Synthesis of Real-Time Embedded Software by Timed Quasi-Static Scheduling[1]

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

Abstract

A formal synthesis method for complex real-time embedded software is proposed in this work. 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 local and global time constraints. Our proposed method called* Timed Quasi-Static Scheduling *(TQSS) 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 in TQSS. We show the feasibility of our method through a master-slave role switch application which is a part of the Bluetooth wireless communication protocol.*

1 Introduction

Current methods for the automatic synthesis of embedded software mostly do not consider temporal constraints [10, 11, 14, 15], 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 [15], by generalizing the system model and the synthesis and code-generation methods based on formal synthesis techniques [7, 10, 11, 14].

Time constraints are classified 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 [5, 9]. 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?

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

Previous works 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 remove restrictions from previously used models. As a motivating example, consider the Petri net model for part of an *Autonomous Cruise Controller* (ACC) [6] 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 [7, 14] 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 [7, 14, 15]. 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 *Timed Quasi-Static Scheduling* (TQSS) 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 TQSS method will be described in Sections 3.1, 3.2, and 3.3, respectively.

TQSS extends previously proposed quasi-static scheduling (QSS) [14] by handling non-free choices (or complex choices) that appear in TCCPN models. Further, TQSS also ensures that limited embedded memory constraints and time constraints are also satisfied. For feasible schedules, real-

Proceedings of the 16th International Conference on VLSI Design (VLSI'03)
1063-9667/03 $17.00 © 2003 IEEE

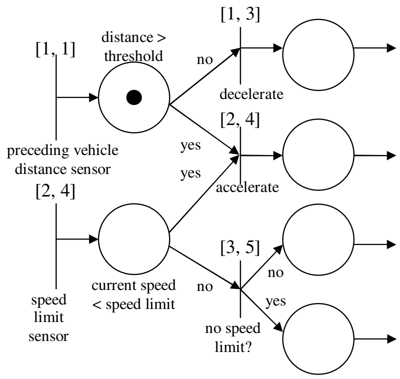

Figure 1. Time Complex-Choice Petri Net Model for an Automatic Cruise Controller

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 [5, 7, 14, 15]. 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 [2, 11, 14], 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 [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 finally generated from the valid schedule.

Later, Hsiung integrated quasi-static scheduling with real-time scheduling to synthesize real-time embedded software [7]. A synthesis method for soft real-time systems was also proposed by Hsiung [8]. The free-choice restriction was first removed by Su and Hsiung in their work [15] on extended quasi-static scheduling. Recently, Gau and Hsiung proposed a more integrated approach called time-memory scheduling [5] 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.

The work presented here extends two research results: (1) Sgroi et al's work [14]: by removing the *free-choice* restriction on the Petri net model, and (2) Su and Hsiung's work [15]: 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 [14], QSS with real-time scheduling [7], and extended QSS [15] 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 $\mathcal{N}$ is the set of positive integers.

Definition 1 *Time Complex-Choice Petri Nets (TCCPN)*
A Time Complex-Choice Petri Net *is a 5-tuple* (P, T, F, M_0, τ)*, where:*

- P *is a finite set of places,*
- T *is a finite set of transitions,* $P \cup T \neq \emptyset$, $P \cap T = \emptyset$,
- $F : (P \times T) \cup (T \times P) \to \mathcal{N}$ *is a weighted flow relation between places and transitions, represented by arcs, with the following characteristics: (1) Synchronization at a transition is allowed between a branch arc of a choice place and another independent concurrent arc, (2) Synchronization at a transition is not allowed between two or more branch arcs of the same choice place, and (3) 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.*
- $M_0 : P \to N$ *is the initial marking (assignment of tokens to places), and*
- $\tau : T \to N \times (N \cup \infty)$*, 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. A place with more than one outgoing transition is called a *choice* place and the transitions are said to be *conflicting*. For example, decelerate and accelerate are conflicting transitions in Fig. 1.

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 real-time constraints, including periods and deadlines. ‖

There are mainly two issues in solving the above defined problem, namely *TCCPN scheduling* and *code generation*. 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. 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

For TCCPN scheduling, we propose a *Timed 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 Timed Quasi-Static Scheduling

To handle complex choices and to satisfy time constraints specified in a TCCPN, we propose the *Timed Quasi-Static Scheduling* (TQSS) method. TQSS is based on the previously proposed QSS [14] and extended QSS [15] methods, which make most scheduling decisions statically, leaving only the data-dependent decisions to run-time.

As QSS cannot handle non-free choices, which we call *complex choices*, thus extended QSS was proposed [15],

Table 1. Timed Quasi Static Scheduling Algorithm

```
TQSS_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 TQSS_i = TQSS_i ∪ X_s; }                      (21)
  if (Check_Sched(S, μ, ψ, TQSS_1, ...) == False)
    return ERROR;                                        (22)
  Gen_Code(S, μ, TQSS_1, ...);                           (23)
}
```

which can handle complex choices, but extended QSS still could not synthesize software satisfying time constraints, thus TQSS is proposed here. The details of our proposed TQSS 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 ψ 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 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 QSS are special cases of TQSS. For each CCS, TQSS analyzes the mutual exclusiveness of the transitions in that CCS and then records their relations into an *Exclusion Table* (Steps (2)–(5)). Based on the exclusion table, a CCS is decomposed into two or more *conflict-free* subsets (Steps (6)–(14)).

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. 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)) ensures that the following conditions are satisfied by the generated set of schedules $\{TQSS_1, \ldots\}$: (1) Each transition t in each of the schedules can be fired within its firing time interval $[\tau_\alpha(t), \tau_\beta(t)]$, (2) Each schedule of a TCCPN A_i can be completed within the deadline d_i of that TCCPN, and (3) The maximum amount of total memory used by each set of concurrent schedules of all the TCCPNs is within the upper bound of μ.

3.3.2 Code Generation with Multiple Threads

Based on the hardware resource configuration provided by a designer, an embedded software code with multiple threads is generated. Each source transition in a TCCPN represents an input event. Corresponding to each source transition, a P-thread is generated. 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;`. Code generation begins by visiting the source transition, once for each of its successor places.

In both the sub-procedures Visit_Trans() and Visit_Place(), 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. 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 TQSS 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 formed of one master device and seven active slave devices. In our TCCPN model of a Master/Slave (M/S) switch between two devices A and B, there are totally four Petri nets. 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.

The proposed TQSS algorithm (Table 1), was applied to the given system of four TCCPN. The results of scheduling are given in Table 2. We observe that each of the two HC/LM models has a CCS $\{t_8, t_9, t_{10}\}$, which is decomposed by TQSS 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, TQSS derived that the system is schedulable under the earliest deadline first scheduling policy. The last column in Table 2 gives the best-case and worst-case execution times of each net schedule.

There are totally six source transitions in the four TCCPN 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.

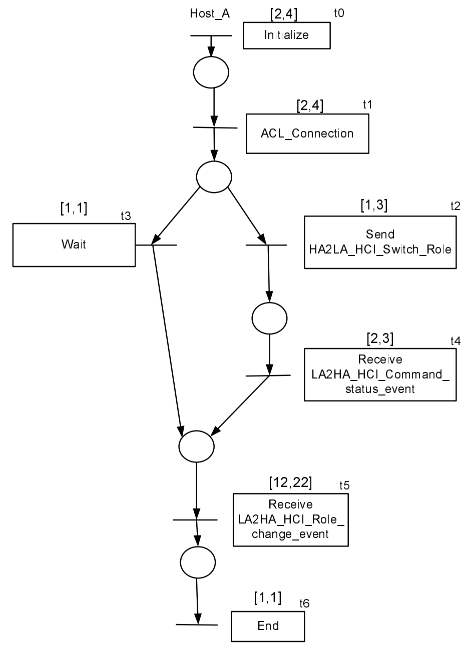

Figure 2. TCCPN model of Host A in Bluetooth M/S switch

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.

[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.

Table 2. Scheduling Results for Bluetooth M/S Role Switch

TCCPN	$\|T\|$	$\|P\|$	d_i	π_i	$\|TQSS\|$	TQSS 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, $|TQSS|$: number of schedules.

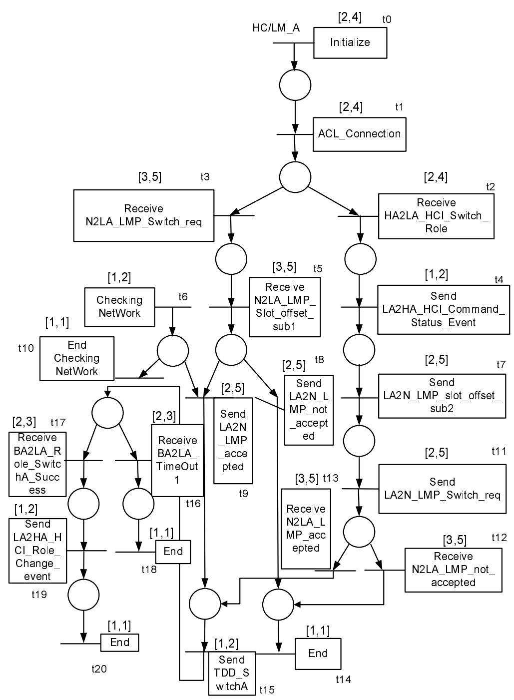

Figure 3. TCCPN model of HC/LM A in Bluetooth M/S switch

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[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 1997.

[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. 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.

[13] 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.

[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.

[15] 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.

Proceedings of the 16th International Conference on VLSI Design (VLSI'03)
1063-9667/03 $17.00 © 2003 **IEEE**