Time-Memory Scheduling and Code Generation of Real-Time Embedded Software [*]

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

Abstract

Increase in software has made embedded systems more accessible and easy to use, but it has also necessitated further research on how a complex, real-time, embedded software can be designed automatically and correctly. Enhancing recent advances in this research, we propose a Time-Memory Scheduling *(TMS) method for formally synthesizing and automatically generating code for real-time embedded software, using the* Colored Time Petri Nets *model. Our method extends previous work in three ways: (1) by allowing the specification of* temporal constraints *in the system description to model* real-time *behaviors of software, (2) by allowing the specification of* colored tokens *in the system description to model different memory usages by data-types, and (3) by proposing an extended algorithm to schedule the enhanced system model and generate static code. A real-time embedded software, which is specified by a set of CTPN, is scheduled using TMS such that the schedules satisfy limited embedded memory requirements and all real-time and task precedence constraints. Finally, a portable embedded software program is generated in the C programming language using the valid TMS schedules. Through a real-world example on the ATM Virtual Private Network server, we illustrate the feasibility and advantages of the proposed TMS method for synthesizing embedded real-time software.*

1 Introduction

With advances in electronic technology, it is now possible to embed a microprocessor in almost any electric appliance such as home appliances, internet appliances, personal assistants, wearable computers, telecommunication gadgets, and transportation facilities. Consequently, the number of embedded systems that a man encounters in a typical day of his or her life has increased dramatically from a few tens in the past to the order of hundreds in the recent few years. Moreover, once an embedded system interacts with a human, there are temporal expectations on its behavior, which may be a soft constraint (such as multimedia servers) or a hard one (such as the braking system in a vehicle). Nowadays, most embedded systems are also *real-time* systems, thus their design must also satisfy all real-time requirements. With this motivation, we propose a *time-memory scheduling* method to formally synthesize and automatically generate code for a real-time embedded system.

A real-time embedded system is a computation unit, installed in a larger system called environment, such that it helps the environment accomplish some dedicated set of tasks with temporal and spatial constraints. In general, an embedded system has both hardware and software parts. Hardware is fabricated as one or more ASICs, ASIPs, or PLDs. Software is executed on one or more microprocessors, with or without an operating system. *Real-time embedded software* (RTES) is a piece of program code that must: (1) satisfy real-time constraints such as response time, deadlines, and periods, and (2) execute within a specified size of memory space. RTES communicates with the embedded hardware either through an interface or through direct connections. There are two main issues in the design of RTES:

- *Bounded Memory Execution*: A processor cannot have infinite amount of memory space for the execution of any software process. This fact is even more emphasized in an embedded system, which generally has only a few hundreds of kilobytes memory installed.
- *Real-Time Constraints*: A processor may have to execute several concurrent tasks with precedence and temporal constraints. Thus, an RTES is generally composed of several concurrent, real-time, computation tasks.

In solution to the above two issues, a synthesis method for RTES must generate program code that can be executed in a bounded amount of memory, while satisfying all given

[*]This work was supported by research project grant NSC-90-2215-E-194-009 from the National Science Council, Taiwan, ROC.

real-time constraints. The proposed solution consists of the following two steps:

- *Time-Memory Scheduling*: A partial reachability tree is computed such that all computations that violate either temporal or spatial constraints are pruned from the tree. The resulting tree guarantees that, for all possible outcomes in a non-deterministic data-dependent execution, the memory utilized for computation is always within limits and the execution of the software is periodic, that is, it always returns to its initial state within its deadline.
- *Code Generation*: The tree obtained after scheduling represents a feasible computation of a system and code can be generated through a direct mapping translation.

In this work, a formal synthesis method based on *Colored Time Petri Nets* (CTPN) is proposed, which employs *Time-Memory Scheduling* (TMS) for satisfying limited embedded memory restrictions and hard real-time constraints. Software code is then generated from TMS schedules. The number of tasks in the software code is minimized to improve efficiency and code-size. Finally, an application example illustrates the feasibility and benefits of our proposed method.

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

2 Previous Work

Currently, *software synthesis* is a hot topic of research in the field of hardware-software codesign of embedded systems [9]. Previously, a large effort was directed towards hardware synthesis and comparatively little attention paid to software synthesis. Partial software synthesis was mainly carried out for communication protocols [17], plant controllers [16], 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, 15, 19].

Lin [15] 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 quasi-static scheduling algorithm was proposed by Sgroi et al. for a class of Petri nets called *Free-Choice Petri Nets* (FCPN) [19]. A necessary and sufficient condition was given for a set of FCPNs to be schedulable. Schedulability was first checked 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 schedules. Based on FCPN, Hsiung proposed an extended scheduling method that incorporated real-time constraints into the synthesis procedure such that code could be generated for hard real-time embedded systems [13]. It was later modified to synthesize code for *soft* real-time embedded systems [14]. Both methods were still restricted by the Free-Choice constraint on the system description model.

Cortadella et al. [6] proposed a reachability graph algorithm for a more general class of Petri nets, which allowed unbounded FIFO channels between two multi-rate communicating processes and synchronization-dependent control on multiple ports. The input consisted of FlowC sources and the output was scheduled embedded software code. No timing constraints were considered in the proposed algorithm.

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 [10], the linear hybrid automata techniques [8, 11], and the mapping strategy [7]. System parameters have also been considered for software synthesis [12].

Among the above related software synthesis work, either they have not considered *real-time* constraints in their system model or their models were restricted so that not all systems could be synthesized. In contrast, our work focuses on how scheduled program code may be generated for *real-time embedded software* without any model restrictions.

3 Real-Time Embedded Software Synthesis

A real-time embedded software is specified as a set of *Colored Time Petri Nets* (CTPN), which are a combination of *Colored Petri Nets* (CPN) [18] and *Time Petri Nets* (TPN) [3, 4]. As mentioned in Section 2, several variations of Petri nets were used for the synthesis of embedded software [6, 15, 19], but neither the modeling of memory usages nor that of timing constraints were allowed explicitly by those models. Thus, we propose to use CTPN, which allows explicit modeling of memory usages and timing constraints.

In the rest of this section, we first define CTPN, give a system model, its semantics, and its scheduling. Then, we formulate our target problem. Finally, we describe our synthesis algorithm, along with code generation.

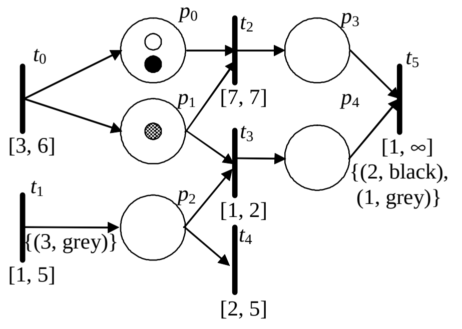

Figure 1: **A Colored Time Petri Net**

3.1 System Model

A real-time embedded system is modeled as a set of *Colored Time Petri Nets*, defined as follows.

Definition 1 *:* **Colored Time Petri Nets** (CTPN)
A *Colored Time Petri Net* is a 6-tuple $(P, T, C, \phi, M_0, \tau)$,where P is a finite set of places, T is a finite set of transitions, $P \cup T \neq \emptyset$, and $P \cap T = \emptyset$, C is a set of colors associated with each token, $\phi : (P \times T) \cup (T \times P) \to 2^{\mathcal{N} \times C}$ is a weighted flow relation between places and transitions, represented by arcs, such that each arc is associated with a set of integer-color pairs $\{(k, c) \mid k \in \mathcal{N}, c \in C\}$, and $\mathcal{N}$ is the set of non-negative integers, $M_0 : P \to 2^{\mathcal{N} \times C}$ is the initial marking (assignment of colored tokens to places), and $\tau : T \to \mathcal{N}^* \times (\mathcal{N}^* \cup \infty)$, i.e., $\tau(t) = (\alpha, \beta)$, $t \in T$, α is the *earliest firing time* (EFT), and β is *latest firing time* (LFT). We will use $\tau_\alpha(t)$ and $\tau_\beta(t)$ to denote EFT and LFT, respectively.

Graphically, a CTPN can be depicted as shown in Fig. 1, where circles represent places, vertical bars represent transitions, arrows represent arcs, dots represent tokens, different shadings of dots represent different colors, and sets of integer-color pairs labeled over arcs represent the weights as defined by ϕ. Here, $\phi(x, y) \neq \emptyset$ implies there is an arc from x to y with a weight of $\phi(x, y)$, where x and y can be a place or a transition. Both *conflicts* and *confusions* are allowed in a CTPN. A conflict occurs when there is a token in a place with more than one outgoing arc such that only one enabled transition can fire, thus consuming the token and disabling all other transitions. For example, $\{t_2, t_3\}$ and $\{t_3, t_4\}$ are pairs of conflicting transitions in Fig. 1. A *confusion* is a result of the coexistence of both concurrency and conflict at the same transition. For example, there is a confusion at transition t_3 in Fig. 1.

Semantically, the behavior of a CTPN is given by a sequence of *markings*, where a marking is an assignment of colored tokens to places. Starting from an initial marking M_0, a CTPN may transit to another marking through the firing of an enabled transition and re-assignment of tokens. A transition is said to be *enabled* when all its input places have the required number of colored tokens for the required amount of time, where the required number of colored tokens is the weight as defined by the flow relation ϕ and the required amount of time is the earliest firing time α as defined by τ. An enabled transition need not necessarily fire. But upon firing, the required number of tokens are removed from all the input places and the specified number of tokens are placed in the output places, where the specified number of colored tokens is that specified by the flow relation ϕ on the outgoing arcs from the transition. An enabled transition may not fire later than β.

To formalize the above semantics description with notations, we give the following basic definitions. A set of integer-color pairs is defined as $\{(n, c) \mid n \in \mathcal{N}, c \in C\}$, where $\mathcal{N}$ is the set of non-negative integers and C is a set of colors. If NC and NC' are two sets of integer-color pairs, then we say $NC' \leq NC$ iff $k' \leq k$ for all $(k', c) \in NC'$, $(k, c) \in NC$, and $k' > 0$. Intuitively, this means for each type of color the number of tokens of that color in NC is not greater than that in NC'. Further, for $NC' \leq NC$, we can also define their difference $NC - NC'$ as a set NC'' of integer-color pairs $\{(k'', c) \mid k'' = k - k', \forall (k, c) \in NC, (k', c) \in NC'$, and $k' \leq k\}$. Similarly, sum can also be defined for two sets of integer-color pairs.

Formally, a marking is a vector $M = \langle NC_1, NC_2, \ldots, NC_{|P|} \rangle$, where $NC_i \subseteq \mathcal{N} \times C$ is a set of integer-color pairs, representing the non-negative number of colored tokens in place $p_i \in P$. Associated with each marking M, there are two attributes: (1) a time-stamp $\psi(M)$, and (2) a memory-usage $\mu(M)$. A time-stamp $\psi(M)$ is defined as the time elapsed for a CTPN to change from the initial marking M_0 to the marking M. Here, $\psi(M_0) = 0$. A memory-usage $\mu(M)$ is defined as the amount of memory used by a CTPN when it is in the marking M.

A transition t is said to be enabled at time κ in a marking M with time-stamp $\psi(M)$ if the following conditions hold: (1) $\phi(p_k, t) \leq NC_k$, for all $\phi(p_k, t) \neq \emptyset$ and $k \in \{1, \ldots, |P|\}$, and (2) $\kappa - \psi(M) \geq \tau_\alpha(t)$. When a transition t fires in some marking M, the state of a CTPN changes to a new marking $M' = \langle NC'_1, NC'_2, \ldots, NC'_{|P|} \rangle$, where $NC'_k = NC_k - \phi(p_k, t) + \phi(t, p_k)$ for all $k \in \{1, \ldots, |P|\}$. The firing of a transition t at time κ in a marking M with time-stamp $\psi(M)$ is called a *valid firing* if it satisfies the following two properties:

- *Transition Deadline*: $\tau_\alpha(t) \leq \kappa - \psi(M) \leq \tau_\beta(t)$,

- *Memory Constraint*: $\mu(M') \leq \mu_{max}$, where M' is the marking obtained by firing t in M and μ_{max} is the maximum amount of memory available.

3.2 Problem Formulation

A user specifies the requirements for a real-time embedded software by a set of CTPNs and an upper bound on memory use, which can be defined formally as follows.

Definition 2 *:* **Real-Time Embedded Software**
A real-time embedded software system $\mathcal{S}$ is defined as a set of CTPNs $\{A_1, A_2, \ldots, A_n\}$, where $A_i = (P_i, T_i, C, \phi_i, M_{0i}, \tau_i)$, and an upper-bound on the amount of system memory, μ_{max}. ||

The following is a formal definition of the real-time embedded software (RTES) synthesis problem.

Definition 3 *:* **RTES Synthesis**
Given the specification of a real-time embedded software system $\mathcal{S}$ modeled by a set of CTPNs $\{A_1, A_2, \ldots, A_n\}$, where $A_i = (P_i, T_i, C, \phi_i, M_{0i}, \tau_i)$, and an upper-bound μ_{max} on memory use, and given a set of real-time constraints such as system period and deadline for each CTPN, a software code is to be generated such that (1) it can be executed on a single processor, (2) it uses memory less than or equal to the upper-bound μ_{max}, and (3) it satisfies all transition EFT, LFT, and real-time constraints. ||

3.3 Synthesis Algorithm

Before going into the details of the synthesis algorithm, some basic concepts and definitions are required and described as follows. Given a CTPN, we define *choice* sets and *exclusion* sets to ensure full coverage of all transitions in a final feasible schedule of the full CTPN.

Definition 4 *:* **Choice Set**
Given a CTPN $A_i = (P_i, T_i, C, \phi_i, M_{0i}, \tau_i)$, a set of transitions $H = \{t_0, t_1, \ldots, t_m\} \subseteq T_i$ is called a *choice set* if there exists a place $p \in P_i$ such that there are arcs connecting p with each of the transitions in H and with none in $T_i \backslash H$. Notationally, $\exists p \in P_i$, $\phi(p, t_k) \neq \emptyset$, $\forall\, k \in \{0, 1, \ldots, m\}$ and $\phi(p, t') = \emptyset$, $\forall\, t' \in T_i \backslash H$. ||

Conflicting transitions as mentioned in Section 3.1 are a special case of a choice set because sets of conflicting transitions are disjoint. However, choice sets are not necessarily disjoint since a transition may belong to two or more choice sets. For example, a *synchronization* transition between two places, each of which has a set of more than one outgoing transitions, belongs to two choice sets. When we merge all non-disjoint choice sets into one set of transitions, it is called an *exclusion* set, which is formally defined as follows.

Definition 5 *:* **Exclusion Set**
Given a CTPN $A_i = (P_i, T_i, C, \phi_i, M_{0i}, \tau_i)$, a set of transitions $H = \{t_0, t_1, \ldots, t_m\} \subseteq T_i$ is called an *exclusion set* if there exists a sequence of the transitions such that each adjacent pair of transitions has a common input place. ||

From the above definition, we can observe that a choice set is a special case of an exclusion set, an exclusion set is always connected, and two or more exclusion sets are disjoint. Intuitively, an exclusion set represents all possible choices of dependent computation (behavior) at a particular system state (CTPN marking). Thus, in our scheduling algorithm to be presented later in this Section, we enforce the fact that an exclusion set should be either completely enabled or completely disabled at a marking before we accept the marking as a feasible state for the system schedule. Partial enabling of an exclusion set will eventually result in a partial system schedule.

Now, we introduce the notions of source transitions and independent tasks. A transition t is called a *source transition* if $\phi(p, t) = \emptyset$ for all places $p \in P$, that is, it has no input place. Physically, a source transition represents an uncontrollable input event from the environment. Two source transitions are said to be *dependent* if they synchronize at some common reachable transition, where a transition t is said to be reachable from another transition t' if there exists a sequence of valid transition firings from the firing of t to the enabling of t'. A set of source transitions is defined as *maximal* if it consists of all source transitions that are inter-dependent and there is no other source transition in a CTPN that is dependent on any transition in that set. For example, in Figure 1, source transitions t_0 and t_1 are dependent because their corresponding computation runs eventually synchronize at t_3. Further, a set of transitions constitute an *independent task* if they are all reachable from some maximal set of dependent source transitions. In Figure 1, the CTPN constitutes a single independent task.

Given the above basic definitions and concepts on the CTPN model, we will now formally present our synthesis algorithm. As introduced in Section 1 and formulated in Definition 3, there are two objectives for an RTES synthesis algorithm, namely bounded memory execution and satisfaction of real-time constraints. The algorithm proposed here gives an integrated solution to the two issues, in the form of a *Time-Memory Scheduling* strategy.

3.3.1 Time-Memory Scheduling

In *Time-Memory Scheduling* (TMS), valid software schedules are generated for a real-time embedded system by creating a process for each independent task, which consists of one or more dependent source transitions. Each process is a sequential schedule generated by creating a reachability tree with markings as nodes and valid transition firings as edges. Several factors are considered when creating a reachability tree such as the bound on maximum memory available, the period of the CTPN in which an independent task belongs, and the corresponding deadline. Each task can be assigned a priority such as execution frequency, thus we do not allow preemption of a task while it is executing. This ensures that transition firing intervals are obeyed according to the sequential schedule of a process.

The details of our proposed TMS algorithm is given in Table 1. The given set of CTPNs is first partitioned into independent tasks, as defined earlier (Step 1). Each independent task is contained within a CTPN, whereas a CTPN may consists of more than one independent task. Then, a reachability tree is generated for each independent task by starting with the initial marking as the root node. Here, the root node is in fact a projection of the CTPN initial marking onto the independent task (Steps 2, 3, 4). Each node of the reachability tree represents a marking of the independent task and each tree edge represents the valid firing of an enabled transition. First, child nodes (1-step successor markings) are generated for the root node (**Spawn_Child**() in Step 6). Second, one of the child nodes of the root is selected for traversal, where selection is based on an evaluation of memory and time usages (**Select_Child**() in Step 7), as described later. Lastly, a reachability tree is generated iteratively (Steps 8–28) until either the root node is marked and thus code can be generated (**Gen_Code**() in Step 9) or all nodes have been deleted (Step 8) and thus no feasible schedule exists.

In the generation of a reachability tree, a *marked* node indicates that starting from the marking represented by that node, there is a valid schedule. For each current node (CNode) under consideration, either it is a complete schedule or not (Step 24). If it is, then it is simply marked (Step 25) and its parent considered as the current node (Step 26). If it is not a complete schedule, then a child node is created (**Spawn_Child**() in Step 27) for each of its 1-step successor marking, which satisfies all constraints including:

- *Transition Deadline*: $\psi(M') - \psi(M) + \tau_\alpha(t) \le \tau_\beta(t)$, where it is assumed that t is a transition which is enabled starting from marking M, represented by CNode, at the time-stamp $\psi(M)$, and t is continuously enabled until another marking M' with time-stamp $\psi(M')$ is reached,

- *CTPN Deadline*: $\psi(M') + \tau_\alpha(t) \le d_i$, where d_i is the deadline of the CTPN to which the current task belongs.

- *Memory Usage*: $\mu(M'') \le \mu_{max}$, where M'' is a new marking reached after firing t from M'.

Table 1: Time-Memory Scheduling Algorithm

```
TM_Schedule($\mathcal{S}, \mu_{max}, E, D$)
$\mathcal{S} = \{A_i \mid A_i = (P_i, T_i, C, \phi_i, M_{0i}, \tau_i), i = 1, 2, \ldots, n\}$;
integer $\mu_{max}$;        // maximum memory
$E = \{\pi_i \mid \pi_i \in \mathcal{N}, i = 1, 2, \ldots, n\}$;     // periods
$D = \{d_i \mid d_i \in \mathcal{N}, i = 1, 2, \ldots, n\}$;     // deadlines
{  $T$ = Independent_Tasks($\mathcal{S}$);                                   (1)
  for each $task \in T$ {     // assume $task \in A_i$,                   (2)
    RTree = Create_New_Reach_Tree($t$);                               (3)
    RTree.root = Project_Marking($M_{0i}, t$);                        (4)
    CNode = RTree.root;     // CNode: Current Node                 (5)
    Spawn_Child(CNode, $\mu_{max}$, $\pi_i$, $d_i$);                          (6)
    CNode = Select_Child(CNode);                                      (7)
    while (RTree.size != 0) {                                         (8)
      if(CNode==RTree.root && CNode.HasChild
        && CNode.AllChildMarked)
        Gen_Code(RTree);                                              (9)
      if(CNode.Spawned) {                                            (10)
        if(CNode.HasChild) {                                         (11)
          Delete_Incomplete_ExSet(CNode);                            (12)
          if(Marked(CNode.CompleteExSet)) {                          (13)
            Delete_Other_Child(CNode);                               (14)
            Mark(CNode);                                             (15)
            CNode = CNode.Parent; }                                  (16)
          else if(Marked(CNode.NonExSet)) {                          (17)
            Delete_Other_Child(CNode);                               (18)
            Mark(CNode);                                             (19)
            CNode = CNode.Parent; }                                  (20)
          else CNode = Select_Child(CNode); }                        (21)
        else { Delete(CNode);                                        (22)
          CNode = CNode.Parent; } }                                  (23)
      else { if(CNode.Is_Complete_Schedule) {                        (24)
          Mark(CNode);                                               (25)
          CNode = CNode.Parent; }                                    (26)
        else Spawn_Child(CNode, $\mu_{max}$, $\pi_i$, $d_i$);               (27)
      }}}}
```

If CNode has some child (Step 11), then all child nodes that represent markings of incomplete exclusion sets are deleted (Step 12). The intuition here is that a partial enabling of an exclusion set will eventually lead to a partial schedule, which is not acceptable. If there is some child node with a complete exclusion set and is also marked (Step 13), then all other child nodes are deleted (Step 14), CNode is marked (Step 15) and its parent considered as the current node (Step 16). The same is done for a single marked child node that does not belong to any exclusion set (Steps 17–20). If there is no marked child node, then one of the child nodes is selected as the current node (**Select_Child**() in Step 21). If no child can be generated for CNode, then it is deleted (Step 22) and its parent considered as the current node (Step 23).

For the selection of a child node (**Select_Child**()) as a feasible next marking in the reachability tree schedule, TMS algorithm adopts the *Earliest Deadline First* (EDF) approach, that is, among all the possible markings, the marking with the earliest deadline is chosen as the next marking in the generated schedule. If two or more markings have equal earliest deadlines, then the marking with the largest execution time is chosen. If two or more markings have equal earliest deadlines as well as equal execution times, then the one with the least memory usage is chosen. Here, the satisfaction of timing constraints is given preference over that of memory constraints because time accumulates over a computation run, whereas memory usage is the maximum of memory usages of all markings in a computation run.

After applying the above method, a reachability tree is created for each independent task. These tasks can then be scheduled non-preemptively according to their priorities.

During scheduling, an estimation of memory usage is made for each new marking and the satisfaction of memory bound is checked by observing if the estimated memory space does not exceed the bound. Memory space used by a program can be classified functionally into the following:

- *Global Memory*: Global variables and data reside in global memory and their life-span is the entire duration of program execution. This space is assumed to be allocated at the very beginning of program execution, thus it is of constant size and can be determined statically. This constant space size must be added to each estimation of memory space.

- *Local Memory*: Local variables used by the user-given code for a transition reside in local memory. This space size differs for each transition and must be estimated a priori through code analysis. The maximum size of local memory spaces used by all transitions, whose firings result in a marking, must be added to the memory size estimate.

- *Buffer Memory*: Intermediate variables or data that are passed from the code of one transition to that of another reside in buffer memory. Since CTPNs have colored tokens with colors from the set C, if the amount of memory occupied by some color c in C is denoted as $\mu_C(c)$, we can estimate the amount of buffer memory used by a marking $M = \langle NC_1, \ldots, NC_{|P|}\rangle$ as follows:

$$\mu_B(M) = \sum_{1 \le i \le |P|} \left(\sum_{(n,c) \in NC_i} (n \times \mu_C(c)) \right) \quad (1)$$

  It is assumed here that garbage collection of released memory space is either performed by each transition (upon consumption of input colored tokens), or by the system such as the Java Virtual Machine.

Table 2: Code Generation Algorithm

```
**Gen_Code**($RTree$)
$RTree$: reachability tree
{
    $X$ = Create_Queue();                              (1)
    Extract(RTree.root);                               (2)
    schedule = Create_Schedule();                      (3)
    while($X \neq$ NULL)                               (4)
        schedule = Concatenate(schedule, $X$.pop());   (5)
    Replace_Code(schedule);                            (6)
}
```

The maximum amount of memory space used by a program code can be estimated as follows:

$$\mu(S) = \max_{R \in S} \left\{ \mu_G(R) + \max_{M \in R} \left[\max_{t \to M}(\mu_L(t)) + \mu_B(M) \right] \right\} \quad (2)$$

where $\mu_G(R)$ is the global memory size for an independent task that is scheduled using the reachability tree R, $\max_{t \to M}(\mu_L(t))$ is the maximum amount of local memory space $\mu_L()$ used by transitions t whose firings result in the marking M, and $\mu_B(M)$ is as defined in Equation (1).

3.3.2 Code Generation

After time-memory scheduling, the schedules (reachability trees) obtained from the set of CTPNs are mapped into software programs by a *code generation procedure* **Gen_Code**() as shown in Table 2. A *real-time process* is created for each independent task in the system. This method of code generation minimizes the number of tasks in a system because the degree of concurrency in a system is equal to the number of independently firing transitions [19], which is the same as the number of independent tasks.

As shown in Table 2, for transforming a reachability tree into software code, a queue is used to store a schedule of the tree (Step 1). An **Extract**() procedure recursively extracts code from the tree and stores it into the queue (Step 2). A schedule is thus generated by popping out all the extracted codes in sequence (Step 3, 4). Finally, all scheduling symbols are replaced by actual user-defined codes (Step 5). For example, the code for each transition is now used to replace scheduling symbols that represented the transition.

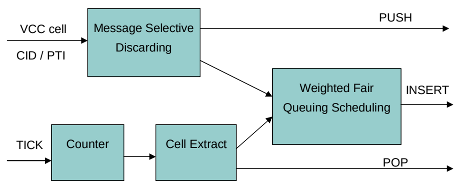
VCC cell
CID / PTI
Message Selective Discarding
PUSH
Weighted Fair Queuing Scheduling
INSERT
TICK
Counter
Cell Extract
POP

Figure 2: **ATM Virtual Private Network Server**

3.3.3 Implementation

The proposed TMS algorithm and code generation procedures were implemented in the Java programming language which generates C code. Due to the portability of Java, our small synthesis program can be installed in different kinds of embedded systems and prototypes so that users can dynamically change features of embedded application software according to their needs. The reasoning for generating C code is because it is more efficient than Java and equally portable on most machines. An example on an ATM server will be given in the next Section, whose code was synthesized and generated by executing our synthesis program.

4 ATM VPN Server Example

To illustrate the feasibility and advantages of our real-time embedded software synthesis method, we have applied it to a real-world system: an ATM Server for Virtual Private Networks (VPN) [5]. An ATM server resides in ATM switching nodes interconnecting LANs via an ATM backbone. An ATM server temporarily stores input cells from *Virtual Channel Connections* (VCCs) and forwards them to *Virtual Path Connections* (VPCs) according to cell header information and internal state tables of VCCs. The functionalities of an ATM server are shown in Figure 2, where CID and PTI are interrupts that carry header information and occur at irregular times when a non-empty cell enters the server, TICK is a periodic event that, after N occurrences, enables the algorithm (Cell Extract) that chooses the next cell to be emitted [20]. According to the specification, CID/PTI and TICK do not have a fixed sampling rate ratio and are thus independently fireable. We thus have two independent tasks for scheduling (reachability tree construction) and code generation. There are two algorithms in ATM: *Message Selective Discarding* (MSD) and *Weighted Fair Queuing* (WFQ) scheduling.

A set of CTPN is given in Figure 3, which models the ATM-VPN server. There are totally 39 places and 44 transitions in the model. It is a compact modified version of that in [20]. As illustrated in Figure 3, the MSD algorithm starts executing whenever it receives both interrupts CID and PTI (synchronized at t1). It first checks the state of the VCC of the incoming cell and the logic queue where the cell is to be forwarded, from the internal tables (READ_STATE_VCC and READ_OUT_QUID). Then, the incoming cell is processed according to the VCC state. At place p9, the value of variable st indicates the state of a VCC: IDLE, ACCEPT, or REJECT.

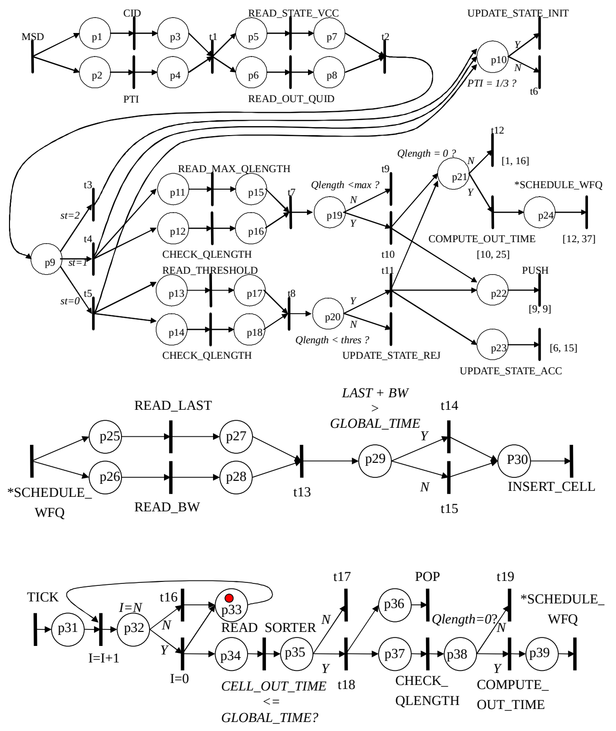

Figure 3: **CTPN models of ATM VPN Server**

For any state of VCC of the incoming cell, the MSD algorithm checks the value of the last bit of the PTI field in the header. If the bit is one, the cell is an end-message cell and the state of the VCC is updated to IDLE (UPDATE_STATE_INIT), otherwise no action is taken (t6).

TICK is also a source transition that fires independently. After N occurrences of TICK (modeled by transitions I=I+1, t16, and I=0), a real-time sorter with time-stamp information of each cell is read and the cell with the smallest timestamp equal to the current global time is popped from its VPC queue. If the queue becomes empty, then a timestamp computation algorithm is invoked to compute the next timestamp and WFQ scheduling is also performed.

The execution time and the memory used by the output data of each transition in the CTPN model of the ATM-VPN server were specified as shown in Table 3, where transitions are grouped according to type.

Table 3: Time and Memory of ATM-VPN Transitions

Transition Type	Transitions	T	M
Interrupt Handling	MSD, CID, PTI, TICK	1	4
Memory Read	READ_STATE_VCC, READ_OUT_QUID, READ_MAX_QLENGTH, CHECK_QLENGTH, READ_THRESHOLD, READ_LAST, READ_BW, READ_SORTER, CHECK_QLENGTH	3	4
Memory Write	UPDATE_STATE_INIT, UPDATE_STATE_REJ, UPDATE_STATE_ACC, INSERT_CELL, I=I+1, I=0	6	4
Synchronization	t1, t2, t7, t8, t13	1	4
Push Queue	PUSH	9	8
Event Triggers	t3, t4, t5, t9, t10, t11, t14, t15, WFQ, t16, t18	1	4
Sink (No-Op)	t6, t12, t17, t19	1	4
Computation	COMPUTE_OUT_TIME, POP	10	8

T (Time) is in number of instructions, **M** (Memory) is in number of bytes

On applying our proposed time-memory scheduling algorithm (Table 1), to the given CTPNs in Figure 3, we obtain three reachability trees, which are omitted due to page-limits. In the reachability tree for the MSD algorithm, there are 49 nodes (reachable markings) and 14 different computation runs (schedules). For WFQ, there are 9 markings and 2 schedules. For TICK, there are 13 markings and 4 schedules. Though there are three CTPNs in the ATM model (Figure 3) and three corresponding reachability trees, there are actually only two independent tasks corresponding to the independently firing CID/PTI pair and TICK source transitions. In the model and trees, WFQ is not an independent task, it is invoked by either of the two tasks and hence the notation *SCHEDULE_WFQ (invocation of one of the two WFQ schedules in the tree). Thus, for the CID/PTI task, there are totally $14 \times 2 = 28$ schedules, and for the TICK task, there are totally $4 \times 2 = 8$ schedules.

Table 4 shows the computation runs in the ATM-VPN server model. The estimates for execution time and memory usage for each computation run are also given. The maximum of those estimates are reported as system execution time (66 instructions) and memory usage (12 bytes). For simplicity, we only consider buffer memory space size estimation in this example. The actual memory usage will be larger than 12 bytes because there will be global memory and local memory usages.

Upon execution of transition t11, there are tokens in places p21, p22, and p23, which concurrently enables transitions PUSH, UPDATE_STATE_ACC, and t12 or COMPUTE_OUT_TIME. During time-memory scheduling, we have a choice here to select one of the child nodes as the next marking. As described in Section 3.3.1, we use earliest deadline first (EDF) as our selection policy. Here, the deadlines are, respectively, 9, 15, 16, and 25, which is also the order for selecting them as next markings (see Table 4).

Software code was then generated for the ATM VPN server using our code generation procedure. Since the code is a straightforward mapping of the reachability tree to a C procedure, we have omitted it here. Branching constructs such as if-then-else or switch-case are inserted at branching nodes of the tree. Nodes are then replaced by actual user-given codes.

5 Conclusion and Future Work

A formal automatic method for the synthesis of *Real-Time Embedded Software* (RTES) was proposed, including a time-memory scheduling algorithm and a code generation procedure. The resulting program code not only satisfied all user specified real-time and memory constraints, but also consisted of a minimum number of scheduled tasks, which minimized both memory usage and execution time. The proposed method was applied to a real-world ATM Virtual Private Network example to illustrate its feasibility and advantages. Future work consists of installing our small synthesis program into an embedded system and into a prototyping platform for on-the-fly synthesis and code-generation of real-time embedded software. Future research directions include the development of methods for automatic code generation and code modifications based on the frequently changing dynamic needs of users, such as web computations.

References

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

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

[3] B. Berthomieu and D. Diaz. Modeling and verification of time dependent systems using time petri nets. *IEEE Transactions on Software Engineering*, 17(3):259–275, 1991.

[4] B. Berthomieu and M. Menasche. An enumerative approach for analyzing time petri nets. In *Proc. of the IFIP Congress*, September 1983.

[5] P. Coppo, M. D'Ambrosio, and V. Vercellone. The A-VPN server: a solution for ATM virtual private networks. In *Proc. Singapore ICCS'94*, November 1994.

[6] J. Cortadella, A. Kondratyev, L. Lavagno, M. Massot, S. Moral, C. Passerone, Y. Watanabe, and A. Sangiovanni-Vincentelli. Task generation and compile-time scheduling for mixed data-control embedded software. In *Proc. Design Automation Conference (DAC'00)*, June 2000.

[7] 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.

Table 4: Computation Runs and Time-Memory Estimates for ATM-VPN

Computation Run	**Time**	**Mem**
⟨MSD, CID, PTI, t1, READ_STATE_VCC, READ_OUT_QUID, t2, t3, t6⟩	13	8
⟨MSD, CID, PTI, t1, READ_STATE_VCC, READ_OUT_QUID, t2, t3, UPDATE_STATE_INIT⟩	18	8
⟨MSD, CID, PTI, t1, READ_STATE_VCC, READ_OUT_QUID, t2, t4, READ_MAX_QLENGTH, CHECK_QLENGTH1, t7, t6, t9⟩	21	12
⟨MSD, CID, PTI, t1, READ_STATE_VCC, READ_OUT_QUID, t2, t4, READ_MAX_QLENGTH, CHECK_QLENGTH1, t7, t6, t10, PUSH, t12⟩	31	12
⟨MSD, CID, PTI, t1, READ_STATE_VCC, READ_OUT_QUID, t2, t4, READ_MAX_QLENGTH, CHECK_QLENGTH1, t7, t6, t10, PUSH, COMPUTE_OUT_TIME, *SCHEDULE_WFQ⟩	55	12
⟨MSD, CID, PTI, t1, READ_STATE_VCC, READ_OUT_QUID, t2, t4, READ_MAX_QLENGTH, CHECK_QLENGTH1, t7, UPDATE_STATE_INIT, t9⟩	26	12
⟨MSD, CID, PTI, t1, READ_STATE_VCC, READ_OUT_QUID, t2, t4, READ_MAX_QLENGTH, CHECK_QLENGTH1, t7, UPDATE_STATE_INIT, t10, PUSH, t12⟩	36	12
⟨MSD, CID, PTI, t1, READ_STATE_VCC, READ_OUT_QUID, t2, t4, READ_MAX_QLENGTH, CHECK_QLENGTH1, t7, UPDATE_STATE_INIT, t10, PUSH, COMPUTE_OUT_TIME, *SCHEDULE_WFQ⟩	60	12
⟨MSD, CID, PTI, t1, READ_STATE_VCC, READ_OUT_QUID, t2, t5, READ_THRESHOLD, CHECK_QLENGTH2, t8, t6, UPDATE_STATE_REJ⟩	26	12
⟨MSD, CID, PTI, t1, READ_STATE_VCC, READ_OUT_QUID, t2, t5, READ_THRESHOLD, CHECK_QLENGTH2, t8, t6, t11, PUSH, UPDATE_STATE_ACC, t12⟩	37	12
⟨MSD, CID, PTI, t1, READ_STATE_VCC, READ_OUT_QUID, t2, t5, READ_THRESHOLD, CHECK_QLENGTH2, t8, t6, t11, PUSH, UPDATE_STATE_ACC, COMPUTE_OUT_TIME, *SCHEDULE_WFQ⟩	61	12
⟨MSD, CID, PTI, t1, READ_STATE_VCC, READ_OUT_QUID, t2, t5, READ_THRESHOLD, CHECK_QLENGTH2, t8, UPDATE_STATE_INIT, UPDATE_STATE_REJ⟩	31	12
⟨MSD, CID, PTI, t1, READ_STATE_VCC, READ_OUT_QUID, t2, t5, READ_THRESHOLD, CHECK_QLENGTH2, t8, UPDATE_STATE_INIT, t11, PUSH, UPDATE_STATE_ACC, t12⟩	42	12
⟨MSD, CID, PTI, t1, READ_STATE_VCC, READ_OUT_QUID, t2, t5, READ_THRESHOLD, CHECK_QLENGTH2, t8, UPDATE_STATE_INIT, t11, PUSH, UPDATE_STATE_ACC, COMPUTE_OUT_TIME, *SCHEDULE_WFQ⟩	66	12
⟨TICK, I=I+1, t16⟩	8	4
⟨TICK, I=I+1, I=0, READ_SORTER, t17⟩	17	4
⟨TICK, I=I+1, I=0, READ_SORTER, t18, POP, CHECK_QLENGTH, t19⟩	31	12
⟨TICK, I=I+1, I=0, READ_SORTER, t18, POP, CHECK_QLENGTH, COMPUTE_OUT_TIME, *SCHEDULE_WFQ⟩	55	12

Time is in number of instructions, Memory (**Mem**) is in number of bytes
*SCHEDULE_WFQ is one of the two computation runs for WFQ scheduling with 15 time units and 8 bytes of memory.

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

[9] P.-A. Hsiung. CMAPS: A cosynthesis methodology for application-oriented parallel systems. *ACM Transactions on Design Automation of Electronic Systems*, 5(1):51–81, January 2000.

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

[11] 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.

[12] 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.

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

[14] P.-A. Hsiung. Formal synthesis and control of soft embedded real-time systems. In *Proc. 21st International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'01), (Cheju Island, Korea)*, pages 35–50. Kluwer Academic Publishers, August 2001.

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

[16] 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, March 1995.

[17] 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 – 25, January 1983.

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

[19] 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.

[20] Marco Sgroi. Quasi-static scheduling of embedded software using free-choice petri nets. Master's thesis, Dept. of Electrical Engineering and Computer Science, Univ. of California at Berkeley, 1999.