Time-Memory Scheduling and Code Generation of Real-Time Embedded Software

Chuen-Hau Gau

A Thesis Submitted To
Institute of Computer Science and Information Engineering
College of Engineering
National Chung Cheng University
For the Degree of
Master
in
Computer Science and Information Engineering
Abstract

Increase in system complexity has made embedded software a very important part of an embedded system. It has 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* (CTPN) 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 both limited embedded memory requirements and all real-time with task precedence constraints. Finally, a 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.

**Keyword:** Time-Memory Scheduling, Colored Time Petri Net, embedded system
## Contents

1 Introduction ......................................................... 1

2 Previous Work .................................................... 4

3 Real-Time Embedded Software Synthesis ....................... 7
   3.1 System Model ................................................... 7
   3.2 Problem Formulation ........................................... 12

4 Time-Memory Scheduling ........................................... 14
   4.1 Time-Memory Scheduling ....................................... 16
   4.2 Code Generation ................................................ 23
   4.3 Implementation ................................................ 25

5 Examples .......................................................... 26
   5.1 Specification of the ATM Server .............................. 26
   5.2 Petri Net model .................................................. 29
   5.3 Results ........................................................... 36
6 Conclusion and Future Work

Bibliography
List of Figures

3.1 A Colored Time Petri Net ........................................ 9
3.2 A medic care system ............................................... 11
5.1 ATM Server example ............................................... 27
5.2 VCC states .......................................................... 28
5.3 CTPN models of ATM VPN server ............................... 30
5.4 Schedules of MSD task generated by program ................ 37
5.5 Schedules of TICK task generated by program ............... 38
List of Tables

4.1 Time-Memory Scheduling Algorithm ........................................ 17
4.2 Time-Memory Scheduling Algorithm (continued) ....................... 18
4.3 Function Select() ............................................................... 21
4.4 Code Generation Algorithm .................................................. 24

5.1 Definition of Transitions ..................................................... 33
5.2 Definition of Transitions (continued) ..................................... 34
5.3 Type of tokens ................................................................. 35
Chapter 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
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 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 Thesis is organized as follows. Chapter 2 gives some previous work related to RTES synthesis. Chapter 3 defines *Colored Time Petri Net* (CTPN) and formulates the RTES synthesis problem. Chapter 4 shows our *Time-Memory Scheduling* (TMS) and code generation algorithm. Chapter 5 illustrates the proposed problem solution through an ATM application example. Chapter 6 concludes the Thesis giving some future work.
Chapter 2

Previous Work

Currently, software synthesis is a hot topic of research in the field of hardware-software codesign of embedded systems [10]. 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 [18], plant controllers [17], and real-time schedulers [1] because they generally exhibited regular behaviors. Only recently has there been some work on automatically generating software code for embedded systems [2, 16, 20, 22].

Lin [16] proposed an algorithm that generates a software program from a concurrent process specification through intermediate Petri-Net representation. This approach is based on the assumption that the Petri-Nets are safe, i.e., buffers can store at most one data unit, which implies that it is always schedulable. The proposed method applies quasi-static scheduling to a set of safe Petri-Nets to produce a set of corresponding state machines, which are then mapped syntactically to the final software code. Later, Zhu and Lin [22] proposed a compositional version of the synthesis method that produced the generated code size and was thus more efficient.
A quasi-static scheduling algorithm was proposed by Sgroi et al. for a class of Petri nets called Free-Choice Petri Nets (FCPN) [20]. 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 [14]. It was later modified to synthesize code for soft real-time embedded systems [15]. Both methods were still restricted by the Free-Choice constraint on the system description model.

Cortadella et al. [7] 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.

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.

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 [11], the linear hybrid automata techniques [9, 12], and the mapping strategy [8]. System
parameters have also been considered for software synthesis [13].

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.
Chapter 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) [19] and Time Petri Nets (TPN) [4, 5]. As mentioned in Chapter 2, several variations of Petri nets were used for the synthesis of embedded software [7, 16, 20], 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 Chapter, we first define CTPN, give a system model, its semantics, and its scheduling. Finally, we formulate our target problem.

3.1 System Model

A real-time embedded software is modeled as a set of Colored Time Petri Nets, which are defined as follows.
Definition 3.1: 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, such that \(P \cup T \neq \emptyset\) and \(P \cap T = \emptyset\),
- \(C\) is a finite set of colors associated with each token,
- \(\phi: (P \times T) \cup (T \times P) \rightarrow 2^{\mathbb{N} \times C}\) 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 \(\{(k, c) | k \in \mathbb{N}, c \in C\}\), and \(\mathbb{N}\) is the set of non-negative integers,
- \(M_0: P \rightarrow 2^{\mathbb{N} \times C}\) is the initial marking (assignment of colored tokens to places),
- \(\tau: \mathbb{N}^* \times (\mathbb{N}^* \cup \infty), i.e., \tau(t) = (\alpha, \beta), t \in T, \alpha\) is the earliest firing time (EFT), and \(\beta\) is the 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 Figure 3.1, 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 \(\phi\). 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.

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 $\phi$ and the required amount of time is the earliest firing time $\alpha$ as defined by $\tau$. 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 $\phi$ on the outgoing arcs from the transition. An enabled transition may not fire later than $\beta$ as defined by $\tau$.

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 possibly disabling other enabled
transitions. For example, \{t_2, t_3\} and \{t_3, t_4\} are pairs of conflicting transitions in Figure 3.1. A \textit{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 Figure 3.1.

Semantically, the behavior of a CTPN is given by a sequence of \textit{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 enable transition and re-assignment of tokens.

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

Formally, a marking is a vector \(M = \langle \mathcal{NC}_1, \mathcal{NC}_2, ..., \mathcal{NC}_{\mid P} \rangle\), where \(\mathcal{NC}_i \subseteq \mathcal{N} \times \mathcal{C}\) 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:

- Time-stamp \(\psi(M)\), which is defined as the time elapsed for a CTPN to change from the initial marking \(M_0\) to the marking \(M\). Here, \(\psi(M_0) = 0\).
Memory-usage $\mu(M)$, which is defined as the amount of memory in bytes used by a CTPN when it is in the marking $M$.

A transition $t$ is said to be enabled at time $\kappa$ 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 $k \in \{1, 2, ..., |P|\}$, 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 $M' = \langle NC'_1, NC'_2, ..., NC'_P \rangle$, where $NC'_k = NC_k - \psi(p_k, t) + \psi(t, p_k)$ for all $k \in \{1, 2, ..., |P|\}$. The firing of a transition $t$ at time $\kappa$ 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:**  \( \mu(M') \leq \mu_{\text{max}} \), where \( M' \) is the marking obtained by firing \( t \) in \( M \) and \( \mu_{\text{max}} \) is the maximum amount of memory available.

To illustrate our CTPN model, we introduce a medic care system example, as shown in Figure 3.2, which analyzes information from patients periodically and (1) shows patient status on an LCD screen, (2) writes status into a LOG file. If there is an abnormal situation, the system (3) activates an alarm, and (4) starts an emergency procedure to help the patient. The system starts from the interrupt transition \( t_0 \), and filters the incoming information (FILTER), shows the information on an LCD screen (LCD) and writes information into a LOG file (LOG). If the patient is in an abnormal situation, the system starts emergency procedure (BLOOD, TEMP, BREATHE) and activate the alarm (ALARM).

In the transitions \( t_2, BLOOD, TEMP, BREATHE \), we define the EFT values the same as the LFT values to denote that these transitions are tightly real-time which must be executed immediately after being enabled (i.e. tolerance of delay time is zero). In this example, we can observe that the best case execution time is 42 (schedule \( \langle t_0, \text{FILTER}, \text{LCD}, \text{LOG}, t_1 \rangle \)), and the worst case execution time is 87 (schedule \( \langle t_0, \text{FILTER}, t_2, \text{BLOOD}, \text{ALARM}, \text{LCD}, \text{LOG} \rangle \)).

### 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 3.2: **Real-Time Embedded Software**

A real-time embedded software system $S$ is defined as a set of CTPNs \( \{A_1, A_2, \ldots, A_n\} \), where \( A_i = \{P_i, T_i, C_i, \phi_i, M_{0i}, \tau_i\} \), and an upper-bound on the amount of system memory, \( \mu_{max} \).

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

**Definition 3.3: RTES Synthesis**

Given the specification of a real-time embedded software system $S$ modeled by a set of CTPNs \( \{A_1, A_2, \ldots, A_n\} \), where \( A_i = \{P_i, T_i, C_i, \phi_i, M_{0i}, \tau_i\} \), and an upper-bound integer \( \mu_{max} \) 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 \( \mu_{max} \), and (3) it satisfies all transition EFT, LFT, and real-time constraints.
Chapter 4

Time-Memory Scheduling

In the Chapter 3, we have introduced the Colored Time Petri Net model and formulated our problem. In this Chapter, we will propose a Time-Memory Scheduling method and code generation algorithm for solving the problem.

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 4.1 : Choice Set

Given a CTPN $A_i = (P_i, T_i, C, \phi_i, M_{0i}, \tau_i)$, a set of transitions $H = \{t_0, t_1, ..., t_m\} \subseteq T_i$ 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 $T_i \setminus H$. Notationally, $\exists p \in P_i, \phi(p, t_k) \neq \emptyset, \forall k \in 0, 1, ..., m$ and $\phi(p, t') = \emptyset, \forall t' \in T_i \setminus H$.

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 4.2 : Exclusion Set**

Given a CTPN $A_i = (P_i, T_i, C, \phi_i, M_{0i}, \tau_i)$, a set of transitions $H = \{t_0, t_1, ..., t_m\} \subseteq T_i$ is called a *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 Chapter, 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 3.1, 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 3.1, 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 and formulated in Chapter 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.

4.1 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
Table 4.1: Time-Memory Scheduling Algorithm

\[
\text{TM Schedule}(S, \mu_{\text{max}}, E, D) \\
S = \{A_i \mid A_i = (P_i, T_i, C, \phi_i, M_{0_i}, \tau_i), i = 1, 2, \ldots, n\}; \\
\text{integer } \mu_{\text{max}}; \quad \text{// maximum memory} \\
E = \{\pi_i \mid \pi_i \in \mathcal{N}, i = 1, 2, \ldots, n\} \quad \text{//periods} \\
D = \{d_i \mid d_i \in \mathcal{N}, i = 1, 2, \ldots, n\} \quad \text{//deadlines} \\
\{ T=\text{Independent\_Tasks}(S) \}; \quad (1) \\
\text{for each task } \in T\{ \quad \text{//assume task } \in A_i \quad (2) \\
\quad \text{RTree } = \text{Create\_New\_Reach\_Tree}(t); \quad (3) \\
\quad \text{RTree.root } = \text{Project\_Marking}(M_{0_i}, t); \quad (4) \\
\quad \text{CNode } = \text{RTree.root}; \quad (5) \\
\quad \text{Spawn}(\text{CNode, } \mu_{\text{max}}, \pi_i, d_i); \quad (6) \\
\quad \text{Select();} \quad (7) \\
\}
// Continued in Table 4.2
\]

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 4.1. The given set of CTPNs is first partitioned into independent tasks (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 (\texttt{Spawn}(CNode, \mu_{max}, \pi_i, d_i) 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 usages (\texttt{Select()} in Step 7), as described later.

As shown in Table 4.2, a reachability tree is generated iteratively (Steps 8-19)
until either the root node is marked and thus code can be generated ($\text{Gen\_Code}(\text{RTree})$ in Step 10) 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 15). If it is, then it is simply marked (Step 16) and its parent considered as the current node (Step 17). If it is not a complete schedule, then a child node is created ($\text{Spawn}(\text{CNode}, \mu_{\text{max}}, \pi_i, d_i)$ in Step 19) for each of its 1-step successor marking, which satisfies all constraints including:

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

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

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

If current node was spawned (Step 11), which may have some child nodes, then all child nodes that represent markings resulting from the firing 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. We check if current node is in a valid schedule path or not and select next node to
continue exploring by the \texttt{select()} function (Step 13) as described in Table 4.3.

In Table 4.3, we delete current node and focus on its parent, if there does not exists any child node (Steps 1, 2, 3, 4), which means this node is in an invalid schedule path of the reachability tree. Otherwise, we check if there is a partial of valid schedule path starting from the current node (Steps 5, 6, 7, 8), which satisfies one of the following two rules.

- Have one marked child and the connecting arc between the current node and the marked child corresponds to a transition that does not belong to any exclusion set.

- Have some marked child nodes and the connecting arcs between the current node and the marked child nodes correspond exactly to the transition in an exclusion set.

If the conditions are satisfied, we remove other child nodes, mark this node and focus on its parent (Steps 9, 10, 11). When we can’t identify current node leads to a valid schedule or not, we find an unmarked child by function \texttt{score()} to continue our exploration (Step 12).

The purpose of function \texttt{score()} is to find an unmarked child to continue exploring. The selection is based on the effect in memory usage status. For example, suppose we have two child nodes representing two enabled transitions to be fired such that one consumes tokens and another generates tokens. We will consider the transition that is consuming tokens as the better one and schedule it first.

After applying the above method, a reachability tree is created for each inde-
**Table 4.3: Function Select()**

Select(CNode, $\mu_{max}$, $\pi_i$, $d_i$)

$H' = \{H_1, H_2, ..., H_n\}$ \hspace{1cm} //Sets of different system Exclusion Sets

$\eta = H'' \setminus T_i$ \hspace{1cm} // which $H'' = H_1 \cup H_2 \cup ... \cup H_n$,

\hspace{1cm} // $T_i$ is the set of transitions in current CTPN $A_i$

$R' = \{r_1, r_2, ..., r_m\}$ \hspace{1cm} //set of outgoing arcs from CNode

$T' = \{t_1, t_2, ..., t_m\} \subseteq T_i$ \hspace{1cm} //$t_i$ is the transition corresponds to $r_i$, $i = 1, 2, ..., m$

$N' = \{n_1, n_2, ..., n_m\}$ \hspace{1cm} //$n_i$ is the node which $r_i$ connects to, $i = 1, 2, ..., m$

{
if($N' == \emptyset$) {

\hspace{1cm} temp=CNode;

\hspace{1cm} CNode=CNode.PreviousNode;

\hspace{1cm} delete(temp) }

\hspace{1cm} else if($\exists R'' \subseteq R'$, and $T''$ is the transition set respect to $R''$, \hspace{1cm} (5)

\hspace{1cm} \hspace{1cm} $N''$ is the node set respect to $R''$

\hspace{1cm} \hspace{1cm} $\&\& (T'' \in H' \hspace{1cm} || \hspace{1cm} (T'' \subseteq \eta \hspace{1cm} \&\& \hspace{1cm} |T''| == 1))$

\hspace{1cm} \hspace{1cm} $\&\& \hspace{1cm} \text{marked}(N'')$) {

\hspace{1cm} \hspace{1cm} \textbf{Delete\_Children\_Excepts}(CNode, $N''$);

\hspace{1cm} \hspace{1cm} \textbf{mark}(CNode);

\hspace{1cm} \hspace{1cm} CNode=CNode.PreviousNode(); }

\hspace{1cm} else

\hspace{1cm} CNode=\textbf{score}(N')

}
pendent 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 $M = \langle NC_1, NC_2, ..., NC_{|\Pi|} \rangle$ as follows:
\[
\mu_B(M) = \sum_{1 \leq i \leq |P|} \left( \sum_{(n,c) \in N C_i} (n \times \mu_C(c)) \right) \tag{4.1}
\]

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.

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

\[
\mu(S) = \max_{R \in S} \left\{ \mu_G(R) + \max_{M \in R} \left[ \max_{t \rightarrow M} (\mu_L(t)) + \mu_B(M) \right] \right\} \tag{4.2}
\]

where \( \mu_G(R) \) is the global memory size for an independent task that is scheduled using the reachability tree \( R \), \( \max_{t \rightarrow M} (\mu_L(t)) \) 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 (4.1).

### 4.2 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 \texttt{Gen_Code()} as shown in Table 4.1 and Table 4.2. 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
Table 4.4: Code Generation Algorithm

```
Gen_Code(RTree)

RTree: reachability tree

{

X = Create_Queue(); (1)

Extract(RTree.root); (2)

schedule = Create_Schedule(); (3)

while(X ≠ NULL) (4)

schedule = Concatenate(schedule, X.pop()); (5)

Replace_Code(schedule); (6)

}
```

the number of independently firing transitions [20], which is the same as the number of independent tasks.

As shown in Table 4.4, 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, 5). Finally, all scheduling symbols are replaced by actual user-defined codes (Step 6). For example, the code for each transition is now used to replace scheduling symbols that represented the transition.
4.3 Implementation

The proposed TMS algorithm and code generation procedures were implemented in the Java programming language which generates user defined code. Due to the portability of Java, our 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. An example on an ATM server will be given in the next Chapter, whose code was synthesized and generated by executing our synthesis program.
Chapter 5

Examples

To illustrate the feasibility and advantages of our real-time embedded software syn-
thesis method, we have applied it to a real-world system: an ATM Server for Virtual
Private Networks (VPN) [6].

5.1 Specification of the ATM Server

The ATM Server is a network component that performs support functions to im-
plement Virtual Private Networks (A-VPNs) in ATM switching nodes and allow the
interconnection of Local Area Networks of multi-site users via an ATM backbone. The
Server performs statistical multiplexing of the input flows, that are Virtual Channel
Connections (VCCs) in ATM networks, by using an internal buffer that temporarily
stores the incoming cells. The buffer is shared by several queues, one for every out-
put Virtual Path Connection (VPC) (Figure 5.1). For every incoming cell the Server
identifies from the header which input VCC the cell belongs to and retrieves from
the internal tables the state of the VCC and the output VPC where the cell is to be
forwarded. Then, according to the state of its VCC, the cell is either discarded or
Figure 5.1: ATM Server example

forwarded to the queue that corresponds to the output VPC.

The main functionalities of the Server are (1) a message discarding technique that avoids buffer overflow by discarding selected incoming cells and (2) a bandwidth control policy for the outgoing flows based on a *Weighted Fair Queueing* (WFQ) scheduling discipline.

- The *Message Selective Discarding* (MSD) algorithm avoids the indiscriminate loss of cells by using a threshold mechanism that preserves the integrity of messages. Every VCC can be in one of the following states: idle, accept or reject (Figure 5.2). At the setup, every VCC is in the idle state. When the first cell of a message arrives, the corresponding VCC enters the accept state if the
length of the queue where the cell is to be forwarded is below a threshold, it enters the reject state if it is equal or above the threshold. While the VCC is in the accept (reject) state, all its cells are accepted (discarded). Upon arrival of an end-message cell, recognized by the value of the last bit of the field PTI of the header, the VCC returns to the idle state. The next incoming cell belonging to the same VCC is necessarily the first cell of a new message and therefore the threshold-based acceptance mechanism is repeated. This buffer management technique ensures that all the cells of a message are either accepted or rejected and therefore avoids retransmission of entire messages when message integrity is required and cell losses occur.

- The control of the bandwidth is realized by using a *Weighted Fair Queueing* (WFQ) Scheduling policy that assigns to every queue a fixed portion of the

\[
\text{Figure 5.2: VCC states}
\]
bandwidth of the output link. This mechanism is implemented by assigning to an incoming cell the timestamp at which it must be emitted, so that each connection is guaranteed to acquire its allocated bandwidth. Then, a pointer to the cell and its output timestamp are inserted into a real-time sorter, i.e. a priority queue that returns, when requested, the pointer to the cell with the smallest timestamp value. Since cells belonging to the same queue are always output in order, the sorter only stores pointers to cells that are at the head of the queue. Upon arrival of a new cell, a cell pointer is inserted in the sorter only if the queue where the cell is stored is empty. At every cell time slot if the current time is equal to the smallest timestamp in the sorter, the cell with this timestamp is popped from the buffer and forwarded to the assigned output link. If a cell is emitted and the queue is not empty, it is necessary for a new timestamp computation and pointer insertion for the cell that now occupies the first place in the queue. Therefore, both the cell admission and emission algorithms may enable the timestamp computation algorithm whenever a new entry in the sorter is to be inserted.

5.2 Petri Net model

A set of CTPN is given in Figure 5.3, 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 [21].

As illustrated in Figure 5.3, Transition MSD is an interrupt that carry header information and occur at irregular times when a non-empty cell enters the Server,
Figure 5.3: CTPN models of ATM VPN server
TICK is a periodic event that after $N$ occurrences enables the algorithm that chooses the next cell to be emitted. According to the specification, transitions MSD and TICK do not have a fixed sampling rate ratio and therefore are independently-firable.

Let us describe the dynamic behavior of the net. After the arrival of a non-empty cell (firing of MSD), information about the state of the VCC of the incoming cell and the logic queue where the cell is to be forwarded is retrieved from the internal tables ($READ.\text{STATE.VCC}$ and $READ.\text{OUT.QUID}$). The cell is processed differently according to the state of the VCC. If the state is reject ($st = 2$), no further action is taken and the cell is discarded ($t3$). If the state is accept ($st = 1$), the actual length of the queue where the cell is forwarded is compared with the maximum queue size and then, if the queue is not full, the cell is pushed into the buffer ($PUSH$). If the queue is full, the cell is discarded ($t9$). If the state is idle ($st = 0$), the queue length is compared with the buffer threshold and if the queue length is smaller then the threshold, the cell is forwarded into the queue ($PUSH$), otherwise it is discarded ($t11$). The state of the VCC is updated ($UPDATE.\text{STATE.REJ}$ and $UPDATE.\text{STATE.ACC}$), respective to the two cases, to either accept or reject. Finally, for any state of the VCC of the incoming cell, the MSD algorithm checks the value of the last bit of the field PTI of the header: if this bit is set to one, the cell is an end-message cell and the state of the VCC is updated to idle ($UPDATE.\text{STATE.INIT}$), otherwise no action is taken ($t6$). When a cell is pushed into the buffer ($PUSH$) and the queue is empty, the computation of the cell output time is enabled ($COMPUTE.\text{OUT.TIME}$).
The second source transition of the system is \textit{TICK}. \textit{TICK} enables a counter module $N$, whose activity is modeled by transitions $I = I + 1$, $t16$ and $I = 0$. After transition \textit{TICK} has fired $N$ times, firing transition \textit{READ\_SORTER}, enables the process of choosing the next cell to be emitted. The smallest timestamp is compared with the global time. If it is equal or smaller, the cell is ready to be emitted and therefore is popped from the buffer and forwarded to the output link (\textit{POP}), otherwise no action is taken and no cell is emitted during this cell time slot ($t17$). Concurrently with the emission of a cell, the timestamp computation process is enabled (\textit{*SCHEDULE\_WFQ}) only if the queue is not empty.

Both the MSD and the Cell Extraction algorithms may enable the cell output time computation algorithm, starting in transition \textit{*SCHEDULE\_WFQ}. To compute the cell timestamp to be inserted into the sorter, the bandwidth of the output VPC (\textit{READ\_BW}) and the time the last cell of the same connection was emitted (\textit{READ\_LAST}) are taken from the internal tables. Then, the output time of the next cell of the queue is computed based on the \textit{Weighted Fair Queueing} algorithm [6]. Finally, the timestamp is inserted into the sorter (\textit{INSERT\_CELL}) that is part of the environment in our model.

Each transition in the CTPN model of the ATM-VPN server was specified as shown in Table 5.1 and Table 5.2, where $\alpha$ and $\beta$ are the execution time and deadline of the transition, as defined in Chapter 3 and Chapter 4. The execution times and deadlines are defined in numbers of instructions. Table 5.3 shows specification of tokens in the CTPN model of the ATM-VPN server.
Table 5.1: Definition of Transitions

<table>
<thead>
<tr>
<th>Transition</th>
<th>$\alpha$</th>
<th>$\beta$</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSD</td>
<td>1</td>
<td>$\infty$</td>
<td>Interrupt handling.</td>
</tr>
<tr>
<td>CID</td>
<td>3</td>
<td>$\infty$</td>
<td>Get cell ID from cell structure.</td>
</tr>
<tr>
<td>PTI</td>
<td>3</td>
<td>$\infty$</td>
<td>Get PTI field from cell structure.</td>
</tr>
<tr>
<td>READ_STATE_VCC</td>
<td>3</td>
<td>$\infty$</td>
<td>Memory Access.</td>
</tr>
<tr>
<td>READ_OUT_QUID</td>
<td>15</td>
<td>$\infty$</td>
<td>Access hash table.</td>
</tr>
<tr>
<td>t1</td>
<td>1</td>
<td>$\infty$</td>
<td>Event trigger or sink.</td>
</tr>
<tr>
<td>t2</td>
<td>1</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>t3</td>
<td>2</td>
<td>$\infty$</td>
<td>Branch.</td>
</tr>
<tr>
<td>t4</td>
<td>2</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>t5</td>
<td>2</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>UPDATE_STATE_INIT</td>
<td>8</td>
<td>8</td>
<td>Branch.</td>
</tr>
<tr>
<td>t6</td>
<td>2</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>READ_MAX_QLENGTH</td>
<td>3</td>
<td>$\infty$</td>
<td>Memory Access.</td>
</tr>
<tr>
<td>CHECK_QLENGTH</td>
<td>3</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>READ_THRESHOLD</td>
<td>3</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>t7</td>
<td>1</td>
<td>$\infty$</td>
<td>Event trigger or sink.</td>
</tr>
<tr>
<td>t8</td>
<td>1</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>t9</td>
<td>2</td>
<td>$\infty$</td>
<td>Branch.</td>
</tr>
<tr>
<td>t10</td>
<td>2</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>t11</td>
<td>2</td>
<td>$\infty$</td>
<td>Branch.</td>
</tr>
<tr>
<td>UPDATE_STATE_REJ</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Continued in Table 5.2.
Table 5.2: Definition of Transitions (continued)

<table>
<thead>
<tr>
<th>Transition</th>
<th>$\alpha$</th>
<th>$\beta$</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>t12</td>
<td>2</td>
<td>$\infty$</td>
<td>Branch.</td>
</tr>
<tr>
<td>COMPUTE_OUT_TIME</td>
<td>12</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>PUSH</td>
<td>9</td>
<td>$\infty$</td>
<td>Push Queue</td>
</tr>
<tr>
<td>UPDATE_STATE_ACC</td>
<td>6</td>
<td>6</td>
<td>Memory access</td>
</tr>
<tr>
<td>WFQ</td>
<td>1</td>
<td>$\infty$</td>
<td>Event trigger or sink.</td>
</tr>
<tr>
<td>t13</td>
<td>1</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>READ_LAST</td>
<td>3</td>
<td>$\infty$</td>
<td>Memory access</td>
</tr>
<tr>
<td>READ_BW</td>
<td>3</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>t14</td>
<td>15</td>
<td>$\infty$</td>
<td>Branch.</td>
</tr>
<tr>
<td>t15</td>
<td>25</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>INSERT_CELL</td>
<td>6</td>
<td>$\infty$</td>
<td>Memory access</td>
</tr>
<tr>
<td>TICK</td>
<td>1</td>
<td>$\infty$</td>
<td>Interrupt handling.</td>
</tr>
<tr>
<td>I=I+1</td>
<td>6</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>t16</td>
<td>2</td>
<td>$\infty$</td>
<td>Branch.</td>
</tr>
<tr>
<td>I=0</td>
<td>6</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>READ_SORTER</td>
<td>30</td>
<td>$\infty$</td>
<td>Call real-time sorter.</td>
</tr>
<tr>
<td>t17</td>
<td>2</td>
<td>$\infty$</td>
<td>Branch.</td>
</tr>
<tr>
<td>t18</td>
<td>2</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>t19</td>
<td>2</td>
<td>$\infty$</td>
<td>Branch.</td>
</tr>
<tr>
<td>COMPUTE_OUT_TIME</td>
<td>12</td>
<td>$\infty$</td>
<td></td>
</tr>
</tbody>
</table>
### Table 5.3: Type of tokens

<table>
<thead>
<tr>
<th>Token</th>
<th>Weight (In bytes)</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temp</td>
<td>0</td>
<td>Represents a control point.</td>
</tr>
<tr>
<td>CellPt</td>
<td>8</td>
<td>A point to cell structure.</td>
</tr>
<tr>
<td>CID</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PTI</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>st</td>
<td>4</td>
<td>VCC states.</td>
</tr>
<tr>
<td>OutQuid</td>
<td>4</td>
<td>Out Queue ID.</td>
</tr>
<tr>
<td>MaxQlength</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Qlength</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Threshold</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>LAST</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>BW</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>TimeStamp</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Sorter</td>
<td>10</td>
<td>The memory which the real-time sorter used.</td>
</tr>
</tbody>
</table>
5.3 Results

On applying our proposed time-memory scheduling algorithm (Table 4.1 and Table 4.2), to the given CTPNs in Figure 5.3, we obtain two independent tasks, starting from the MSD and TICK transitions, respectively, and we generate two reachability trees. In the reachability tree for the MSD task, there are 83 nodes (reachable markings) and 18 different computation runs (schedules). For TICK, there are 20 markings and 5 schedules.

Figure 5.4 shows the computation runs generated by program of MSD task. As shown in Figure 5.4. There are 18 schedules generated. The worst case execution time is 112 instructions and the maximum memory usage is 20 bytes. For simplicity, we only consider buffer memory space size estimation in this example. The actual memory usage will be larger than 20 bytes because there will be global memory and local memory usages. The computation runs of TICK task are shown in Figure 5.5.

Software code was then generated for the ATM VPN server using our code generation procedure. The code is a straightforward mapping of a reachability tree to a procedure. Transition codes are insert into each arc, which represents transitions in the CTPN. Branching constructs such as if-then-else or switch-case are inserted at branching nodes of the tree.
Figure 5.4: Schedules of MSD task generated by program.
<table>
<thead>
<tr>
<th>Exe</th>
<th>Mem</th>
<th>Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>4</td>
<td>tick/i=i+1/t16/</td>
</tr>
<tr>
<td>45</td>
<td>14</td>
<td>tick/i=i+1/i=0/read_sorter/t17/</td>
</tr>
<tr>
<td>60</td>
<td>14</td>
<td>tick/i=i+1/i=0/read_sorter/t18/pop/check_qlength3/t19/</td>
</tr>
<tr>
<td>99</td>
<td>14</td>
<td>tick/i=i+1/i=0/read_sorter/t18/pop/check_qlength3/compute_out_time/wfq/read_bw/read_last/t13/t14/insert_cell/</td>
</tr>
<tr>
<td>109</td>
<td>14</td>
<td>tick/i=i+1/i=0/read_sorter/t18/pop/check_qlength3/compute_out_time/wfq/read_bw/read_last/t13/t15/insert_cell/</td>
</tr>
</tbody>
</table>

Figure 5.5: Schedules of TICK task generated by program
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.

The current work on Time-Memory Scheduling can be further enhanced on improved along the following further research direction.

- The current proposed TMS algorithm and code generation procedure analyze and decompose the concurrent behaviors and map it into a procedure without concurrent behavior to increase software efficiency. The decomposing was optimize by considering of memory usage. But TMS do not guarantee the output result is the best one, which has minimal worse case memory usage. Thus we may improve the algorithm by some estimations of CTPN model.
• The problems of infinite computation runs was mentioned in [7]. Making a reachability tree that with a circle to represent a marking that was repeated in some schedule is not feasible in a CTPN model which with time features on the transition. Because timestamps on the each marking effects fires of transitions, we may further research in how to find closure of schedules represents the repeating markings.

• To develop methods for automatic code generation and code modifications based on the frequently changing dynamic needs of users, such as web computations.
Bibliography


