Pos-Graduation in Computer Science

"A Time Petri Net Based Approach for Software Synthesis in Hard Real-Time Embedded Systems with Multiple Processors"

By

Eduardo Antônio Guimarães Tavares

MSc Dissertation

Federal University of Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/∼posgraduacao

Recife, March/2006

FEDERAL UNIVERSITY OF PERNAMBUCO

INFORMATIC CENTER

POS-GRADUATION IN COMPUTER SCIENCE

Eduardo Antônio Guimarães Tavares

"A Time Petri Net Based Approach for Software Synthesis in Hard Real-Time Embedded Systems with Multiple Processors"

ADVISOR: Dr. Paulo Romero Martins Maciel
CO-ADVISOR: Dr. Raimundo Barreto da Silva

Recife, March/2006

Tavares, Eduardo Antônio Guimarães

A time Petri net based approach for software synthesis in Hard Real-Time embedded systems with multiple processors / Eduardo Antônio Guimarães Tavares. – Recife : O Autor, 2006.

xi, 118 pages : il., fig., tab.

Dissertação (mestrado) – Universidade Federal de Pernabuco. CIn. Ciência da Computação, 2006.

Inclui bibliografia.

1. Ciência da Computação – Sistemas Operacionais. 2. Síntese de Software – Sistemas de tempo real críticos. 3. Redes de Petri – Modelagem de sistemas. 4. Múltiplos processadores – Passagem de mensagens. I. Título.

004.415.2	CDU (2.ed.)	UFPE
005.273	CDD (22.ed.)	BC2006-308

This dissertation is dedicated to
my mother,
and
my sister.

Acknowledgments

First of all, I would like to thank God for all conquests that I have been achieving. Also, I am very grateful to my mother, who has always encouraged me to do what I dreamed. Many thanks to my sister, who has been at my side at all difficult moments. Thanks to my father.

Many thanks to my advisor and friend professor Paulo Maciel. He gave me the chance to be part of his research group, and has helped me at all moments. A special thanks to professor and friend Raimundo Barreto who helped me at all stages of this research. Even away from UFPE, he has never stopped to give his support and encouragement; to professor Meuse Oliveira Jr. who provided the hardware implementation for validating the experiments; to Fernando Rocha and Bruno Souza for testing and validating the experiments; to professor Ricardo Massa who helped with valuable reviews; to all members of our research group (MODCS - Modeling of Distributed and Concurrent Systems); and to Tomaz Barros for his valuable contribution and service as a comittee member.

Thanks to several colleagues I made at CIn/UFPE. In order to not forget any name, feel acknowledged all who read this dissertation. Thanks to all professors and staff in the center for informatics.

Resumo

Atualmente, sistemas embarcados são ubíquos. Em outras palavras, eles estão em todos os lugares. Desde utilitários domésticos (ex: fornos microondas, refrigeradores, videocassetes, máquinas de fax, máquinas de lavar roupa, alarmes) até equipamentos militares (ex: mísseis guiados, satélites espiões, sondas espaciais, aeronaves), nós podemos encontrar um sistema embarcado. Desnecessário afirmar que a vida humana tem se tornado mais e mais dependente desses sistemas.

Alguns sistemas embarcados são classificados como sistemas de tempo real, onde o comportamento correto depende não somente da integridade dos resultados, mas também nos tempos em que tais resultados são produzidos. Em sistemas embarcados de tempo real críticos, se as restrições temporais não forem satisfeitas, as conseqüências podem ser desastrosas, incluindo grandes danos aos equipamentos ou mesmo perdas de vidas humanas.

Devido a tarefas que possuem alta taxa de utilização de processador, alguns sistemas embarcados (ex: dispositivos médicos) precisam ser compostos de mais de um processador para obter performance aceitável e, no caso de sistemas embarcados de tempo real críticos, para satisfazer as restrições temporais críticas. Entretanto, questões adicionais precisam ser consideradas para lidar com um ambiente multiprocessado, tal como comunicação entre processadores e sincronização .

Nessa dissertação, um método de síntese de software baseado no formalismo matemático redes de Petri com tempo é apresentado para lidar com sistemas embarcardos de tempo real críticos com múltiplos processadores. A abordagem inicia a partir de uma especificação (usualmente composta de tarefas concorrentes e comunicantes) e automaticamente gera o código fonte de um programa considerando: (i) as funcionalidades e restrições; e (ii) o suporte operacional para execução das tarefas em um ambiente multiprocessado. Síntese de software é uma alternativa para sistemas operacionais especializados para dar suporte a execução de um programa. Sistemas operacionais são usualmente genéricos e podem introduzir atrasos no tempo de execução, e ao mesmo tempo produzir alto consumo de memória. Por outro lado, a síntese de software é uma alternativa de projeto, dado que este método automaticamente gera o código fonte do programa, satisfazendo a funcionalidade, as restrições especificadas, o suporte para execução, e a minimização dos atrasos e uso de memória.

Abstract

Nowadays, embedded systems are ubiquitous. In other words, they are everywhere. From household appliances (e.g. microwave ovens, refrigerators, VCRs, fax machines, dishwashers, burglar alarms) to military equipments (e.g. guide missiles, spy satellites, deep-space probes, aircrafts), we may find an embedded system. Needless to say, the human life has become more and more dependent of these systems.

Some embedded systems are classified as real-time systems, where the correct behavior depends not only on the integrity of the results, but also on the time in which such results are produced. In *hard* real-time systems (e.g. medical devices), if timing constraints are not met, the consequences can be disastrous, including great damage of resources or even loss of human lives.

Due to CPU-bound tasks, some embedded systems (e.g. medical devices) need to be composed of more than one processor for achieving acceptable performance and, in the case of hard real-time embedded systems, for meeting stringent timing constraints. However, additional issues should be considered when dealing with a multiprocessing environment, such as inter-processor communication and synchronization.

In this dissertation, a software synthesis method based on a mathematical formalism, namely, time Petri Net, is presented for handling hard real-time embedded systems with multiple processors. The approach starts from a specification (usually composed of concurrent and communicating tasks) and automatically generates a program source code considering: (i) the functionalities and constraints; and (ii) the operational support to the tasks' execution in a multiprocessing environment. Software synthesis is an alternative to specialized operating system for supporting the software execution. Operating systems are usually very general and may introduce delays in the execution time, at the same time in which produces a higher rate of memory utilization. On the other hand, the software synthesis is a design alternative, since this method automatically generates the program source code, satisfying the functionality, the specified constraints, the typical runtime support, and the minimization of both delays and use of memory.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, embedded systems are ubiquitous. In other words, they are everywhere. From household appliances (e.g. microwave ovens, refrigerators, VCRs, fax machines, dishwashers, burglar alarms) to military equipments (e.g. guide missiles, spy satellites, deep-space probes, aircrafts), we may find an embedded system. Needless to say, the human life has become more and more dependent of these systems.

Depending on the purpose of the application, the design of embedded systems may have to take into account several constraints, for instance, time, size, weight, cost, reliability and energy consumption. Additionally, many embedded systems need to be composed of more than one processor, since some CPU-bound tasks surpass the computation power of a single processor. In this way, other issues should be considered, for instance, inter-processor communication and synchronization.

Embedded systems that have timing constraints are classified as real-time systems (e.g. cell phones and medical devices). In real-time systems, not only is the logical result of the computation important, but also the time in which it was obtained [46]. Such systems are categorized as hard or soft real-time systems. In soft real-time systems, timing constraints may occasionally not be reached. In this case, the system may just degradate its behavior and this may be tolerated. On the other hand, consequences can be disastrous in hard real-time systems, which is the focus of this dissertation, if timing constraints are not met. In this case, the consequences can be resource damages or even loss of human life.

Lately, the automatic synthesis of embedded software has been receiving much attention. According to [40], most part of an embedded system is composed of software, more specifically, more than 80% of functionalities considering standard embedded systems. This leads to a software-oriented design, which leverages some advantages such as flexibility, lower cost and accessibility. However, not many works deal with embedded software synthesis, more specifically, time-critical embedded software synthesis considering multiple processors. Generating code that guarantees meeting all timing and resource constraints is not a trivial task, even more when considering multiple

processors. This research area has several open issues, mainly related to generation of predictable scheduled code considering multiple processors . In order to cope with those stringent requirements, formal development methodologies play an important role.

1.1 Problem Description

According to Cornero et.al. [12], software synthesis is the task of converting automatically a specification (typically composed of concurrent and communicating tasks) into a software considering: (i) the specified functionalities; and (ii) the typical runtime support required. In other words, the software synthesis method translates a high-level specification into a program source code with all the operational support code required for its execution.

Software synthesis is necessary since specifications have special characteristics which are not found in traditional programming languages. For instance, specifications are generally composed of several concurrent tasks, so, scheduling and synchronization of multiple tasks are important issues. Considering multiple processors, additional issues should be considered, such as inter-processor communications. In this particular situation, software synthesis should provide an appropriate scheduler, in such a way that all specification constraints are satisfied. Thus, software synthesis consists of two main activities [12]: (i) task handling, and (ii) code generation. Task handling takes into account tasks scheduling, resource management, and intertask communication. Code generation is responsible for static generation of source code for each individual task.

In general, complex systems adopt a specialized operating system to support the software. However, this solution is very general and usually introduces overheads, mainly in execution time and memory requirements. On the other hand, the software synthesis method is an alternative to such operating systems usage, since this method automatically generates customized codes, in such a way that all constraints are met and overheads are minimized.

The problem considered in this dissertation is software synthesis for hard real-time embedded systems with multiple processors.

1.2 Motivation

Hard real-time embedded systems have stringent timing constraints that must be met for the correct functioning of the system. Some of these systems need to be composed of more than one processor in order to satisfy all timing constraints.

It is not a trivial task developing software for hard-real time embedded systems, even more when considering multiple processors. In many embedded systems, the adoption of multiple processors is a feasible solution:

- some systems are composed of several critical tasks that are CPU-bound. For ensuring that all deadlines are met, critical tasks must be placed on different processors. As an simple example, sampling a keyboard might interfere with the calculation of the oxygen saturation in a pulse oximeter;
- using several small processors may be cheaper than using one large CPU. In many cases, several 8-bit controllers can be purchased for the price of one 32-bit processor.
- many embedded computing systems require a large number of devices. Implementing the device interfaces through a set of microcontrollers may result in a low cost and flexible project.

Additionally, the market pressure has demanded, at the same time, high complex applications, and short time-to-market. Software synthesis methods play an important role, since code can be generated automatically, meeting all timing constraints and handling all inter-processor communication issues. Moreover, adopting formal methods in the software synthesis permits the verification and analysis of properties as well as facilitate system validation.

An alternative method is hand-crafted solutions. However, this alternative is error-prone, very time consuming, and generally leads to unacceptable results when considering hard real-time systems.

1.3 Objectives

Considering the problem stated previously, the main objective of this dissertation is to propose a methodology starting from a high-level specification, translating such specification into a predictable source code taking in account a multiprocessing environment.

More specifically, the objectives are:

1. proposing a specification model that captures code, timing constraints, inter-task relations (such as precedence and exclusion relations), and inter-processor communications;
2. modeling the specification using a formal method;
3. providing a scheduling synthesis framework that produces schedules guaranteeing that timing, precedence and exclusion constraints, and inter-processor communications are satisfied;

4. developing a code generator for multiple processors that generates scheduled code with guaranteeing that the implemented code satisfies the specified properties.

1.4 Proposed Method

The specification is composed of a set of tasks, which may be executed in one or more processors. For each task, timing constraints are specified, as well as inter-task relations, scheduling method, and allocation of task to processors. Additionally, inter-processor communications need to be considered. This dissertation represents each inter-processor communication as a special task, namely, communication task.

In this work, the system's tasks are modeled using a *transition-annotated* time Petri nets (TPN), that is, a time Petri net with code associated with transitions. The TPN model is used in the scheduling phase, which adopts the pre-runtime scheduling policy. In hard real-time systems, pre-runtime scheduling may provide better results than runtime policies, mainly when considering inter-task relations. In other words, pre-runtime policies are more predictable than runtime methods, therefore more suitable to hard real-time systems. Starting from a TPN model, a feasible schedule (one that satisfies all constraints) is synthesized, and a scheduled code is generated in accordance with the found schedule.

The generated code can be seen as a *cyclic executive*[5], since tasks are recurrently executed in accordance with the previously computed schedule. Cyclic executive consists of a single control loop where the execution of several periodic process is statically interleaved (or scheduled) on each CPU. The interleaving is done in a deterministic fashion so that execution time is predictable.

Figure 1.1 presents a diagram of the phases composing the proposed methodology. The phases are:

- **Specification**. The specification is composed of a set of tasks. Each task contains: (i) timing constraints (phase, release time, worst-case execution time, deadline, and period); (ii) scheduling method (preemptive or non-preemptive); (iii) inter-tasks relations, such as precedence and exclusion relations; (iv) processor allocated; and (v) source code. As described before, an inter-processor communication is represented by a special task, namely, communication task. Each communication task contains: (i) communication worst-case time; (ii) bus (the channel); (iii) sender; and (iv) receiver.

- **Modeling**. This phase deals with the translation from specifications to the respective time Petri net (TPN) models. This modeling is based on building blocks composition. There are specific blocks for modeling the task structure (preemptive or non-preemptive), deadline-checking, processor(s), communication channel(s), periodic arrival of tasks, precedence relations, exclusion relations, and

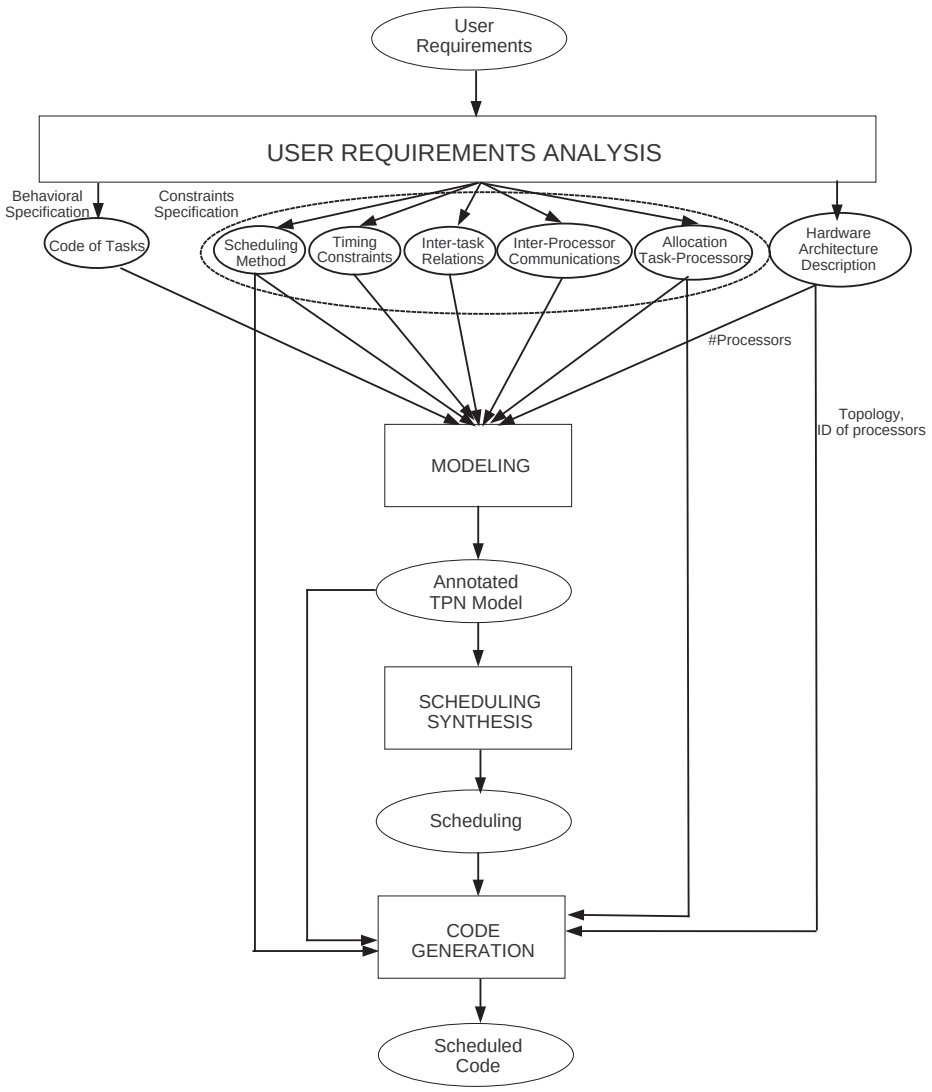

Figure 1.1: Proposed Software Synthesis Methodology Phases

inter-processor communication between tasks. The resulting model is considered in the scheduling synthesis phase. Part of this model, that is the code of tasks, is also used in the code generation phase.

- **Scheduling Synthesis**. Since this work deals with time-critical systems, a pre-runtime scheduling is adopted. Starting from the TPN model, a schedule is entirely computed during design time. This work adopts the algorithm proposed by Barreto [8], which is based on depth-first search method.

- **Code Generation**. This phase aims to generate the respective scheduled code,

considering the previously computed schedule. In addition to the tasks' code, the runtime support is also taken into account in the code generation phase. In this work, a special architecture is adopted to provide real-time clock synchronization between processors .

1.5 Contributions

This dissertation presents a methodology for development of predictable scheduled code for hard real-time embedded systems with multiple processors. This work extends the approach proposed by Barreto [8] for dealing with a multiprocessing environment. Specific contributions are depicted as follows:

1. **Modeling**. The proposed method translates the specification into a time Petri net model. As stated before, the modeling phase is based on composition of building blocks. This dissertation defines a new block, namely, the inter-processor comunication block.

2. **Schedule Synthesis**. The pre-runtime scheduling algorithm presented in [8] adopts a tagging scheme in order to avoid visiting a state more than once. Although this solution improves the algorithm execution, the tagging scheme needs to save all visited states in the memory, leading to a huge memory consumption. In this dissertation, a compressing function is utilized to mitigate such problem.

3. **Code Generation**. This dissertation presents a code generation framework for embedded hard real-time systems with multiple processors, that considers release time, deadline, periods, arbitrary precedence and exclusion relations, and inter-processor communications.

1.6 Outline

Chapter 2 overviews the main concepts of concern in this dissertation, such as embedded systems, real-time systems, and scheduling. Chapter 3 reviews the related works, and Chapter 4 introduces Petri nets. Afterwards, Chapter 5 describes the method for modeling embedded hard real-time systems. Next, Chapter 6 explains the method for synthesizing the software. Chapter 7 shows experiments conducted using the proposed methodology. Finally, Chapter 8 concludes this dissertation and presents future works.

Chapter 2

Background

This chapter introduces the main concepts needed to the understanding of this dissertation. Firstly, an overview of embedded systems is presented. Next, concepts related to real-time systems are shown. Finally, the relevant scheduling methods for real-time systems are described.

2.1 Embedded Systems

According to [53], an embedded system is a system whose principal function is not computational, but which is controlled by a computer embedded within it. Such computer maybe a microprocessor or microcontroller. The meaning of the word *embedded* implies that the computer lies inside the overall system, hidden from view, forming an integral part of a greater whole. As a result, the user may be unaware of the computer's existence. Unlike a general-purpose personal computer, embedded system computer is usually purpose designed, or at least customized, for the single function of controlling its system. Besides, embedded systems do not terminate, unless it fails [27].

Nowadays, embedded systems are everywhere, from home appliances to spaceships. More specifically, we may find an embedded system in washing machines, dishwashers, microwave ovens, burglar alarms, avionics, and so on. Due to this diversity of applications, the design of embedded systems can be subject to many different types of constraints, including timing, size, weight, power consumption, reliability, and cost.

Originally, most part of embedded systems were hardware-based, using for instance ASICs (Application Specific Integrated Circuit). However, the technology evolved, processors increased computational power and, correspondingly, decreased size and cost. Consequently, the software is responsible for 80% of an embedded system development nowadays [40]. The moving of functionalites from hardware to software brings some advantages such as flexibility, lower cost, and accessibility. On the other hand, functionalities implemented in hardware still have better performance and consume less

energy.

Over the last years, the market pressure has demanded, at the same time, high complexity embedded systems, and short time-to-market. In order to cope with these issues, design methodologies play an important role. There are some methodologies that have been used for embedded systems development. However, this work only concentrates on hardware-software co-design [18, 52, 54, 16, 26, 32], which is detailed in the next section.

2.1.1 Hardware-Software Codesign

Hardware/Software Codesign can be defined as the cooperative design of hardware and software. One of the purposes of such methodology is to deal with the problem of designing heterogeneous systems. Additionally, codesign aims to reduce time-to-market, the design effort and costs of the designed system.

One important concern in codesign methodologies is the process of selecting which functionalities should be implemented in software and in hardware. Several advantages may be obtained using software, since it is more flexible and cheaper than hardware. This flexibility allows late design changes and simplified debugging opportunities. Moreover, the possibility of reusing software by porting it to other processors, reduces the time-to-market and the design effort. Finally, in most cases, the processor-based implementation is cheaper compared to the development costs of ASICs, because processors are often produced in high-volume, leading to a significant price reduction. However, hardware is always used by the designer when processors are not able to meet the required performance and energy constraints.

Codesign Phases

A hardware-software codesign methodology may be depicted by four main phases (Figure 2.1): *Specification*, *Partitioning*, *Co-Synthesis*, and *Estimation Analysis and Validation*. These phases are described below:

a) *Specification.* This phase describes the system requirements at a high abstraction level, taking into account functional and non-functional requirements. Such description should be performed using an appropriate language or formalism.

b) *Partitioning.* This phase selects functionalities that should be implemented in hardware or in software. Such process can be executed either manually, automatically, or combining both approaches. The output of this phase is a set of communicating modules, where some of them should be implemented in hardware and others in software. Partitioning methods often take into account quality metrics as mean for partitioning the system specification.

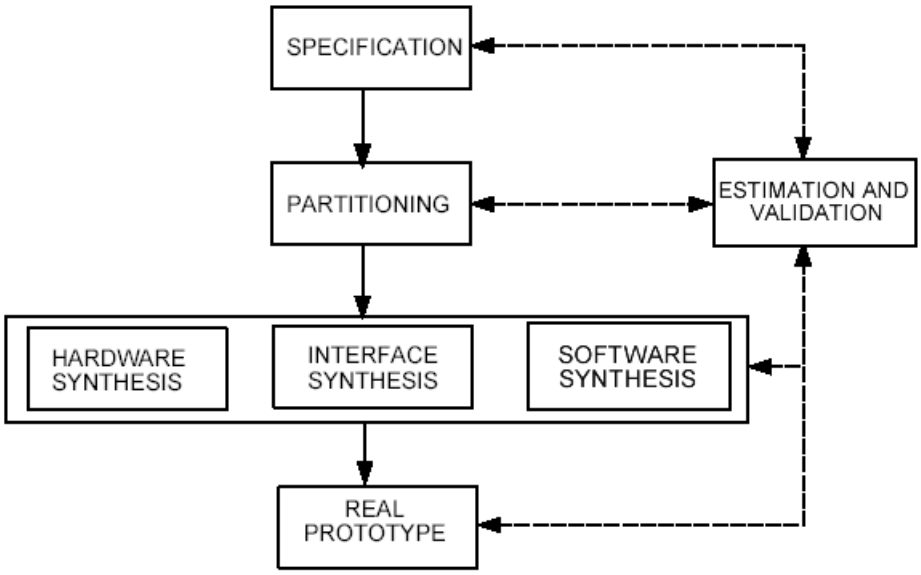

Figure 2.1: Main Phases of a Hardware-Software Codesign Methodology

c) *Co-Synthesis.* The co-synthesis phase maps the partitioning output into a real prototype in such a way that all system constraints are satisfied. The synthesis decisions might be considered in this phase. For instance, the specific processor to be used, the interconnection network, communication protocols, interface between hardware and software, concurrent processes scheduling, and so on. This phase comprises the hardware synthesis, software synthesis and the interface synthesis (when one module is implemented in hardware and the other in software) and communication synthesis (when both modules are implemented in software but in different processors).

d) *Estimation and Validation.* The analysis of a system consists of providing several quality metrics. These estimations are evaluated in order to make good design decisions. The validation can be carried out after each phase, since before each design refinement, the product of each phase may be validated through simulation or considering a real prototype evaluation. Once systems having hardware and software components are considered, this methodology may require the interaction between different simulation environments in a process called co-simulation. In this case, the methodology should permit the concurrent utilization of several simulators.

2.1.2 Embedded Software

Over the last years, embedded systems have enlarged their functionalities and complexity, in such a way that the development time has become hard to predict and control. The increasing complexity along with evolving specifications has forced designers to consider implementations that can be easily modified. Because hardware manufacturing is expensive and time consuming, much attention has been given to embedded software recently. Moving functionalities from hardware to software has been possible since processors have increased computational power and decreased the size and cost.

According to [40], nowadays, software accounts for more than 80% of a embedded system development. Differently from PC-like software applications, embedded softwares may have to take into account several constraints such as reaction speed, memory footprint, power consumption, and so on. In this way, traditional software design methodologies can not be applied directly to the development of embedded softwares.

Problems with Embedded Software

In [40], Sangiovanni-Vincentelli and Martin present some problems related to the development of embedded software, which are summarized below.

Embedded software also has some drawbacks in spite of providing more flexibility than hardware. One of these drawbacks is related to performance, since hardware implementations can provide better response time than software. For dealing with performance issues, programmers usually adopt hand-crafted solutions using low-level programming languages, such as C or assembly. Nevertheless, these solutions may affect the time-to-market, readability, and maintainability of the resultant software. In addition, the tools utilized for developing embedded systems are basically the same as those for PC-like software applications. However, embedded software also needs hardware support for debugging and performance evaluation, which are more important for embedded software than PC-like software applications.

Another drawback is the increasing difficulty in verifying design correctness in embedded software. Such verification is critical, since several application domains, such as transportation and environment monitoring, are characterized by safety considerations. Moreover, little attention has been given to hard real-time constraints, memory utilization, and power consumption of embedded software.

Many companies have adopted object-oriented and other syntactically driven methods. Such methods are certainly very important for dealing with embedded software structure and documentation, but they are not sufficient for guaranteeing quality assurance and meeting time-to-market.

Embedded Software Design Methodology

Considering a simple embedded software implementation, there is no need for a sophisticated design method. Nevertheless, taking into account complex embedded software applications, primitive methods becomes a bottleneck.

Recently, much attention has been given to embedded software, more specifically, embedded software development methodologies. In [41], Sangiovanni-Vicentelli and Martin describe the challenges, which a software development methodology has to consider:

- reusing;
- hardware/software co-design;
- modeling non-functional properties;
- extensive use of software components;
- system and SW architecture;
- system level validation and verification;
- adoption of HW and SW reconfigurable architectures and component plug and play;
- composition of SW systems using reusable SW components;
- support for parallel development via integration technology
- development of process standards and common workflows.

Summarizing, Sangiovanni-Vicentelli and Martin envision a embedded software design methodology that *have an optimized, semi-automated, transparent, verifiable, and mathematically correct flow from product specification through to implementation for software-dominated products implemented with highly programmable platform.*

This section depicts the main stages of design (specification, refinement and decomposition, and analysis), implementation (target platform definition, mapping, and links to implementation) and verification, considering the software project perspective.

Specification

Specification is the entry point of the design process. It should contain three basic elements:

- description of the system functionality without implying an implementation. This description should be expressed using a formalism, in such a way that ambiguity is avoided;
- a set of constraints that have to be satisfied by the final implementation of the system. Example of constraints are weight, cost, size, and so on; and
- A set of design criteria. The difference between constraints and criteria is that constraints have to be met (e.g. maximum power dissipation allowed), while criteria you do your best to optimize it (e.g. higher autonomy).

Refinement and Decomposition

Once a specification is obtained, the design process should progress toward implementation via well-defined stages. The idea is to manipulate the description by introducing additional details while preserving the original functionality, its properties and meeting the constraints. Considering smaller steps, it is easier to formally prove that constraints are met and properties are satisfied. This process is called successive refinement. It is one of the main features of embedded software design methodology proposed in [40].

Moreover, it is often convenient to split parts of the design description in smaller parts so that optimization techniques might have a large design space and hence better chance of producing interesting results. This is called decomposition. The main idea is to determine whether the decomposed system satisfies the original specification The designer, however, should be concerned that larger design space demands longer evaluation.

Analysis

While advancing in the direction of the final implementation, designers may take decisions that lead to designs that do not satisfy some of the constraints. In this way, the adoption of tools is extremely important, in such a way that intermediate results are evaluated with respect to such constraints.

Target Platform Definition

Because most embedded systems are defined to map onto a target platform, it is essential to find the right specification form and notations with which a target platform can be described. When a platform offers re-configurable logic, new methods of describing the service and configuration are required.

Mapping

The mapping associates parts of the specification (generally refined) with specific implementation components of the target platform.

Link to Implementation

The implementation process comprises not only the selection of reusable components, but also generation of software for programmable components, generation of the static or dynamic configuration of reconfigurable hardware, and generation of the synthesis of appropriate hardware modules. Although the selection of the components could be done by hand, part of the work for obtaining an implementation should indeed be automatically done for minimizing human errors and for optimizing productivity.

Verification

This phase consists of verifying if the system is in accordance with the original specification. The adoption of formal specification makes easier the application of formal methods and may eliminate the need for verifying properties that are satisfied by construction. Moreover, automatic synthesis can reduce the need for implementation verification.

2.2 Real-Time Systems

There are several embedded systems for many different purposes, such as home appliances, guide missiles, medical devices, and so on. Some of these systems are time sensitive, in the sense that not only is the logical result of the computation important, but also the time in which the results are produced [46]. These systems are denominated *real-time systems*. Two approaches have been adopted for designing such systems: event-triggered and time-triggered. In event-triggered systems, the system promptly reacts to external events. On the other hand, time-triggered systems react to external events at predetermined time instants [25].

It is worth explaining that real-time computing is not fast computing. Fast computing aims to get the results as fast as possible, while real-time computing aims at obtaining the results within prescribed timing constraints.

2.2.1 Characteristics of Real-Time Systems

Real time systems are characterized by the timely response to external stimuli, predictable behavior, dependability or robustness, accuracy of the outputs, and concur-

rency [46]. This section overviews each characteristic.

Timely Response. The most important characteristic of a real-time system is that it must respond to some external stimuli within prescribed time constraints. Getting a correct output is not the only goal. This output must also be produced in a timely manner otherwise disastrous consequences may arise.

Predictability. Another requirement of real-time systems is that they must have predictable performance. Each execution of the system should run in a more or less similar manner, and one should be able to deterministically say when each of the tasks is executed.

Robustness. The system should be immune to minor changes in its state and should be able to run without degradation as when it was originally designed. In this way, machine overloads, execution delays, change in environment, and hardware failure should be dealt with in such a way that the overall system performance is not degraded.

Accuracy. Not only the system should be predictable and dependable, but it should also give accurate results. In case of most real-time systems, inaccurate results can be as bad as not meeting timing constraints and also can have serious consequences. Sometimes it is impossible to compute accurate results in the given timing constraints. In this way, a trade-off between computation time and accuracy results is very important.

Concurrency. Viewing a real-time system as a collection of concurrent process is quite common. Real-time systems should be distributed and provide parallel processing, since a system may have multiple sensors, each independent of the other, providing stimuli to the system and requiring response from the system within a given time frame.

Nevertheless, parallelism in real-time systems introduces additional complexities, such as: (i) parallel process must be scheduled correctly to meet timing constraints; (ii) synchronization between tasks in such environment may not be easy; (iii) communication models can introduce significant amount of overhead into the system; and (iv) the system is more susceptible to failures, since there are several processing units.

2.2.2 Types of Real-Time Systems

Real-time systems can be classified into two categories: hard real-time systems and soft real-time systems. The main difference between both types is the stringency of the predictability requirements.

Hard real-time systems require guaranteed predictable responses and behaviors. These systems are also called safety-critical real-time systems, since they are often used to control life-critical operations. In such cases, any deadline missing may result disastrous consequences, or even loss of human life. Moreover, any lateness in the execution of hard real-time tasks is not allowed under any circumstances, and such systems also have to employ a high degree of robustness and fault tolerance. Examples

of hard real-time systems are aircraft navigation, medical devices, nuclear power plant control, and so on.

Differently, soft real-time systems have a trade-off between computation time and the accuracy of the desired results. In such systems, if a timing constraint is not met, nothing critical happens, the system may only have its performance degraded. Examples of soft real-time systems are on-line transaction systems, electronic games, telephone switches, and so on.

2.2.3 Types of Real-Time Tasks

In real-time systems, there are, generally, three types of tasks:

- **Periodic** tasks perform a computation that are executed once in each fixed period of time;
- **Aperiodic** tasks are activated randomly;
- **Sporadic** tasks are executed randomly, but the minimum interval between two consecutive activations is known a priori.

In order to provide predictability in hard-real-time systems, the timing constraints of all tasks must be previously known, otherwise it is impossible to guarantee that all constraints will be met [56]. In this way, periodic and sporadic tasks are the most adopted tasks in hard real-time systems. The timing constraints of a periodic task is generally composed of periodicity, computation time (worst case execution time - WCET), deadline (the maximum time by which a task must finish its execution in a period) and release time (the earliest time that a task can start in a period). Timing constraints of sporadic tasks are composed of its computation time, deadline, and the minimum interval between two activations.

It is worth explaining that worst case execution time (WCET) of each task must be known previously, since its the only way to guarantee that each task finishes its execution before reaching its respective deadline [25]. The WCET is an upper bound for the time interval between task's activation and task's termination instants. This period (WCET) must be valid for all possible input data and execution scenarios of the task.

2.2.4 Specification and Verification of Real-Time Systems

As stated by Singhal [46], the fundamental challenge in the specification and verification of real-time systems is how to incorporate the time metric. Formal methods must be developed to incorporate these timing criteria into the specifications of a real-time

system specification. Additionally, verification methods must guarantee that these timing constraints are being met and that the system is robust, predictable and accurate. Such problem is made even more difficult in the face of concurrency issues inherently present in real-time applications.

Many formal methods for specifying, analyzing, and verifying real-time systems have been proposed over the years. Most of them have not being adopted due to two main reasons: (i) the difficulty of using such formalisms; and (ii) the lack of enthusiasm from the developers' community. Nevertheless, this should not take away from their importance in real-time systems, since many accidents can be avoided if formal methods are adopted [43, 46].

2.3 Scheduling in real-time systems

Real-time systems are generally composed of a set of concurrent tasks, where each task has its respective timing constraints. In hard real-time systems, these constraints must be met for the correct functioning of the system. In light of this consideration, scheduling plays an important role in software synthesis for hard real-time systems, since the code must be generated in order to provide predictable execution and resources use(e.g. buses, processors).

Scheduling is one of the most active research areas in real-time systems. Since this work considers only safety time-critical systems, this session presents a summary of the most relevant methods related to scheduling with stringent timing constraints.

In a broad sense, there are two scheduling approaches, named: runtime (also called dynamic or on-line), and pre-runtime (also called static or off-line). In addition to this classification, a scheduling method can also be characterized whether an executing task may be interrupted (e.g. if a more urgent task needs to be executed - *preemptive scheduling*) or not (*non-preemptive scheduling*). This section give an overview of runtime and pre-runtime approaches and, afterward, a comparison between them is presented.

2.3.1 Runtime Method

Runtime scheduling method make decisions at run time by selecting one of the current ready tasks for execution. The selection is based on priorities, in a such way that the task with higher priority is selected. For each task, a fixed priority is assigned.

Hereafter, a set of runtime method algorithms is concisely introduced: (i) Rate Monotonic Scheduling, (ii) Earliest Deadline First and (iii) Priority Ceiling Protocol. In the following presentation, it assumed that n is the number of tasks, c_i is the computation time, d_i is the deadline, and p_i is the period of task t_i.

Rate Monotonic Scheduling. In [31], Liu et. al proposed a dynamic preemptive algorithm based on static task priorities, namely, Rate Monotonic Scheduling (RMS). The approach makes the following assumptions about the task model: (i) all tasks are periodic and independent; (ii) the deadline of every task is equal to its period; (iii) the computation time of each task is previously known and it is constant; and (iv) the context-switching overhead is ignored.

In rate monotonic scheduling, the task with shorter period have the higher priority. Moreover, a schedulability analysis (verification whether a given schedule satisfies all deadlines) is performed before run-time:

$$U = \sum_{i=1}^{n} (c_i/p_i) \leq n(2^{1/n} - 1)$$

In other words, if the utilization factor U is less than $n(2^{1/n} - 1)$, the set of tasks is schedulable. In [28], Leung et al. extend RMS in order to permit deadlines to be less than or equal to periods. This extension is called Deadline Monotonic Scheduling (DMS).

Earliest Deadline First. The earliest deadline first (EDF) scheduling [31] is an dynamic scheduling algorithm based on dynamic priorities. The assumptions are the same as RMS. After any significant event, the task closest to its deadline (considering the current time) receives the highest priority, and it is selected for execution. According to EDF, a set of tasks is schedulable if:

$$U = \sum_{i=1}^{n} (c_i/p_i) \leq 1$$

Priority Ceiling Protocol. The Priority Ceiling Protocol (PCP) [43] is utilized to schedule a set of periodic tasks that may have critical sections protected by semaphores. It makes the same assumptions as RMS.

A set of n periodic processes using PCP can be scheduled by the rate-monotonic algorithm if the following condition is satisfied:

$$\frac{c_1}{p_1} + \frac{c_2}{p_2} + \cdots + \frac{c_n}{p_n} + \max\left(\frac{B_1}{p_1}, \cdots, \frac{B_{n-1}}{p_{n-1}}\right) \leq n(2^{1/n} - 1)$$

where B_i is the worst-case blocking time of task τ_i due to any lower priority process.

2.3.2 Pre-runtime method

Pre-runtime method performs scheduling decisions at compile time. It aims at generating a schedule table for a runtime component, namely, dispatcher, which is responsible for controlling the tasks during system execution. In order to adopt such method, the major characteristics of the tasks must be know in advance.

This approach can only be used to schedule periodic tasks. It computes a schedule considering the least common multiple (LCM) of the periods of the given set of tasks. Nevertheless, it is possible to translate a sporadic task to a periodic task using the technique described in [36, 56].

Several algorithms and techniques were proposed. In [55], Xu and Parnas propose an algorithm that adopts the branch-and-bound technique, where a large number of possible schedules are analyzed in order to find the optimal solution. That work is the first attempt to formalize a method of pre-runtime scheduling for real-time tasks with arbitrary exclusion and precedence relations. In [44], Shepard et al. extended Xu's approach in order to deal with multiprocessors. In the same way, Albelzaher and Shin [1] proposed an extension to Xu's algorithm for dealing with distributed real-time systems.

This work adopts the pre-runtime scheduling algorithm proposed in [8], which is based on depth-first search method. Such algorithm is described in Chapter 6.

2.3.3 Runtime versus Pre-runtime Scheduling

Runtime as well as pre-runtime method have advantages and disadvantages. However, pre-runtime method has become a more suitable approach for embedded hard real-time systems, since this method provides more predictable behavior than runtime approach. This section aims to provide a brief comparison between both scheduling classes. For more details, readers are referred to [57].

When adopting a runtime method, the amount of system resources required (e.g. memory) is much greater than a pre-runtime approach, since a schedule is computed entirely online. Moreover, the runtime scheduling takes time, which leads to overheads that directly affect the system predictability. In other words, tasks may miss their respective deadlines. On the other hand, pre-runtime methods compute the schedule in advance. When using this method, overheads are greatly reduced, since just a tiny dispatcher will be executing in addition to the real-time tasks.

Real-world time-critical applications are composed of several tasks with their respective timing constraints and inter-tasks relations (e.g. precedence and exclusion relations). In a runtime method, it is usually difficult to extend schedulability analysis to consider additional constraints, such as inter-tasks relations, because additional applications constraints are likely to conflict with existing priorities. In general, it is unfeasible to map application constraints into a fixed hierarchy of priorities. In contrast, a pre-runtime approach can compute an off-line schedule considering additional constraints without being restricted by any priority scheme.

The runtime scheduling approach requires complex run-time mechanisms for providing task synchronization and prevent simultaneous access to shared resources. In addition to the overheads, such mechanisms may conduct the system to a deadlock state. In contrast, in pre-runtime scheduling, there is no need of concern related to

deadlocks, since a feasible schedule is guaranteed to be deadlock-free whenever it is found.

Another drawback of runtime methods is that they have less chance of finding a feasible schedule than a pre-runtime scheduling algorithm. For instance, let us consider the task set consisting of two tasks, A, B, and the respective timing constraints (release, computation, and deadline): $A = (0, 10, 12)$; $B = (1, 1, 2)$. This specification also considers that B can not preempt A.

Figure 2.2(a) shows that a runtime method could not find a feasible schedule, since task B misses its deadline. However, a pre-runtime method finds a feasible schedule (Figure 2.2(b)). It is worth observing that the processor must be left idle between time 0 and 1, even though A's release time is 0.

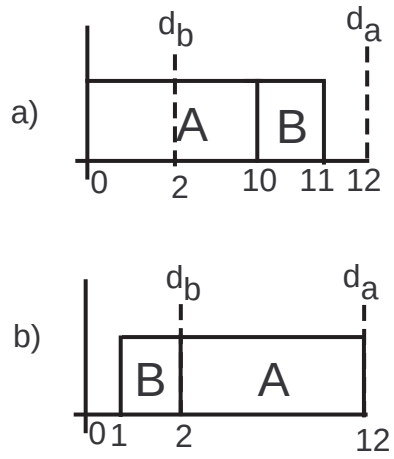

Figure 2.2: Comparison between runtime and pre-runtime scheduling

2.4 Summary

This chapter showed the main concepts needed for understanding this dissertation. Firstly, concepts related to embedded systems were presented. Since design methodologies play an important role in the development of an embedded system, an overview of hardware/software co-design was presented. Besides, some issues related to embedded software were described. Next, real-time systems were introduced, including characteristics and types. Lastly, scheduling in real-time systems was explained. In order to show the benefits and drawbacks of the scheduling methods (runtime and pre-runtime), a comparison was shown.

Chapter 3

Related Works

This chapter aims to show a summary of the relevant related works. Since this dissertation deals with software synthesis, this section presents some works that are related to code generation for single processor and multiple processors.

3.1 Code Generation for a Single Processor

Several works address the problem of code generation for embedded systems with a single processor. However, few works consider hard timing constraints.

Cornero et al. [12] present a software synthesis method for real-time information processing systems. As stated by the authors, such systems have the distinctive characteristic of the coexistence of two different types of functionalities: digital signal processing and control functions. In that work, the specification is composed of concurrent processes with their respective timing constraints, data dependencies and communications. Such specification is translated into a set of program threads, which may or may not start with a non-deterministic operation. Program threads are represented by a constraint graph model, where the nodes represent threads and the edges capture data dependency, control precedence and timing constraints between threads. Initially, constraint graphs are partitioned into disjoint clusters, called thread frames. Next, static scheduling is performed for determining the relative ordering of threads in the same thread frame. Lastly, the static information is used at runtime by the dynamic scheduler, whose aim is to combine different thread frames according to runtime system evolution. Although Cornero's work seems an interesting approach, the method can not be applied to hard real-time systems, since it considers non-deterministic delays. In safety time-critical systems, predictability is essential.

Lin [29] presents a software synthesis approach for implementing asynchronous process-based specifications without the need of a run-time scheduler. The specification is captured using a C-like programming language extended with mechanisms for

concurrency and communication. The extension is based on Communicating Sequential Process formalism, since it provides mechanisms not found in standard C language. Starting from the specification, an intermediary model based on Petri nets is generated in order to provide explicitly the ordering relations across process boundaries. Lastly, the software synthesis is performed using the intermediary model. The approach has some drawbacks when considering time-critical applications, since timing and resources constraints are not considered.

Balarin and Chiodo [6] present the software synthesis approach adopted in POLIS co-design framework [7]. In that work, the approach is focused on reactive embedded systems. In POLIS, systems are specified as networks of communicating processes, called Codesign Finite State Machines (CFMS), which are finite state machines with arithmetic and relational operators. Moreover, an intermediary data structure, namely, s-graph (software graph), is adopted to describe the reactive behavior. Such structure is translated into C code together with a simple Real-Time Operating System (RTOS), which is responsible for performing the scheduling (e.g. Rate-Monotonic or Deadline-Monotonic). Although the approach seems to be very interesting, the authors do not show how inter-task relations are dealt with, and no code example is shown.

Bokhnoven et al. [11] propose a software synthesis for system level design using process execution trees. The approach aims to translate a specification described in a process algebra language, namely, Parallel Object-Oriented Specification Language (POOSL), into an imperative programming language (C++). The process execution trees are adopted for capturing the real-time and concurrent behaviour of processes. The work proposed by Bokhoven et al. has some drawbacks. Firstly, it does not show how mutual exclusions are handled. In addition, nothing is said about preemption.

Sgroi et. al. [42] propose a software synthesis methodology based on a Petri net subclass, namely, Free Choice Petri Nets (FCPN). A FCPN model is adopted to represent a system specification, which is taken as input in a quasi-static scheduling algorithm for partitioning the model in a set of tasks. Basically, the algorithm decomposes the model in a set of conflict-free nets with the purpose of finding possible resolutions for each non-deterministic choice. Each resolution is represented by a cyclic firing sequence, which is used to compose a feasible schedule, in a such way that memory constraints are met. After obtaining a feasible schedule, a C code is synthesized by traversing the schedule and replacing transitions with the respective associated code. Although the approach seems very promising, it does not deal with real-time constraints, which are left to a real-time operating system (RTOS).

Su and Hsiung [47] extends the approach proposed by Sgroi et. al. [42], in the sense that they do not use free-choice Petri net, but a complex-choice Petri net (CCPN). As stated by the authors, CCPN models can describe more complex systems than FCPN, since CCPN models can have confusions, which are mixing of conflict and concurrent transitions. Additionally, the quasi-static scheduling algorithm was modified in order to cope with CCPN models. Briefly, a CCPN model is decomposed into conflict-free

subnets, in a such way that a finite complete cycle is constructed for each subnet. A conflict-free subnet is said to be schedulable if a finite complete cycle can be found for it and it is deadlock-free. Once all conflict-free subnets are scheduled, a valid schedule for the CCPN can be generated as a set of finite complete cycles. Starting from the schedule, a multi-threaded code is generated, including the synchronized access to variables that are utilized concurrently. Although the approach evolved the concepts presented by Sgroi [42], it is still not suitable for hard real-time systems because timing constraints are not considered in the scheduling process. Moreover, a generated code example is not presented.

Hsiung [19] proposes another extension of Sgroi et. al. work [42] considering timing constraints. The new approach adopts another model, namely, Time Free-Choice Petri Net (TFCPN), which is a Free-Choice Petri Net extended with time. As described by the author, the TFCPN time semantics is equal to time Petri net [34]. In that work, two scheduling are carried out: (i) quasi-static scheduling (for dealing with task generation with limited memory), and (ii) dynamic fixed-priority scheduling (for satisfying hard real-time constraints). Firstly, the quasi-static scheduling is performed, which is the same as one presented in [42]. For each finite complete cycle of the conflict-free subnets, the execution time interval is calculated by summing up all earliest firing time and latest firing time values, respectively, of each transition in the sequence. Among all the execution time intervals of conflict-free subnets, the maximum latest firing time is selected as the worst-case execution time of the TFCPN. In this way, a real-time scheduling algorithm, such as rate monotonic or deadline monotonic, may be used to schedule the TFCPN. Lastly, the code generation is performed. For each TFCPN, a real-time process is created. In each process, a task is created for each transition with independent firing rate. Although the approach deals with stringent timing constraints, there are drawbacks:(i) no real-world experiments are presented; and (ii) it is not shown how to add preemption in the proposed methodology.

Amnell et al. [3] propose a framework for the development of real-time embedded systems based on timed automata extended with a notion of real-time tasks. They describe how to compile from the design model to executable programs with predictable behavior. In addition, the approach rely on a real-time operating system for controlling tasks execution during system runtime. The framework utilizes a fixed-priority scheduling, which is well suited for independent tasks. However, such policy may not reach feasible schedules when considering arbitrary intertask relations (such as precedence and exclusion relations).

3.2 Code Generation for Multiple Processors

Few works aim to generate code for embedded systems with multiple processors. Additionally, just a subset of these works take into account hard real-time constraints.

Pino et at. [21] present the software synthesis approach adopted in Ptolemy. Ptolemy is an environment for simulation, prototyping and software synthesis for heterogeneous systems. The approach is based on object-oriented software methodology, and the code generation is focused on digital signal processors (DSP). The code generation framework utilizes a set of classes, which describes the system behavior and the features of the hardware architecture. Considering a multiprocessing environment, hardware resources essential for inter-processor communication, such as bus, are also taken into account in the scheduling. For an inter-processor communication, two tasks are generated: (i) send (which is responsible for transmitting the data at sender side) and (ii) receive (which gets the respective data at receiver side). Although the approach contains a scheduling phase, nothing is said about stringent timing constraints.

Altenbernd [2] presents CHaRy, a software system to support the synthesis of periodic controller applications, considering hard real-time constraints and parallel embedded computers. Essentially, CHaRy receives as input a description of a controller algorithm implemented by a C subset, and, next, it partitions the algorithm in several tasks using a task graph. Each vertex of the graph represents a program statement, whereas the edges represent the dependencies between the statements. Afterwards, a timing analysis process is started for estimating the worst-case execution time (WCET) of each task. The next step comprises the allocation of periodical tasks to processors using a method, namely, Slack Method. The processors are considered to be identical and interconnected using a point-to-point network. In order to verify timing constraints after the allocation, a schedulability analysis is performed using deadline monotonic scheduling (DMS). Additionally, the approach allocates one timer for each task during system execution. CHaRy seems a powerful tool for safety time-critical controller applications. However, DMS is a runtime policy, and, in this way, there are cases where a feasible schedule may not be found, even if one exits (Chapter 2). Moreover, nothing is said about exclusion relations.

Thoen et al. [48] propose a system model for real-time embedded software synthesis, namely, Multi-Thread Graph (MTG). The authors extend the approach proposed in [12] with new features, mainly, the support of multiple processors, data communication and multiple threads of control. A MTG model is composed of operation nodes (e.g. program threads, events) and control edges. Operation nodes have single control *entry* or *exit* points or both. A control edge $e_{i,j}$ between an *exit* point of an operation node o_i and *entry* point of an operation node o_j enforces the start of execution of o_j to be after the completion of o_i. In this way, control precedences and data dependencies are modeled. Additionally, each node has an attribute that specifies its execution time (δ_i). In that model, non-deterministic timing delays are also captured, which are related to external synchronization (e.g. wait for peripherals and wait for communication), and they are modeled as event nodes.In MTG models, all data communications are based on shared memory paradigm. The authors state that other communication paradigms (e.g. message passing) may be modeled applying some refinements, since

they utilize physical memory in some way. The approach presented by Thoen appears to be very interesting. However, it can not be applied to time-critical systems since non-deterministic timing delays are considered. Moreover, details about the code generation are not shown.

Böke [10] presents a software synthesis of real-time communication system for distributed embedded applications. The aim is the automatic synthesis of communication software for embedded systems from reliable software fragments. The specification is captured using a communication graph, which describes the communication behavior, and a resource graph, which describes the given hardware topology. Starting from the specification, a generator tool is adopted to provide the communication code for each CPU. Even though the approach is very promising, it does not seem suitable for hard real-time systems, since there is no scheduling phase (as stated by the author). Moreover, no real-world experiments are shown, and examples of the generated communication code are not depicted.

Kang et al. [24] present a software synthesis tool for designing distributed embedded systems. The authors concentrate on embedded systems that are dominated by throughput and latency requirements, such as digital signal processing systems. In addition, the approach takes into account off-the-shelf processors. In that work, stochastic time is considered, since each task has associated a discrete probability distribution function (PDF). In this way, the run-time system model assumes soft real-time constraints. The approach is not feasible for embedded hard real-time systems because timing constraints may not be met. In time-critical systems, timing correctness is fundamental.

Anand et al. [4] propose a code generation approach for distributed embedded systems from hybrid system models. Hybrid systems allow the state to change discretely as well as in a continuous manner. In that work, the modeling is performed using a modeling language called CHARON. Starting from a high-level model, a C++ code is generated and compiled along with the runtime support. Although the work seems a promising approach, there are some issues that do not allow the approach to be adopted in hard real-time systems. In that approach, the communication times are not taken into account. Futhermore, a schedulability analysis is not performed when considering a distributed environment, although, in a centralized environment, Earliest Deadline First scheduling (EDF) is adopted.

3.3 Summary

This chapter summarized the main works related with code generation. Several works address the problem of code generation for embedded systems with a single processor. However, few works consider hard real-time timing constraints. Besides, just a subset of these works take into account multiple processors. This research area has several

open issues, mainly related to generation of predictable scheduled code considering multiple processors.

Chapter 4

Petri Nets

This chapter presents an introduction to the formal model adopted in this dissertation, namely, Petri nets. First of all, one of the best studied class of Petri nets, namely, Place-Transitions nets, is presented, including the respective transition enabling and firing rules. Next, elementary nets and some Petri net subclasses are described. In order to show the practical utilization of Petri nets, some examples of models are shown. After that, time Petri net is presented, which is the extension adopted in this dissertation. Finally, this chapter shows some behavioral and structural properties that some Petri net models may contain.

4.1 Introduction

In the sixties, Carl Adams Petri [38] proposed the Petri net theory in his PhD thesis at Technical University of Darmstandt, Germany. Petri net can be defined as a mathematical formalism that allows specification and verification of systems. Initially, the theory aimed to model and analyze communication systems. Later, Petri's work came to the attention of the scientific community, and since then, the theory has been adapted and extended in several directions. Many extensions has been proposed to the original Petri net model in order to model and analyze different kinds of systems. The main extensions are inhibitor arcs, deterministic and stochastic timed nets [58], and high-level nets, such as object-oriented nets [50] and colored nets [22]. As described in [37], Petri net is adopted for modeling several kinds of systems, such as: discrete-event systems, multiprocessor memory systems, operating systems, manufacturing/industrial control systems and so on.

For the practical utilization of a formalism, a set of tools are essential for automatizing several tasks, such as modeling, analysis and verification. Petri net is a well-established formalism, and several tools are available for all Petri net extensions.

4.2 Place-Transition Net

Place/Transition Petri nets are one of the most prominent and best studied class of Petri nets. This class is sometimes called just Petri net [13].

Place/Transition Petri net is a bipartite directed graph, represented by a tuple $\mathcal{N}= (P, T, F, W, m_0)$ such that,

- $P = \{p_1,p_2,...,p_n\}$;
- $T = \{t_1,t_2,...,t_m\}$;
- $F \subseteq (P \times T) \cup (T \times P)$ is a flow relation for the set of arcs;
- $W : F \rightarrow \mathbb{N}^+$ is a weight function for the set of arcs;
- m_0 is the initial marking.

This class of Petri net consists of two kinds of nodes, called *places* (P) and *transitions* (T), such that $P \cap T = \emptyset$. Places represent local states and transitions represent actions. The set of arcs F is used to denote the places connected to a transition, and transitions to a place. W is a weight function, which assigns a weight to each arc in F. In this case, each arc is said to have *multiplicity* k, where k represents the respective weight of the arc. if $k > 1$, the arc weight can be interpreted as the set of k parallel arcs. A marked Petri Net contains *tokens*, which reside in places, and their flow through the net is regulated by transitions.

Figure 4.1 depicts graphically the basic components of a Petri Net. Places (a) are represented by circles, transitions (b) are depicted as bars or rectangles, arcs (c) are represented by arrows, and (d) tokens are generally represented by filled small circles.

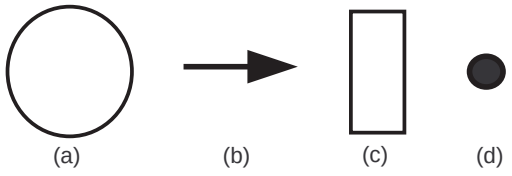

Figure 4.1: The Basic Components of a Petri Net: (a) place, (b) arc, (c) transition, and (d) token

Places and transitions may have several interpretations. Using the concept of conditions and events, places represent conditions, and transitions represent events, such that, an event may have several pre-conditions and post-conditions. For more interpretations, Table 4.1 shows other meanings for places and transitions [37].

Table 4.1: Interpretation for places and transitions

input places	transitions	output places
pre-conditions	events	post-conditions
input data	computation step	output data
input signals	signal processor	output signals
resource granting	tasks	resource releasing
conditions	logical clauses	conclusions
buffers	processor	buffers

The set of reachable markings is denoted by $m = \{m_0, m_1, \cdots, m_i, \cdots\}$, where m_0 represents the initial marking. Although the definition of reachable marking set may have infinite markings, in the context of this dissertation it is assumed that this set is finite. A marking m_i of a Petri net is an assignment of tokens to the places in that net. The vector $m_i = (m_{i_1}, m_{i_2}, \cdots, m_{i_n})$ gives, for each place in the Petri net, the number of tokens in that place at respective marking m_i. Therefore, the number of tokens in place p_j at marking m_i is m_{i_j}, for $j = 1, ..., n$. It may also be defined a marking function $m_i : P \rightarrow \mathbb{N}$, from the set of places to the natural numbers. This allows using the notation $m_i(p_j)$ to specify the number of tokens in place p_j at marking m_i. In this case, for a marking m_i, $m_{i_j} = m_i(p_j)$. In this dissertation both notations are used interchangeably.

The set of input transitions (also called pre-set) of a place $p_i \in P$ is:

$$\bullet p_i = \{t_j \in T \mid (t_j, p_i) \in F\}$$

and the set of output transitions (also called post-set) is:

$$p_i \bullet = \{t_j \in T \mid (p_i, t_j) \in F\}$$

The set of input places of a transition $t_j \in T$ is:

$$\bullet t_j = \{p_i \in P \mid (p_i, t_j) \in F\}$$

and the set of output places of a transition $t_j \in T$ is:

$$t_j \bullet = \{p_i \in P \mid (t_j, p_i) \in F\}$$

4.2.1 Transition Enabling and Firing

The behavior of many systems can be described in terms of system states and their changes. In order to simulate the dynamic behavior of a system, a state (or marking) in a Petri net is changed according to the following transition firing rule:

1. A transition t is said to be *enabled*, if each input place p of t is marked with at least the number of tokens equal to the multiplicity of its arc connecting p with t. Using a mathematical notation, an enabled transition t for given marking m_i is denoted by $m_i[t >$, if $m_i(p_j) \geq W(p_j, t), \forall p_j \in P$.

2. An enabled transition may or may not fire. It depends on whether or not the respective event takes place.

3. The firing of an enabled transition t removes tokens (equal to the multiplicity of the input arc) from each input place p, and adds tokens (equal to the multiplicity of the output arc) to each output place p'. Adopting a mathematical notation, the firing of a transition is represented by the equation $m_j(p) = m_i(p) - W(p, t) + W(t, p), \forall p \in P$. If a marking m_j is reachable from m_i by firing a transition t, it is denoted by $m_i[t > m_j$.

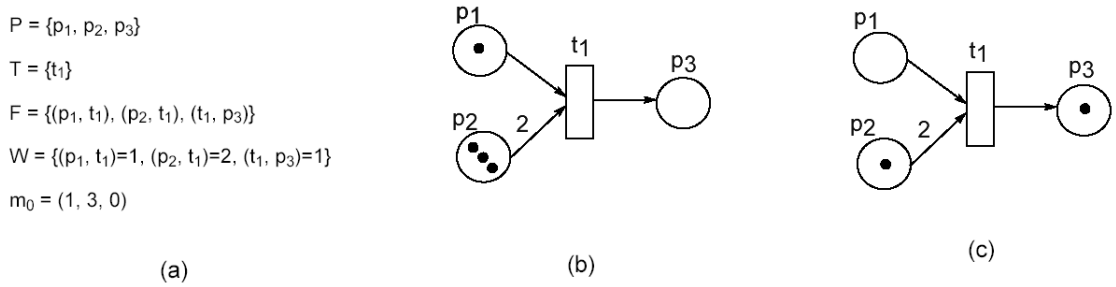

Figure 4.2: Petri net. (a) Mathematical formalism; (b) Graphical representation before firing of t_1; (c) Graphical representation after firing of t_1

Figure 4.2(a) shows a Petri net mathematical formalism for a model with three places and one transition. Figure 4.2(b) outlines its respective graphical representation, and 4.2(c) provides the same graphical representation after the firing of t_1. For this example, the set of reachable markings is $m = \{m_0 = (1, 3, 0), m_1 = (0, 1, 1)\}$. m_1 was obtained by firing t_1, such that, $m_1(p_1) = 1 - 1 + 0$, $m_1(p_2) = 3 - 2 + 0$, and $m_1(p_3) = 0 - 0 + 1$.

A transition without any input place is called a *source* transition, and one without any output place is called a *sink* transition. A source transition is unconditionally enabled, and the firing of a sink transition consumes tokens, but does not produce any. Figure 4.3 shows source and sink transitions before and after the respective firing. A pair of a place p and transition t is called a *self-loop* if p is both an input and output place of t. A Petri net is said to be *pure* if it has no self-loops. Figure 4.4 depicts a self-loop.

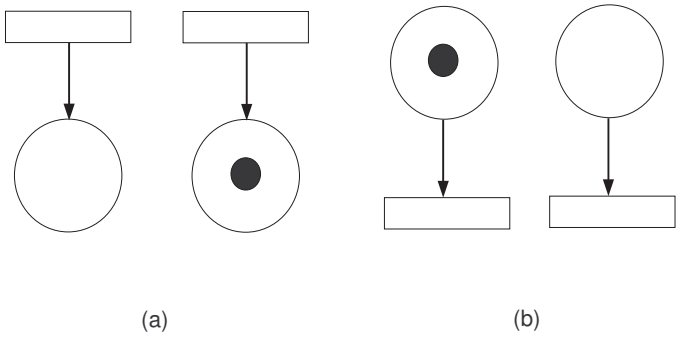

Figure 4.3: Source and sink transition before and after the firing

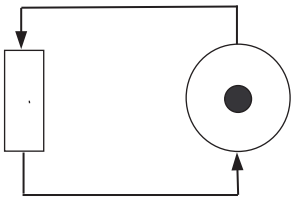

Figure 4.4: Self-Loop

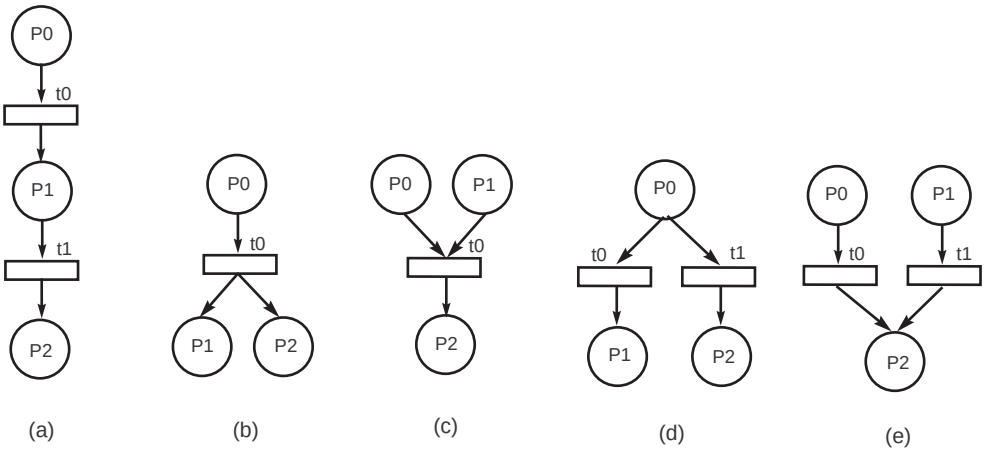

Figure 4.5: Elementary Structures

4.2.2 Elementary Nets

Elementary nets are used as building blocks in the specification of more complex applications. Figure 4.5 shows five structures, namely, (a) sequence, (b) fork, (c) synchronization, (d) choice, and (e) merging.

Sequence

The sequence is a structure that represents sequential execution of action execution, provided that a condition is satisfied. After the firing of a transition, another transition is enabled to fire. In Figure 4.5(a) a mark in place p_0 enables transition t_0, and with the firing of this transition, a new condition is established (p_1 is marked). This new condition allows the firing of transition t_1.

Fork

This net allows the creation of parallel processes. As it can be seen in Figure 4.5(b), the firing of transition t_0 removes the token of place p_0, and adds one token in place p_1 and another in place p_2. Considering that places p_1 and p_2 are pre-conditions of two distinct processes, the new state allows that both processes can execute in parallel.

Synchronization (or Join)

Generally, concurrent activities need to synchronize with each other. This net (Figure 4.5(c)) combines two or more nets, allowing that another process continues its execution only after the end of predecessor processes.

Conflict (or Choice)

If two (or more) transitions are in conflict, the firing of one transition disables the other(s). As you can see in Figure 4.5(d), the firing of transition t_0 disables transition t_1. This building block is suited for modeling `if-then-else` statement.

Merging

The merging is an elementary net that allows the enabling of the same transition by two or more processes. In the case of Figure 4.5(e) the two transitions (t_0 and t_1) are independent, but they have an output place in common. Therefore, after the firing of any of these two transitions, a condition is created (p_2 is marked) which may allow the firing of another transition (not shown in the figure).

Confusions

The mixing between conflict and concurrency is called *confusion*. While conflict is a local phenomenon in the sense that only the pre-sets of the transitions with common input places are involved, confusion involves firing sequences. Two types of confusions are shown in Figure 4.6: (a) *symmetric confusion*, where two transitions t_1 and t_3

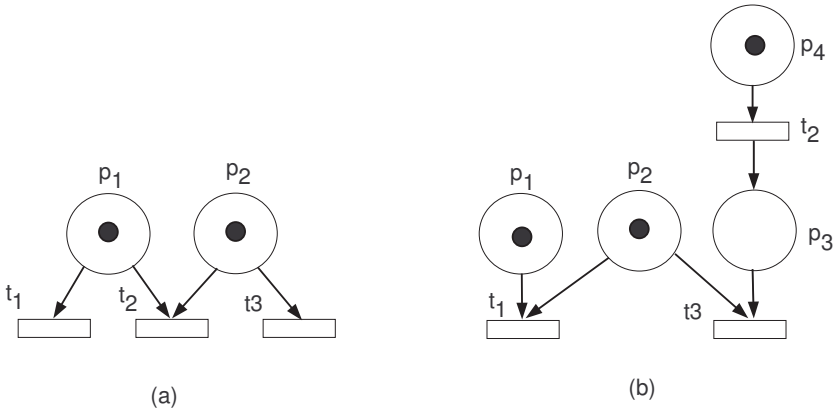

Figure 4.6: Confusions. (a) symmetric confusion; (b) asymmetric confusion

are concurrent while each one is in conflict with transition t_2; and (b) *asymmetric confusion*, where t_1 is concurrent with t_2, but will be in conflict with t_3 if t_2 fires first.

4.2.3 Petri Net Subclasses

Net subclasses is defined exclusively by introducing constraints on the structure of the nets [45]. By restricting the generality of the model, it may improve the study of its behavior. In particular, powerful structural results allow us to fully characterize some properties, such as liveness and reversibility.

Based on [37], let us introduce five important subclasses depicted in Figure 4.7.

State Machine

In this subclass (*state machine*-SM) (Fig. 4.7(a)) each transition has just one input and output place, i.e.,

$$|\bullet t| = |t \bullet| = 1 \qquad \text{for all } t \in T.$$

State machines can represent conflict and merging structures, but not fork and synchronization. Several properties are obvious in this Petri net class. For instance, the number of tokens are always the same (conservative property), which results in a finite system.

Marked Graph

The subclass called *marked graph* (MG) (Fig. 4.7(b)) restricts each place p to have exactly one input transition and one output transition, i.e.,

$$|\bullet p| = |p \bullet| = 1 \qquad \text{for all } p \in P.$$

Marked graphs can represent concurrency and synchronization, but not conflict and merging structures.

Free-Choice Petri Nets

The *free-choice* (FC)) (Fig. 4.7(c)) is a Petri net such that every arc from a place is either a unique outgoing arc or a unique incoming arc to a transition, i.e.,

$$p_1 \bullet \cap p_2 \bullet \neq \emptyset => |p_1 \bullet| = |p_2 \bullet| = 1 \qquad \text{for all } p_1, p_2 \in P.$$

In other words, a place may be input for several transitions, however, it is the only input for these transitions. Free-choice allows the modeling of conflict as well as modeling concurrency and synchronization. However, this subclass is more restricted when compared with general Petri nets, since when a conflict exists either all conflicting transitions are enabled or not. Therefore, the choice is made freely.

Extended Free-Choice Petri Nets

Extended free-choice nets (EFC) (Fig. 4.7(d)) extend free-choice nets allowing more complex conflict structures. EFC models the conflict of two or more transitions even if they have more than one input places. However, in such case, the input set of each of these conflicting transitions should be the same, i.e.

$$p_1 \bullet \cap p_2 \bullet \neq \emptyset => p_1 \bullet = p_2 \bullet \qquad \text{for all } p_1, p_2 \in P.$$

Asymmetric Choice (or Simple Net)

An *asymmetric choice* (AC) (Fig. 4.7(e)) is a Petri net such that

$$p_1 \bullet \cap p_2 \bullet \neq \emptyset => p_1 \bullet \subseteq p_2 \bullet \ \textit{ or } \ p_1 \bullet \supseteq p_2 \bullet \qquad \text{for all } p_1, p_2 \in P.$$

In other words, asymmetric choice nets allow that each transition has at most one input place shared with other transitions. The typical basic example of an asymmetric choice net is the model of a system in which a resource is shared by two or more processes [45].

In summary, SMs admit no synchronization, MGs admit no conflict, FCs admit no confusion, and ACs allow asymmetric confusion (Fig. 4.6(b)), but disallow symmetric confusion (Fig. 4.6(a)) [37].

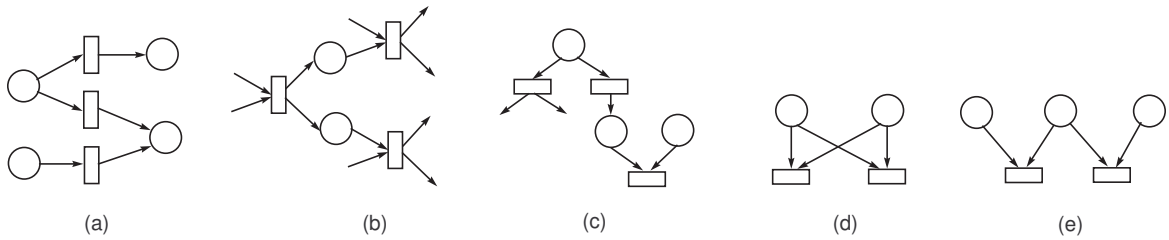

Figure 4.7: Five fundamental Petri net subclasses

4.3 Modeling with Petri Nets

This section shows some classical problems and their respective Petri net models. These models are represented by using elementary net structures presented in previous section.

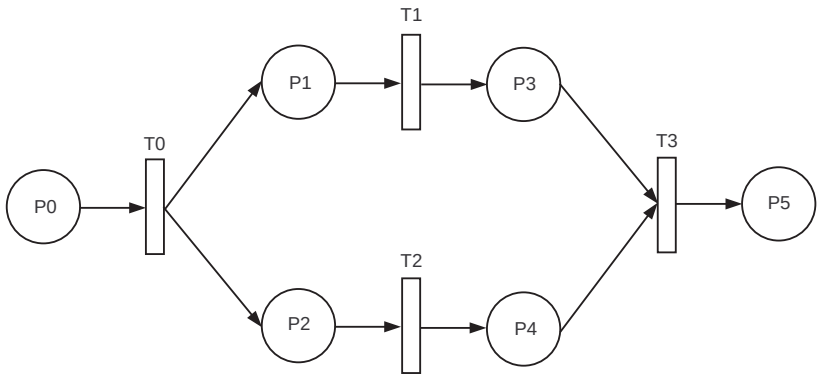

Figure 4.8: Transitions T_1 and T_2 represents parallel activities

4.3.1 Parallel Processes

In order to represent parallel processes, a model may be obtained by composing the model for each individual process with a fork and synchronization models. Two transitions are said to be parallel (or concurrent), if they are *causally independent*, i.e., one transition may fire either before (or after) or in parallel with the other.

Figure 4.8 shows an example of parallel activity, where transitions t_1 and t_2 represent parallel activities. When transition t_0 fires, it creates marks in both output places (p_1 and p_2), representing a concurrency. When t_1 and t_2 are enabled for firing, they may fire independently. The firing of t_3 depends on two pre-conditions, p_3 and p_4, implying that the system can only continue whether t_1 and t_2 have been fired.

4.3.2 Mutual Exclusion

Some applications require sharing of resources or data or both. Most of resources and data should be accessed in a mutual exclusive way. Usually, the resource (or data variable) is modeled by a place with tokens representing the amount of resources. This place is seen as pre-conditions for all transitions that need this resource. After the use of the resource, it must be released.

Figure 4.9 shows an example of a machine accessed in a mutual exclusive way.

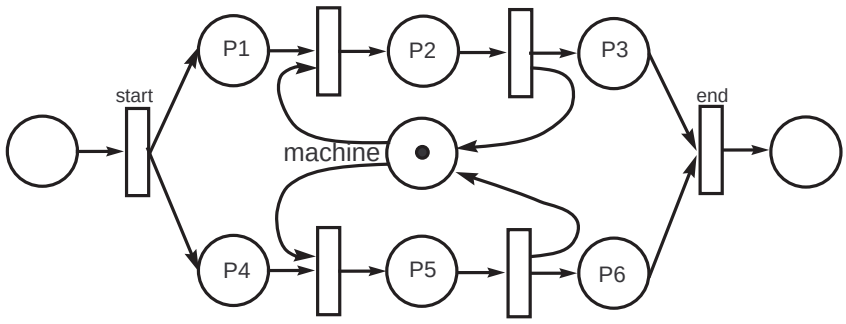

Figure 4.9: Mutual Exclusion

4.3.3 Communication Protocols

Communication protocols are another area where Petri nets have been widely used to represent and specify systems' features as well as analysis of properties.

Communicating entities may be modeled in several ways: (i) a single transition representing the communication (Fig. 4.10(a)); (ii) the explicit representation of message flow (Fig. 4.10(b)); or (iii) representing the sending of message and the rspective acknowledgment (Fig. 4.10(c)).

4.3.4 Producer-Consumer

The producer-consumer problem represents two kinds of processes: producers and consumers.

Producer process generates objects that are stored in a buffer. A consumer process waits until one (or more) object is stored in the buffer in such a way that it can consume such an object. The net that models the producer-consumer problem is depicted in Figure 4.11, where we can see the producer, the consumer, and the buffer. The number of tokens in p_0 and p_2 indicate the number of producers and consumers, respectively. Transition t_0 represents production of items and transition t_1 the storage of this item

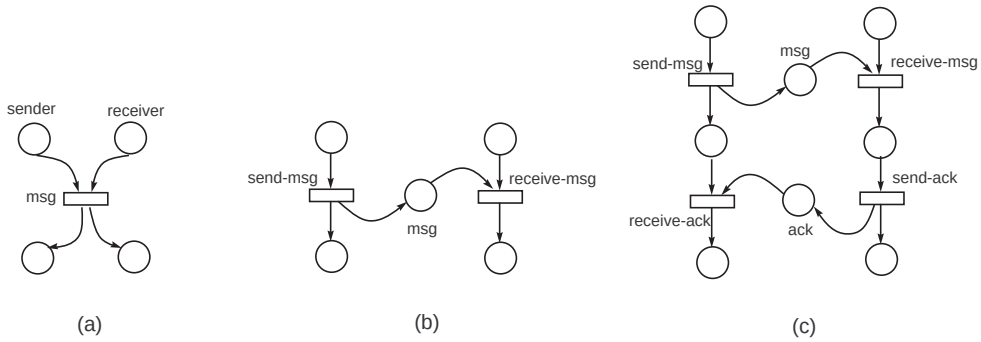

Figure 4.10: Communication Protocols

into the buffer. The same way, transition t_2 represents the item removal from the buffer by the consumer, and t_3 the consumption of the item.

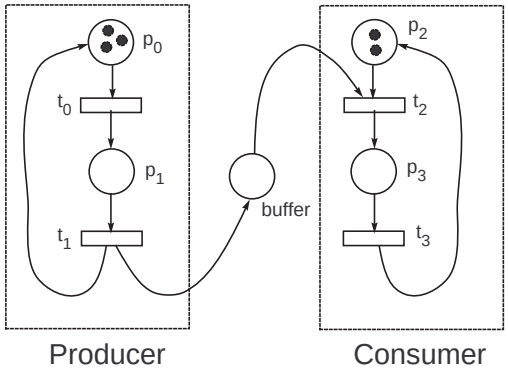

Figure 4.11: Producer/Consumer

4.3.5 Dining Philosophers

The dining philosophers is a classical problem that was proposed by Dijsktra in [14]. Briefly, three philosophers are arranged in a ring with one fork (resource) between each pair of neighbors, and for eating, a philosopher must have exclusive access to both of its adjacent forks. If all philosophers take at the same the right fork and wait the left fork to be free, the system will enter a *deadlock* state.

In Figure 4.12 it is presented a solution for this problem. It is represented the resources (forks) by marks in the places fork_1, fork_2, and fork_3. The state of each philosopher is represented by the places eating (pc_i), hungry (pcf_i), and thinking(pp_i). The event start-to-eat is represented by transition tcc_i, as well as, is-hungry, and start-to-think are represented by ttf_i and tcp_i, respectively.

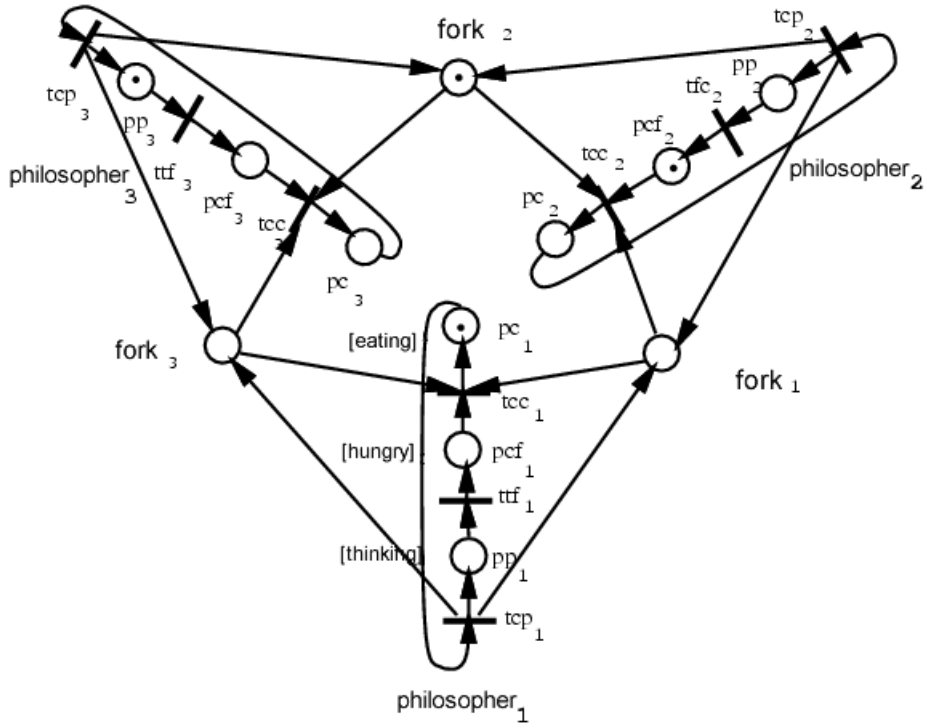

Figure 4.12: Dining Philosophers

4.4 Time Extensions

Originally, the Petri net theory did not deal with time, since the aim was only the logical behavior of systems by describing the causal relations between events. However, time is essential in many systems because some applications have requirements not only related to logical correctness, but also associated to the time at which results are produced. The introduction of time into Petri nets allows the modeling of real-time controls, communication protocols, and so on, where timing is essential for assuring that systems are correct.

When introducing time into Petri net models, it should not modify the basic behavior of the underlying untimed model. In this way, it is possible to study the Petri net extended with time exploiting the properties of the basic model as well as the available theoretical results.

There are different ways for incorporating timing in Petri Nets. Time may be associated with places, tokens, arcs, and transitions. Since transitions represent activities that change the state (marking) of the net, it seems natural to associate time to transitions.

The firing of a transition in a Petri net model corresponds to an event that changes

the state of the real system. There are two different firing policies:

- Three-phase firing: firstly, the tokens are consumed from input places when the transition is enabled, then a delay elapses; finally, tokens are generated in output places. Such delay is called *duration*;
- Atomic firing: when the transition is enabled, tokens remain in input places until the delay elapses; after that period they are consumed from input places and generated in output places when the transition fires. The firing itself does not consume any time.

The memory policies represent the way transitions are affected whenever a transition fires [33]:

- Resampling: the timers of all transitions are discarded (restart mechanism). No memory of the past is recorded. New values of timers are set for all enabled transitions at a new marking;
- Enabling memory: the timers of transitions that are disabled are restarted whereas the timers of all transitions that are not disabled hold their present value.
- Age memory: at each transition firing, the timers of all transitions hold their present values (continue mechanism).

Several time extensions have been proposed and adopted by the research community. This section only describes the extension adopted in this work, namely, time Petri net.

Time Petri net [34] is defined by (PN, I), where PN is an underlying Petri net, and I is a deterministic time interval expressing timing constraints, such that each transition t_i has associated a time interval $I_i = (EFT_i, LFT_i)$. EFT stands for *earliest firing time* and LFT stands for *latest firing time*. This non-negative interval expresses the minimum and maximum time for firing the respective transition. The firing policy adopted is the atomic firing.

An enabled transition t_i may only fire in the interval $EFT_i \leq \delta \leq LFT_i$, that is, t_i must be continuously enabled for at least EFT_i time units. But what happen when a transition t_i is enabled for LFT_i time units? The firing mode concept is related to this issue.

There are two firing modes: strong and weak firing modes. Consider that transition t_i is enabled at time θ. According to the *strong firing mode*, a transition is forced to fire at time $\theta + LFT_i$, if t_i has not fired and has not been disabled by other transition firing. The *weak firing mode*, on the other hand, does not force an enabled transition to fire, that is, an enabled transition may or may not fire.

The reader should note that time Petri nets are equivalent to the standard Petri nets if all $EFT = 0$ and all $LFT = \infty$. It is also important to note that the set of reachable markings of the time Petri nets is either equal to or a subset of the equivalent untimed model. This is true because the enabling rules for the timed model are the same for the untimed model. The only difference is due to the timing restrictions imposed on the firing rules. Thus, the time information may restrict the set of reachable markings, but never increase it.

4.5 Petri nets Properties

Petri nets are not just a modeling tool for describing systems. A major strength of Petri nets is their support for analysis of many interesting properties. Two types of properties can be studied with a Petri net model: behavioral properties (those which depend on the marking) and structural properties (those which do not depend on the marking).

4.5.1 Behavioral Properties

This section, based on [37], describes behavioral properties.

Reachability

A marking m_i is said to be reachable from marking m_0 (initial marking), if there exists a sequence of firings that transforms m_0 to m_i. A firing (or occurrence) sequence is denoted by $\sigma = t_1 t_2 \cdots t_i$. In this case, m_i is reachable from m_0 by σ. It is denoted by $m_0[\sigma > m_i$. The set of all possible reachable markings from m_0 in a net (PN, m_0) is denoted by $R(PN, m_0)$, or simply $R(m_0)$. The set of all possible firing sequence from m_0 in a net (PN, m_0) is denoted by $L(PN, m_0)$, or simply $L(m_0)$.

It has been shown [30] that the reachability problem is decidable, although it needs exponential space for system verification in the general case.

Boundedness and Safeness

A Petri net is said to be *k-bounded* (or simply *bounded*) if the number of tokens in each place does not exceed a finite number k for any reachable marking from m_0. A Petri net is said to be *safe* if it is *1-bounded*.

Places in a Petri net are often used to represent buffers for storing intermediate data. By verifying that a net is bounded (or safe), it is guaranteed that there will be no overflows in the buffers, no matter what firing sequence is taken.

Liveness

A Petri net is said to be *live* if, no matter what marking has been reachable from m_0, it is possible to fire any transition of the net by progressing through some further firing sequence. This means that a live Petri net guarantees deadlock-free operation, no matter what firing sequence is chosen. In Petri nets, deadlock means that all transitions are unable to fire.

Liveness is an ideal property for many real systems. However, it is impractical and too costly to verify this strong property for some systems. Thus, the liveness condition is relaxed in different levels. A transition t is said to be live at the following levels:

- L0-Live (dead), if t can never be fired in any firing sequence in $L(m_0)$.
- L1-Live (potentially firable), if it can be fired at least once in some firing sequence in $L(m_0)$.
- L2-Live if, given any positive integer k, t can be fired at least k times in some firing sequence in $L(m_0)$.
- L3-Live if there is an infinite-length firing sequence in $L(m_0)$ in which t is fired infinitely.
- L4-Live (or simply live), if it is L1-Live for every marking m in $R(m_0)$.

A net is classified as live at level i, if every transition is live at the same level i. It is worth noting that transition live at level 4, is also live at levels 3, 2, 1.

Reversibility and Home State

A Petri net is said to be *reversible* if, for each marking (or state) m in $R(m_0)$, m_0 is reachable from m. Thus, in a reversible net one can always get back to the initial marking (or state). This property is very important mainly in the context of control systems.

In many applications, however, it is not necessary to get back to the initial state as long as one can get back to some (home) state. Therefore, the reversibility condition is relaxed in such a way that the net can always get back to another marking m_k (where $m_k \neq m_0$). Marking m_k is defined as home state.

Coverability

Coverability is closed related to reachability. A marking m is said to be *coverable*, if there exists a marking m' in $R(m_0)$ such that $m'(p) \geq m(p)$, for each place p in the Petri net. If a marking m' covers marking m, it means that m may be reached from m'.

Persistence

A Petri net is said to be *persistent* if, for any two enabled transitions, the firing of one transition will not disable the other. A transition in a persistent net, once it is enabled, will stay enabled until it fires. Persistency is closed related to conflict-free nets. It is worth noting that all marked graph are persistent, but not all persistent nets are marked graphs. Persistence is a very important property when dealing with parallel system design.

Fairness

Petri net literature presents many different points of view about the fairness concept (e.g.[9, 49, 37]). This section presents two of them: *bounded-fairness* and *unconditional-fairness*. According to the bounded-fairness (B-fair) concept, two transitions t_1 and t_2 are said to be bounded if the maximum number of times that one fires, while the other does not fire, is bounded. A Petri net is said to be a *B-fair net* if every pair of transitions in the net are in a B-fair relation.

A firing sequence σ is said to be *unconditionally* (or *globally*) *fair* if it is (i) finite; or (ii) every transition in the net appears infinitely often in σ. A Petri net is said to be *unconditionally fair net* if every firing sequence σ from m in $R(m_0)$ is unconditionally fair.

4.5.2 Structural Properties

This section, based on [32], aims to describe structural properties.

Structural Boundedness

A net is bounded if the bound of each of its places is finite for a given initial marking. A net is *structurally bounded* if it is bounded for any initial marking.

Conservation

A Petri net is said to be conservative if any transition firing does not change the number of tokens. In other words, resources are neither created nor destroyed. In this case, the net is called *strictly conservative*.

However, this property is not only restricted to conservation of the number of tokens. There exist nets that are not classified as strictly conservative, but they can be converted into strictly conservative nets. Such nets are said to be conservative.

One token in one place may represent several resources that may later be used to create multiple tokens by firing a transition with more output arcs than input arcs.

These nets may provide a weighted sum of tokens for all reachable markings of the net. A conservative net is one in which the weighted sum of tokens is constant.

Repetitiveness

A marked net is classified as *repetitive* if there is an initial marking m_0 and a sequence σ from m_0 such that every transition of the net is infinitely fired. If only some of these transitions are fired infinitely often in the sequence σ, this net is called *partially repetitive*.

Consistence

A net is classified as consistent if there is an initial marking m_0 and an enabled firing sequence from m_0 back to m_0 such that every transition of the net is fired at least once. If only some of these transitions are not fired in the sequence σ, this net is called *partially consistent*.

4.5.3 Coverability (Reachability) Tree

Given a Petri net, from the initial marking m_0 many new markings can be obtained due to the firing of enabled transitions. This process results in a tree representation of the markings, namely, coverability tree. Nodes represent markings generated from m_0 (root) and its successors, and each arc represents a transition firing, which transforms one marking to another. However, such tree representation will grow infinitely large if the net is unbounded. To keep this tree finite, a special symbol ω, is introduced which can be thought of "pseudo-infinite", which represents a number of tokens that can be made very large. Therefore, for any integer n, $\omega > n$, $\omega + n = \omega$, $\omega - n = \omega$, and $\omega \geq \omega$. The coverability tree can be built using the following algorithm [37]:

1. Label the initial marking as the "root" and tag it as "new";
2. `While` "new" markings exist `do`:
   - 2.1. Select a "new" marking M ;
   - 2.2. `If` no transitions are enabled at M, tag M as "dead-end";
   - 2.3. `If` M is identical to a marking on the path from the root to M, label M as "old" and go to another "new" marking;
   - 2.4. `For` all transitions enabled at M `do`:
     - 2.4.1. Obtain the marking M' by firing a transition t enabled at M;

2.4.2. `If` from the "root" to M there exists a marking M'' such that $M'(p_i) \geq M''(p_i)$ for each place p_i and $M' \neq M''$ then replace $M'(p_i)$ by ω wherever $M'(p_i) > M''(p_i)$;

2.4.3. Introduce M' as a node, labeling the arc with t and tag M' as "new".

For a bounded Petri net, the coverability tree is called reachability tree (reachability graph) since it contains all possible reachable markings. Using the coverability (reachability) tree method, several properties can be analyzed, for instance, boundedness, safeness, and reachability.

4.6 Summary

Petri nets are well-established tools for modeling and analyzing several kinds of systems, such as concurrent and parallel systems. This chapter introduced several concepts related to this subject, from the basic aspects to the properties analysis. Additionally, an extension of Petri nets was presented, namely, time Petri net, which is adopted in this dissertation. More details about time Petri net are described in the next chapter.

Chapter 5

Modeling Embedded Hard Real-Time Systems

Modeling is defined as the process of creating a representation of objects considering only characteristics of interest. This chapter aims to present the method adopted for modeling embedded hard real-time systems. The modeling method utilizes time Petri net as the computational model.

This work is interested in the modeling of specifications composed of inter-processor communications and hard real-time tasks, more specifically, their timing constraints, inter-task relations, scheduling method, and allocated processor. This dissertation extends the modeling method proposed in [8] with inter-processor communications. In this way, a new building block is presented for representing such communications.

5.1 Formal Model

This section presents the computational model that assures timing constraints. As stated before, time Petri net is being adopted. In the following subsections, concepts related to the formal model are described.

Definition 5.1 (Time Petri Net) *A time Petri net (TPN) is a tuple* $\mathcal{P} = (\mathcal{N}, I)$*, where* $\mathcal{N}$ *is the underlying marked Petri net, and* $I : T \rightarrow \mathbb{N} \times \mathbb{N}$*, is a bounded static firing interval that represents the timing constraints, such that* $I(t) = (EFT(t), LFT(t))$ $\forall t \in T$ *and* $EFT(t) \leq LFT(t)$*.*

This definition is an extension of ordinary Petri nets in which time intervals are attached to transitions. This extension introduces the static firing interval $I(t)$ associated with each transition $t \in T$. $I(t)$ is the allowed timing interval for the respective transition firing. The lower and upper bound of $I(t)$ are called *earliest* and *latest firing*

time, respectively. EFT is the minimal time that must elapse, starting from the respective transition enabling, until this transition can fire. LFT denotes the maximum time during which the respective transition can be enabled without being fired.

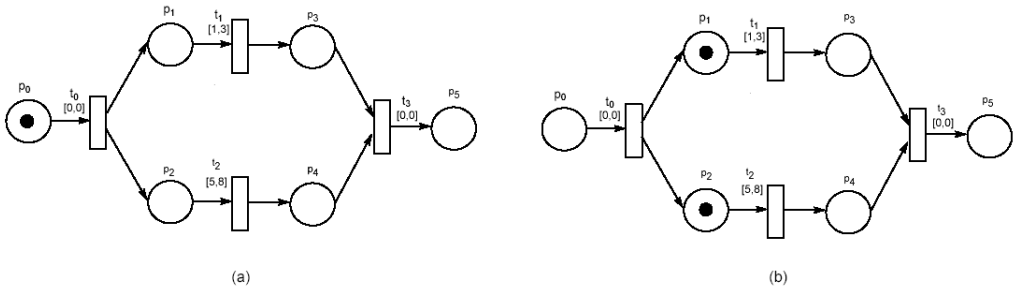

Figure 5.1: A Simple Example of Time Petri Net: (a) initial marking (m_0); (b) new marking after firing of t_0

Figure 5.1(a) shows a simple time Petri net, where:

- $P = \{p_0, p_1, p_2, p_3, p_4, p_5\}$;
- $T = \{t_0, t_1, t_2, t_3\}$;
- $F = \{(p_0, t_0),\ (t_0, p_1),\ (t_0, p_2),\ (p_1, t_1),\ (p_2, t_2),\ (t_1, p_3),\ (t_2, p_4),\ (t_3, p_5)\}$;
- $W(x, y) = 1,\ \forall (x, y) \in F$;
- $m_0(p_0) = 1,\ m_0(p_i) = 0,\ 1 \leq i \leq 7$; and
- $I = \{I(t_0), I(t_1), I(t_2), I(t_3)\}$, where $I(t_0) = (0, 0), I(t_1) = (1, 3), I(t_2) = (5, 8)$, and $I(t_3) = (0, 0)$.

Firing intervals $I(t)$ of some transitions may be equal to (0,0), which means that the respective delay from the instant in which the transition became enabled and its firing is 0.

Definition 5.2 (Enabled Transition Set) *Let $\mathcal{P}$ be a time Petri net, and m_i a reachable marking. The set of enabled transitions at marking m_i is denoted by:*

$$ET(m_i) = \{t \in T \mid m_i(p_j) \geq W(p_j, t)\},\ \forall p_j \in P.$$

Taking into account the net depicted in Figure 5.1(a), only transition t_0 is enabled at the initial marking m_0.

Definition 5.3 (Clocks) *A clock is defined by* $c_i : ET(m_i) \rightarrow \mathbb{N}$*, where* c_i *is a clock function (or vector) , which represents the time elapsed since the respective transition enabling.*

In time Petri nets, each enabled transition has an implicit clock. The clock starts counting at the moment the transition is enabled. This dissertation adopts the firing semantics described in [33], namely, *single server semantics with restart*. This semantics states that just one transition may be fired at once, and only its respective clock and the clocks of transitions in conflict with the fired transition are reset to zero after the firing. Since *enabling memory* is adopted, the clocks of other transitions are not reset. Additionally, the tokens remain in places up to the firing of the transition [34]. In this dissertation, it is considered that this firing is instantaneous, that is, the firing takes no time.

In addition, it is important to differentiate between static and dynamic firing intervals for a transition t in time Petri nets. As defined before, $I(t)$ is the static firing interval. The dynamic firing interval is defined as $(I_D(t) = (DLB(t), DUB(t)))$, where DLB stands for dynamic lower bound, and DUB stands for dynamic upper bound. I_D is computed as follows: $DLB(t) = \max(0,\ EFT(t) - c(t))$, and $DUB(t) = LFT(t) - c(t)$. As it can be seen, $I_D(t)$ is dynamically modified whenever the respective clock variable is incremented, and t is not fired. Initially, when a transition t becomes enabled, $I_D(t) = I(t)$.

Definition 5.4 (States) *Considering* $\mathcal{P}$ *a time Petri net,* M *the set of all reachable markings of* $\mathcal{P}$*, and* C *the set of all clock vectors of* $\mathcal{P}$*, the set of states* S *of a time Petri net is defined by* $S \subseteq (M \times C)$*. A single state is defined by a pair* (m, c)*, where* m *is a marking, and* c *is its respective clock vector for* $ET(m)$*. The initial state is* $s_0 = (m_0, c_0)$*, where* $c_0(t) = 0,\ \forall\ t \in ET(m_0)$*.*

In time Petri nets, there are two types of transitions between states. The first takes into account the *time elapsing*. In this case, there is no marking changing, but only clock incrementation. The second regards state change due to transition firings. In this case, changes happen in both marking and clock. Although, by definition, the first type represents a state change, this is not considered in this dissertation in order to optimize the state space generation. This work is only concerned with the time instants where the transitions are fired.

Definition 5.5 (Fireable Transition Set) *Let* $s = (m, c)$ *be a state of a* TPN*.* $FT(s)$ *is the set of fireable transitions at state* s*, which is defined by:*

$$FT(s) = \{t_i \in ET(m) \mid DLB(t_i) \leq min(DUB(t_k))\ \forall t_k \in ET(m)\}.$$

This definition enforces the *strong firing semantics*. In such semantics, an enabled transition t cannot fire before it has been enabled for $EFT(t)$ time units and no later than $LFT(t)$ time units.

Definition 5.6 (Firing Domain) *Let $s = (m, c)$ be a state of a TPN. The firing domain for a fireable transition t at a specific state s, is defined by the following time interval:*

$$FD_s(t) = [DLB(t),\ \min(DUB(t_k))],\ \forall t_k \in ET(m).$$

A transition t at state s can only fire in the interval expressed by $FD_s(t)$.

Definition 5.7 (Reachable States) *Let $\mathcal{P}$ be a time Petri net, $s_i = (m_i, c_i)$ a reachable state, t a fireable transition ($t \in FT(s_i)$), and θ a specific time value in the firing domain of t ($\theta \in FD_{s_i}(t)$). A new reachable state $s_j = \texttt{fire}(s_i, (t, \theta))$ denotes that firing a transition t at time θ from the state s_i, a new state $s_j = (m_j, c_j)$ is reached, such that:*

- $\forall p \in P,\ m_j(p) = m_i(p) - W(p,t) + W(t,p)$, *as usual in Petri nets;*
- $\forall t_k \in ET(m_j), C_j(t_k) = \begin{cases} 0, & if (t_k = t) \\ 0, & if (t_k \in ET(m_j) - ET(m_i)) \\ C_i(t_k) + \theta, & otherwise \end{cases}$

Definition 5.8 (Timed Labeled Transition System) *A timed labeled transition system is a quadruple $\mathcal{L}$= (S,Σ,$\rightarrow$,s_0), where S is a finite set of discrete states, Σ is an alphabet of labels representing activities (or actions), $\rightarrow \subseteq S \times \Sigma \times S$ is the transition relation, and $s_0 \in S$ is the initial state.*

The semantics of a time Petri net $\mathcal{P}$ is defined by associating a timed labeled transition system ($TLTS$). The $TLTS$ is utilized to represent a feasible firing schedule, which is a firing sequence that guarantees all constraints are met. In order to find a feasible firing sequence, a schedule synthesis algorithm is adopted. Next chapter presents in details this algorithm.

Definition 5.9 (Feasible Firing Schedule) *Let $\mathcal{L}$ be a timed labeled transition system of a time Petri net $\mathcal{P}$, s_0 its initial state, $s_n = (m_n, c_n)$ a final state, and $m_n = M^F$ is the desired final marking. $s_0 \xrightarrow{(t_{k1},\theta_{k1})} s_1 \xrightarrow{(t_{k2},\theta_{k2})} s_2 \rightarrow ... \rightarrow s_{n-1} \xrightarrow{(t_{kn},\theta_{kn})} s_n$ is defined as a feasible firing schedule, where $s_{i+1} = \texttt{fire}(s_i, (t_{ki}, \theta_{ki}))$, $i \geq 0$, $t_{ki} \in FT(s_i)$, and $\theta_{ki} \in FD_{s_i}(t_{ki})$.*

Definition 5.10 (Code-Labeled Time Petri Net) *A code-labeled time Petri net (CTPN) is represented by $\mathcal{P}_c = (\mathcal{P}, \mathcal{CS})$. $\mathcal{P}$ is the underlying time Petri net, and $\mathcal{CS}$:$T \nrightarrow \mathcal{ST}$ is a partial function that assigns transitions to behavioral source code, where $\mathcal{ST}$ is a set of source code.*

5.2 Specification Model

As stated previously, this work is concerned with hard real-time embedded systems, more specifically, hard real-time embedded systems with multiple processors. A specification model for such systems is obtained through the user requirement analysis. A specification can be composed of several elements:

1. Timing constraints of a set of hard real-time tasks;
2. Inter-task relations, such as precedence and exclusion relations;
3. Scheduling method of each task (preemptive or non-preemptive);
4. Processors where each task is allocated;
5. Inter-processor communications;
6. Source code of each task.

In the following subsections, the specification model is detailed.

5.2.1 Task Constraints Specification

The scheduling approach adopted in this dissertation is pre-runtime, which only deals with periodic tasks. Considering $\mathcal{T}$ the set of tasks in a system and $\mathcal{P}$ the set of processors, the definition of periodic tasks is as follows.

Definition 5.11 (Periodic Task Constraints) *Let $\tau_i \in \mathcal{T}$ be a periodic task. The constraints of τ_i is defined by (ph_i, r_i, c_i, d_i, p_i, $proc_i$), where ph_i is the phase time; r_i is the release time; c_i is the worst-case execution time(WCET); d_i is the deadline; p_i is the period; and $proc_i \in \mathcal{P}$ is the processor allocated to such task.*

The phase (ph_i) is the delay associated to the first time request of task τ_i after the system starting. The periodicity in which τ_i is executed is defined by the period p_i. Release time r_i, WCET c_i, and deadline d_i, are time instants related to the beginning of a period. In this way, c_i is the WCET required for executing task τ_i; and d_i is the time at which task τ_i must be completed. This work considers that $c_i \leq d_i \leq p_i$. All these timing constraints (phase, release, computation, deadline, and period) are non-negative integer values, that is, ph_i, r_i, c_i, d_i, $p_i \in \mathbb{N}$.

When adopting an architecture with multiple processors, the previous allocation of tasks to processors becomes necessary. This dissertation considers that this allocation have to be performed by the designer.

It is work stating that all timing constraints are expressed in task time units (TTUs), where each TTU has a correspondence with some multiple of a specific timing unit (e.g.

millisecond). A TTU is the smallest indivisible granule of a task, during which a task cannot be preempted by any other task. The granularity of the TTU is a design decision.

In real applications, there are some situations where the arrival of tasks is not periodic, since these tasks arrive randomly. These tasks are called aperiodic tasks. As stated before, pre-runtime approach only deals with periodic tasks. Nevertheless, there is a class of aperiodic tasks called sporadic tasks, where the minimum period between two activations is known. In [35], Mok describes a translation technique from sporadic to periodic. Using this technique, converted sporadic tasks can also be taken in account in the scheduling process. The definition of sporadic tasks is as follows.

Definition 5.12 (Sporadic Task) *Let $\tau_k \in \mathcal{T}$ be a sporadic task defined by τ_k = (c_k, d_k, min_k, $proc_k$), where c_k is the worst-case execution time; d_k is the deadline; min_k is the minimum time interval between two activations of task τ_k; and $proc_k$ is the respective processor.*

The tecnique proposed by Mok converts a sporadic task $(c_s, d_s, min_s, proc_s)$ into a corresponding periodic task (ph_p, c_p, d_p, p_p, $proc_p$) using the steps below:

1. $ph_p = 0$;
2. $c_p = c_s$;
3. $d_s \geq d_p \geq c_s$;
4. $c_s \leq p_p \leq \min(d_s - d_p + 1, min_s)$; and
5. $proc_p = proc_k$.

5.2.2 Inter-tasks Relations

The inter-tasks relations are represented by precedence and exclusion relations.

A task τ_i preceding task τ_j (τ_i `PRECEDES` τ_j) means that τ_j can only start executing after τ_i has finished. In general, this kind of relation is suitable whenever a task (successor) needs information that is produced by another task (predecessor). This relation imposes equal period for both tasks involved.

A task τ_i excluding task τ_j (τ_i `EXCLUDES` τ_j) means that τ_j can not start while task τ_i is executing. In other words, if a single processor is being utilized, task τ_i could not be preempted by task τ_j. In this work it is considered that the exclusion relation is symmetrical, that is, when A `EXCLUDES` B it implies that B `EXCLUDES` A.

5.2.3 Scheduling Method

Two scheduling methods are considered in this work: (i) preemptive and (ii) non-preemptive.

A task τ_i is said to be preemptive if its execution can be suspended by another tasks. A task τ_i is said to be non-preemptive if its execution cannot be suspended by another tasks. In the non-preemptive case, the task runs without interference until its conclusion.

It is up to the designer the selection of the best suited method.

5.2.4 Task Source Code

The purpose of this dissertation is to provide automatic code generation for embedded hard-real time systems. In this way, the source code associated with each task also needs to be specified.

This work adopts C language as the standard programming language for the code generation. Although C has standard constructs, there are some C extensions that contain additional functionalities for specific hardware platforms. This work considers that the designer is responsible to provide the compiler for the desired plataform. For this dissertation, Keil 8051 compiler is adopted, since all experiments were performed using the 8051 microcontroller. However, as described later, the approach may be easily ported to other plataforms.

5.2.5 Communication Task

When adopting a multiprocessing environment, the inter-processor communications have to be taken into account, since these communications affect the system predictability. A inter-processor communication is represented by a special task, namely, communication task, which is described below.

Definition 5.13 (Communication Task) *Let $\mu_m \in \mathcal{M}$ be a communication task defined by $\mu_m = (\tau_i, \tau_j, ct_m, bus_m)$, where $\tau_i \in \mathcal{T}$ is the sending task, $\tau_j \in \mathcal{T}$ is the receiving task, ct_m is the worst case communication time, $bus_m \in \mathcal{B}$ is the bus, $\mathcal{B}$ is the set of buses, and $proc_i \neq proc_j$.*

It is worthwhile observing that the definition enforces point-to-point communication, since the communication can only occur between two different tasks, where each task is allocated to a different processor. In addition, the bus is an abstraction for a communication channel used for providing communication between tasks from different processors.

Table 5.1: Specification Example

task	phase	release	wcet	deadline	period	proc/bus	from	to
A	0	0	2	10	30	proc1	-	-
B	0	2	3	20	30	proc1	-	-
C	0	2	3	30	30	proc1	-	-
D	0	0	2	10	30	proc2	-	-
E	0	2	3	30	30	proc2	-	-
F	0	0	3	10	30	proc3	-	-
M1	-	-	1	-	-	bus1	F	A
M2	-	-	2	-	-	bus1	C	D
Intertask Relations								
A PRECEDES B, B EXCLUDES C								

5.2.6 Specification Example

Table 5.1 depicts parts of a specification, which is composed of periodic and communication tasks. In addition, the intertask relations are shown. It is worthwhile observing that buses also need to be specified, since they are considered in the scheduling synthesis.

5.3 Modeling the Specification

This section presents the modeling phase of the proposed methodology. In order to model a system specification, a mathematical formalism is being adopted, namely, time Petri net.

This work adopts the modeling proposed in [8], which applies composition rules on building blocks models. These blocks are specific for the scheduling policy adopted, that is, pre-runtime scheduling policy. In that work, the building blocks are: (i) periodic task arrival; (i) task structure, which considers preemptive or non-preemptive task scheduling method; (ii) deadline checking; (iii) inter-task relations, such as precedence. In this dissertation, a new building block is defined: inter-processor communication block. In order to depict the modeling phase, an overview of the composition rules and building blocks proposed in [8] are presented, and next, a detailed explanation is presented for modeling inter-processor communications.

In addition, it is worth explaining that pre-runtime algorithm schedules tasks considering a schedule period that corresponds to the least common multiple between all periods in the task set. Thus, the modeling has to be adjusted to consider such requirement.

5.3.1 Scheduling Period

When adopting a pre-runtime policy, the task scheduling is performed using a finite period. This finite period is obtained by calculating the least common multiple (LCM) among periods of the given set of tasks. The LCM is called *schedule period*.

Considering this new period, there are several *tasks instances* of the same task, which are obtained by calculating $\mathcal{N}(\tau_i) = P_S/p_i$, where $\mathcal{N}(\tau_i)$ represents the number of instances of task τ_i, P_S the schedule period, and p_i the task period. For example, let us consider the task set in Table 5.2. In this particular case, $P_S = 24$, hence implying that the two periodic tasks are replaced by seven new periodic tasks ($\mathcal{N}(\tau_0) = 3$, and $\mathcal{N}(\tau_1) = 4$), where the timing constraints of each task instance has to be adjusted to consider that new period.

Table 5.2: Timing Constraints for a Simple Task Set

task	ph	r	c	d	p
τ_0	0	0	2	7	8
τ_1	0	2	2	6	6

Table 5.3 depicts the modified timing constraints. For the j^{th} task execution of τ_i, the corresponding release time is $r_i^j = r_i + p_i * (j-1)$; and deadline is $d_i^j = d_i + p_i * (j-1)$.

Table 5.3: Modified Timing Constraints for a Simple Task Set

	τ_0^1	τ_0^2	τ_0^3	τ_1^1	τ_1^2	τ_1^3	τ_1^4
r	0	8	16	2	8	14	20
c	2	2	2	2	2	2	2
d	7	15	23	6	12	18	24
p	24	24	24	24	24	24	24

5.3.2 Composition Rules

The modeling method adopted is conducted by applying composition rules on building block models in order to form larger Petri net models. This section gives an overview of the operators that represent the composition rules. These operators are: place merging, serial place refinement, place addition, arc addition, arc removing, and net union operator. For a more detailed explanation about the operators, the reader is referred to [8].

Definition 5.14 (Place Merging) *Consider the following nets:*

$N_1 = (P_1, T_1, F_1, W_1, M_{01}, I_1)$;

$N_2 = (P_2, T_2, F_2, W_2, M_{02}, I_2)$;

$N_c = (P_c, T_c, F_c, W_c, M_{0c}, I_c)$.

The composition by place merging is denoted by $N_c = \langle \texttt{Pmerg} \rangle\ (N_1,\ N_2)$. *The net* N_c *is composed in the following way:*

- $P_c = P_1 \cup P_2$
- $T_c = T_1 \cup T_2$
- $\forall t \in T_c :\ I_c(t) = \begin{cases} I_1(t), & if\ t \in T_1 \\ I_2(t), & if\ t \in T_2 \end{cases}$
- $F_c = F_1 \cup F_2$
- $\forall p \in P_c :\ M_{0c}(p) = \begin{cases} M_{01}(p) & if\ p \in P_1 \\ M_{02}(p) & if\ p \in P_2 \\ \max\left(M_{01}(p), M_{02}(p)\right) & if\ p \in P_1, P_2 \end{cases}$

This operator combines two nets by merging a set of places with the same name, resulting in a single net. Figure 5.2 shows a simple example of place merging operator applied to two nets.

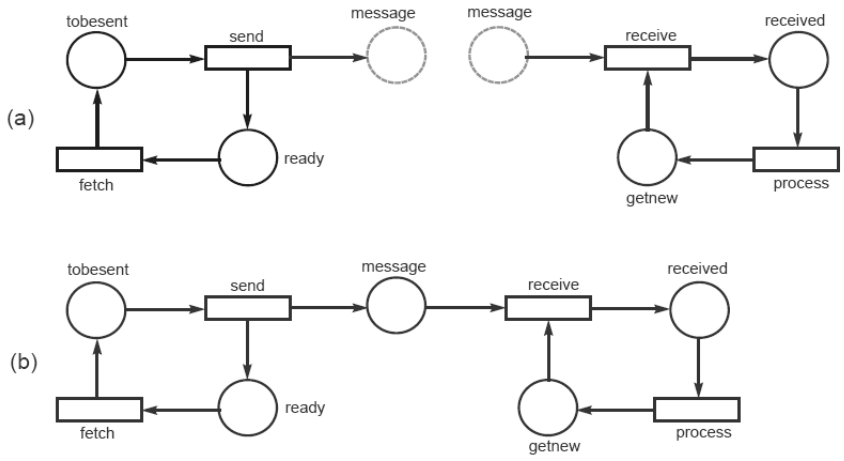

Figure 5.2: An Example of Place Merging

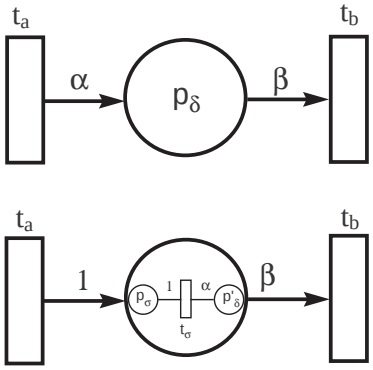

Figure 5.3: Place Refinement

Definition 5.15 (Serial Place Refinement) *Considering the following time Petri nets* $N = (P, T, F, W, M_0, I)$*, and* $N_c = (P_c, T_c, F_c, W_c, M_{0_c}, I_c)$*, the serial place refinement is defined by* $N_c = \langle \texttt{Pref} \rangle (N, p_\delta, p_\sigma, t_\sigma, p'_\delta)$*, where* $p_\delta \in P$*. The new net* N_c *is composed in the following way:*

- $P_c = (P \cup \{p_\sigma, p'_\delta\}) - \{p_\delta\}$
- $T_c = T \cup \{t_\sigma\}$
- $F_c = (F - F^1) \cup F^2 \cup F^3$*, where:*
  - $F^1 = \{(t_i, p_\delta), (p_\delta, t_j) \mid t_i \in {\bullet p_\delta},\ t_j \in p_\delta{\bullet}\}$
  - $F^2 = \{(t_i, p'_\delta), (p'_\delta, t_j) \mid t_i \in {\bullet p_\delta},\ t_j \in p_\delta{\bullet}\}$
  - $F^3 = \{(p_\sigma, t_\sigma), (t_\sigma, p'_\delta)\}$
- $\forall p \in P_c$*:* $M_{0_c}(p) = \begin{cases} M_0(p_j), & \text{if } p = p_j \wedge p_j \in P \\ 0, & \text{if } p = p_\sigma \vee p = p'_\delta \end{cases}$
- $\forall t \in T_c$*:* $I_c(t) = \begin{cases} I(t_j), & \text{if } t = t_j \wedge t_j \in T \\ [0,0], & \text{if } t = t_\sigma \end{cases}$

$$\star\ \forall f \in F_c\text{: } W_c(f) = \begin{cases} 1, & if\ \ f = (g, p_\sigma),\ \forall g \in T \\ 1, & if\ \ f = (p_\sigma, t_\sigma) \\ W(\bullet p_\delta, p_\delta), & if\ \ f = (t_\sigma, p'_\delta) \\ W(p_\delta, \bullet p_\delta), & if\ \ f = (p'_\delta, p_\delta \bullet) \\ W(f), & otherwise \end{cases}$$

The serial place refinement operator replaces a single place by a sequence of one place, one transition, and a second place. Figure 5.3 depicts the final result of using the place refinement operator.

Definition 5.16 (Arc Addition) *Considering the following time Petri nets $N = (P, T, F, W, M_0, I)$, and $N_c = (P_c, T_c, F_c, W_c, M_{0_c}, I_c)$, arc addition is an operator that adds a single arc (x, y), such that $(x \in P \wedge y \in T) \vee (x \in T \wedge y \in P)$. Arc addition is represented by $N_c = \langle \texttt{Aadd} \rangle\, (N, (x, y), n)$, where N is the original net, (x, y) is the arc, w is the weight of the arc, and N_c is the resultant net. N_c is generated in the following way:*

$\star\ P_c = P; \quad T_c = T; \quad M_{0_c} = M_0; \quad I_c = I;$

$\star\ F_c = (F \cup \{(x, y)\};$

$$\star\ \forall f \in F_c\text{: } W_c(f) = \begin{cases} W(f), & if\ f \in F \\ n, & if\ f = (x, y) \end{cases}$$

This operator adds an arc from a place to a transition or from a transition to a place.

Definition 5.17 (Arc Removing) *Considering the following time Petri nets $N = (P, T, F, W, M_0, I)$, and $N_c = (P_c, T_c, F_c, W_c, M_{0_c}, I_c)$, arc removing is an operator that removes a single arc (x, y), such that $(x \in P \wedge y \in T) \vee (x \in T \wedge y \in P)$. Arc removing is represented by $N_c = \langle \texttt{Arem} \rangle\, (N, (x, y))$, where N is the original net, (x, y) is the removed arc, and N_c is the resultant net. N_c is generated in the following way:*

$\star\ P_c = P; \quad T_c = T; \quad M_{0_c} = M_0; \quad I_c = I;$

$\star\ F_c = (F - \{(x, y)\};$

$\star$ $\forall f \in F_c$: $W_c(f) = W(f)$

Arc removing removes an arc from a place to transition, or from a transition to a place.

Definition 5.18 (Place Addition) *Considering the following time Petri nets* $N = (P, T, F, W, M_0, I)$, *and* $N_c = (P_c, T_c, F_c, W_c, M_{0_c}, I_c)$, *place addition is an operator that adds a single place into the respective net. Place addition is represented by* $N_c = \langle \texttt{Padd} \rangle (N, p_\delta, m_\delta)$, *where* N *is the original net,* p_δ *is the place,* m_δ *is its respective marking, and* N_c *is the output net.* N_c *is generated in the following way:*

$\star$ $T_c = T; \quad F_c = F; \quad W_c = W; \quad I_c = I;$

$\star$ $P_c = P \cup \{p_\delta\}$

$\star$ $\forall p \in P_c,\ M_{0c}(p) = \begin{cases} M_0(p), & if\ p \in P \\ m_\delta, & if\ p = p_\delta \end{cases}$

This operator adds a single place to a net.

Definition 5.19 (Net Union) *Considering the following time Petri nets* $N_1 = (P_1, T_1, F_1, W_1, M_{0_1}, I_1)$, $N_2 = (P_2, T_2, F_2, W_2, M_{0_2}, I_2)$, *and* $N_c = (P_c, T_c, F_c, W_c, M_{0_c}, I_c)$, *the net union is an operator that unifies two nets. It is represented by* $N_c = N_1 \sqcup N_2$. N_c *is computed in the following way:*

$$P_c = P_1 \cup P_2; \quad T_c = T_1 \cup T_2; \quad F_c = F_1 \cup F_2$$

$$\forall f \in F_c,\ W_c(f) = \begin{cases} W_1(f), & if\ f \in F_1 \\ W_2(f), & if\ f \in F_2 \end{cases}$$

$$\forall p \in P_c,\ M_{0c}(p) = \begin{cases} M_{01}(p), & if\ p \in P_1 \\ M_{02}(p), & if\ p \in P_2 \end{cases}$$

$$\forall t \in T_c,\ I_c(t) = \begin{cases} I_1(t), & if\ t \in T_1 \\ I_2(t), & if\ t \in T_2 \end{cases}$$

Net union joins two nets into a single one.

5.3.3 Tasks' Modeling

In order to show how to represent the specification using time Petri net, this section describes the building blocks that are used to model the tasks.

Periodic Task Arrival Block

This block models the periodic invocation for all task instances in the schedule period (P_S). As can be seen in Figure 5.4, the transition t_{ph_i} models the initial phase of the first task instance. Additionally, t_{a_i} models the periodic arrival for the remaining instances.

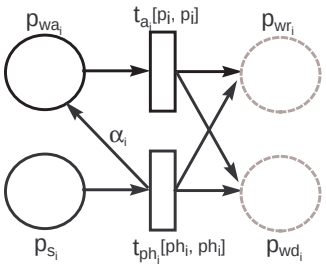

Figure 5.4: Building Block Arrival

Task Structure Block

For modeling release and computation, the task structure block is defined for such purpose. Processor granting and releasing are also modeled in this block, since a processor needs to be accessed in mutual exclusion.

For each scheduling method (preemptive or non-preemptive), there is a respective task structure block. For modeling non-preemptive method, the task structure block for non-preemptive tasks should be considered (see Figure 5.5). Transitions t_{ri} and t_{ci} represent release and computation, respectively. Processor granting is represented by transition t_{gi} and, after the computation, processor releasing is performed (t_{ci} firing). In order to model preemptive tasks, the task structure block depicted in Figure 5.6 is adopted. It is worthwhile observing that, for preemptive tasks, the computation time is broken in several task time units (TTU). For example, considering a task with computation time equals to 10, this value is represented by 10 TTUs, where, after each TTU, the task may be preempted by another task. This situtation is represented by the arc weight (c_i) from t_{ri} to p_{wgi}.

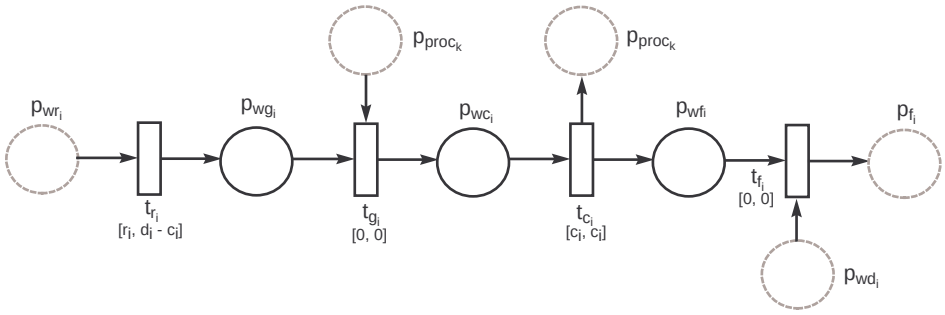

Figure 5.5: Non-Preemptive Task Structure Building Block

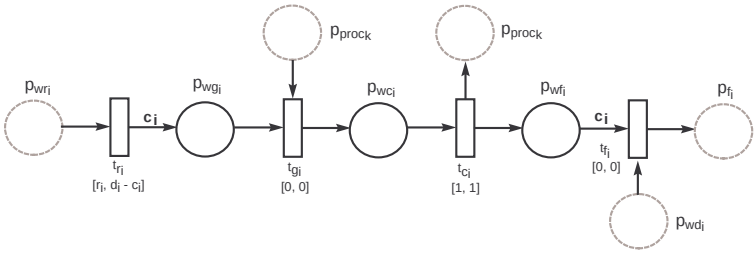

Figure 5.6: Preemptive Task Structure Building Block

Deadline Checking Block

This block models the deadline missing, which is an undesirable state in hard real-time systems. Although this state is modeled, one of the goals of the scheduling algorithm (Chapter 6) is to eliminate states that represent undesirable situations like this one. Figure 5.7 depicts this block. Transition t_{di} represents the deadline timing constraint, and t_{pci} is an auxiliary transition, which removes any token that enables a computation transition.

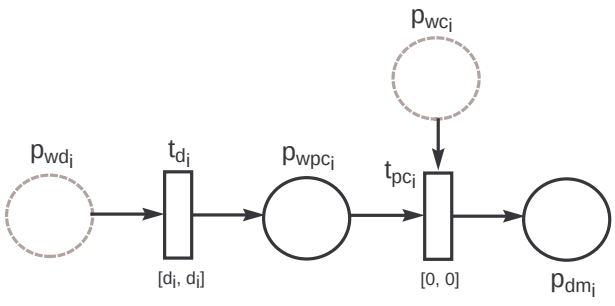

Figure 5.7: Deadline Checking Building Block

Resource Block

This work explicitly models processors and buses, where each one is represented by a single place. Figure 5.8 shows an example of a processor (P_{proc_i}) and a bus (P_{bus_k}). As described before, each task is previously allocated to a processor, and the task migration to another processor is not allowed.

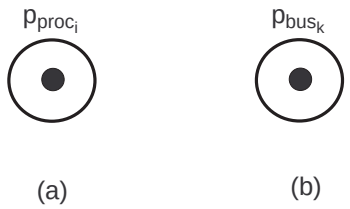

Figure 5.8: Resources Modeling: (a) Processor; (b) Bus

Fork Block

The building block fork (Fig. 5.9) is responsible for starting all tasks instances occurring in the schedule period. In other words, this block consists of creating of n concurrent processes.

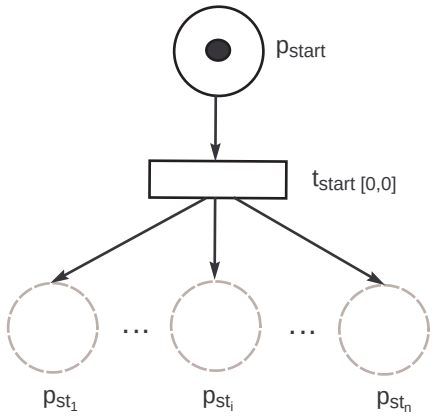

Figure 5.9: Building Block Fork

Join Block

The join block represents the synchronization of executions of all tasks in the schedule period. Figure 5.10 presents the join block.

It is worth stating that a marking in place p_{end} represents the desirable final marking (or M^F). In this case, $M(p_{end}) = 1$ indicates that a feasible schedule was found.

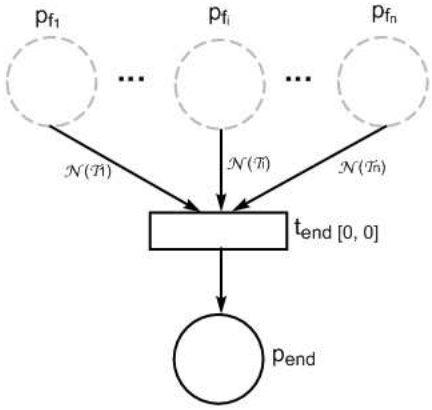

Figure 5.10: Building Block Join

A Complete Example Considering Two Tasks

In order to show the practical utilization of the building blocks and composition rules, this section shows an example considering the specification depicted in Table 5.2. The specification is composed of two tasks, τ_0 and τ_1, and both share the same processor.

In this example, two nets (N_0 and N_1), representing both tasks (τ_0 and τ_1), are being considered, and the net $N_{aux} = (P_{aux}, T_{aux}, F_{aux}, W_{aux}, M_{0_{aux}}, I_{aux})$ represents the system. Additionally, the fork and join blocks (N_f and N_j) are instantiated (N'_f and N'_j) considering that n = 2, that is, there are two tasks in the system.

For the proposed example, p_{proc} is a single place, where $M(p_{proc}) = 1$. Using the modeling method proposed in [8], the TPN representing this system is modeled as follows:

1. $N_{aux} = (N_0 \sqcup N_1)$;
2. $N_{aux} = \langle \texttt{Padd} \rangle \, (N_{aux}, p_{proc}, 1)$
3. $N_{aux} = \langle \texttt{Aadd} \rangle \, (N_{aux}, (p_{proc}, t_{g0}), 1)$
4. $N_{aux} = \langle \texttt{Aadd} \rangle \, (N_{aux}, (p_{proc}, t_{g1}), 1)$
5. $N_{aux} = \langle \texttt{Aadd} \rangle \, (N_{aux}, (t_{c0}, p_{proc}), 1)$
6. $N_{aux} = \langle \texttt{Aadd} \rangle \, (N_{aux}, (t_{c1}, p_{proc}), 1)$

7. $N_{aux} = \langle \texttt{Pmerg} \rangle (N_{aux}, N'_f)$

8. $N_{aux} = \langle \texttt{Pmerg} \rangle (N_{aux}, N'_j)$

Initially, two nets N_0 and N_1, representing the two tasks, are joined in a new net called N_{aux} (Step 1). Next, place p_{proc} is added into the N_{aux} net (Step 2). The next four steps (Steps from 3 to 6) add the respective arcs that represents processor granting ((p_{proc}, t_{g0}) and (p_{proc}, t_{g1})) and processor releasing ((t_{c0}, p_{proc}) and (t_{c1}, p_{proc})). Finally, place merging is used in order to compose the two tasks with the fork (Step 7) and join (Step 8) nets.

Figure 5.11 shows the resultant model taking into account the non-preemptive scheduling method. It is worth restating that the timing constraints are faithfully modeled using the building blocks. As presented before, the periodic arrival of each task is represented by the respective arrival block, release and computation are represented by the task structure block, the deadline is modelded using the deadline checking block, and the processor is modeled using the resource block. From the Petri net model, the aim is to obtain a feasible schedule, such that a firing sequence results in a marking in place p_{end}. In this case, the marking states that a feasible schedule was found. In order to search for a feasible schedule, the scheduling algorithm described in Chapter 6 is responsible for this role.

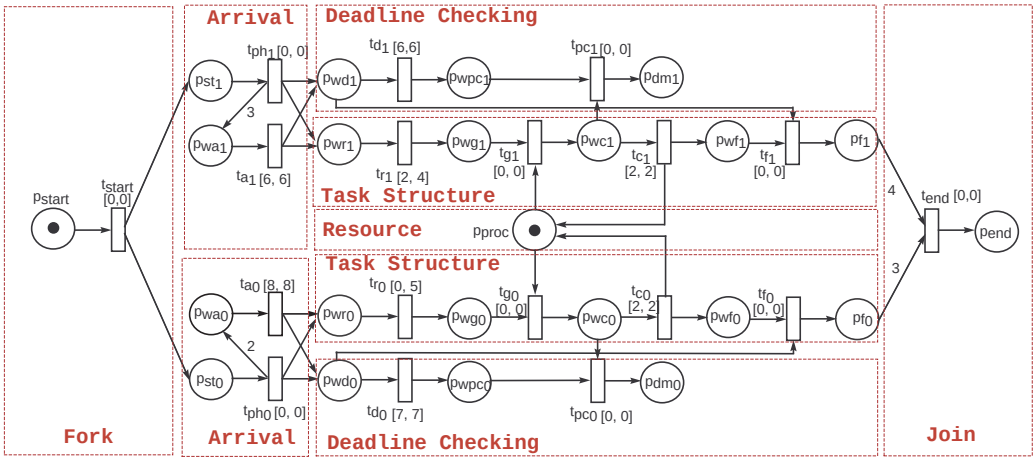

Figure 5.11: Complete Model for τ_0 and τ_1 Non-preemptive Tasks

Inter-task Relations Modeling

This section presents how to model precedence and exclusion relations. Precedence relations are defined between pairs of tasks, such that one task can only start executing

after the other has been finished. Figure 5.12 shows the TPN model for tasks T_i and T_j, such that T_i `PRECEDES` T_j.

Exclusion relations are also defined between pairs of tasks, such that two tasks cannot be executing at the same time. Figure 5.13 depicts an example considering that τ_i `EXCLUDES` τ_j.

For more details, the reader is referred to [8].

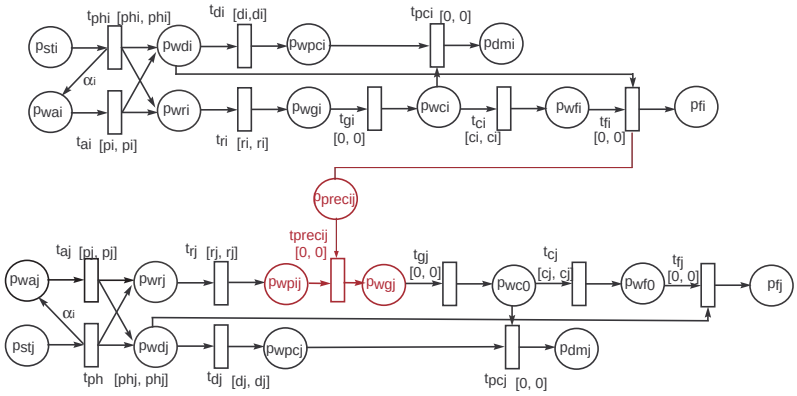

Figure 5.12: Precedence Relation Model Example

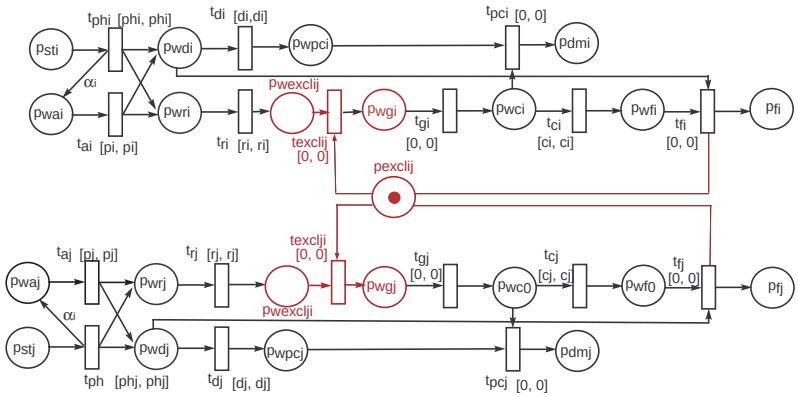

Figure 5.13: Exclusion Relation Model Example

5.3.4 Inter-processor Communication

One of the main contributions of this dissertation is the modeling of inter-processor communications. It is extremely important that inter-processor communications are

taken into account in order to provide predictability in embedded hard real-time systems with multiple processors.

This work considers that communication time between tasks allocated to the same processor is taken into account in each task computation time, since, in embedded systems, communication is usually performed through shared memory. In this case, such communication is simply dealt with as precedence relation.

However, when considering inter-processor communication the method is different, since communication is usually performed through a shared channel (e.g. bus). This dissertation adopts message-passing paradigm as the standard way for communication between processors.

The proposed method schedules the communication for avoiding network contention. Otherwise, it could result in different execution times for different runs of the same system, which is not appropriated for hard real-time systems.

The proposed method for inter-processor communication considers that:

1. after the execution of the sending task, the message transmission is performed;
2. the receiving task can only execute after receiving the complete message;
3. both the sending and receiving processors are ready in the beginning of the communication. In other words, when the sender is transmitting the data, a special task is executing at receiver side at the same moment for getting the respective data. This mechanism may be viewed as a synchronous communication. Since interrupts may affect the system predictability, the proposed approach adopts polling rather than interrupt handling to implement the receive operation;
4. point-to-point communication (or unicasting);
5. buses are reliable;
6. before communication takes place, the bus and, both sending and receiving processors have to be granted;
7. communication time is represented by the respective communication transition.

Next sections define a new building block, namely, inter-processor communication block, and describe how inter-processor communications are modeled.

Inter-processor communication Block

As previously introduced in Section 5.2 (Specification Model), the specification considers that all inter-processor communications are dealt with as a new communication

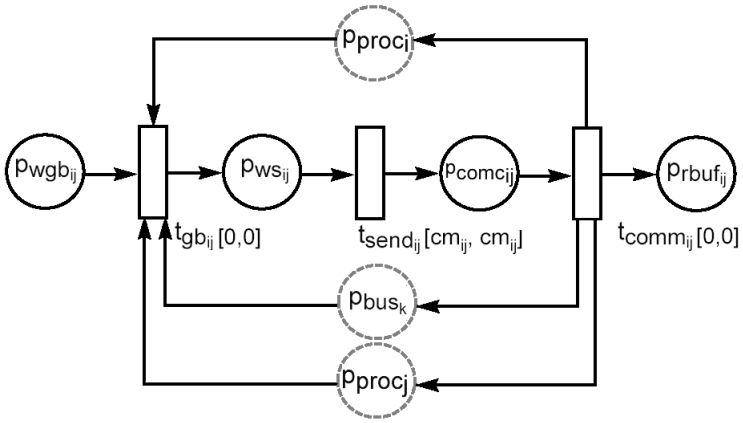

Figure 5.14: Building Block Message Sending

task. This section aims to present a new building block for modeling inter-processor communication. Figure 5.14 depicts this building block.

The building block inter-processor communication is a TPN $N_{ipc} = (P_{ipc}, T_{ipc}, F_{ipc}, W_{ipc}, M_{0_{ipc}}, I_{ipc})$, such that:

- $P_{ipc} = \{p_{wgb_{ij}}, p_{ws_{ij}}, p_{comc_{ij}}, p_{rbuf_{ij}}, p_{proc_i}, p_{proc_j}, p_{bus_k}\}$. These places model the following situations:

  $p_{wgb_{ij}}$: waiting for bus and processors granting;

  $p_{ws_{ij}}$: waiting for sending a message;

  $p_{comc_{ij}}$: communication concluded;

  $p_{rbuf_{ij}}$: receiving buffer;

  p_{proc_i}: processor of the sending task;

  p_{proc_j}: processor of the receiving task; and

  p_{bus_k}: bus.

- $T_{ipc} = \{t_{gb_{ij}}, t_{send_{i,j}}, t_{comm_{ij}}\}$. These transitions model the following actions:

  $t_{gb_{ij}}$: bus and processors granting;

  $t_{send_{i,j}}$: sending the message; and

  $t_{comm_{i,j}}$: bus and processors releasing.

- $F = \{(p_{wgb_{ij}}, t_{gb_{ij}}),\ (t_{gb_{ij}}, p_{ws_{ij}}),\ (p_{ws_{ij}}, t_{send_{i,j}}),\ (t_{send_{i,j}}, p_{comc_{ij}}),\ (p_{comc_{ij}}, t_{comm_{i,j}}),\ (t_{comm_{i,j}}, p_{rbuf_{ij}}),\ (p_{proc_i}, t_{gb_{ij}}),\ (t_{comm_{i,j}}, p_{proc_i}),\ (p_{proc_j}, t_{gb_{ij}}),\ (t_{comm_{i,j}}, p_{proc_j}),\ (p_{bus_k}, t_{gb_{ij}}),\ (t_{comm_{i,j}}, p_{bus_k})\}$

- $W_{ipc}(x,y) = 1\ \forall (x,y) \in F_{ipc}$.
- $M_{0_{ipc}}(p_{bus_k}) = M_{0_{ipc}}(p_{proc_i}) = M_{0_{ipc}}(p_{proc_j}) = \beta,\ \beta \in \mathbb{N}^+; \quad M_{0_{ipc}}(p) = 0\ \forall p \in P \wedge p \neq p_{bus_k} \wedge p \neq p_{proc_i} \wedge p \neq p_{proc_j}$.
- $I_{ipc}(t_{send_{ij}}) = [ct_m, ct_m]; \quad I_{ipc}(t_{gb_{ij}}) = I_{ipc}(t_{comm_{ij}}) = [0,0])$

The timing interval of transition $t_{send_{ij}}$ is fulfilled by the timing constraint specification, in this case, ct_m (worst-case communication time) of the respective communication task $\mu_m \in \mathcal{M}$. The timing intervals of transitions $t_{gb_{ij}}$ and $t_{comm_{ij}}$ are constant ([0,0]).

It is worthwhile to point out that the the transition $t_{send_{ij}}$ represents both the message sending and receiving.

Modeling Inter-processor Communications

Communication tasks (Definition. 5.13) are specified by $\mu_m = (\tau_i, \tau_j, ct_m, bus_m)$. In this case, the communication is from task τ_i to task τ_j, the worst-case communication time is ct_m, and the bus to be used is bus_m. Figure 5.15 applies the building block inter-processor communication for modeling the sending task τ_i as well the receiving task τ_j.

Taking into account that: (i) a bus is specified ($\{p_{bus_m}\}$ represents the bus bus_m); (ii) nets N_i and N_j represent tasks τ_i and τ_j, respectively, with their respective allocated processors ($\{p_{proc_i}$,$p_{proc_j}\}$). Formally, inter-processor communication model is obtained by applying the following steps:

1. instantiate the block inter-processor communication (let us call $N_{sm_{ij}}$);
2. join three nets N_i, N_j and $N_{sm_{ij}}$ ($N_{aux} = ((N_i \sqcup N_j) \sqcup N_{sm_{ij}})$)
3. For the sending task (N_i), do:
   (a) refine place p_{f_i} $\quad (N_{aux} = \langle \texttt{Pref} \rangle\, (N_{aux}, p_{f_i}, p_{fc_i}, t_{endcomm_i}, p_{f_i}))$;
   (b) remove the arc from t_{f_i} to p_{fc_i} $\quad (N_{aux} = \langle \texttt{Arem} \rangle\, (N_{aux}, t_{f_i}, p_{fc_i}))$;
   (c) add an arc from t_{f_i} to $p_{wgb_{ij}}$ $\quad (N_{aux} = \langle \texttt{Aadd} \rangle\, (N_{aux}, t_{f_i}, p_{wgb_{ij}}, 1))$.
   (d) add an arc from $t_{comm_{ij}}$ to p_{fc_i} $\quad (N_{aux} = \langle \texttt{Aadd} \rangle\, (N_{aux}, t_{comm_{ij}}, p_{fc_i}, 1))$.
   (e) add an arc to the bus p_{bus_m} $\quad (N_{aux} = \langle \texttt{Aadd} \rangle\, (N_{aux}, (p_{bus_m}, t_{gb_{ij}}), 1))$;
   (f) add another arc to the bus p_{bus_m} $\quad (N_{aux} = \langle \texttt{Aadd} \rangle\, (N_{aux}, (t_{comm_{ij}}, p_{bus_m}), 1))$.
   (g) add an arc to the processor p_{proc_i} $\quad (N_{aux} = \langle \texttt{Aadd} \rangle\, (N_{aux}, (p_{proc_i}, t_{gb_{ij}}), 1))$;
   (h) add another arc to the processor p_{proc_i} $\quad (N_{aux} = \langle \texttt{Aadd} \rangle\, (N_{aux}, (t_{comm_{ij}}, p_{proc_i}), 1))$.

4. For the receiving task (N_j), do:

   (a) refine place p_{wg_j} $(N_{aux} = \langle \texttt{Pref} \rangle (N_{aux}, p_{wg_j}, p_{rec_{ij}}, t_{rec_{ij}}, p_{wg_j}))$;

   (b) add an arc to the processor p_{proc_j} $(N_{aux} = \langle \texttt{Aadd} \rangle (N_{aux}, (p_{proc_j}, t_{gb_{ij}}), 1))$;

   (c) add another arc to the processor p_{proc_j} $(N_{aux} = \langle \texttt{Aadd} \rangle (N_{aux}, (t_{comm_{ij}}, p_{proc_j}), 1))$;

   (d) add an arc from $p_{rbuf_{ij}}$ to t_{rec_j} $(N_{aux} = \langle \texttt{Aadd} \rangle (N_{aux}, p_{rbuf_{ij}}, t_{rec_{ij}}))$.

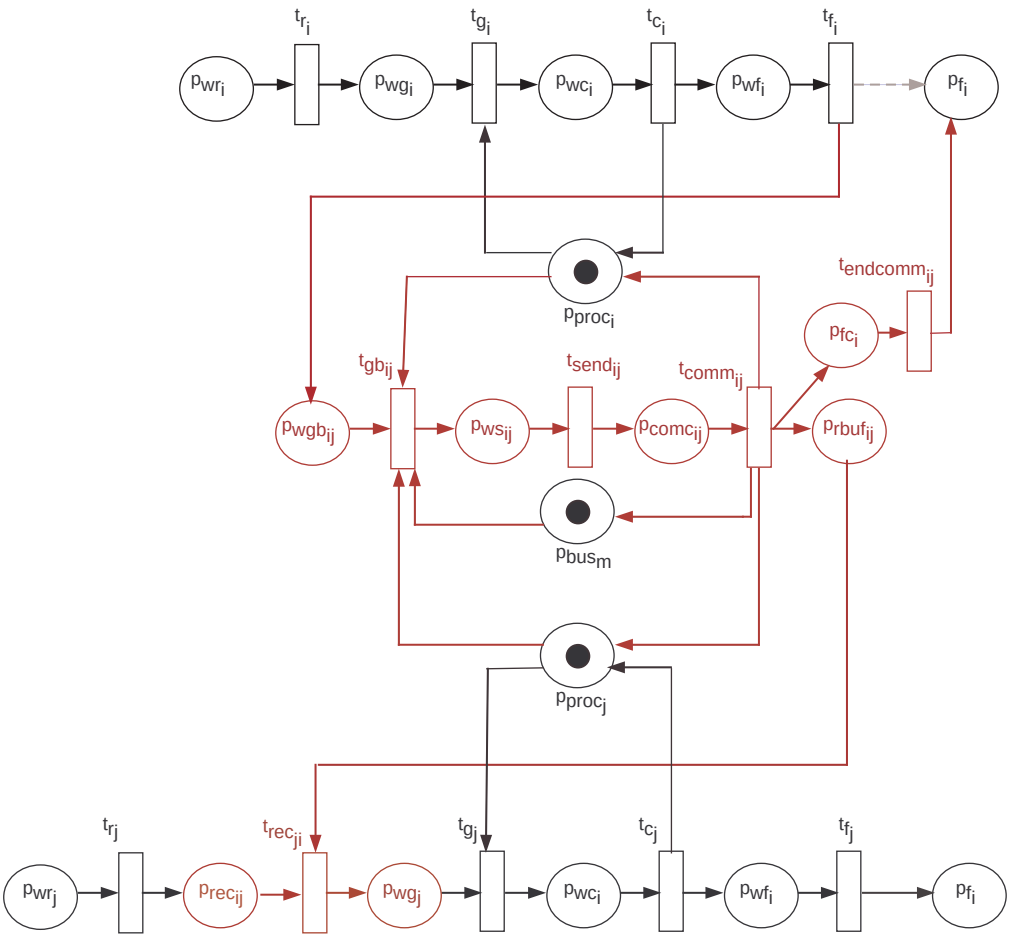

Figure 5.15: Modeling of the Sending and Receiving Tasks

As it can be observed in this procedure, the modeling of communication between tasks in different processors is performed by composing the nets representing both tasks with the inter-processor communication block (Step 2). Considering the modeling of the sending task, a place refinement is performed (Step 3a) in order to represent the end

of all communications performed by the respective task. This situation is represented by the place p_{fc_i} and the transition $t_{endcomm_i}$. In addition, the arc weight from p_{fc_i} to $t_{endcomm_i}$ is equal to the number of inter-processor message sending performed by τ_i. In this case, the arc weight of p_{fc_i} to $t_{endcomm_i}$ is 1, since just one message sending is done by τ_i. The requirement of such refinement is more clear when taking into account more than one communication task. Additionally, one arc is removed (Step 3b) and six arcs are added (Steps from 3c to 3h). Considering the modeling of the receiving task, one place is refined (Step 4a), and three arcs are added (Steps from 4b to 4d).

Considering another communication task $\mu_{m2} = (\tau_i, \tau_k, ct_{m2}, bus_m)$, where the communication is from task τ_i to task τ_k, the worst-case communication time is ct_{m2}, and the bus to be used is bus_m. The composition, taking into account the second message sending block from task τ_i, is shown in Figure 5.16. Considering both communication tasks (μ_{m1} and μ_{m2}), inter-processor communication model is obtained by carrying out the following steps:

1. instantiate the first block inter-processor communication (let us call $N_{sm_{ij}}$);
2. instantiate the second block inter-processor communication (let us call $N_{sm_{ik}}$);
3. join five nets N_i, N_j, N_k, $N_{sm_{ij}}$ and $N_{sm_{ik}}$ $((N_{aux} = ((((N_i \sqcup N_j) \sqcup N_k) \sqcup N_{sm_{ij}}) \sqcup N_{sm_{ik}}))$
4. For the sending task (N_i), do:
   (a) refine place p_{f_i} $(N_{aux} = \langle \texttt{Pref} \rangle (N_{aux}, p_{f_i}, p_{fc_i}, t_{endcomm_i}, p_{f_i}))$;
   (b) remove the arc from t_{f_i} to p_{fc_i} $(N_{aux} = \langle \texttt{Arem} \rangle (N_{aux}, t_{f_i}, p_{fc_i}))$;
   (c) remove the arc from p_{fc_i} to $t_{endcomm_i}$ $(N_{aux} = \langle \texttt{Arem} \rangle (N_{aux}, p_{fc_i}, t_{endcomm_i}))$;
   (d) add an arc from p_{fc_i} to $t_{endcomm_i}$ $(N_{aux} = \langle \texttt{Aadd} \rangle (N_{aux}, (p_{fc_i}, t_{endcomm_i}), 2))$. It is worth observing that the arc weight is equal to the number of inter-processor message sending performed by τ_i. For this case, the weight is 2;
   (e) add an arc from t_{f_i} to $p_{wgb_{ij}}$ $(N_{aux} = \langle \texttt{Aadd} \rangle (N_{aux}, t_{f_i}, p_{wgb_{ij}}, 1))$.
   (f) add an arc from $t_{comm_{ij}}$ to p_{fc_i} $(N_{aux} = \langle \texttt{Aadd} \rangle (N_{aux}, t_{comm_{ij}}, p_{fc_i}, 1))$.
   (g) add an arc from p_{bus_m} to $t_{gb_{ij}}$ $(N_{aux} = \langle \texttt{Aadd} \rangle (N_{aux}, (p_{bus_m}, t_{gb_{ij}}), 1))$;
   (h) add another arc to the bus p_{bus_m} $(N_{aux} = \langle \texttt{Aadd} \rangle (N_{aux}, (t_{comm_{ij}}, p_{bus_m}), 1))$.
   (i) add an arc to the processor p_{proc_i} $(N_{aux} = \langle \texttt{Aadd} \rangle (N_{aux}, (p_{proc_i}, t_{gb_{ij}}), 1))$;
   (j) add another arc to the processor p_{proc_i} $(N_{aux} = \langle \texttt{Aadd} \rangle (N_{aux}, (t_{comm_{ij}}, p_{proc_i}), 1))$.
   (k) add an arc from t_{f_i} to $p_{wgb_{ik}}$ $(N_{aux} = \langle \texttt{Aadd} \rangle (N_{aux}, t_{f_i}, p_{wgb_{ik}}, 1))$.
   (l) add an arc from $t_{comm_{ik}}$ to p_{fc_i} $(N_{aux} = \langle \texttt{Aadd} \rangle (N_{aux}, (t_{comm_{ik}}, p_{fc_i}), 1))$.

(m) add an arc to the bus p_{bus_m} $(N_{aux} = \langle \texttt{Aadd} \rangle (N_{aux}, (p_{bus_m}, t_{gb_{ik}}), 1))$;

(n) add another arc to the bus p_{bus_m} $(N_{aux} = \langle \texttt{Aadd} \rangle (N_{aux}, (t_{comm_{ik}}, p_{bus_m}), 1))$.

(o) add an arc to the processor p_{proc_i} $(N_{aux} = \langle \texttt{Aadd} \rangle (N_{aux}, (p_{proc_i}, t_{gb_{ik}}), 1))$;

(p) add another arc to the processor p_{proc_i} $(N_{aux} = \langle \texttt{Aadd} \rangle (N_{aux}, (t_{comm_{ik}}, p_{proc_i}), 1))$.

5. For both receiving tasks (N_j and N_k), do:

(a) refine place p_{wg_j} $(N_{aux} = \langle \texttt{Pref} \rangle (N_{aux}, p_{wg_j}, p_{rec_{ij}}, t_{rec_{ij}}, p_{wg_j}))$;

(b) add an arc to the processor p_{proc_j} $(N_{aux} = \langle \texttt{Aadd} \rangle (N_{aux}, (p_{proc_j}, t_{gb_{ij}}), 1))$;

(c) add another arc to the processor p_{proc_j} $(N_{aux} = \langle \texttt{Aadd} \rangle (N_{aux}, (t_{comm_{ij}}, p_{proc_j}), 1))$;

(d) add an arc from $p_{rbuf_{ij}}$ to $t_{rec_{ij}}$ $(N_{aux} = \langle \texttt{Aadd} \rangle (N_{aux}, (p_{rbuf_{ij}}, t_{rec_{ij}}), 1))$.

(e) refine place p_{wg_k} $(N_{aux} = \langle \texttt{Pref} \rangle (N_{aux}, p_{wg_k}, p_{rec_{ik}}, t_{rec_{ik}}, p_{wg_k}))$;

(f) add an arc to the processor p_{proc_k} $(N_{aux} = \langle \texttt{Aadd} \rangle (N_{aux}, (p_{proc_k}, t_{gb_{ik}}), 1))$;

(g) add another arc to the processor p_{proc_k} $(N_{aux} = \langle \texttt{Aadd} \rangle (N_{aux}, (t_{comm_{ik}}, p_{proc_k}), 1))$;

(h) add an arc from $p_{rbuf_{ik}}$ to $t_{rec_{ik}}$ $(N_{aux} = \langle \texttt{Aadd} \rangle (N_{aux}, (p_{rbuf_{ik}}, t_{rec_{ik}}), 1))$.

Table 5.4: A Simple Example of Task Timing Specification with Two Communication Tasks

TaskID	ph	r	c	d	p	proc/bus	from	to
T0	0	0	10	100	250	proc1	-	-
T1	0	0	15	100	150	proc1	-	-
T2	0	0	20	150	250	proc1	-	-
T3	0	0	40	200	250	proc1	-	-
T4	0	0	20	50	250	proc2	-	-
T5	0	0	10	100	250	proc3	-	-
M1	-	-	5	-	-	bus1	T4	T2
M2	-	-	5	-	-	bus1	T3	T5
Intertask Relations								
T0 PRECEDES T2, T1 PRECEDES T2, T2 PRECEDES T3, T4 PRECEDES T5								

A more complex example is depicted in Table 5.4, where five tasks are allocated in three processors, and two communication tasks perform the inter-processor communications. For this specification, Figure 5.17 shows the respective TPN model. For sake of readability, this net does not show the processors. It is worth observing that communications between tasks in the same processor is dealt with as precedence relation.

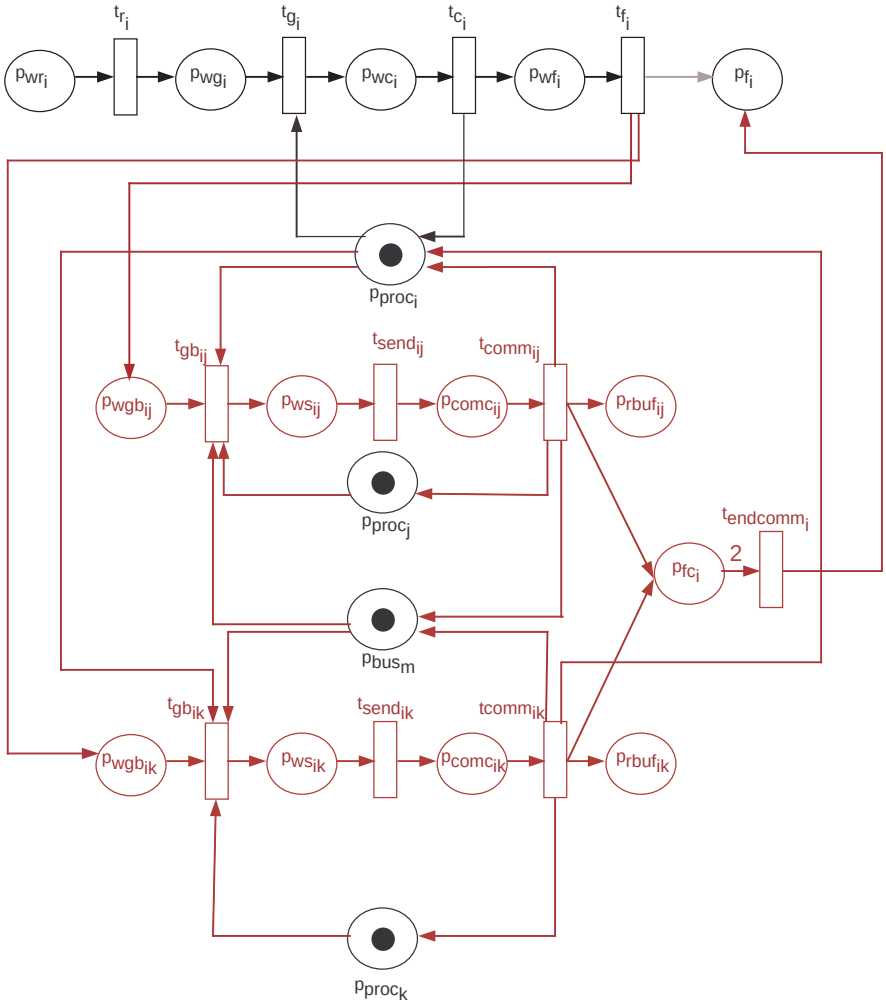

Figure 5.16: Modeling of two communication tasks

5.4 Summary

This chapter detailed the method adopted for modeling embedded hard real-time systems. Firstly, aspects related to the computation model, namely, time Petri net, were presented. Next, the specification model was shown. The specification is composed of periodic task timing constraints, inter-task relations, allocation of tasks to processors, tasks source code, and inter-processor communications. Lastly, the modeling of embedded hard real-time systems using time Petri net was demonstrated. The modeling approach applies composition rules on building block models. In this dissertation, a new building block was presented: inter-processor communication block. This block

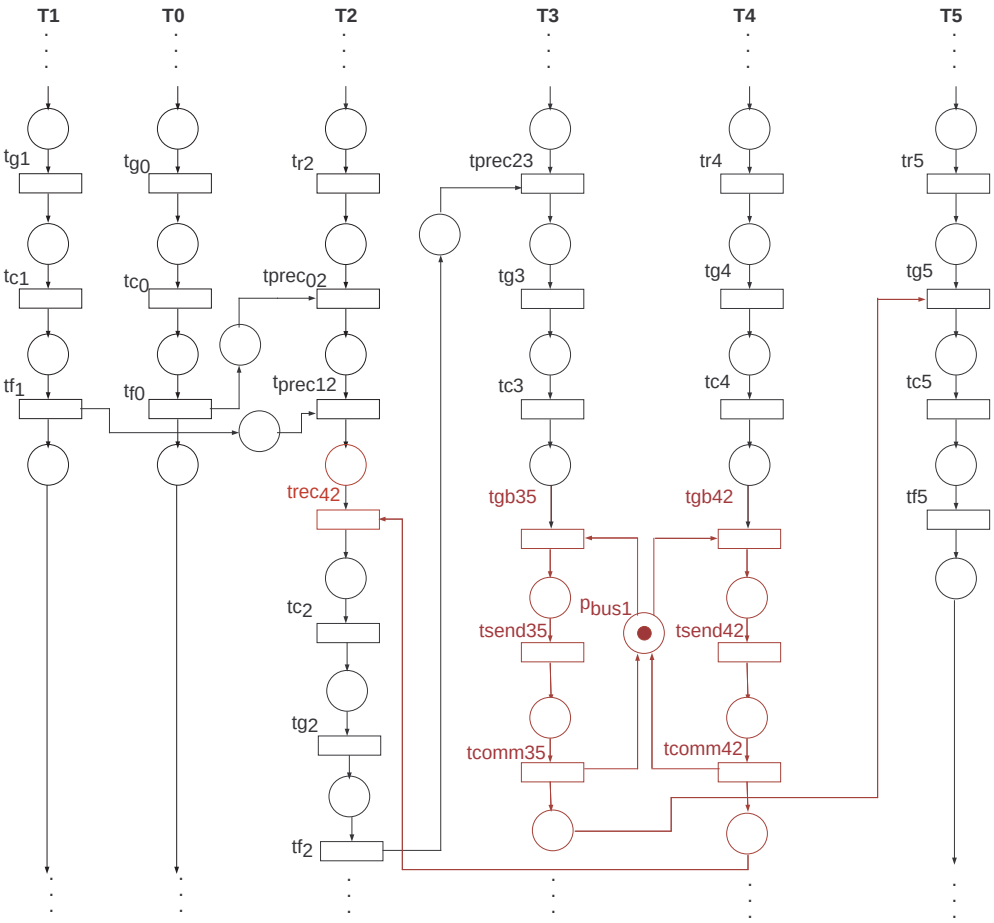

Figure 5.17: A Simple Example of Inter-processor Communication

represents the communication between two tasks allocated to different processors.

Chapter 6

Software Synthesis

This chapter aims to present the software synthesis approach proposed by this dissertation. Software synthesis is the task of converting a high-level specification into software, with lower overheads and satisfying all timing constraints. Firstly, a scheduling synthesis algorithm is presented for finding a feasible schedule that satisfies all timing constraints. As stated before, this work uses the pre-runtime scheduling policy, where schedules are computed entirely off-line. Starting from the found feasible schedule, the code generation phase can be started. Firstly, the code generation considering a single processor is shown. After that, the method for multiple processors is presented.

6.1 Scheduling Synthesis

Starting from the time Petri net model, a scheduling synthesis algorithm is utilized to generate a timed labeled transition system that represents a feasible pre-runtime schedule, considering that such schedule exists. This dissertation adopts the algorithm proposed in [8].

In a nutshell, the algorithm consists in recursively checking all successor states, starting from an initial state, by firing all enabled transitions in each state. This method is called *state space exploration*. Although using this simple mechanism a feasible schedule can be found, this solution leads to a well-known problem, namely, *state explosion problem* [17, 51]. In order to mitigate this problem, the algorithm applies some techniques for reducing the state space size.

The scheduling algorithm proposed in [8] has one shortcoming, which is the huge set of visited states that need to be stored in memory due to the tagging scheme (see Section 6.1.2). Therefore, this dissertation extends the implementation of this algorithm applying a method for compressing visited states in order to reduce memory consumption.

In the next subsection, an overview of the minimization techniques adopted by the

proposed method for reducing the state space size is presented. After that, the scheduling synthesis algorithm is explained. Finally, an example showing the application of the algorithm is depicted.

6.1.1 Minimizing State Space Size

In order to avoid the analysis of all possible states, the scheduling algorithm adopts two minimization techniques: partial-order reduction and elimination of undesirable states.

The first technique (partial-order reduction) associates for each transition class a choice-priority. The main idea is to avoid the verification of all interleaving possibilities of a set of firable transitions. When changing from one state to another, it is sufficient to analyze the class with the highest choice-priority and prunning the other ones, in such a way that the exploration occurs only in part of the state space. The highest choice-priorities are given to transitions that do not disable other transitions. These transitions are called independent. In other words, it does not matter in which order independent transitions fire, the final result is always the same. Examples of these transitions are arrival and release transitions. On the other hand, transitions that disable other firable transitions have the lowest choice-priority. Examples of these transitions are processor-granting and exclusion transitions. The choice-priorities adopted in this work is depicted in Table 6.1. For more information about partial-order reduction, the interested reader is encouraged to refer to [17], [51] , and [8].

Table 6.1: Choice-priorities for each transition class

Choice-priority	Transition
1	Final
2	Arrival
3	Release
4	Precedence
5	Computation
6	Exclusion
7	ProcessorGranting
8	BusGranting

The second technique aims to remove undesirable states, more specifically, states that represent deadline missing. This technique is very simple, it just prunes firable transitions that lead to undesirable states, for instance, deadline-checking transitions. Deadline-checking transitions should not fire unless deadlines are not met.

```
1 scheduling-synthesis(S,M^F,TPN)
2 {
3   if (S.M = M^F) return TRUE;
4   tag(S);
5   PT = remove-undesirable(partial-order(firable(S)));
6   if (|PT| = 0) return FALSE;
7   for each (⟨t,θ⟩ ∈ PT) {
8     S'= fire(S, t, θ);
9     if (untagged(S') ∧ scheduling-synthesis (S',M^F,TPN)){
10      add-in-trans-system (S,S',t,θ);
11      return TRUE;
12    }
13  }
14  return FALSE;
15 }
```

Figure 6.1: Scheduling Synthesis Algorithm

6.1.2 Pre-Runtime Scheduling Algorithm

As stated before, the pre-runtime scheduling algorithm used in this dissertation was proposed in [8]. This algorithm is a depth-first search method that generates as output a TLTS (Definition 5.9), which represents a feasible schedule.

In [8], the author shows some properties of the proposed Petri net model. One property of interest is boundedness (Chapter 4), which implies that the TLTS is finite and thus the proposed algorithm always finishes.

Figure 6.1 depicts the algorithm, which was implemented using C language. Although presented as a recursive function, the algorithm was iteratively implemented due to performance issues. The algorithm execution is described as follows. Firstly, the algorithm checks if the desired final marking (M^F) was reached. By reaching this marking, it means that a feasible schedule satisfying all timing constraints was found (line 3). For avoiding the state space explosion, the partial-order reduction and the prunning of undesirable states is applied in the set of firable transitions (line 5). `PT` is a set of ordered pairs $\langle t, \theta \rangle$ representing, for each post-pruning fireable transition, all possible firing time in the firing domain (Definition 5.6). The *tagging scheme* (lines 4 and 9) ensures that no state is visited more than once. The function `fire` (line 8) returns a new generated state (Definition 5.7) due to the firing of transition t at time θ. The feasible schedule is represented by a TLTS generated by the function `add-in-trans-system` (line 10). The whole state space is visited only when the system does not have a feasible schedule (line 14), where the algorithm returns `FALSE`.

Tagging Scheme

As described before, the tagging scheme is adopted to avoid visiting a state more than once. Such scheme considerably improves the algorithm execution. As explained

in [8], an experiment was performed without the tagging scheme. The algorithm took more than 48 hours to return a feasible schedule. Using the tagging scheme, the same schedule was found in just 2.5 seconds.

Although the tagging scheme optimizes the algorithm execution, memory consumption can be very high, since the set of visited states may be very large. In order to mitigate this problem, a function to compress the visited states was implemented. The set of visited states stores three information: marking(M), set of enabled transitions (ET) and a clock (C). All these information are stored as a contiguous list. Figure 6.2 depicts the structure that represents a state.

```
struct state {
    int *M;
    int *ET;
    int *VC;
    int sizeM;
    int sizeET;
};
```

Figure 6.2: State Structure

```
struct state_compressed {
    unsigned short int *elements;
    unsigned int size;
};
```

Figure 6.3: State Compressed Structure

Analyzing the generated states of some experiments, the most important aspect observed is the great amount of zeros values in the marking vector(M). Hence, the main idea of the compressing function is to reduce the representation of zero sequences. A new data structure was implemented to represent the compressed state (*state_compressed*), which is composed of an vector of `unsigned short int`. Figure 6.3 shows the structure that represents a compressed state. The compressing function acts in the M vector as follows. A sequence of more than one zero is represented by two elements in the list: the identifier `65,535` (reserved number) followed by the number of zeros in the sequence; a sequence of just one zero or any other number directly copied from the M vector to the new structure. As an example, using the state depicted in Figure 6.4, the respective compressed state is shown in Figure 6.5. Supposing that an `int` variable requires 4 bytes of memory and an `unsigned short`

`int` requires 2 bytes, the state size was reduced from $32 \times 4 = 128$ bytes to $22 \times 2 = 44$ bytes. For this simple example, the compression rate was 65%.

```
state->M = {0,0,0,0,0,0,0,0,0,0,1,0,1,1,1,0,3,0,0,0};
state->ET = {1,2,4,6,7};
state->VC = {0,0,0,2,1};
state->sizeM = 20;
state->EC = 5;
```

Figure 6.4: State example

```
compressedState->elements = {65535,10,1,0,1,1,1,0,3,65535,3,
                             1,2,4,6,7,0,0,0,2,1};
compressedState->size = 21;
```

Figure 6.5: Compressed State example

Considering the experiments adopted in this work, this very simple method results in compression rate of about 70%. This compression rate increases so as the amount of zero sequences in the M vector. Therefore, the larger the number of places the higher compression rate. Chapter 7 describes some experiments adopted to show the feasiability of the compressing function.

6.1.3 Application of the Algorithm

Table 6.2: Simple Specification

task	ph	r	c	d	p
τ_1	0	0	2	7	8
τ_2	0	2	2	6	6

The simple task set shown at Table 6.2 produces the TPN model outlined in Figure 6.6. For this example, the LCM is 24, resulting in 7 task instances. Applying the scheduling algorithm, the feasible firing schedule found for the TPN model of Figure 6.6, expressed as a TLTS, is depicted as follows: $s_0 \xrightarrow{(t_{start},0)} s_1 \xrightarrow{(t_{ph1},0)} s_2 \xrightarrow{(t_{ph2},0)} s_3 \xrightarrow{(t_{r1},0)} s_4 \xrightarrow{(t_{p1},0)} s_5 \xrightarrow{(t_{r2},2)} s_6 \xrightarrow{(t_{c1},0)} s_7 \xrightarrow{(t_{f1},0)} s_8 \xrightarrow{(t_{p2},0)} s_9 \xrightarrow{(t_{c2},2)} s_{10} \xrightarrow{(t_{f2},0)} s_{11} \xrightarrow{(t_{a2},2)} s_{12} \xrightarrow{(t_{a1},2)} s_{13} \xrightarrow{(t_{r1},0)} s_{14} \xrightarrow{(t_{r2},0)} s_{15} \xrightarrow{(t_{p2},0)} s_{16} \xrightarrow{(t_{c2},2)} s_{17} \xrightarrow{(t_{f2},0)} s_{18} \xrightarrow{(t_{p1},0)} s_{19} \xrightarrow{(t_{a2},2)} s_{20} \xrightarrow{(t_{c1},1)} s_{21} \xrightarrow{(t_{f1},0)} s_{22} \xrightarrow{(t_{r2},1)} s_{23} \xrightarrow{(t_{p2},0)} s_{24} \xrightarrow{(t_{c2},2)} s_{25} \xrightarrow{(t_{f2},0)} s_{26} \xrightarrow{(t_{a1},1)} s_{27} \xrightarrow{(t_{r1},0)} s_{28} \xrightarrow{(t_{p1},0)} s_{29} \xrightarrow{(t_{a2},1)} s_{30} \xrightarrow{(t_{c1},1)} s_{31} \xrightarrow{(t_{f1},0)} s_{32} \xrightarrow{(t_{r2},1)} s_{33} \xrightarrow{(t_{p2},0)} s_{34} \xrightarrow{(t_{c2},2)} s_{35} \xrightarrow{(t_{f2},0)} s_{36} \xrightarrow{(t_{end},0)} s_{37}$.

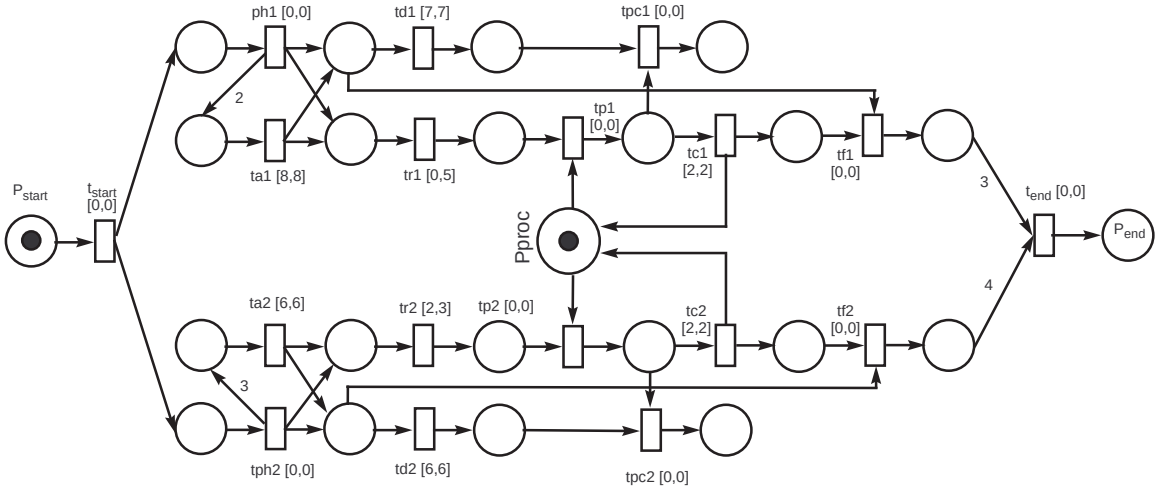

Figure 6.6: TPN for the task set in Table 6.2

6.2 Scheduled Code Generator Framework for One Processor

As presented before (Definition 5.9), the feasible firing schedule is expressed as a TLTS. The code is generated by traversing the TLTS, and detecting the time when the tasks should be executed. Thus, the generated code should execute the tasks in accordance with the previously computed schedule. A special data structure called *pre-runtime schedule table* is created for defining information about each task instance, for example, start time, and a pointer to a C function containing the code. More details about this data structure is presented in this Chapter. In the proposed method, the code for each task comes directly from the code associated with each computation transition in the TPN model.

In order to manage the execution of tasks, the code generation includes a small dispatcher to treat this activity. The timer is programmed by the dispatcher to interrupt the processor at the time instant where the next task must be executed (or resumed). It is worth observing that just one *timer* is needed since the generated code is already scheduled.

The code generation presented in this section considers one processor. Next section describes the code generation framework considering multiple processors.

6.2.1 Scheduled Code Generation

The proposed method for code generation includes not only the code of tasks (implemented by C functions), but also includes a timer interrupt handler, and a small dispatcher. Such dispatcher is adopted to automate several controls needed to the execution of tasks. Timer programming, context saving, context restoring, and tasks' calling are examples of such additional controls. The timer interrupt handler always transfers the control to the the dispatcher, which evaluates the need for performing either context saving or restoring, and calling the specific task. It is worthwhile to remind that, as presented before (Chapter 5), the proposed method considers that the timer is always programmed by a multiple of the TTU.

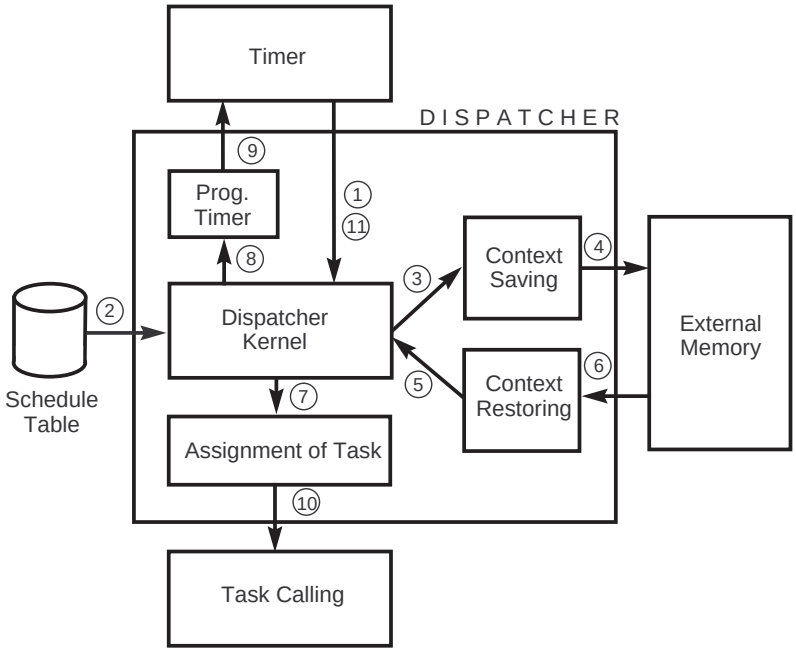

Figure 6.7: Proposed Code Generator Overview

Figure 6.7 overviews the proposed code generator framework, where the dispatcher is the main component. Figure 6.8 shows a simplified version of the proposed dispatcher function. Using Figure 6.7, the description of the code generator framework can be summarized as follows.

1. When the system starts, the timer is programmed using the first entry in the schedule table. Whenever the timer overflows, the timer interrupt handler is called, and the control is transferred to the dispatcher kernel. This dispatcher kernel uses the current clock (line 4 of Figure 6.8) to check if there is a task to be executed at this time;

2. The dispatcher kernel consults the schedule table for evaluating when and which is the next task to be executed. This table is stored as an array of `struct scheduleItem`. This array, representing the schedule table, is accessed as a circular list (line 13 of Figure 6.8);

3. The dispatcher kernel saves the context of the current task (line 7 of Figure 6.8) if the current task is being preempted by the new task. This information is obtained by a global variable called `existTaskInExecution` (line 6 of Figure 6.8). This variable has true value if, at a specific time instant, any task is running, and false otherwise;

4. The dispatcher kernel uses the external memory for storing such context;

5. The dispatcher kernel restores the context of the new task (line 10 of Figure 6.8), if it is returning from a preemption. This information comes from the schedule table;

6. The dispatcher kernel accesses the external memory in order to get such context;

7. Using the schedule table, the dispatcher kernel assign the next task function code (`functionPointer` at line 12 of Figure 6.8) to the global pointer variable `taskFunction`. At this point, the next task becomes the current task;

8. The dispatcher kernel uses the information of the schedule table for programming the timer to interrupt at the beginning of the next task execution (line 14 of Figure 6.8). It is worth observing that `scheduleIndex` was incremented at line 13 of Figure 6.8.

9. The timer is activated (line 15 of Figure 6.8);

10. A C-function, that corresponds to the current task, is executed.

11. When the timer interrupts, the control is again transferred to the dispatcher.

As defined before in this section, the schedule table is stored in an array of `struct ScheduleItem`. In particular, there is one entry in the array for each *execution part* of a task instance. That is, in case of preemption, a task instance may have more than one execution part. The `struct ScheduleItem` contains the following information: (i) start time; (ii) a flag indicating if either it is a preemption returning or not; (iii) task id; and (iv) a pointer to a function that represents the code of the respective task. Figure 6.9 shows the schedule table for a preemptive example that contains 7 task instances and 4 preemptions. Thus, the array has 11 entries. Figure 6.10 presents the respective timing diagram.

```
 1 void dispatcher()
 2 {
 3   struct ScheduleItem newTaskInfo = scheduleTable[scheduleIndex];
 4   globalClock = newTaskInfo.clock;
 5
 6   if(existTaskInExecution) {
 7     // context saving
 8   }
 9   if(newTaskInfo.isPreemptionReturn) {
10     // context restoring
11   }
12   taskFunction = newTaskInfo.functionPointer;
13   scheduleIndex = ((++scheduleIndex) % SCHEDULE_SIZE);
14   programmingTimer(scheduleTable[scheduleIndex].clock);
15   activateTimer();
16 }
```

Figure 6.8: Simplified Version of the Dispatcher

```
struct ScheduleItem  scheduleTable [SCHEDULE_SIZE] =
  {{ 1, false, 1, (int *)TaskA},
   { 4, false, 2, (int *)TaskB},
   { 6, false, 3, (int *)TaskC},
   { 8, true,  2, (int *)TaskB},
   {10, false, 4, (int *)TaskD},
   {11, true,  2, (int *)TaskB},
   {13, true,  1, (int *)TaskA},
   {18, false, 1, (int *)TaskA},
   {20, false, 3, (int *)TaskC},
   {22, false, 2, (int *)TaskB},
   {28, true,  1, (int *)TaskA}
};
```

Figure 6.9: Example of a Schedule Table

The generated code has a set of global variables. Figure 6.8 shows some of them which stores, for instance, the number instances of tasks (`SCHEDULE_SIZE`), information of the task currently executing (`newTaskInfo`), global clock value (`globalClock`); and a pointer to the task function to be executed (`taskFunction`).

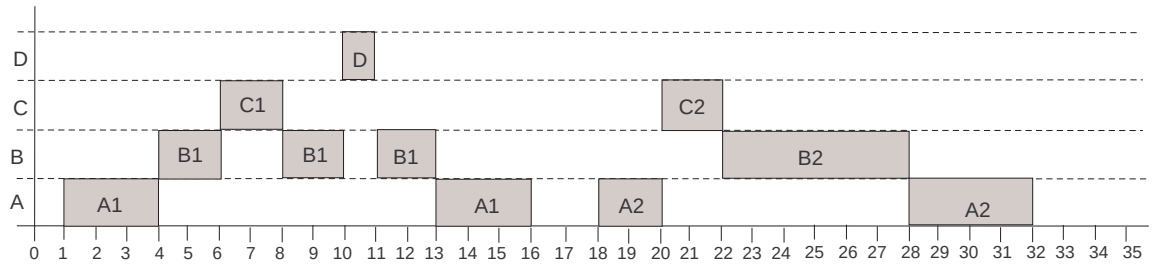

Figure 6.10: Timing Diagram for Schedule Table in Figure 6.9

6.3 Scheduled Code Generation Framework for Multiple Processors

This section presents the code generation approach for embedded hard real-time systems with multiple processors.

In the proposed hard real-time method considering multiple processors, a special mechanism is necessary for dealing with real-time clock synchronization. In this way, this section presents an architecture for solving the problem related to real-time clock synchronization between processors.

6.3.1 An Architecture for Embedded Hard Real-Time Systems with Multiple Processors

Whenever considering hard real-time embedded systems design based on multiprocessor platforms, one fundamental concern is the mechanism for synchronizing processors. As an example, supposing a system composed of two traffic lights is hard time constrained. Each traffic light is controlled by one particular processor, which is responsible for turning on/off the set of lights (red, yellow, green). In addition, at the same time, when one traffic light changes from yellow to red, the other traffic light must change from red to green. How to keep both real-time clocks synchronized? Obviously, if both are green at the same, accidents may happen.

In order to tackle this problem, an specific architecture is proposed. The architecture is based on *Central Master Synchronization* [25], in the sense that a central master processor peforms the time counting and it sends periodically synchronization messages to slave processors for updating their respective real-time clock values. More specifically, the architecture is a modified version of a pattern, namely, *Shared-Clock Schedulers Using External Interrupts* [39]. Figure 6.11 depicts the pattern. The pattern defines a master processor for performing the time counting for the entire system. The master processor is responsible for propagating the real-time clock values to the slave processors through external interrupts. Additionally, there is a runtime scheduler

running in each processor (including the master processor) for performing the tasks scheduling.

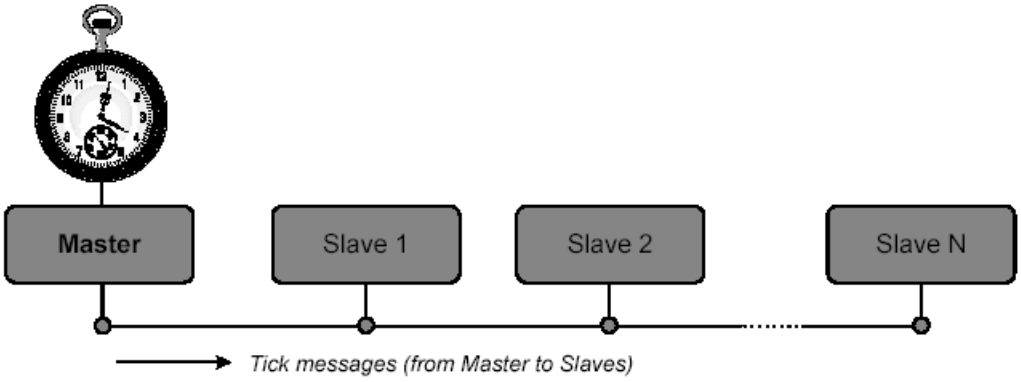

Figure 6.11: Shared-Clock Schedulers Using External Interrupts

The proposed architecture differs from the *Shared-Clock Schedulers Using External Interrupts* pattern in two points, summarized in the following steps:

- there is no runtime scheduler running in each processor, but a runtime dispatcher. The scheduling is performed using a pre-runtime approach, as described previously;

- the master processor is only responsible for performing the time counting and for notifying the slave processors. The master processor does not execute any task, but only a special dispatcher. In addition, the slave processors do not have their own real-time clocks. The real-time clock only resides in the master processor. Therefore, the master processor is called **Central Time Counting Processor - CTC Processor**. The slave processors are being named as **Node Processors**. Figure 6.12 depicts the proposed architecture.

It is worth stating that the terms multiprocessing and multiple processors are employed to refer to both multiprocessor and multicomputer architectures. As described in [20], a multiprocessor architecture is composed of a set of processors and a shared common memory. In contrast, a multicomputer architecture is composed of a set of processors and unshared distributed memories, and the processors are interconnected by a message-passing network. Although the terms multiprocessing and multiple processors are being employed, the architecture adopted in this work is multicomputer.

Figure 6.13 depicts a system composed of three 8051 microcontrollers adopting the proposed architecture. One 8051 is responsible for performing the time counting (CTC Processor), and the other two are responsible for performing the tasks' execution

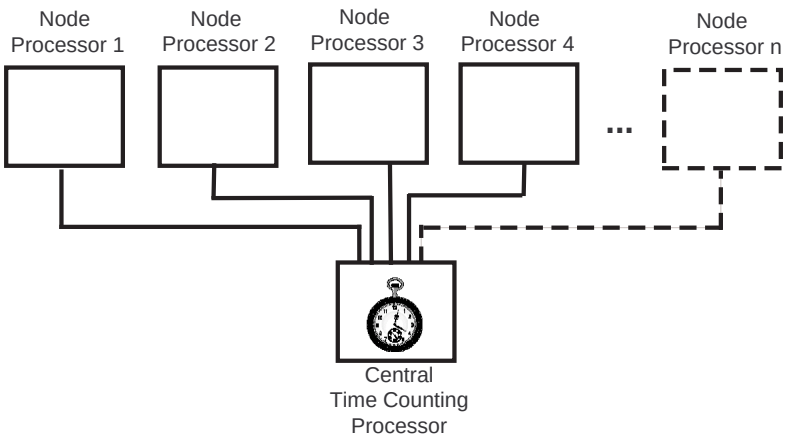

Figure 6.12: Proposed Multicomputer Architecture

(Node Processors). The CTC processor is connected with the node processors through a parallel port. A different pin of the parallel port is connected to the external interrupt of each node processor. Figure 6.13 shows the node processor 1 connected with the CTC processor using the pin P2.0 and node processor 2 connected using the pin P2.1 of the parallel port P2. If one or more node processors need to be interrupted, the CTC processor just sends a byte value to the parallel port in order to perform the interruption. Additionally, Figure 6.13 depicts two node processors connected by a serial channel. However, other channels may be adopted to perform the communication between the node processors, for instance, parallel communication. A comparison between different communication means is beyond of the scope of this work.

Although the current implementation is using 8051 microcontrollers, the architecture may be ported to other platforms. Moreover, the CTC processor does not need to be a powerful processor. This processor just need to have a timer and a parallel port. Another alternative is performing the synthesis of a specific hardware for doing this job.

Next section details the code generation taking into account this proposed architecture.

6.3.2 Scheduled Code generation

As presented previously, the feasible firing schedule is expressed as a TLTS. The code generation is performed by traversing the TLTS, and detecting the time where the tasks should be executed.

Adopting a multiprocessing environment, the communication tasks are also taken into account in the code generation. As stated before, this work does not consider

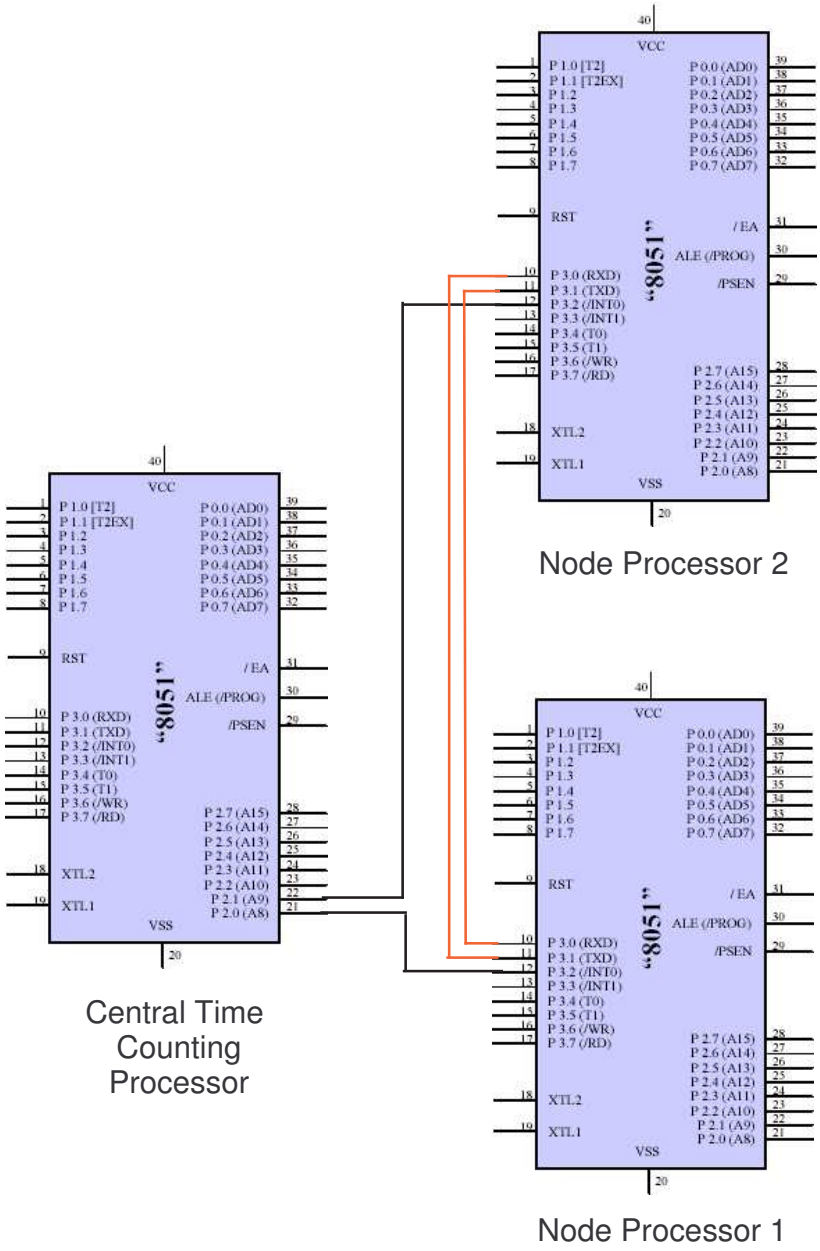

Figure 6.13: Proposed Architecture using 8051 Microcontroller

the data receiving via interrupt handling, since interrupts may affect the system predictability. The proposed approach adopts polling. Thus, a communication task (μ_m) is translated into two special tasks: `sendM`$_m$ and `receiveM`$_m$. Both tasks are executed

at the same time for guaranteeing the correct data transmission. `sendM`$_m$ is executed at sender side and `receiveM`$_m$ at receiver side. In addition, `sendM`$_m$ and `receiveM`$_m$ are considered in the *pre-runtime schedule table* and both can not be preempted.

In order to manage the tasks' execution, a dispatcher is automatically generated for performing the management. However, when adopting a multiprocessing environment, the dispatcher and the *pre-runtime schedule table* are divided in two parts :

- The first one is deployed in the CTC processor. The dispatcher is responsible for interrupting the node processors and to perform the timer programming, in accordance with the schedule table. In the CTC processor, the schedule table only contains the clock values and the node processors to be interrupted.
- The second one is deployed in the node processors. The dispatcher is responsible for performing the task calling, context saving and context restoring. In the node processors, each entry in the schedule table contains: (i) a flag indicating if either it is a preemption returning or not; (ii) task id; and (iii) a pointer to a function that represents the code of the respective task. It is worthwhile to pointing out that the schedule table only contains the tasks to be executed in the respective node processor.

Figure 6.14 overviews the proposed code generator framework considering a multiprocessing environment. Figure 6.15 shows a simplified version of the proposed dispatcher for the CTC processor and Figure 6.16 depicts the dispatcher, which will be running in each node processor. Using Figure 6.14, the description of the code generator framework can be summarized as follows.

1. When the system starts, the timer is programmed using the first entry in the schedule table. Whenever the timer overflows, the timer interrupt handler is called, and the control is transferred to the dispatcher kernel of the CTC processor.
2. The dispatcher kernel consults the schedule table for evaluating which node processors should be interrupted. This table is stored as an array of `struct ScheduleItemCTC`. This array, representing the schedule table in CTC processor, is accessed as a circular list (line 7 of Figure 6.15);
3. A byte value is sent to a parallel port (line 5 of Figure 6.15), where the nodes are connected, for interrupting a subset of the node processors;
4. The dispatcher uses the information of the schedule table for programming the timer to interrupt node processor(s) at the beginning of the next task execution (line 8 of Figure 6.15). It is worth observing that `scheduleIndexCTC` was incremented at line 7 of Figure 6.15.

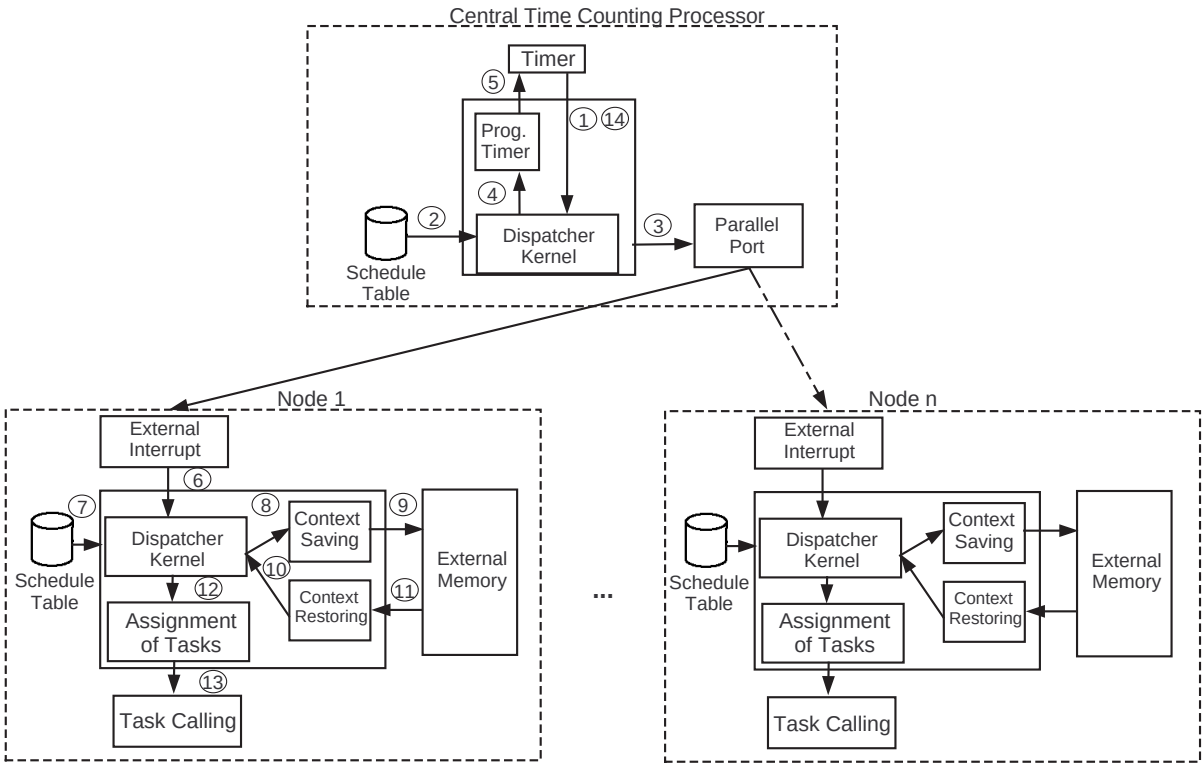

Figure 6.14: Proposed Code Generation Overview

5. The timer is activated (line 9 of Figure 6.15);

6. When a node receives an external interrupt performed by the CTC processor, the external interrupt handler is forced to be called, and the control is transferred to the dispatcher kernel in the node processor.

7. The dispatcher kernel consults the schedule table for evaluating which is the next task to be executed. This table is stored as an array of `struct ScheduleItemNode`. This array, representing the schedule table in a node processor, is accessed as a circular list (line 13 of Figure 6.16);

8. The dispatcher kernel saves the context of current task (line 6 of Figure 6.16) if the current task is being preempted by a new task. This information is obtained from a global variable called `existTaskInExecution` (line 5 of Figure 6.16). This variable has true value if, at a specific time instant, any task is running, and false otherwise;

9. The dispatcher kernel uses the external memory for storing such context;

10. The dispatcher kernel restores the context of the new task (line 9 of Figure 6.16), if it is returning from a preemption. This information comes from the schedule table;

11. The dispatcher kernel accesses the external memory in order to get such context;

12. Using the schedule table, the dispatcher kernel assign the next task function code (`functionPointer` at line 11 of Figure 6.16) to the global pointer variable `taskFunction`. At this point, the next task becomes the current task. If it is returning from a preemption, a new task instance will not be created, but the restored context will be considered.

13. A C-function, that corresponds to the current task, is executed.

14. When the timer interrupts, the control is again transferred to the dispatcher in the CTC processor.

```
1 void ctcDispatcher()
2 {
3   struct ScheduleItemCTC nodesInfo = scheduleTableCTC[scheduleIndexCTC];
4
5   parallelPort = nodesInfo.nodes;
6
7   scheduleIndexCTC = ((++scheduleIndexCTC) % SCHEDULE_SIZE_CTC);
8   programmingTimer(scheduleTableCTC[scheduleIndexCTC].clock);
9   activateTimer();
10 }
```

Figure 6.15: Simplified Version of the CTC Processor Dispatcher

Even in a multiprocessing environment, it is worth observing that just one timer is needed since the generated code is already scheduled.

Let us take a look at how to apply the proposed code generation method considering the specification depicted in Table 6.3. This specification is automatically translated into the TPN model of Figure 6.17. For a better understanding, this figure does not show the timing constraints, the deadline checking, the bus, the precedence relation, and the shared processors (P1 and P2). Moreover, the LCM is 500, which results in 7 tasks instances. Applying the scheduling synthesis algorithm (Figure 6.1) on the TPN model, the following feasible firing schedule was found: $s_0 \xrightarrow{(t_{start},0)} s_1 \xrightarrow{(t_{ph0},0)} s_2 \xrightarrow{(t_{r0},0))} s_3 \xrightarrow{(t_{ph1},0)} s_4 \xrightarrow{(t_{r1},0)} s_5 \xrightarrow{(t_{ph2},0)} s_6 \xrightarrow{(t_{r2},0)} s_7 \xrightarrow{(t_{ph3},0)} s_8 \xrightarrow{(t_{r3},0)} s_7 \xrightarrow{(t_{ph4},0)} s_8 \xrightarrow{(t_{r4},0)} s_{10} \xrightarrow{(t_{g0},0)} s_{11} \xrightarrow{(t_{g4},0)} s_{12} \xrightarrow{(t_{c0},10)} s_{13} \xrightarrow{(t_{c4},0)} s_{14} \xrightarrow{(t_{f0},0)} s_{15} \xrightarrow{(t_{f4},0)} s_{16} \xrightarrow{(t_{g1},0)} s_{17} \xrightarrow{(t_{c1},40)} s_{18} \xrightarrow{(t_{f1},0)} s_{19} \xrightarrow{(t_{gb2},0)}$

```
 1 void nodeDispatcher()
 2 {
 3   struct ScheduleItemNode newTaskInfo = scheduleTableNode[scheduleIndexNode];
 4
 5   if(existTaskInExecution) {
 6     // context saving
 7   }
 8   if(newTaskInfo.isPreemptionReturn) {
 9     // context restoring
10   } else {
11     taskFunction = newTaskInfo.functionPointer;
12   }
13   scheduleIndexNode = ((++scheduleIndexNode) % SCHEDULE_SIZE_NODE);
13 }
```

Figure 6.16: Simplified Version of the Node Processor Dispatcher

Table 6.3: Task Timing Specification Considering 2 Processors

Task	Release	Comp.	Deadline	Period	Processor/Bus	From	To
T_0	0	10	10	250	P1	-	-
T_1	0	40	200	500	P1	-	-
T_2	0	20	150	500	P2	-	-
T_3	0	10	450	500	P2	-	-
T_4	0	10	10	250	P2	-	-
M_1	-	2	-	-	bus1	T_1	T_2
Intertask Relations							
T_2 PRECEDES T_3							

$s_{20} \xrightarrow{(t_{send2},2)} s_{21} \xrightarrow{(t_{comm2},0)} s_{22} \xrightarrow{(t_{endcomm1},0)} s_{23} \xrightarrow{(t_{rec2},0)} s_{24} \xrightarrow{(t_{g2},0)} s_{25} \xrightarrow{(t_{c2},20)} s_{26} \xrightarrow{(t_{f2},0)} s_{27} \xrightarrow{(t_{prec3},0)}$
$s_{28} \xrightarrow{(t_{g3},0)} s_{29} \xrightarrow{(t_{c3},10)} s_{30} \xrightarrow{(t_{f3},0)} s_{31} \xrightarrow{(t_{a0},168)} s_{32} \xrightarrow{(t_{r0},0)} s_{33} \xrightarrow{(t_{a4},0)} s_{34} \xrightarrow{(t_{r4},0)} s_{35} \xrightarrow{(t_{g0},0)} s_{36} \xrightarrow{(t_{g4},0)}$
$s_{37} \xrightarrow{(t_{c0},10)} s_{38} \xrightarrow{(t_{c4},0)} s_{39} \xrightarrow{(t_{f0},0)} s_{40} \xrightarrow{(t_{f4},0)} s_{41} \xrightarrow{(t_{end},0)} s_{42}$. Figure 6.18 depicts the respective timing diagram, considering both processors and their respective tasks, including the communication task.

Analyzing the schedule, which is represented by a TLTS, it has been found out that task $T0$ is executed twice, for clock values equal to 0 and 250, and task $T1$ is executed once, when the clock value is equal to 10. The same way, $T2$ is executed once, for clock value 52, $T3$ is executed once, for clock value 72, and $T4$ is executed twice, for clock values 0 and 250. Additionally, the communication task M_1 is executed once, when the clock value is equal to 50.

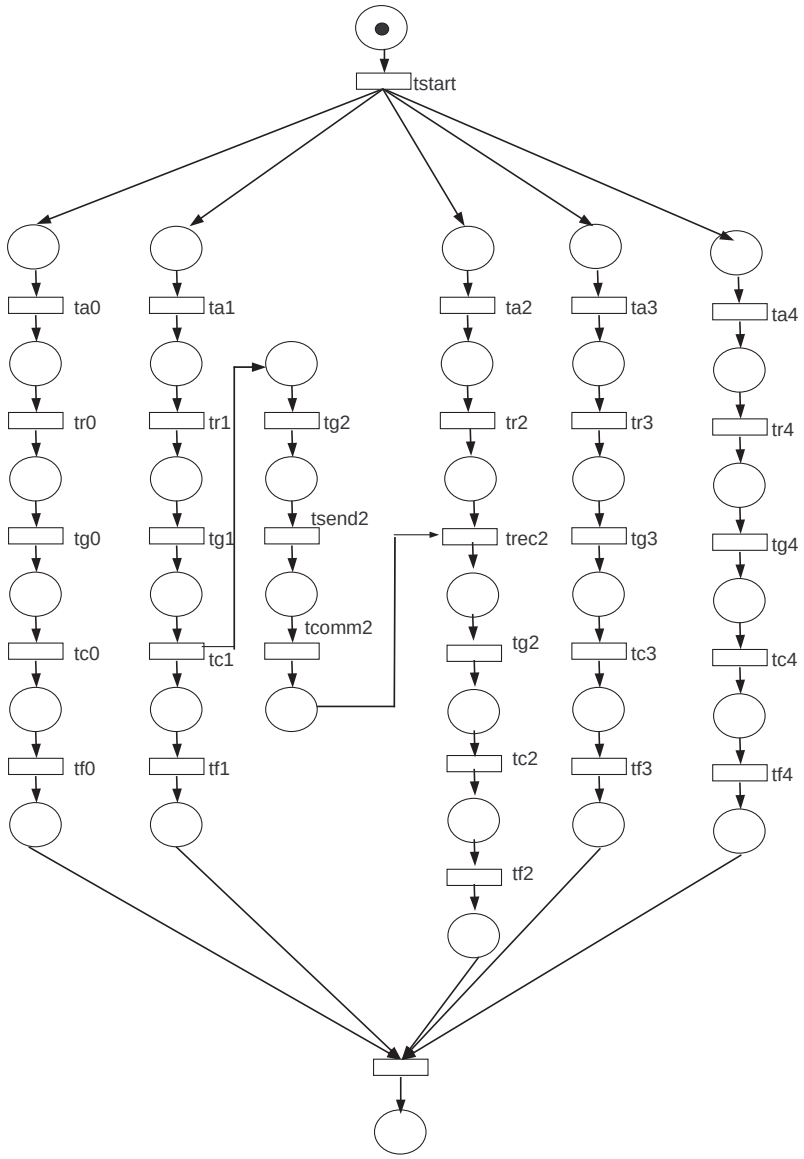

Figure 6.17: TPN for the task specification in Table 6.3

Examining the task specification (Table 6.3), three processors are required: one for performing the time counting (CTC Processor), and two for executing the tasks (Node Processors). Moreover, the processors need to be linked using the architecture described in Section 6.3.1. Figure 6.19 shows the code generated for the node processor 1, and Figure 6.20 shows the code generated for the node processor 2. As described previously, it is worth observing that each entry in the schedule table of each node processor contains: (i) a flag indicating if either it is a preemption returning or not;

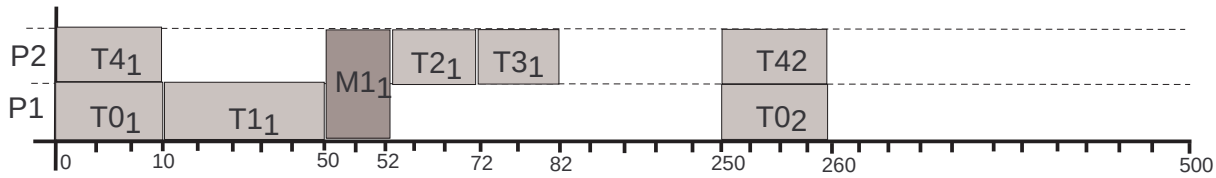

Figure 6.18: Timing Diagram Considering Both Processors

```
void sendM1() {...} void taskT0() {...}
void taskT1() {...}

#define SCHEDULE_SIZE_NODE 4

struct ScheduleItemNode schedule[SCHEDULE_SIZE_NODE] =
{
  {false, 0, (int *)taskT0},
  {false, 1, (int *)taskT1},
  {false, 5, (int *)sendM1},
  {false, 0, (int *)taskT0}

};
```

Figure 6.19: Generated code for the node processor 1

```
void receiveM1() {...} void taskT2() {...}
void taskT3() {...}    void taskT4() {...}

#define SCHEDULE_SIZE_NODE 5

struct ScheduleItemNode schedule[SCHEDULE_SIZE_NODE] =
{
  {false, 4, (int *)taskT4},
  {false, 6, (int *)receiveM1},
  {false, 2, (int *)taskT2},
  {false, 3, (int *)taskT3},
  {false, 4, (int *)taskT4}
};
```

Figure 6.20: Generated code for the node processor 2

(ii) task id; and (iii) a pointer to a function that represents the code of the respective task. Figure 6.21 depicts the code generated for the CTC processor. For this processor, the schedule table contains: (i) clock value; and (ii) node processors to be interrupted. In order to interrupt the node processors, 0 bits are being used, since all experiments were performed using the 8051 platform. In this platform, the external interrupts

```
#define SCHEDULE_SIZE_CTC 6

struct ScheduleItemCTC schedule[SCHEDULE_SIZE_CTC] =
{
  {0,   252}, //11111100 - Proc 1 and 2
  {10,  254}, //11111110 - Proc 1
  {50,  252}, //11111100 - Proc 1 and 2
  {52,  253}, //11111101 - Proc 2
  {72,  253}, //11111101 - Proc 2
  {250, 252} //11111100 - Proc 1 and 2

};
```

Figure 6.21: Generated code for the CTC processor

were activated using the low level sensitive setting. As an example, let us consider the first entry in the schedule table depicted in Figure 6.21. The node processors to be interrupted are represented by the value 252. Converting to binary, 252 means 11111100. The value indicates that two processors need to be notified at clock value 0, more specifically, the processors connected at pin 0 and 1 of the parallel port. Additionally, it should be mentioned that when the clock value is equal to 50, both processors are interrupted in order to execute the data transmission, which is performed by tasks `sendM1` and `receiveM1`.

Table 6.4: Task Timing Specification Considering 2 Communication Tasks

Task	Release	Comp.	Deadline	Period	Processor/Bus	From	To
T_0	0	10	100	250	P1	-	-
T_1	0	40	200	500	P1	-	-
T_2	0	20	150	500	P2	-	-
T_3	0	10	450	500	P2	-	-
T_4	0	20	500	500	P3	-	-
M_1	-	2	-	-	bus1	T_1	T_2
M_2	-	4	-	-	bus1	T_3	T_4

Let us examining a second example composed of two communication tasks (Table 6.4). This specification is automatically translated into the TPN model of Figure 6.22. Examining the task specification (Table 6.4), four processors are required: one for performing the time counting (CTC Processor), and three for executing the tasks (Node Processors). For this example, Figure 6.23 depicts the timing diagram that represents the feasible schedule. Figure 6.24, Figure 6.25, and Figure 6.26 show, respectively, the code generated for the node processor 1, node processor 2, and node processor 3. Lastly, Figure 6.27 depicts the generated code for the CTC processor.

For this example, Figure 6.28 shows the proposed architecture considering four 8051 microcontrollers. It is worth observing the way that all node processors are connected. Diodes are placed in each transmission pin (P3.1) in order to avoid equipment damage due to the transmission of multiple messages at the same time in the same communication channel. Diode is a component that allows an electric current to flow in one direction, but block it in the opposite direction. However, as the communication channel is taken into account in the scheduling synthesis, the problem stated should never occur.

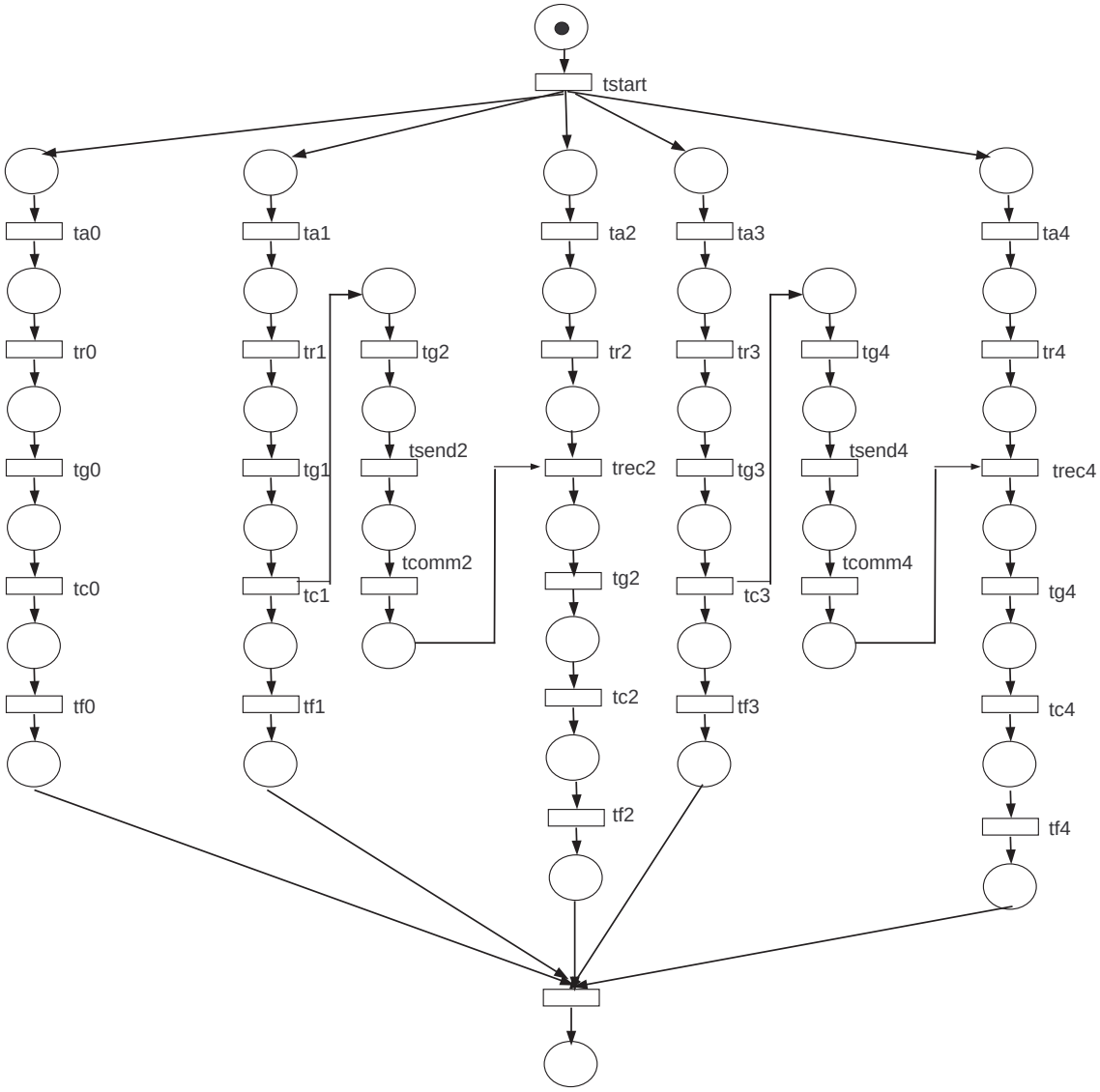

Figure 6.22: TPN for the task specification in Table 6.4

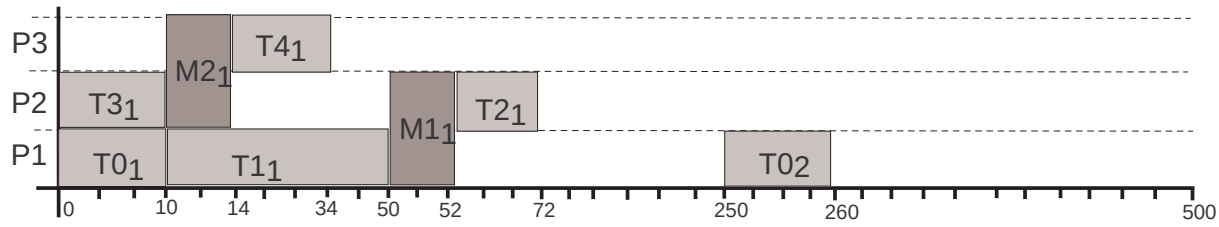

Figure 6.23: Timing Diagram for the TPN Model in Figure 6.22

```
void sendM1() {...}   void taskT0(){...}
void taskT1() {...}

#define SCHEDULE_SIZE_NODE 4

struct ScheduleItemNode schedule[SCHEDULE_SIZE_NODE] =
{
  {false, 0, (int *)taskT0},
  {false, 1, (int *)taskT1},
  {false, 4, (int *)sendM1},
  {false, 0, (int *)taskT0}

};
```

Figure 6.24: Generated code for the node processor 1

```
void sendM2() {...} void receiveM1() {...}
void taskT2() {...} void taskT3() {...}

#define SCHEDULE_SIZE_NODE 4

struct ScheduleItemNode schedule[SCHEDULE_SIZE_NODE] =
{
  {false, 3, (int *)taskT3},
  {false, 6, (int *)sendM2},
  {false, 5, (int *)receiveM1},
  {false, 2, (int *)taskT2}

};
```

Figure 6.25: Generated code for the node processor 2

```
void sendM2() {...} void receiveM1() {...}
void taskT2() {...} void taskT3() {...}

#define SCHEDULE_SIZE_NODE 2

struct ScheduleItemNode schedule[SCHEDULE_SIZE_NODE] =
{
  {false, 6, (int *)receiveM2},
  {false, 4, (int *)taskT4}

};
```

Figure 6.26: Generated code for the node processor 3

```
#define SCHEDULE_SIZE_CTC 6

struct ScheduleItemCTC schedule[SCHEDULE_SIZE_CTC] =
{
  {0,   252}, //11111100 - Tasks T3, T0
  {10,  248}, //11111000 - Tasks sendM2, receiveM2, T1
  {14,  251}, //11111011 - Task T4
  {50,  252}, //11111100 - Tasks sendM1 and receiveM1
  {52,  253}, //11111101 - Task T2
  {250, 254}  //11111110 - Task T0

};
```

Figure 6.27: Generated code for the CTC processor

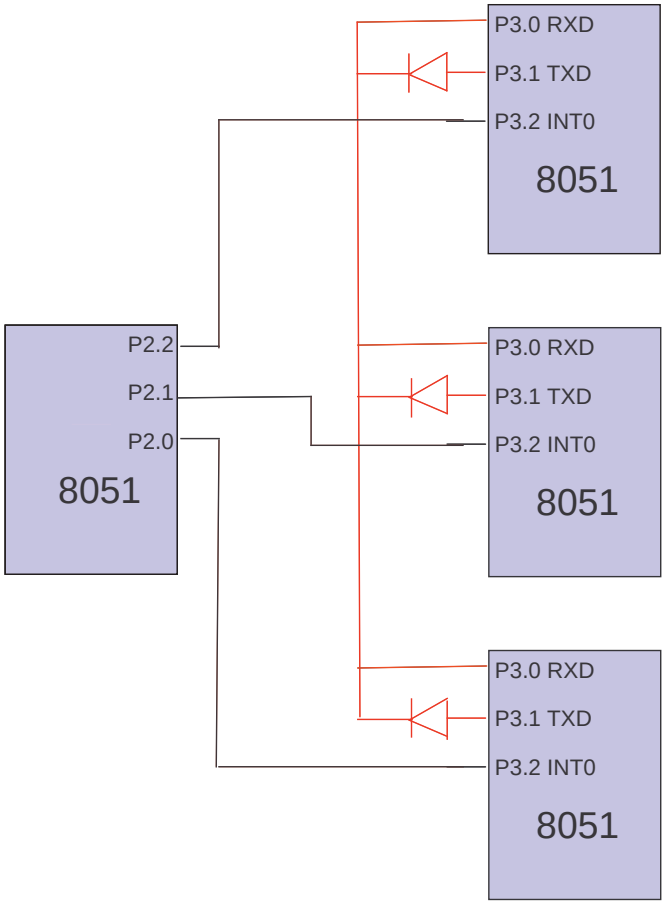

Figure 6.28: Proposed Architecture with four 8051 Microcontroller

6.4 Summary

This chapter described the proposed software synthesis approach. First of all, it was presented the pre-runtime scheduling algorithm, which is considered for finding feasible schedules from the formal model. As a contribution, an algorithm was developed for compressing the visited states with the purpose of reducing memory consumption.

Next, the code generation framework for one processor was presented. The feasible schedule is translated into a schedule table, which contains all information about the tasks' execution. Additionally, a dispatcher and a timer interrupt handler were described in order to provide several controls, for instance, timer programming, context saving, context restoring, and tasks calling

Finally, the code generation framework for multiple processors was described. Firstly, an architecture for dealing with real-time clock synchronization between processors was

adopted and adapted. Next, the code generation method was presented in order to provide predictable code considering a Central Time Counting Processor and Node Processors. In order to control the tasks' execution in a multiprocessing environment, two special dispatchers were described. Finally, the handling of communication tasks was presented.

The proposed code generation framework may be applied to several processor platforms. It is sufficient to make each dispatcher available for each respective platform. Furthermore, the Central Time Counting Processor does not need to be a powerful processor. This processor just need to have a timer and a parallel port.

Chapter 7

Experiments

This chapter aims to present the experiments carried out in this work in order to show the practical usability of the proposed methodology. First of all, Table 7.1 depicts some experiments adopted for demonstrating the feasibility of the compressing function, which is utilized to reduce the state space size. In this table, *instances* represent the number of task instances; *min* is the minimum number of states to be verified, which is equal to the number of transitions to be fired; *found* counts the number of states actually verified for finding a feasible schedule; *time* expresses the algorithm execution time in seconds without the compressing function; *size* is the state space size in kilobytes without compression; *time comp.* expresses the algorithm execution time in seconds with the compressing function; *comp. size* is the state space size in kilobytes with compression; and *method* states the chosen (preemptive (P) or non-preemptive (nP)) scheduling method. It is worth observing that the compressing function has provided a compression rate of about 70% for those case studies considered during this research. However, it increases the algorithm execution time.

In order to show the practical feasibility of the proposed software synthesis method, three case studies are presented in details : (i) simple control application, (ii) pulse oximeter, and (iii) vehicle monitoring system. Although all case studies considered adopts the 8051-based plataform, the approach may be ported to other plataforms, as described in Chapter 6.

All executions of the scheduling synthesis algorithm were performed on a Duron 1.2 GHz, 256 MB RAM, OS Linux, and compiler GCC 3.3.2.

7.1 Simple Control Application

The simple control application consists of a sensory device mounted on a motorized platform that must detect and track specific objects in the environment. This application was originally described in [15], and later used in [8]. Four processors are connected by a single bus. The model consists of 6 tasks split into 22 subtasks, which

Table 7.1: Compressing Function Experimental Results

Example	inst.	min	found	time (s)	size	time comp.	comp. size	method
Simple Example	7	30	30	0.002	3.589	0.002	1.427	nP
Xu&Parnas (example 3)	4	171	1558	0.118	250.616	0.121	93.422	P
Xu&Parnas (figure 9)	5	281	2406	0.164	647.671	0.166	213.097	P
Pulse-Oximeter	178	850	850	0.079	631.438	0.080	171.265	nP
Mine Pump Control	782	3130	3255	0.260	1575.667	0.273	603.231	nP
Unmanned Ground Vehicle	433	4701	14761	0.978	8665.132	1.036	3086.281	P

exchanges 10 messages, 6 of them are sent across processor boundaries.

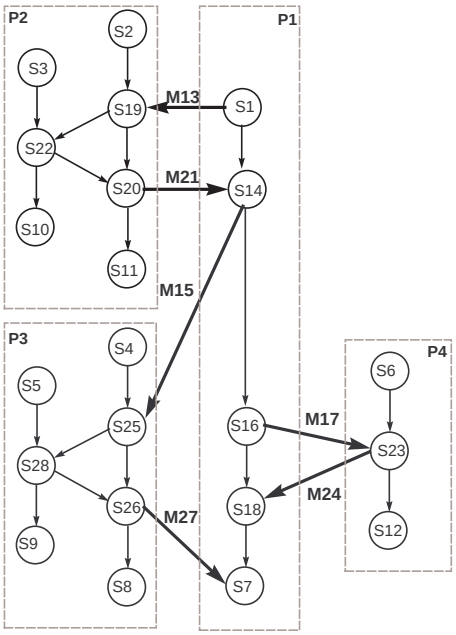

Figure 7.1: The Simple Control Application Graph

Figure 7.1 shows the communication graph, which depicts the communication between tasks. The graph shows the subtasks allocated to processors, and its communication pattern, where the interprocessor communications are labeled with "M" in the figure. Communication in the same processor is treated as precedence relation. Table 7.2 gives the worst-case execution time and deadline for each subtask as well as the worst-case communication time for each inter-processor communication. In this work, the period of all tasks is 200 and the task time unit is 1 second. Figure 7.2 presents a simplified time Petri net model for this case study using a non-preemptive scheduling method. Transitions GP stands for granting-processor, and GB stands for granting-bus. For a better understanding, this figure does not show the timing constraints, the

Table 7.2: Task Set for the Simple Control Application

Segment	C_i	D_i	Segment	C_i	D_i	Segment	C_i	D_i	Segment	C_i	D_i
S_1	3	100	S_8	2	100	M_{15}	1		S_{22}	6	40
S_2	3	200	S_9	2	100	S_{16}	10	100	S_{23}	10	200
S_3	3	40	S_{10}	2	40	M_{17}	1		M_{24}	1	
S_4	3	100	S_{11}	2	200	S_{18}	1	100	S_{25}	2	100
S_5	3	100	S_{12}	2	200	S_{19}	5	200	S_{26}	1	100
S_6	3	200	M_{13}	1		S_{20}	7	100	M_{27}	1	
S_7	2	100	S_{14}	15	100	M_{21}	1		S_{28}	7	100

deadline checking block, the bus, and the shared processors (P1, P2 and P3).

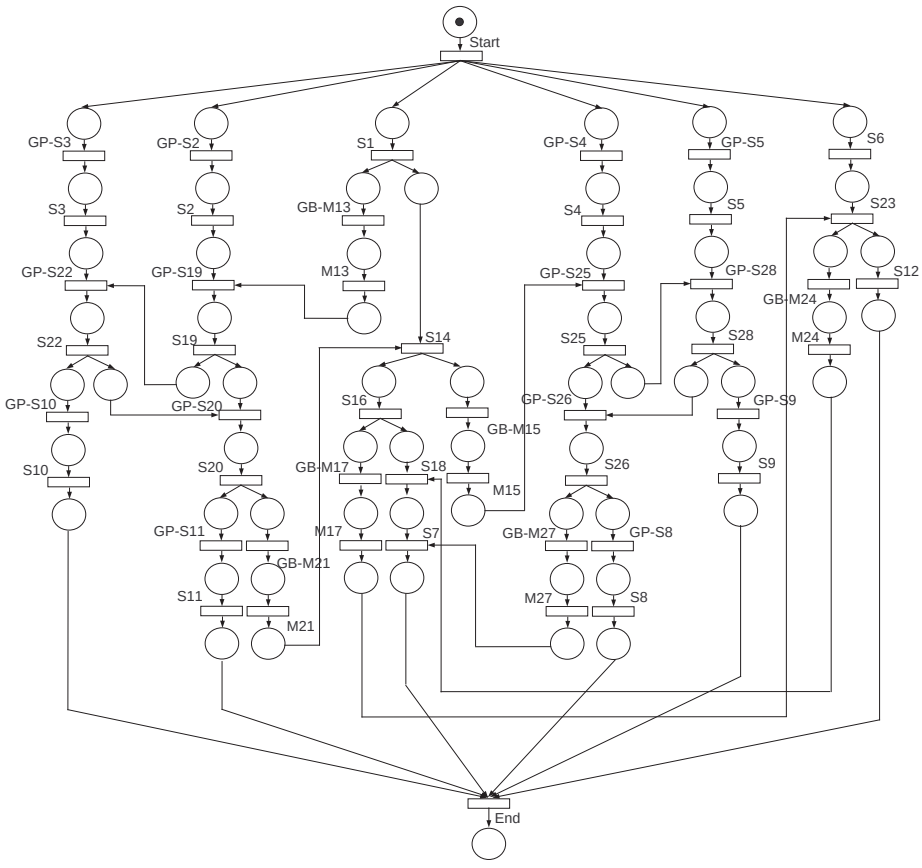

Figure 7.2: Simplified Simple Control Application Time Petri Net Model

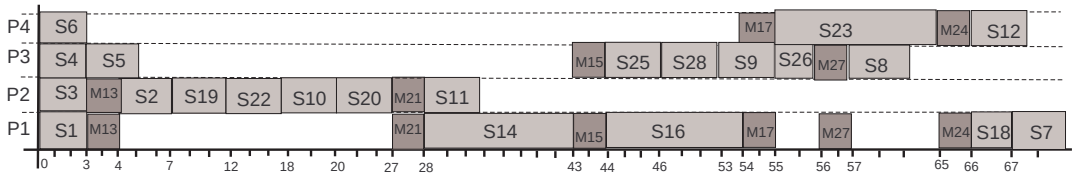

Figure 7.3: Timing Diagram for the Simple Control

The scheduling synthesis algorithm found a feasible schedule after examining 152 states in just 0.0181001 seconds. Figure 7.3 shows the timing diagram that graphically represents the found feasible schedule. The timing diagram depicts 4 processors with their respective task instances, including the communication tasks.

In order to apply the proposed software synthesis method, the system architecture needs to be compatible with the architecture presented in Chapter 6. Therefore, an additional processor needs to be considered for performing the time counting.

Traversing the feasible firing schedule, the C code generated for the processor 1 is depicted in Figure 7.4. For the processor 2 and 3, Figure 7.5 and Figure 7.6 depict the respective codes. Additionally, Figure 7.7 shows the generated code for the processor 4. Finally, the C code generated for the Central Time Counting Processor is shown in Figure 7.8.

7.2 Pulse Oximeter

The pulse oximeter [23] is an equipment responsible for measuring the oxygen saturation in the blood system using a non-invasive method. A pulse-oximeter may be used in many circumstances, like checking if the oxygen saturation is lower or not than the acceptable level, when a patient is sedated with anesthetics for a surgical procedure. This equipment is widely used in center care units (CCU) in hospitals.

Originally, a particular pulse oximeter was developed using a single microcontroller. Since this equipment is composed of several CPU-bound tasks, a pricey, powerful microcontroller was adopted. However, using the proposed approach, some cheap microcontrollers can be utilized instead of an expensive microcontroller. The original architecture of this equipment can be seen in Figure 7.9. The architecture consists of a microcontroller unit, a spectrophotometric sensor (which is composed of a infrared led, a red led, and a photo-diode), a digital/analog interface, a led driver, a converter, a pre-amplifier, a demultiplex, a demodulator, a selector signal/test, two filters, a programmable amplifier, an interface, an attenuator, and a selector control.

In order to show the practical usability of the proposed software synthesis approach for multiple processors, the pulse oximeter is being composed of two microcontrollers. One microcontroller controls the synchronization and amplitude of the led driver, which

```
#define SCHEDULE_SIZE_CTC 20

void taskS1() { ... }        void sendM13(){ ... }

void receiveM21() { ... }    void receiveS14() { ... }

void sendM15() { ... }       void taskS16() { ... }

void sendM17() { ... }       void receiveM27() { ... }

void receiveM24() { ... }    void taskS18() { ... }

void taskS7() { ... }

#define SCHEDULE_SIZE_NODE 11

struct ScheduleItemNode schedule[SCHEDULE_SIZE_NODE] =
{
  {false, 1,  (int *)taskS1},
  {false, 13, (int *)sendM13},
  {false, 21, (int *)receiveM21},
  {false, 14, (int *)taskS14},
  {false, 15, (int *)sendM15},
  {false, 16, (int *)taskS16},
  {false, 17, (int *)sendM17},
  {false, 27, (int *)receiveM27},
  {false, 24, (int *)receiveM24},
  {false, 18, (int *)taskS18},
  {false, 7,  (int *)taskS7},

};
```

Figure 7.4: Simple Control Generated Code for Node Processor 1

```
void taskS3() { ... }          void receiveM13() { ... }

void receiveS2() { ... }       void receiveS19() { ... }

void receiveS22() { ... }      void receiveS10() { ... }

void receiveS20() { ... }      void sendM21() { ... }

void taskS11() { ... }

#define SCHEDULE_SIZE_NODE 9

struct ScheduleItemNode schedule[SCHEDULE_SIZE_NODE] =
{
  {false, 3,   (int *)taskS3},
  {false, 13,  (int *)receiveM13},
  {false, 2,   (int *)taskS2},
  {false, 19,  (int *)taskS19},
  {false, 22,  (int *)taskS22},
  {false, 10,  (int *)taskS10},
  {false, 20,  (int *)taskS20},
  {false, 21,  (int *)sendM21},
  {false, 11,  (int *)taskS11}
};
```

Figure 7.5: Simple Control Generated Code for Node Processor 2

```
void taskS4{ ... }     void taskS5{ ... }
void receiveM15{ ... } void taskS4{ ... }
void taskS25{ ... }    void taskS28{ ... }
void taskS9{ ... }     void taskS26{ ... }
void sendM27{ ... }    void taskS8{ ... }
#define SCHEDULE_SIZE_NODE 9
struct ScheduleItemNode schedule[SCHEDULE_SIZE_NODE] =
{
  {false, 4,   (int *)taskS4},
  {false, 5,   (int *)taskS5},
  {false, 15,  (int *)receiveM15},
  {false, 25,  (int *)taskS25},
  {false, 28,  (int *)taskS28},
  {false, 9,   (int *)taskS9},
  {false, 26,  (int *)taskS26},
  {false, 27,  (int *)sendM27},
  {false, 8,   (int *)taskS8}

};
```

Figure 7.6: Simple Control Generated Code for Node Processor 3

```
void taskS6{ ... } void receiveM17{ ... }

void taskS23{ ... } void sendM24{ ... }

void taskS12{ ... }

#define SCHEDULE_SIZE_NODE 5

struct ScheduleItemNode schedule[SCHEDULE_SIZE_NODE] =
{
  {false, 6,   (int *)taskS6},
  {false, 17,  (int *)receiveM17},
  {false, 23,  (int *)taskS23},
  {false, 24,  (int *)sendM24},
  {false, 12,  (int *)taskS12}


};
```

Figure 7.7: Simple Control Generated Code for Node Processor 4

```
#define SCHEDULE_SIZE_CTC 20

struct ScheduleItemCTC schedule[SCHEDULE_SIZE_CTC] =
{             //Comments
  {0, 240},  //11110000 - Tasks S1,S3,S4,S6
  {3, 248},  //11111000 - Tasks S5, sendM13, recM13
  {4, 253},  //11111101 - Task S2
  {7, 253},  //11111101 - Task S19
  {12, 253}, //11111101 - Task S22
  {18, 253}, //11111101 - Task S10
  {20, 253}, //11111101 - Task S20
  {27, 252}, //11111100 - Tasks sendM21, receiveM21
  {28, 252}, //11111100 - Tasks S11, S14
  {43, 250}, //11111010 - Tasks sendM15, receiveM15
  {44, 250}, //11111010 - Tasks S25,S16
  {46, 251}, //11111011 - Task S28
  {53, 244}, //11111011 - Task S9
  {54, 246}, //11110110 - Tasks sendM17, receiveM17
  {55, 243}, //11110011 - Tasks S26, S23
  {56, 250}, //11110100 - Tasks sendM27, receiveM27
  {57, 251}, //11111011 - Task S8
  {65, 246}, //11110110 - Tasks sendM24, receiveM24
  {66, 246}, //11110110 - Tasks S12,S18
  {67, 254}, //11111110 - Task S7
};
```

Figure 7.8: Simple Control Generated Code for CTC Processor

dispatches non-simultaneous stream pulses to the infrared and red leds. Both leds generate, respectively, infrared and red radiation pulses that cross the finger of a patient. After crossing the finger, a photo-diode catches the radiations level. A sequence of operations occurs until data reaches the microcontroller. The acquired data is sent to a second microcontroller, which is responsible for performing the calculation related to oxygen saturation level based on data received, and shows the result on a display. Additionally, there is an extra microcontroller for performing the time counting.

Table 7.3 shows the pulse oximeter task specification. In this work, the oximeter specification was based on 3 task sets: excitation (TE), acquisition (TA) , and control (TC). The excitation tasks are responsible to dispatch stream pulses to the leds in order to generate radiation pulses. The acquisition tasks capture radiations crossing patient's finger. Finally, the control tasks perform the calculation of oxygen saturation level.

All tasks are non-preemptive. Additionally, there is a communication task (M1) responsible to send the acquired data to the second microcontroller. Moreover, the

Table 7.3: Task Specification for the Pulse Oximeter

TaskID	Task Name	r	c	d	p	proc/bus	from	to
TE1	SetExcitationLedRed	0	43	1000	2500	P1	-	-
TE2	ResetExcitationLedRed	371	43	1000	2500	P1	-	-
TE3	SetExcitationLedInfra	576	43	1000	2500	P1	-	-
TE4	ResetExcitationLedInfra	947	43	1000	2500	P1	-	-
TE5	ResetOutputDAConversor	0	47	2000	2500	P1	-	-
TA1	StartChannelACRed	0	43	5000	16000	P1	-	-
TA2	ReadChannelACRed	141	52	5000	16000	P1	-	-
TA3	StartChannelACInfra	191	43	5000	16000	P1	-	-
TA4	ReadChannelACInfra	323	52	5000	16000	P1	-	-
TA5	StartChannelADRed	382	43	5000	16000	P1	-	-
TA6	ReadChannelADRed	523	52	5000	16000	P1	-	-
TA7	StartChannelADInfra	573	43	5000	16000	P1	-	-
TA8	ReadChannelADInfra	714	52	5000	16000	P1	-	-
TC1	StoreDataArrays	764	62	5000	16000	P2	-	-
TC2	ScreenScanTime	0	52	10000	16000	P2	-	-
TC3	SaturAndBPMPresentation	0	52	10000	16000	P2	-	-
TC4	StorePointsDetectionBeep	764	47	10000	16000	P2	-	-
TC5	GenerateAlarmSignaling	0	92	10000	16000	P2	-	-
TC6	ProgrammingTiming	0	62	10000	16000	P2	-	-
TC7	Control	0	92	10000	160000	P2	-	-
TC8	GenerateCardioBeep	0	92	10000	80000	P2	-	-
M1	-	-	7	-	-	bus1	TA8	TC1

Intertask Relations
TE1 PRECEDES TE2, TE2 PRECEDES TE3, TE3 PRECEDES TE4,
TA1 PRECEDES TA2, TA2 PRECEDES TA3, TA4 PRECEDES TA4,
TA4 PRECEDES TA5, TA5 PRECEDES TA6, TA6 PRECEDES TA7,
TA7 PRECEDES TA8, TA8 PRECEDES TA9

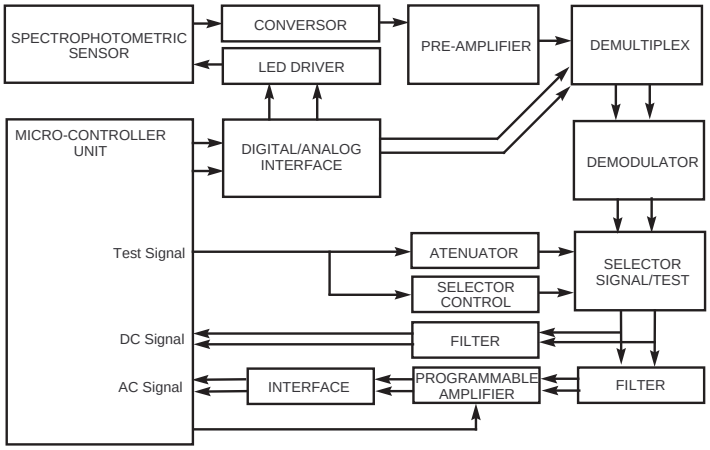

Figure 7.9: Pulse Oximeter Architecture

dispatcher overheads (CTC dispatcher and node dispatcher) are being considered in each task computation time. The overheads totalize 200 microseconds. Lastly, the task time unit (TTU) adopted in this example is 100 microseconds.

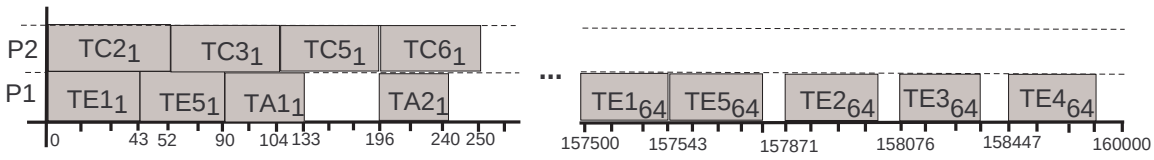

Figure 7.10: Pulse Oximeter Timing Diagram

Using the proposed approach, a feasible schedule was found in 0.272232 seconds. The amount of visited states was 2619 states. Due to the size of the feasible schedule (463 task instances), just part of the timing diagram is depicted in Figure 7.10. Also, for sake of readability, just part of the generated code is shown. Figure 7.11 depicts the code generated for the CTC processor. Figure 7.12 and Figure 7.13 show, respectively, the code for the node processor 1 and node processor 2. In addition, it is worth explaining that only the oximeter control part was adopted for testing and validating this experiment.

7.3 Vehicle Monitoring System

The vehicle monitoring system is a system composed of a set of sensors, which are used to verify whether the car components are correctly working. Whether a component fails or works erroneously, the system notifies the driver through the dashboard. The vehicle

```
#define SCHEDULE_SIZE_CTC 449

struct ScheduleItemCTC schedule[SCHEDULE_SIZE_CTC] =
{             //Comments
  {0, 252},   //11111100 - P1 and P2(Tasks TE1,TC2)
  {43, 254},  //11111110 - P1 (Task TE5)
  {52, 253},  //11111101 - P2 (Task TC3)
  {90, 254},  //11111110 - P1 (Task TA1)
  {104, 253}, //11111101 - P2 (Task TC5)
  {196, 252}, //11111100 - P1 and P2 (Task TA2, TC6)

        .
        .
        .

  {157500, 254}, //11111110 - P1 (Task TE1)
  {157541, 254}, //11111110 - P1 (Task TE5)
  {157871, 254}, //11111110 - P1 (Task TE2)
  {158076, 254}, //11111110 - P1 (Task TE3)
  {158447, 254}  //11111110 - P1 (Task TE4)

};
```

Figure 7.11: Pulse Oximeter Generated Code for CTC Processor

```
void taskTE1() { ... }       void taskTE2() { ... }
void taskTE3() { ... }       void taskTE4() { ... }
void taskTE5() { ... }       void taskTA1() { ... }
void taskTA2() { ... }       void taskTA3() { ... }
void taskTA4() { ... }       void taskTA5() { ... }
void taskTA7() { ... }       void taskTA8() { ... }
#define SCHEDULE_SIZE_NODE 400
struct ScheduleItemNode schedule[SCHEDULE_SIZE_NODE] =
{
  {false, 1,  (int *)taskTE1},
  {false, 5, (int *) taskTE5},
  {false, 6, (int *) taskTA1},
  {false, 7, (int *) taskTA2},
          .
          .
          .
  {false, 1, (int *)taskTE1},
  {false, 5, (int *)taskTE5},
  {false, 2, (int *)taskTE2},
  {false, 3, (int *)taskTE3},
  {false, 4, (int *)taskTE4}
};
```

Figure 7.12: Pulse Oximeter Generated Code for Node Processor 1

```
void taskTC1() { ... }       void taskTC2() { ... }
void taskTC3() { ... }       void taskTC4() { ... }
void taskTC5() { ... }       void taskTC6() { ... }
void taskTC7() { ... }       void taskTC8() { ... }
#define SCHEDULE_SIZE_NODE 73
struct ScheduleItemNode schedule[SCHEDULE_SIZE_NODE] =
{
  {false, 15,  (int *)taskTC2},
  {false, 16, (int *) taskTC3},
  {false, 18, (int *) taskTC5},
  {false, 19, (int *) taskTC6},
          .
          .
          .
};
```

Figure 7.13: Pulse Oximeter Generated Code for Node Processor 2

monitoring system relies on multiple processors, since several sensors are considered and the microcontroller adopted (8051) contains only four 8-Bit I/O ports. In this way, two processors are utilized for interfacing with the sensors.

A set of tasks periodically checks the status of the engine (TV and TR), the breaks (TB), the gearing (TG), the water (TW), and the temperature (TT). Next, the data are packaged and sent to an external system (TRA), which is responsible for notifying the driver. The specification model (Table 7.4) is composed of 14 tasks, including a communication task (M_1). The communication task is used to send the data acquired by the processor 2 to the processor 1, which is connected to the notifier system. Additionally, the task time unit is 1 μs and the dispatcher overhead is 200 μs, which is being considered in the computation time of each task .

Table 7.4: Task Timing Specification of Considering 2 Processors

TaskID	Task Name	Release	Comp.	Deadline	Period	Proc/Bus	From	To
TV_0	ReadVelocity	0	231	20000	120000	P1	-	-
TV_1	MapVelocity	20000	5487	40000	120000	P1	-	-
TB_0	ReadBreaks	20000	221	40000	120000	P1	-	-
TB_1	MapBreaks	40000	236	60000	120000	P1	-	-
TR_0	ReadRPM	40000	232	60000	120000	P1	-	-
TR_1	MapRPM	60000	238	80000	120000	P1	-	-
TRA	NotifierSystemTrans.	80000	2444	120000	120000	P1	-	-
TW_0	ReadWater	0	227	20000	120000	P2	-	-
TW_1	MapWater	20000	241	40000	120000	P2	-	-
TT_0	ReadTemperature	20000	259	40000	120000	P2	-	-
TT_1	MapTemperature	40000	234	60000	120000	P2	-	-
TG_0	ReadGearing	40000	224	60000	120000	P2	-	-
TG_1	MapGearing	60000	236	80000	120000	P2	-	-
M_1	-	-	1700	-	-	bus1	TG_1	TRA

Using the proposed approach, the specification model is automatically translated into a TPN model. Applying the scheduling synthesis algorithm on the TPN model, a feasible firing schedule (Figure 7.14) was found in 0.007141 seconds, analyzing 78 states.

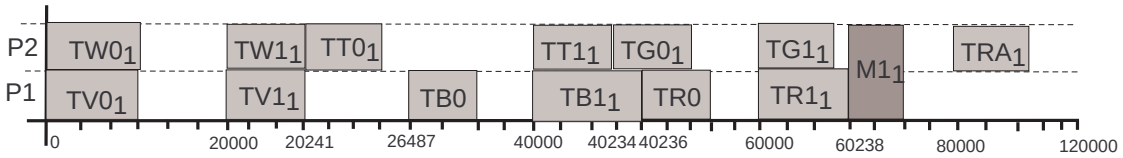

Figure 7.14: Timing Diagram for the Vehicle Monitoring System

Figure 7.15 and Figure 7.16 depict the code generated for the processor 1 and 2, respectively. Figure 7.17 shows the code for the CTC processor.

```
void taskV0() {...} void taskV1() {...}
void taskTB0() {...} void taskTB1() {...}
void taskTR0() {...} void taskTR1() {...}
void receiveM1() {...} void taskTRA() {...}

#define SCHEDULE_SIZE_NODE 8

struct ScheduleItemNode schedule[SCHEDULE_SIZE_NODE] =
{
  {false, 0, (int *)taskTV0},
  {false, 1, (int *)taskTV1},
  {false, 2, (int *)taskTB0},
  {false, 3, (int *)taskTB1},
  {false, 4, (int *)taskTR0},
  {false, 5, (int *)taskTR1},
  {false, 6, (int *)receiveM1},
  {false, 7, (int *)taskTRA},

};
```

Figure 7.15: Code for the node processor 1

```
void taskTW0() {...} void taskTW1() {...}
void taskTT0() {...} void taskTT1() {...}
void taskTG0() {...} void taskTG1() {...}
void sendM1() {...}

#define SCHEDULE_SIZE_NODE 7

struct ScheduleItemNode schedule[SCHEDULE_SIZE_NODE] =
{
  {false, 8, (int *)taskTW0},
  {false, 9, (int *)taskTW1},
  {false, 10, (int *)taskTT0},
  {false, 11, (int *)taskTT1},
  {false, 12, (int *)taskTG0},
  {false, 13, (int *)taskTG1},
  {false, 14, (int *)sendM1}

};
```

Figure 7.16: Code for the node processor 2

```
#define SCHEDULE_SIZE_CTC 10

struct ScheduleItemCTC schedule[SCHEDULE_SIZE_CTC] =
{
  {0,   252}, //11111100 - Proc 1 and 2
  {20000,  252}, //11111100 - Proc 1 and 2
  {20241,  253}, //11111101 - Proc 2
  {25487,  254}, //11111110 - Proc 1
  {40000,  252}, //11111100 - Proc 1 and 2
  {40234,  253}, //11111101 - Proc 2
  {40236,  254}, //11111110 - Proc 1
  {60000,  252}, //11111100 - Proc 1 and 2
  {60238,  252}, //11111100 - Proc 1 and 2
  {80000,  254}, //11111110 - Proc 1

};
```

Figure 7.17: Code for the CTC processor

7.4 Summary

This chapter presented in details the experiments conducted in this work. Firstly, some experiments were depicted for showing the feasibility of the compression function. After that, three software synthesis experiments were detailed. The first experiment was a simple control application, which is composed of 4 processors. The second experiment was conducted taking into account a pulse-oximeter case study, which is an electro-medical equipment responsible for measuring the oxygen saturation in the blood system using a non-invasive method. The third experiment was applied in a vehicle monitoring system, which is responsible for checking whether the car components, such as engine and breaks, are properly working.

These experiments empirically shows that the proposed code generator may be applied to several real-world case studies. In all these experiments, the performance was acceptable and results are very promising.

Chapter 8

Conclusions

Over the last years, the market has demanded, at the same time, high complexity embedded systems, and short time-to-market. Since software implementations have some advantages than hardware, due to flexibility and lower cost, nowadays, 80% of a embedded system development is related to software. Due to this fact, the design of embedded software has significantly increased its complexity. Moreover, little attention has been given to correctness and timeliness verification, which are issues that must be concerned, since several applications demand safety properties.

In light of these considerations, formal methodologies play an important role, because verification and/or analysis of qualitative as well as quantitative properties can be performed at design time, improving the degree of confidence of critical systems.

The dissertation presented a methodology for generating predictable source code in a suitable programming language for multiple processors, satisfying resource access and stringent timing constraints.

The proposed method started from a specification model, which is automatically translated into a formal model, in this case a TPN model. The model is then analyzed for finding a feasible schedule using a pre-runtime approach. Next, the code generation phase is performed.

In order to show the practical feasibility of the proposed approach, this work presents the application of the methodology into three case studies. In all experiments, the performance was acceptable and the results are very promising.

This chapter shows the main contributions proposed during the research and presented in this dissertation as well as future works.

8.1 Contributions

This work extended Barreto's approach [8] in order to consider multiple processors. Three contributions were proposed, which are summarized as follows.

Modeling

The modeling phase presented in this work adopts time Petri net formalism for modeling hard real-time embedded systems. Time Petri Net allows the modeling of several features presented in concurrent and real-time systems. Moreover, several tools adopts such formalism, in order to allow verification and/or analysis of properties.

The modeling is based on composition of building blocks. Several blocks were proposed by Barreto [8], such as (i) periodic task arrival, (ii) task structure (preemptive or non-preemptive), (iii) deadline checking, (iv) resources (e.g. processors and buses), (v) fork, and (vi) join. This dissertation presented a new building block for modeling a communication task, which represents the worst case communication time between two tasks allocated to different processors. The proposed building block was named *inter-processor communication block*. In order to provide predictable executions, it is assumed that all communications adopt the message passing paradigm and the transmission is synchronous. Furthermore, a bus is explicitly modeled and taken into account in the inter-processor communication block, since this resource may be utilized by several processors.

Scheduling

Predictability is fundamental in hard real-time embedded systems, since equipment damage or even loss of human life may occur if timing constraints are not met. In this way, scheduling plays an important role in the development of timing-critical systems. This work adopts a pre-runtime scheduling, which always finds a feasible schedule, considering if one exists, satisfying all constraints. This work adopts Barreto's scheduling algorithm, which utilizes the time Petri Model in order to generate a time labeled transition system that represents a feasible schedule.

The algorithm proposed by Barreto adopts a tagging scheme in order to avoid a state to be visited more than once. This simple technique improves the algorithm execution, but it leads to a huge memory consumption, since the visited states need to be stored in memory.

For mitigating such problem, a compressing function was implemented. The solution resulted in compression rate of 70% in average, and only increased slightly the algorithm execution time.

Code Generation

Starting from the found feasible schedule, a C-code is generated satisfying all timing constraints. This dissertation proposed a code generation method for multiple processors, in such a way that the system execution is predictable. The code generation also comprises the synthesis of schedule tables that contain all information about tasks'

execution, and the synthesis of dispatchers, which are responsible for controlling the tasks during runtime.

In order to provide the real-time clock synchronization between processors, a special architecture was presented. The architecture takes into account a central time counting processor (CTC Processor), which is responsible for performing the time counting and the propagation of real-time clock values to the node processors.

In addition, two new dispatchers were presented. One is deployed in the CTC processor, and it is responsible for performing the time counting and for notifying the node processors. The other one is deployed in the node processors, and it is responsible for controlling the tasks's execution.

In order to provide predictable behavior, each communication task is translated into two special tasks (**send** and **receive**), which are also taken into account in the schedule table.

8.2 Future Works

Software synthesis for hard real-time embedded systems remains a relatively fertile topic in systems research. Consequently, it has several opportunities for further improvement.

The adoption of multiple processors in hard real-time embedded systems emerges additional problems to be handled. Some of them were tackled in this work. However, new issues remain. For instance, many hard real-time systems have constraints on autonomy (e.g. mobile pulse oximeter). Therefore, such systems cannot exceed their respective energy constraints for executing their associated tasks in order to prolong the battery charge usage. Furthermore, processors may fail (e.g. short-circuit), hence fault-tolerance mechanisms based on multiple operational modes seem to be an interesting approach to be considered. Each operational mode is a pre-runtime schedule, which is selected at runtime by the dispatcher due to the occurrence of an external event.

Additionally, some improvements proposed in [8] were not addressed in this work. They are described as follows.

The possibility of deadlock in the precedence relation was not directly addressed. In this case, deadlock-detection may be performed by a cycle search in the graph that represents the precedence relation.

Analysis of properties in large dimension nets is not trivial. Therefore, methods that allow transforming models while preserving system properties has been largely studied. Usually, these transformations are reductions that are applied to larger models in order to obtain smaller ones while preserving properties. Reduction rules is a further work to be investigated.

Bibliography

[1] T. F. Abdelzaher and K. G. Shin. Optimal combined task and message scheduling in distributed real-time systems. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 162–171, December 1995.

[2] P. Altenbernd. Chary: the c-lab hard real-time system to support mechatronical design. *1997 Workshop on Engineering of Computer-Based Systems (ECBS '97)*, page 271, 1997.

[3] T. Amnell, E. Fersman, P. Pettersson, H. Sun, and W. Yi. Code synthesis for timed automata. *Nordic Journal of Computing*, 2003.

[4] M. Anand, J. Kim, and I. Lee. Code generation from hybrid systems models for distributed embedded systems. *Proceedings of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing 2005 (ISORC 2005)*, pages 166–173, 2005.

[5] T.P. Baker and A. Shaw. The cyclic executive model and ada. In *Proceedings of the IEEE Real-Time Systems Symposium*. December 1988.

[6] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli, E. Sentovich, and K. Suzuki. Synthesis of software programs for embedded control applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):834–849, June 1999.

[7] F. Balarin and et al. *Hardware-software Co-design of Embedded Systems: the POLIS approach*. Kluwer Academic Publishers, 1997.

[8] R. Barreto. *A Time Petri Net-Based Methodology for Embedded Hard Real-Time Systems Software Synthesis*. PhD Thesis, Centro de Informática. Universidade Federal de Pernambuco, April 2005.

[9] E. Best. Fairness and conspiracies. In *Information Processing Letter*, volume 18, pages 215–220. Elsevier, 1984.

[10] C. Boke. Software synthesis of real-time communication system code for distributed embedded applications. *In Proc. of the 6th Annual Australasian IFIP Conf. on Parallel and Real-Time Systems (PART'99)*, December 1999.

[11] L.J. Van Bokhoven, J.P.M. Voeten, and M.C.W. Geilen. Software synthesis for system level design using process execution trees. *25th Euromicro Conference (EUROMICRO '99)*, 1:1463, 1999.

[12] M. Cornero, F. Thoen, G. Goossens, and F. Curatelli. Software synthesis for real-time information processing systems. *Code Generation for Embedded Processors*, pages 260–279, 1995.

[13] J. Desel and W. Reisig. Place/transition nets. *Lectures on Petri Nets I: Basic Models, LNCS 1491*, pages 122–173, June 1998.

[14] E. W. Dijsktra. *Hierarchical ordering of sequential processes*, volume 1. Acta Informatzca, 1971.

[15] M. DiNatale and J. A. Stankovic. Dynamic end-to-end guarantees in distributed realtime systems. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 216–227, 1994.

[16] R. Ernst. Codesign of embedded systems: Status and trends. *IEEE Design and Test of Computers*, pages 45–54, April-June 1998.

[17] P. Godefroid. *Partial Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* PhD Thesis, University of Liege, Nov. 1994.

[18] R. Gupta and G. De Micheli. Hardware /software co-synthesis for digital systems. *IEEE Design and Test*, 3:26–41, September 1993.

[19] P.-A. Hsiung. Formal synthesis and code generation of embedded real-time software. *9th Int. Symp. Hw/Sw Codesign (CODES'01)*, pages 208–213, April 2001.

[20] K. Hwang. *Advanced computer architecture: paralelism, scalability, programmability.* McGraw-Hill, July 1993.

[21] Pino J, Ha S., Lee E., and Buck J. Software synthesis for dsp using ptolemy. *Journal of VLSI Signal Processing*, (9):7–21, May 1995.

[22] K. Jensen. Coloured petri nets and the invariant method. *Theoretical Computer Science*, 1:317–336, 1981.

[23] M. Nogueira Oliveira Júnior. *Desenvolvimento de Um Protótipo para a Medida Não Invasiva da Saturação Arterial de Oxigênio em Humanos - Oxímetro de Pulso (in portuguese)*. MSc Thesis, Departamento de Biofísica e Radiobiologia, Universidade Federal de Pernambuco, August 1998.

[24] D.-I. Kang, R. Gerber, L. Golubchik, J. K. Hollingsworth, and M. Saksena. A software synthesis tool for distributed embedded system design. In *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, pages 87–95. ACM Press, 1999.

[25] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[26] L. Lavagno, A. Sangiovanni-Vincentelli, and H. Hsieh. Embedded system co-design: Synthesis and verification. In G. DeMicheli and M. Sami, editors, *Hardware/Software Co-Design*, pages 213–242. Kluwer Academic Publishers, 1996.

[27] E. A. Lee. Embedded software. In M. Zelkowitz, editor, *Advances in Computers*, volume 56. 2002.

[28] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250, December 1982.

[29] B. Lin. Efficient compilation of process-based concurrent programs without run-time scheduling. *Design Automation and Test in Europe Conference (DATE'98)*, February 1998.

[30] R. J. Lipton. The reachability problem requires exponential space. Research report 62, Department of Computer Science. Yale University, January 1976.

[31] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *ACM Journal*, 20(1):46–61, January 1973.

[32] P. Maciel. *Petri Net Based Estimators for Hardware/Sofware Codesign*. PhD Thesis, Centro de Informática. Universidade Federal de Pernambuco, Dec 1999.

[33] M. Marsan, A. Bobbio, and D. Donatelli. Petri nets in performance analysis: An introduction. *LNCS: Lectures on Petri Nets I: Basic Models*, 1491:211–256, June 1998.

[34] P. Merlin and D. J. Faber. Recoverability of communication protocols: Implicatons of a theoretical study. *IEEE Transactions on Communications*, 24(9):1036–1043, Sept. 1976.

[35] A. K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment*. PhD Thesis, Dept Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1983.

[36] A. K. Mok. The design of real-time programming systems based on process models. *IEEE Real-Time Systems Symposium*, pages 5–17, 1984.

[37] T. Murata. Petri nets: Properties, analysis and applications. *Proc. IEEE*, 77(4):541–580, April 1989.

[38] C. A. Petri. *Kommunikation mit Automaten*. PhD Dissertation, Darmstad University, Germany, 1962.

[39] M. J. Pont. *Patterns for Time-Triggered Embedded Systems: Building Reliable Applications with the 8051 Family of Microcontroller*. Addison-Wesley Professional, 1 edition, July 2001.

[40] A. Sangiovanni-Vincentelli and G. Martin. Platform-based design and sofware design methodology for embedded systems. *IEEE Design and Test of Computers*, pages 23–33, November-December 2001.

[41] A. Sangiovanni-Vincentelli and G. Martin. A vision for embedded software. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'01)*, pages 1–7. Atlanta, Georgia, November 16-17 2001.

[42] M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli. Synthesis of embedded software using free-choice petri nets. *Design Automation Conference*, 1999.

[43] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchonization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[44] T. Shepard and J. A. Gagné. A pre-run-time scheduling algorithm for hard real-time systems. *IEEE Trans. Soft. Engineering*, 17(7):669–677, July 1991.

[45] M. Silva. Introducing petri nets. In F. DiCesare, G.Harhalakis, J. M. Proth, M. Silva, and F. B. Vernadat, editors, *Practice of Petri Nets in Manufacturing*, chapter 1. Chapman & Hall, 1993.

[46] A. Singhal. Real time systems: A survey. Technical report, Computer Science Department. University of Rochester, December 1996.

[47] F.-S. Su and P.-A. Hsiung. Extended quase-static scheduling for formal synthesis and code generation of embedded software. *International Symposium on Hardware/Software Codesign (CODES'02)*, May 2002.

[48] F. Thoen, J. Van Der Steen, G. de Jong, G. Goossens, and H. De Man. Multithread graph: A system model for real-time embedded software synthesis. *European Design and Test Conference (ED&TC '97)*, page 476, 1997.

[49] R. Valk. Infinite behavior and fairness. In *Lecture Notes in Computer Science*, volume 254, pages 377–396. Springer-Verlag, 1987.

[50] R. Valk. Petri nets as token objectsan introduction to elementary object nets. *LNCS 1420*, pages 1–24, 1998.

[51] A. Valmari. The state explosion problem. *LNCS: Lectures on Petri Nets I: Basic Models*, 1491:429–528, June 1998.

[52] Wolf. W. Hardware-software co-design of embedded systems. *Proceedings of the IEEE*, 82(7):967–989, July 1994.

[53] T. Wilmshurtz. *An Introduction to the Design of Small-scale Embedded Systems.* PALGRAVE, 2001.

[54] W. Wolf. Essential issues in codesign. In J. Staunstrup and W. Wolf, editors, *Hardware/Software Co-Design: Principles and Practice*, pages 1–45. Kluwer Academic Publishers, 1997.

[55] J. Xu and D. Parnas. Scheduling processes with release times, deadlines, precedence, and exclusion relations. *IEEE Trans. Soft. Engineering*, 16(3):360–369, March 1990.

[56] J. Xu and D. Parnas. On satisfying timing constraints in hard real-time systems. *IEEE Trans. Soft. Engineering*, 19(1):70–84, January 1993.

[57] J. Xu and D. Parnas. Priority scheduling versus pre-run-time scheduling. In *Real-Time Systems*, volume 18, pages 7–23. Kluwer Academic Publishers, January 2000.

[58] W. Zuberek. Timed petri nets - definitions, properties and applications. *Microelectronics and Reliability (Special Issue on Petri Nets and Related Graph Models)*, 31:627–644, 1991.

SERVIÇO PÚBLICO FEDERAL
UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Ata de Defesa de Dissertação de Mestrado do Centro de Informática da Universidade Federal de Pernambuco, 10 de março de 2006.

Ao décimo dia do mês de março do ano dois mil e seis, às quinze horas, no Centro de Informática da Universidade Federal de Pernambuco, teve início a **qüingentésima trigésima quarta** defesa de dissertação de Mestrado em Ciência da Computação intitulada **"Uma Abordagem Baseada em Redes de Petri Temporizadas para Síntese de Software em Sistemas Embarcados de Tempo Real Críticos com Múltiplos Processadores"**, do candidato **Eduardo Antonio Guimarães Tavares,** o qual já havia preenchido anteriormente as demais condições exigidas para a obtenção do grau de mestre. A Banca Examinadora, composta pelos professores Paulo Romero Martins Maciel, pertencente ao Centro de Informática desta Universidade, Tomaz de Carvalho Barros, pertencente ao Departamento de Eletrônica e Sistemas desta Universidade e Ricardo Massa Ferreira Lima pertencente à Escola Politécnica da Universidade de Pernambuco, sendo o primeiro presidente da Banca Examinadora e orientador do trabalho de dissertação, resolveu: **Aprovar por unanimidade e dar o prazo de trinta dias para a entrega da versão final do trabalho**. E para constar lavrei a presente ata que vai por mim assinada e pela Banca Examinadora. Recife, 10 de março de 2006.

Maria Lília Pinheiro de Freitas
(secretária)

Prof. Paulo Romero Martins Maciel
(primeiro examinador)

Prof. Tomaz de Carvalho Barros
(segundo examinador)

Prof. Ricardo Massa Ferreira Lima
(terceiro examinador)