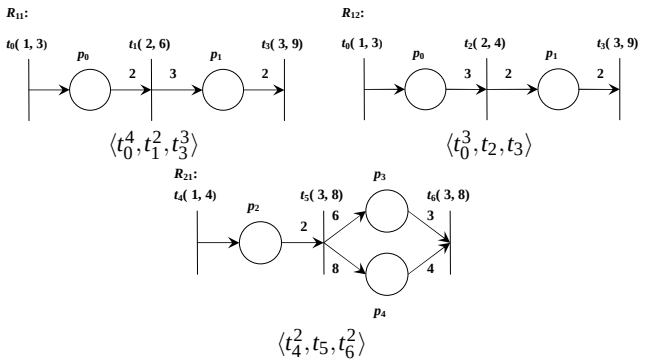

Figure 3: EQSS schedules for Illustration Example

Table 4: QDS scheduling for R_{12} and R_{21}

n	G	α	ρ	*STime*	*SMem*	G'	fire	next
0	t_0	1	3	0	0	Y	t_0	1
	t_4	1	4			Y		
1	t_0	1	3	1	1	Y	t_0	2
	t_4	1	3			Y		
2	t_0	1	3	2	2	Y	t_0	3
	t_4	1	2			Y		
3	t_2	2	4	3	3	N		
	t_4	1	1			Y	t_4	4
4	t_2	2	3	4	4	Y	t_2	5
	t_4	1	4			Y		
5	t_3	3	9	6	3	N		
	t_4	1	2			Y	t_4	6
6	t_3	3	8	7	4	Y	t_3	7
	t_5	3	8			Y		
7	t_5	3	5	10	2	Y	t_5	8
8	t_6	3	8	13	14	Y	t_6	9
9	t_6	3	8	16	7	Y	t_6	Found!
Time & Memory				19	14	$\langle t_0^3, t_4, t_2, t_4, t_3, t_5, t_6^2\rangle$		

n: node, ρ: time left before deadline, Y: Yes, $\in G'$, N: No, $\notin G'$

5. APPLICATION EXAMPLE

The QDS method for software synthesis was applied to several real-world applications such as ATM virtual private network scheduling, Bluetooth wireless communication protocol, motor speed control system, and medic-care system. For purpose of illustration, we describe one of the examples, which is a real-time embedded software driver for the master-slave role switch between two wireless Bluetooth devices. In the Bluetooth wireless communication protocol [3], a *piconet* is formed of one master device and seven active slave devices. There are three situations in which a master device and a slave device would attempt to perform a Master/Slave (M/S) role switch. 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.

In our RTPN model of an M/S switch between two devices *A* and *B*, there are totally four Petri nets [12]: 2 host devices and 2 Host Control / Link Manager (HC/LM) models. Timings for the transitions are allocated as follows. A Bluetooth device times out after 32 slots of 625μs each, which is totally 0.02 second. Thus in our model, we take 0.01 second as one unit of time.

The proposed QDS algorithm (Table 1), was applied to the given

Table 5: EQSS Schedules for Bluetooth M/S Role Switch

RTPN	H: EQSS Schedules	Time
Host A,B	$A_{11} = \langle t_0,t_1,t_2,t_4,t_5,t_6\rangle$,	[20,41]
	$A_{12} = \langle t_0,t_1,t_2,t_4,t_7\rangle$	[8,40]
	$A_{13} = \langle t_0,t_1,t_3,t_5,t_6\rangle$	[18,34]
	$A_{14} = \langle t_0,t_1,t_3,t_7\rangle$	[6,33]
HC/LM A,B	$A_{21} = \langle t_0,t_1,t_2,t_4,t_6,t_7,t_{10},t_{11},t_{12},t_{14}\rangle$	[17,35]
	$A_{22} = \langle t_0,t_1,t_3,t_5,t_6,t_8,t_{10},t_{14}\rangle$	[15,29]
	$A_{23} = \langle t_0,t_1,t_2,t_4,t_6,t_7,t_{10},t_{11},t_{13},t_{15},t_{16},t_{18}\rangle$	[20,40]
	$A_{24} = \langle t_0,t_1,t_2,t_4,t_7,t_{11},t_{13},t_{15},t_{16},t_{18}\rangle$	[18,37]
	$A_{25} = \langle t_0,t_1,t_2,t_4,t_6,t_7,t_{10},t_{11},t_{13},t_{15},t_{17},t_{19},t_{20}\rangle$	[21,42]
	$A_{26} = \langle t_0,t_1,t_3,t_5,t_6,t_9,t_{15},t_{17},t_{19},t_{20}\rangle$	[18,35]

system of four RTPN. First, EQSS is applied. The results of EQSS scheduling are given in Table 5. The last column in Table 5 gives the best-case and worst-case execution times of each net EQSS schedule. Further, reachability trees were constructed for all the 24 different configurations. All deadlines and periods are given as 45 time units. For illustration purpose, the application QDS to one of the configurations $\{A_{11},A_{25}\}$ is given partially in Table 6, which has a schedule time of 41 time units and memory usage of 2 memory units for the schedule $\langle t_{2,6},t_{2,10},t_{1,0},t_{2,0},t_{1,1},t_{2,1},t_{1,2},t_{2,2},t_{2,4},t_{1,4},t_{2,7},t_{2,11},t_{2,13},t_{2,15},t_{2,17},t_{2,19},t_{2,20},t_{1,5},t_{1,6}\rangle$. It is finally derived that the system is schedulable.

6. CONCLUSION

No more workarounds are needed when both local and global deadlines are to be satisfied because quasi-dynamic scheduling (QDS) has solved this problem in the context of real-time embedded software synthesis. QDS has integrated static and dynamic scheduling to efficiently derive an optimal schedule time or memory based on some simple heuristics. Application examples show that we can avoid the worst case analysis when QDS can used for scheduling. Through a real-world example on the master/slave role switch between two wireless Bluetooth devices, we have shown the feasibility of our approach. In the future, we plan to extend QDS in several ways: to handle dissimilar periods and deadlines, to handle interrupts during scheduling, and to estimate transition parameters.

7. 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'99)*. IEEE Computer Society Press, 1999.

Table 6: QDS scheduling for A_{11} and A_{25} (partial)

n	G	α	ρ	*STime*	*SMem*	G'	fire	next
0	$t_{1,0}$	2	4	0	0	N		
	$t_{2,0}$	2	6			N		
	$t_{2,6}$	1	2			Y	$t_{2,6}$	1
1	$t_{1,0}$	2	3	1	1	N		
	$t_{2,0}$	2	5			N		
	$t_{2,10}$	1	1			Y	$t_{2,10}$	2
2	$t_{1,0}$	2	2	2	0	Y	$t_{1,0}$	3
	$t_{2,0}$	2	4			N		
... (intermediate executions omitted)								
15	$t_{2,19}$	1	2	26	2	Y	$t_{2,19}$	16
	$t_{1,5}$	12	14			N		
16	$t_{1,5}$	12	13	27	2	N		
	$t_{2,20}$	1	1			Y	$t_{2,20}$	17
17	$t_{1,5}$	12	12	28	1	Y	$t_{1,5}$	18
18	$t_{1,6}$	1	1	40	1	Y	$t_{1,6}$	Found!
Time & Memory				41	2	$\langle t_{2,6},t_{2,10},\ldots,t_{1,5},t_{1,6}\rangle$		

n: node, ρ: time left before deadline, Y: Yes, $\in G'$, N: No, $\notin G'$

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

[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. 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] 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.

[11] 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.

[12] P.-A. Hsiung, T.-Y. Lee, and F.-S. Su. Formal synthesis and code generation of real-time embedded software using timed quasi-static scheduling. In *Proc. of the 9th Asia-Pacific Software Engineering Conference (APSEC)*, pages 395–404. IEEE CS Press, December 2002.

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

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

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

[16] W.-B. See, P.-A. Hsiung, T.-Y. Lee, and S.-J. Chen. Modular mobile dispatching system (MMDS) and logistics. In *Proc. of the 2002 Annual Conference on National Defense Integrated Logistics Support (ILS)*, pages 365–371, August 2002.

[17] 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.

[18] 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. ACM Press, May 2002.