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

A Time Petri Net-based Methodology for Embedded Hard Real-Time Software Synthesis

by

Raimundo da Silva Barreto

PhD Thesis

Recife
April 29^{th}, 2005

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

Raimundo da Silva Barreto

A Time Petri Net-based Methodology for Embedded Hard Real-Time Software Synthesis

A thesis submitted to the Centro de Informática of Universidade Federal de Pernambuco in partial fulfillment of requirements for the degree of Doctor of Philosofy.

Advisor: Paulo Romero Martins Maciel

Recife
April 29^{th}, 2005

Then Samuel took a stone,
and put it up between Mizpah and Jeshanah,
naming it Ebenezer, and saying,
Up to now the Lord has been our help.

I Samuel 7:12.

This thesis is dedicated to
my mother,
my wife, and
my three children.

Acknowledgments

Whatever you do, work at it with all your heart, as working for the Lord, not for men.

Colossians 3:23

This journey, which is pleased and difficult at the same time, has not been a solitary one. First of all, I would like to acknowledge that my ability and patience to complete this work comes from God. He has blessed me with the intellectual ability, the yearning for knowledge, and the environments to allow those to grow. It is for Him that I work and live.

Many thanks to my wife Lu. Thanks to her unconditional love. She has stood me and encouraged me throughout this process. Thanks also for patiently listening to me about this thesis, even when she did not understand a thing what I said. Without her love, care, and encouragement, I could not be where I am today. I am very grateful to my three children, Lucas, Elizanne and Jessica. Many thanks to all my family, in particular, my mother, father (*in memoriam*), and my sister Maria, for their part in my education. They have always done what they could to encourage and support my desire to learn more and more.

Many thanks to my advisor professor Paulo Maciel. I could not have completed this thesis without the support, encouragement, friendship, and tireless efforts of him. He believed in the project, even when did not exist any possibility of good results. Certainly, he helped me focus on the light at the end of the tunnel.

For their contributions and service as committee members, I thank Paulo Cunha, Nelson Rosa, Ricardo Massa, Siang Wun Song, and Antonio Otávio Fernandes. I also wish to thank professor Sergio Cavalcante for his support in the beginning of this research.

It has been very pleased to work with the ALUPA (informal name of our research group). Thanks to all of you. Each one was very encouraging about my work. A special thanks to Eduardo Tavares who helped me with several implementations; to Meuse who helped me with case studies; Marília for implementing the translator from specification to time Petri net model; and Adilson and Gabriel for implementing part of the tools in the EZPetri environment. I could not forget to thank Sergio Murilo, a person who, in the beginning of this research, was the first to constantly encourage me for writing papers.

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

On a more personal level, there are several people that deserve acknowledgment. Thanks to several friends (outside the University) I made during my stay in Recife, mainly the members of the Presbyterian Church in San Martin, which is a very pleasant church. Thanks to all of you. I certainly learn a lot with you. Many thanks to Rev. Gilberto Barbosa Silva and his wife Ana Lucia, by your friendship and affection. Thanks to Rev. Silvandro and his

wife Helenilda, the first Pastor we had in Recife. Many thanks to Rev. Edmilson Marinho, his wife Ilza, and their children Priscila and Gabriel, for their support and friendship.

Abstract

The problem to be addressed in this thesis is expressed in the following question: *can a specification be translated into a computer program, in such a way that it executes in a group of processors with all specified constraints satisfied?*

This work considers embedded hard real-time systems development methodologies, more specifically, the software generation phase. Regarding real-time systems, the correct behavior depends not only on the integrity of the results, but also the time in which such results are produced. In *hard* real-time systems, if timing constraints are not met, the consequences can be disastrous, including great damage of resources or even loss of human lives.

Nowadays, the human life has become more and more dependent of embedded systems. This includes not only critical systems, such as automotive, railway, aircraft, spaceships and medical devices, but also, household appliances, network printers, automatic teller machines, cellular telephones, among others. Due to this great diversity of applications, the design of embedded systems can be subject to several kinds of different constraints, including timing, size, weight, energy consumption, reliability, and cost. An alternative to treat with such problem is the adoption of formal methods. Such methods are important mechanisms for the analysis and verification of properties, as well as, facilitate system validation. However, for the effective use of formalisms, the availability of automatic tools for attending the designer is an important issue and needs to be considered.

In this thesis, the software synthesis takes a specification (composed of concurrent and communicating tasks) and automatically generates a program source code considering: (i) functionalities and constraints; and (ii) operational support for task's execution. Usually, complex systems adopt a general-purpose operating system to support the software execution. However, this solution is excessively general and may introduce delays in the execution time. Moreover, it produces a higher rate of memory usage. The software synthesis is a design alternative for these drawbacks. This method automatically generates the program source code, satisfying the desired functionality, the specified constraints, the runtime support, and the minimization of both delays and memory consumption.

The embedded software synthesis has been receiving much attention. However, few works treat with software synthesis for hard real-time systems considering arbitrary precedence and exclusion relations. Code generation for meeting all timing and resource constraints is not a trivial task. Thus, this research area has several open issues, mainly related to generation of predictable-guaranteed scheduled code.

The main aim of this work is to propose a methodology and support tools to translate a higher layer specification into a predictable program source code, such that, timing constraints, energy, and resource access constraints are satisfied. The specific objectives are: (i) to introduce a specification model that captures the information of each task in the system, as well as the relations between tasks; (ii) to model the specification using a formal model based on time Petri net; (iii) to provide a scheduling mechanism for guaranteeing that timing and energy constraints, and precedence and exclusion relations are satisfied; and (iv) to develop a method for code generation that tackles specified constraints and maintain certain properties of interest, such as, mutual exclusive access to resources, and deadlock and starvation-freedom.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Day after day, our lives become more dependent on embedded systems. This includes not only safety-critical systems such as automotive, railways, aircraft, spaceships, and medical devices, but also home automation, video game consoles, household appliances, disk drives, network printers, automatic teller machines, cellular telephones, 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, the most part of embedded systems were hardware-based, using for instance ASIC's (*Application Specific Integrated Circuit*). The processors computational power increasing and the size and cost reduction have allowed moving more functionalities to software. Nowadays, the software is responsible for more than 80% of functionalities considering modern embedded systems [81]. In spite of the execution time increment (due to the moving of functionalities from hardware to software), this new method brought some advantages such as flexibility, lower cost and accessibility. Indeed, the current microcontroller and microprocessor technology has allowed the development of extreme fast processors making software-oriented design possible for most embedded real-time systems.

Due to the increasing complexity and diversity of requirements, embedded software has become much harder to design. For instance, since several applications demand safety properties, the correctness and timeliness verification are an issue to be concerned. Moreover, the complexity of software is greatly increased, since consumers have demanded more functionalities. Another commonly required feature is reconfigurability. Thus, systems can download new modules in order to adapt themselves in a mutable environment. Service demands, computing resources, and sensors may appear and disappear. In the same way, demands for quality of service may change as well

as conditions change. The system is therefore continuously being redesigned while it operates, and all the time it must not fail [51]. These are problems that designers of embedded software have to deal with.

Several embedded systems are reactive, i.e., they react to stimulus produced by the environment. If such reaction is time-sensitive, such systems are called embedded real-time systems. In real-time systems, the correct behavior depends not only on the integrity of the results, but also on the time the results are produced. Real-time systems can be divided in two categories: hard and soft real-time systems. In hard real-time systems, consequences can be disastrous if timing constraints are not met, including resource damages or risks for human life. On the other hand, such constraints may occasionally not be reached in soft real-time systems. In this case, the system may just degradate its behavior, which may be tolerated in some cases.

Currently, automatic synthesis of embedded software is an important research field. However, not many works deal with time-critical embedded software synthesis. Generating code that guarantees meeting all timing and resource constraints is not a simple task. This research area has several open issues, mainly related to the generation of predictable-guaranteed scheduled code. Formal development methodologies play an important role to cope with those stringent requirements.

This chapter shows the context of this thesis, describes the problem to be solved, presents the motivation and objectives, explains how the problem is planned to be solved, summarizes the contributions, and finally, shows the outline of this thesis.

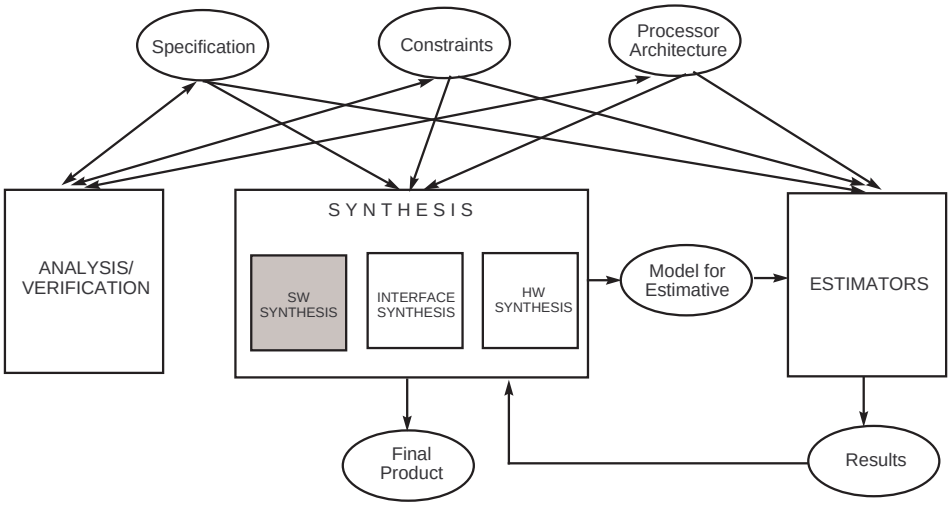

Figure 1.1: Project Overview

1.1 Context

The context of this work is on embedded systems design methodologies. This thesis is specifically interested in the software part of such methodologies. Usually, most embedded systems have to consider timing aspects. The scope of this thesis is restricted to hard real-time embedded software.

Figure 1.1 shows an overview of the project that our research group is investigating, and the context of this work in this global project. The core of this project is the synthesis. The synthesis is divided into three parts, that is, hardware synthesis, software synthesis, and interface between hardware and software synthesis. The input of the proposed methodology is composed of three parts, namely, specification, constraints, and processor architecture. Processor architecture specifies the name of processors, topology (interconnections, and distributed or shared memories) and instruction set (size, execution time, and energy consumption of each instruction). The estimators are used for estimating several metrics of interest, such as energy consumption, size, area, execution time, and communication. The result of the estimation process is used for feedbacking the synthesis phase. The estimator's input is a model for this specific purpose. Our research group is particularly interested in development of power-aware embedded software synthesis, where estimators are responsible for evaluating power consumption due to software. The analysis/verification of properties is performed and the results may be used to adapt the specification, constraints, or processors.

1.2 Problem Description

The aim of any synthesis method is to implement the specification with the minimum cost. In accordance with Gajski *et.al.* [34], software synthesis is the task of converting a complex executable specification into a conventional software program in such a way that this software program can be compiled by conventional compilers. Balarin *et.al.* [11] defines software synthesis as an optimized translation process from a high-level specification into C or assembly code, where the specification describes the function that must be performed, rather than the way it must be implemented. Both definitions are, in some sense, incomplete. In this thesis, the definition of software synthesis provided by Cornero *et.al* [23] is adopted: "software synthesis starts from a specification (typically composed of concurrent communicating processes) and automatically generates source code 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 programing language with, additionally, all the operational support code required for its execution".

Software synthesis is important since it automatically generates code satisfying desired properties, and it may also reduce the time-to-market. Software synthesis becomes necessary since specifications have special characteristics which are not found in traditional programming languages. For instance, specifications are generally composed by several concurrent tasks, so, scheduling and synchronization of multiple tasks are important issues. 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 [23]: (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.

Complex embedded systems adopt a conventional operating system to support the software execution. However, this solution is typically very general and introduces overheads, in execution time and memory requirements. On the other hand, the software synthesis method represents an alternative to such operating systems usage. This method automatically generates customized codes with the following benefits: (i) functionality is attended; (ii) all constraints are met; (iii) typical runtime support is provided; and (iv) overheads are minimized.

The problem considered in this thesis is expressed in the following question: *can a specification be translated into a program, in such a way that it can be directly executed in a specific set of processors with all specified constraints satisfied?*

1.3 Motivation

The principal role of embedded software is the interaction with the physical world, rather than the transformation of data. In this case, embedded software must acquire some properties of the physical world, for instance, it takes time, it consumes power, it access shared resources, and it does not terminate (unless it fails) [51].

One of the motivations of this thesis is related to the complexity of applications, and consequently the size of programs, which is growing rapidly. Most of current devices are networked. Even some programmable DSPs now run a TCP/IP protocol stack. Besides, the applications are getting much more dynamic, with downloadable and migrating code. In the mean time, reliability for embedded software remains very high, unlike general-purpose software. In addition, the market pressure has demanded,

at the same time, high complexity, and short time-to-market. Two conflicting criteria, someway. Thus, in this context, writing assembly code by hand may not be sufficient. Tools are necessary to assist the designer, starting from a high-level specification.

In accordance with the high complexity of embedded systems, another motivation of this thesis is to deal with the increasing difficulty in verifying embedded systems design correctness. This verification is critical due to safety considerations in several application domains. Hence, code generators are necessary to guarantee correct designs and, at the same time, to increase software quality and productivity.

1.4 Objectives

Considering the problem stated in the previous section, the main objective of this thesis is to propose a methodology *starting from a high-level specification, translating such specification into a predictable source code, that is, a code where timing, energy and resource constraints are satisfied.*

The specific objectives are:

1. to propose a specification model that captures code, timing and energy information of tasks, and inter-task relations such as precedence and exclusion relations;

2. to model the specification using a formal method;

3. to provide a scheduling synthesis framework that produces schedules guaranteeing that timing, energy, precedence and exclusion constraints are satisfied;

4. to develop a code generator that generates scheduled code with guarantees that the code maintain specified properties, such as mutual-exclusive resource access, and deadlock and starvation-freedom.

Tasks of embedded real-time systems have timing characteristics that must be guaranteed. Usually, these tasks have also relations between them, such as precedence and exclusion relations. One of the aims of this theses is to propose a specification model that also captures such inter-task relations.

In order to improve the degree of confidence of time-critical systems, formal methods are important mechanisms that allow analysis and verification of properties as well as facilitate system validation. For this reason, this thesis aims to apply a formal method in the system modeling phase. This model is intended to be used in the subsequent phases of the proposed software synthesis method.

Meeting timing constraints is fundamental in embedded hard real-time systems. Therefore, scheduling plays an important role. However, not all scheduling policies are able to find a feasible schedule, even if such schedule exists. This situation is hardened when considering arbitrary precedence and exclusion relations. The scheduling strategy has to be carefully chosen.

In general, embedded software programs runs forever. Therefore, liveness is crucial in such systems. In this case, in order to avoid runtime overheads, another aim of this thesis is to generate already scheduled code.

Some properties are inherent to specifications. For instance, it is not interesting to generate code that do not take into account mutual exclusive access to shared resources. Another undesirable characteristic is deadlock and starvation. Thus, this thesis is of interest in generating code that satisfies such properties.

1.5 Proposed Method

The specification considered in this thesis is composed by a set of tasks. These tasks are 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.

This thesis proposes modeling tasks of a system using an *transition-annotated* time Petri nets (TPN), that is, a TPN with code associated with transitions. Starting from the resulting model, the proposal is to synthesize a feasible schedule (one that satisfies all constraints), and generate a scheduled code in accordance with the found schedule.

In general, scheduling policies are classified as: (i) Pre-runtime scheduling; or (ii) Runtime scheduling. However, considering time-critical systems, predictability is an important concern. In order to guarantee that all critical tasks meet their deadlines, pre-runtime scheduling is used, since runtime methods may constrain the possibility of finding feasible schedules, even if such a schedule exists, especially when considering arbitrary precedence and exclusion relations.

The generated code can be seen as a *cyclic executive*[10], 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 a single CPU. The interleaving is done in a deterministic fashion so that execution time is predictable. In this case, when the periods are different, the period of the cyclic executive is equal to the least common multiple between all periods of processes.

As presented before, embedded systems are becoming highly complex and hard

to verify design correctness. An alternative to tackle this problem is the adoption of formal methods. In order to improve the degree of confidence of critical system designs, formal methods are important mechanisms that allow precise specification, verification and/or analysis of qualitative as well as quantitative properties. However, for effective use of formalisms, the availability of automated tools to assist the embedded software design is an important issue. This work proposes the use of a Petri net-based formal method in order to provide tools for software synthesis of embedded hard real-time systems.

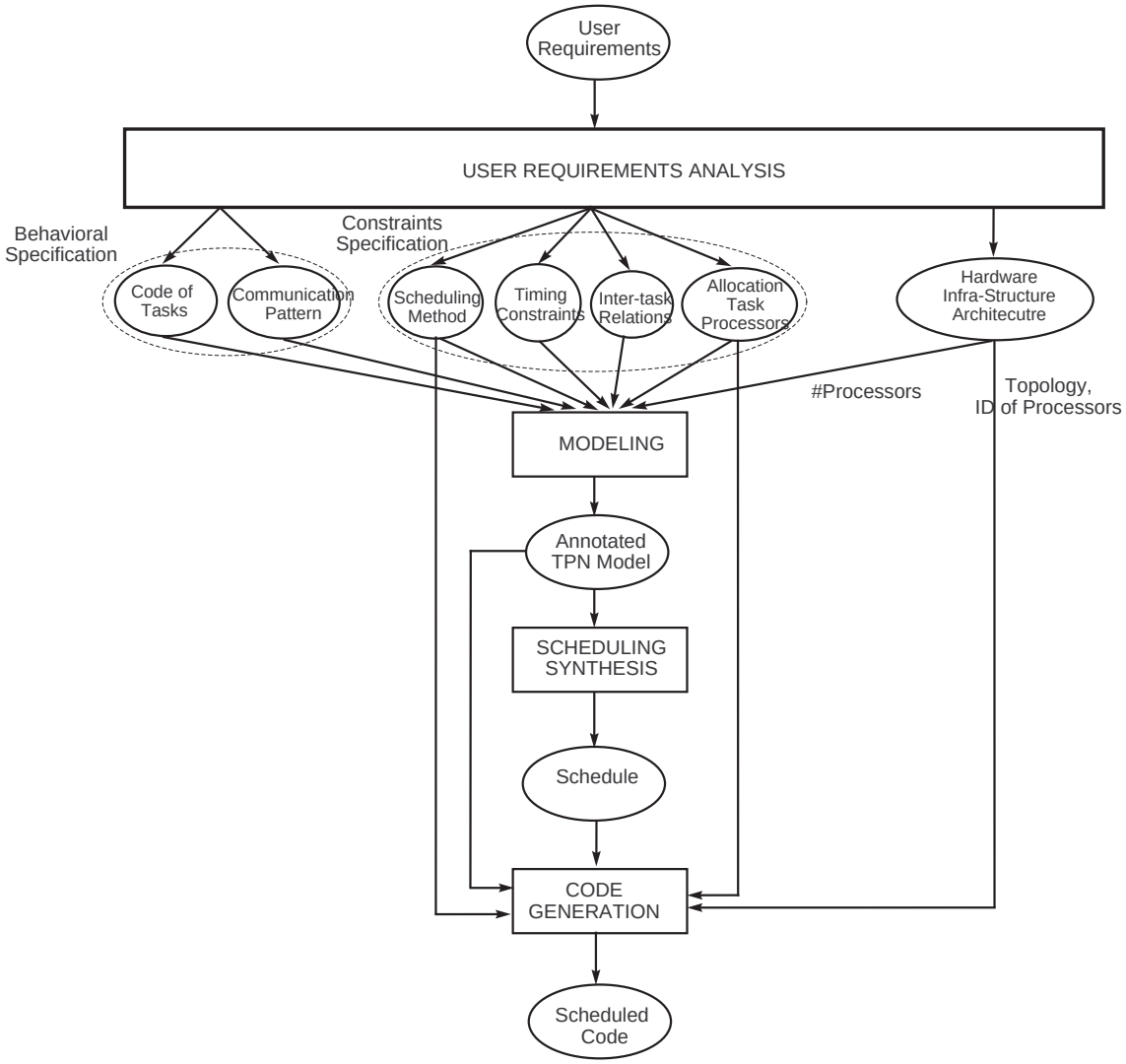

Figure 1.2: Proposed Software Synthesis Methodology Phases

Figure 1.2 presents a diagram of the phases composing the proposed methodology. This section presents only a summary of each phase. More details is presented in the next chapters. The phases are:

- **Specification**. In this phase, the result of the user requirement analysis is composed of three parts:

    1. *Behavioral Specification.* The behavioral specification is responsible for specifying both the code of each task, and the possible communication pattern (if adopted a multi-processor architecture). The code of each task is specified by using a C programming language augmented with communication constructs. Except from these constructs, the C-code has to be compliant with the respective compiler. For instance, there are subtle differences between Keil C51 and GNU CC compilers. If the architecture chosen is a multi-processor, then the communication pattern has to be specified by a *communication graph*.

        The code of each task is used in both modeling and code generator phases. The communication graph is used in the modeling phase.

    2. *Constraints Specification.* The constraints to be captured for each task are: (i) timing constraints (phase, release time, worst-case execution time, energy consumption, deadline, and period); (ii) scheduling method (preemptive or non-preemptive); and (iii) inter-tasks relations, such as precedence and exclusion relations. Additionally, when adopting a multi-processor architecture, the allocation of tasks to processors has to be included. The proposed methodology considers that this allocation is performed in advance by the designer, and such allocation is beyond the scope of this thesis. It is worth noting that the constraints specification already provides the worst-case execution time for each task. The worst-case execution time is calculated in the estimation phase (Figure 1.1).

        All timing constraints and inter-tasks relations are considered in the modeling phase. Scheduling method and the (possible) allocation of tasks to processors are used in two phases, modeling and code generation.

    3. *Hardware Infra-Structure Architecture.* This phase defines the architecture of the hardware infra-structure, specifying the amount of processors, the instruction set description (processor ID, instruction set, instruction execution time, instruction energy consumption, etc) for each processor, and topology of interconnection between processors. The instruction set description may be used, for instance, for calculating the worst-case execution time of tasks. The ID of each processor and topology are required in the code generation phase. Each processor has intrinsic characteristics that have to

be considered when generating code. As an example, consider the timer programming, which may be completely different when adopting different processors.

In the code generation phase, topology is essential for the communication synthesis, and the amount of processors is used in the modeling phase.

- **Modeling**. This phase deals with the translation from specification into the respective time Petri net (TPN) model. In order to allow portability, the time Petri net model is expressed in PNML (Petri Net Markup Language) [100] format. 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 inter-processor communication between tasks. The resulting model is used 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. The algorithm is based on depth-first search method.

- **Code Generation**. This phase aims to generate the respective scheduled code, considering the previously computed schedule, constraints and processor architecture. If a mono-processor architecture is adopted, the code generation requires the schedule found, code of each task (from the model), scheduling method (from the specification of constraints), and processor name (from the hardware infra-structure architecture). Additionally, if the hardware infra-structure has more than one processor, the code generation phase requires the task to processors allocation (from the constraints specification) and topology (from the hardware infra-structure architecture). In order to automate the code generation phase, a code generator engine based on templates was developed. Several templates are provided, where dispatcher, interrupt handler, code-of-tasks, types, constants, and schedule are examples of templates.

1.6 Contributions

This thesis provides a methodology for development of predictable scheduled code for embedded hard real-time systems. The methodology *per si* is a contribution. The more

specific contributions are depicted as follows:

1. **Specification**. The user has requirements that have to be captured by the designer. Generally, the specification is composed of a set of concurrent tasks, where each task has its own attributes, e.g. task behavior, timing constraints, and tasks communication pattern. Although the proposed specification model is not new, this work joins different requirements into a complete and coherent model. Additionally, a tool for inputing the specification is another contribution.

2. **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. These building blocks are a contribution. The automatic translation from the specification to the respective time Petri net model is another contribution. Usually, the time and energy consumed by the dispatcher and interrupt handler are often neglected. This thesis considers such overheads at the modeling phase, which leads to a more precise behavior of the system. This is a very interesting contribution.

3. **Schedule Synthesis**. A pre-runtime scheduling that considers timing constraints (such as phase, release time, deadline, and period) and arbitrary intertask relations is not new. However, there is no similar work that finds a feasible pre-runtime scheduling based on Petri net formalism considering timing and energy constraints. This is an important contribution, since it is the base for any software synthesis method.

4. **Code Generation**. At the best of the present knowledge, there is no work that generates scheduled code for embedded hard real-time systems, that considers release time, deadline, periods, energy consumption, dispatcher overheads, and arbitrary precedence and exclusion relations. This is the main contribution of this thesis.

1.7 Outline

Following this introduction, Chapter 2 overviews the main concepts needed to understand this thesis, such as the main models of computation, embedded systems, real-time systems, and scheduling. Chapter 3 reviews the related works with the real-time scheduling and code generation areas. Chapter 4 introduces Petri nets, presenting how to model several situations present in most systems, the main timing extensions,

methods for properties analysis, and Petri net synthesis (bottom-up, top-down, and hybrid). Chapter 5 describes the method for modeling embedded hard real-time systems. This chapter is composed by three main sections, namely, formal model, specification model, and how to model the specification by the formal model. The formal model syntax is given by an annotated time Petri net and the semantics by a timed labeled transition system. The specification model consists in a set of tasks and their interrelations, where such tasks are executed in one or more processors. The modeling of specification explains the method adopted for translating from the specification model to a time Petri net model. Chapter 6 explains the method for synthesize the software. Firstly, it presents a novel method to compute pre-runtime schedules based on the time Petri net model. Later, this chapter describes how the code is generated starting from a feasible found schedule. Chapter 7 shows several tools provided by this thesis. In particular, this chapter details the integration between the software synthesis method with EZPetri, which is a tool suite based on PNML and plug-in technology that permits editing Petri nets, integration of Petri net tools, as well as importing/exporting Petri nets from/to different Petri net tools. Chapter 8 shows experiments conducted using the proposed methodology. Finally, Chapter 9 concludes this thesis and presents future works.

Chapter 2

Background

This chapter introduces the main concepts needed to the understanding of this thesis. It is divided into four sections: formal models, embedded systems, real-time systems, and scheduling. The first section shows the main models used for specifying embedded real-time systems. The next section deals with embedded systems and development methodologies. After that, real-time systems is presented. Finally, scheduling considering timing constraints is surveyed.

2.1 Formal Models

Most often the set of activities on embedded system design are not specified in a rigorous and unambiguous fashion, so the design process requires several iterations to obtain the final result. Thus, one or more formal methods is highly recommended for designing embedded real-time systems. These methods are used for describing the behavior of the system at a high level of abstraction, and before a decision on its decomposition into hardware and software is taken. Managing the design complexity and heterogeneity are also key problems. Therefore, the use of formal methods and high level synthesis improve design process reliability and productivity.

This section aims to present a model taxonomy and some representative models of each class of such taxonomy, such as automata and extensions, Petri nets, program-state machine, and process algebras.

2.1.1 Model Taxonomy

System designers may use many different models in hardware or software design methodologies. In general, these models fall into five distinct categories [35].:

1. state-oriented;

2. activity-oriented;

3. structure-oriented;

4. data-oriented; and

5. heterogeneous.

State-oriented models, such as an automata, is one that represents the system as a set of states and a set of transitions between them, which usually are triggered by external events. Such model is most suitable for control systems, such as reactive systems, where the response to events is the most important aspect of the design.

Activity-oriented models, such as a dataflow graph, is one that describes a system as a set of activities related by data or execution dependencies. This model is most applicable to transformational systems, such as digital signal processing systems, where data passes through a set of transformations at a fixed rate.

Structure-oriented models, such as a block diagram, describes a system as physical modules and interconnections between them. Unlike state-oriented and activity-oriented models, which primarily reflect functionalities of a system, the structure-oriented models focus mainly on the physical composition of a system.

Data-oriented model, such as an entity-relationship diagram, is another class of models for representing the system as a collection of data related by their attributes, class membership and interactions. This model is most suitable for information systems, such as databases, where the function of the system is less important than the data organization of the system.

Finally, a designer could use a *heterogeneous model*, that is, a model that integrates many characteristics of the previous models, whenever it needs to represent a variety of different point of views of a complex system.

In the rest of this section, some common models adopted for representing the behavior of embedded systems are briefly discussed.

2.1.2 Representative Models

Many different models have been considered in the system design. This section presents some representative models for capturing behavior of systems.

Automata

Automata is more suitable for modeling discrete event systems, that is, systems where the state space is naturally described by a discrete set, and state transitions are only observed at discrete points in time [21]. Considering the state evolution of a system, usually the first concern is with the sequence of visited states and the associated events causing these state transitions. In general, the system behavior is described in terms of event sequences $e_1 e_2 \dots e_n$, where such behavior is modeled by a *language*. Automata is one of the formal models widespread used for modeling languages, which is appropriate for performing analysis and control of (discrete event) systems.

Usually, systems have an underlying finite event set E, which is the "alphabet" of a language, and event sequences are "words" or "strings" in that language. So, a language (defined over an event set E) is a set of finite-length strings formed from events in E. A language specifies all admissible sequence of events that the system is capable of generating.

Automaton is a device that is able to represent a language in accordance with well-defined rules. The automaton may be represented as a directed graph (or state transition diagram), where the states are associated with nodes and the labeled arcs are associated with transition between states. Operations usually applied to a language are those applied to a set, that is: union, intersection, difference and complement with respect to E^*, where E^* is the set of all strings of elements of E, including the empty string ε.

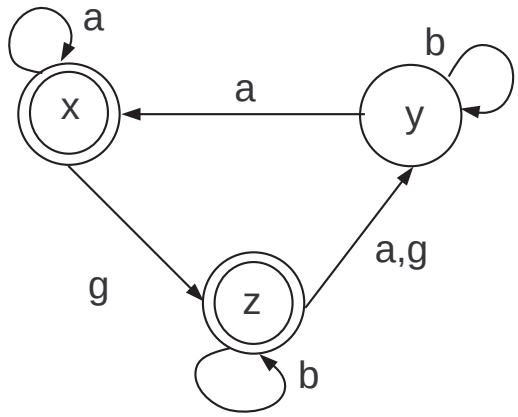

Figure 2.1: An Example of State Transition Diagram

For instance, supposing that the state set is $X = \{x, y, z\}$ and the event set is $E = \{a, b, g\}$. According with Figure 2.1, the transition functions are represented by $f(x, a) = x$, $f(x, g) = z$, $f(y, a) = x$, $f(y, b) = y$, $f(z, b) = z$ and $f(z, a) = f(z, g) = y$.

Three observations are worth making about such example. First, an event may occur without changing the state, as in $f(x, a) = x$. Second, two distinct events may occur at a given state resulting the exact same transition, as in $f(z, a) = f(z, g) = y$. Third, the function f is a partial function on its domain $X \times E$, that is, there is no need of a transition to be defined for each event in E at each state of X. For instance, $f(x, b)$ and $f(y, g)$ are not defined.

In order to completely define an automaton, two more ingredients are necessary: an initial state, denoted by x_0, and a subset X_m of X that represents the states of X that are marked. Marked states are also referred as "accepting" states or "final" states. The initial state x is represented by an arrow and the marked states, x and z, are represented by double circles.

The formal definition of a deterministic automaton is as follows [21].

Definition 2.1 (Deterministic Automaton) *A deterministic automaton, denoted by G, is a six-tuple $G = (X, E, f, \Gamma, x_0, X_m)$, where:*

X is the set of states

E is the finite set of events associated with transitions in G

$f : X \times E \to X$ is the transition function: $f(x, e) = y$ means that there is a transition labeled by event e from state x to state y; in general, f is a partial function on its domain

$\Gamma : X \to 2^E$ is the active event function (or feasible event function), where 2^E is the power set of the set E: $\Gamma(x)$ is the set of all events e for which $f(x, e)$ is defined and it is called the active event set (or feasible event set) of G at x.

$x_0 \in X$ is the initial state

X_m is the set of marked states.

Some remarks about this definition are as follows:

- the words *state machine* and *generator* are also often used to describe an automaton;
- if X is a finite set, G is called *deterministic finite-state automaton* (DFA).
- the automaton is said to be deterministic, since f is a function over $X \times E$. In contrast, the transition structure of a non-deterministic automaton is defined by means of a relation over $X \times E \times X$ or, equivalently, a function from $X \times E$ to 2^X

- the inclusion of Γ in the definition of G is superfluous in the sense that Γ is defined from f. The contents of $\Gamma(x)$ for state x helps in distinguishing between events e that are feasible at x but cause no state transition, that is, $f(x, e) = x$, and events e' that are not feasible at x, that is, $f(x, e')$ is undefined

- f may be extended from domain $X \times E$ to $X \times E^*$, in the following recursive manner:

$$f(x, \varepsilon) := x$$

$$f(x, se) := f(f(x, s), e) \text{ for } s \in E^* \text{ and } e \in E.$$

The connection between languages and automata is easily made by inspecting the state transition diagram of an automaton. This observation leads to the notion of languages generated and marked by an automaton.

Definition 2.2 (Language Generated) *The language generated by* $G = (X, E, f, \Gamma, x_0, X_m)$ *is*

$$\mathcal{L}(G) := \{s \in E^* : f(x_0, s) \text{ is defined } \}$$

The language $\mathcal{L}(G)$ represents all directed paths that can be followed along the state transition diagram, starting at the initial state. The string s, corresponding to a path, is the concatenation of the event labels of the transitions composing the path. Therefore, s is in $\mathcal{L}(G)$ if and only if it corresponds to an admissible path in the state transition diagram, or equivalently, if and only if f is defined at (x_0, s)

Definition 2.3 (Language Marked) *The language marked by* $G = (X, E, f, \Gamma, x_0, X_m)$ *is*

$$\mathcal{L}_m(G) := \{s \in \mathcal{L}(G) : f(x_0, s) \in X_m\}.$$

The second language represented by G, $\mathcal{L}_m(G)$, is the subset of $\mathcal{L}(G)$ consisting only of the strings s for which $f(x_0, s) \in X_m$, that is, these strings corresponds to paths that end at a marked state in the state transition diagram. The language marked is also called the language *recognized* by the automaton, and the given automaton is a *recognizer* of the given language.

One limitation of the finite automaton defined so far is that its output is limited to a binary signal "accept" (1) or "do not accept" (0). However, models in which the output is chosen from some other alphabet have been considered. In this case, there

are two distinct approaches; the output may be associated with the state (called Moore automata) or with the transition (called Mearly automata).

In Moore automata there is an output function ($h : X \rightarrow O$) that assigns an output (from the output alphabet O) to each state. In other words, an output symbol is assigned to each state, and outputted when it enters in the respective state. The DFA may be viewed as a special case of a Moore automata where the output alphabet is $\{0, 1\}$ and state x is accepting if and only if $h(x) = 1$.

Mealy automata, on the other hand, are input/output automata. Thus, transitions are labeled by "events" of the form *input event/output event*. In this case, h maps $X \times E$ to O. Therefore, the interpretation of a transition e_i/e_o from state x to state y is as follows: when the system is in state x, if the automaton receives input event e_i, it will make a transition to state y and in that process will emit the output event e_o.

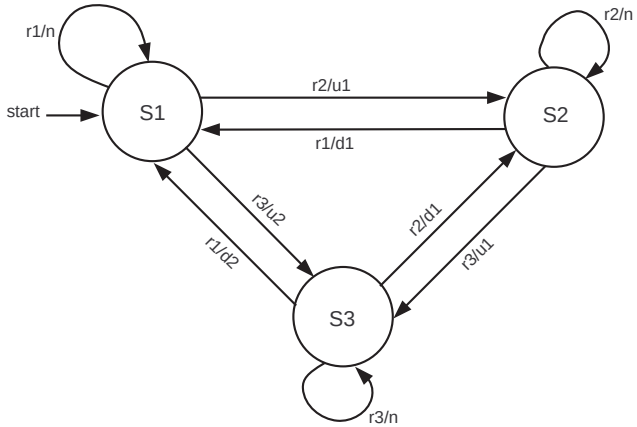

Figure 2.2: Mearly Automata for an Elevator Controller

Figure 2.2 shows a Mearly automaton that models the elevator controller in a building with three floors. In this model, the set of events $E = \{r1, r2, r3\}$ represents the requested floor. For example, $r2$ means that floor 2 is requested. The set of outputs $O = \{d2, d1, n, u1, u2\}$ represents the direction and number of floors the elevator should go. For example, $d2$ means that the elevator should go down 2 floors, $u2$ means that the elevator should go up 2 floors, and n means that the elevator should stay idle. The set of states represents the floors. In Figure 2.2, it can be seen that if the current floor is 2 (i.e., the current state is S_2), and floor 1 is requested, then the output will be $d1$.

Figure 2.3 shows the Moore automata for the same elevator controller, in which the value of the output is indicated in each state. Each state has been split into three states representing each of the output signals that the automaton in Figure 2.2 will output when entering that particular state.

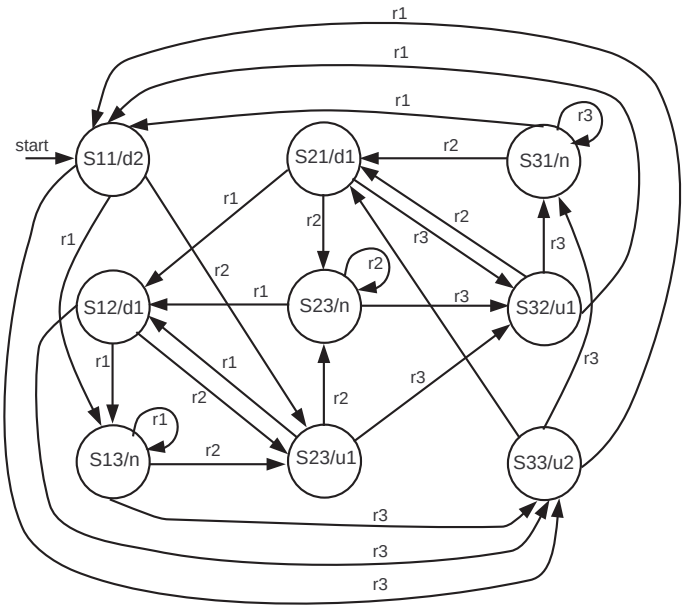

Figure 2.3: Moore Automata for an Elevator Controller

In practical terms, the primary difference between these two models is that the Moore automaton certainly will require more states than the Mearly automaton.

Finite State Machine (or Automaton) with Datapath

As presented before, another name used for representing automata is state machine. In this, and following sections, automaton and state machine (or FSM that stands for finite state machine) is used interchangeably.

If an automaton has to represent integer (or floating-point) numbers, a state-explosion problem may occur, since if each possible value for a number requires its own state, then the automaton could require an enormous number of states. For example, a 16-bit integer can represent 2^{16} or 65536 different states. One way to solve this problem is extending an automaton with integer and floating-point variables.

This kind of finite state machine with datapath (FSMD) can model the elevator controller example in Figure 2.2 with only one state, as shown in Figure 2.4. This reduction in the number of states is possible because a variable $cfloor$ is designated to store the state value of the FSM in Figure 2.2 and $rfloor$ to store the values of $r1$, $r2$ and $r3$.

In general, the FSM is suitable for modeling control-dominated systems, while the FSMD can be suitable for both control and computation-dominated systems. However, it is worth noting that neither the FSM nor the FSMD model is suitable for complex

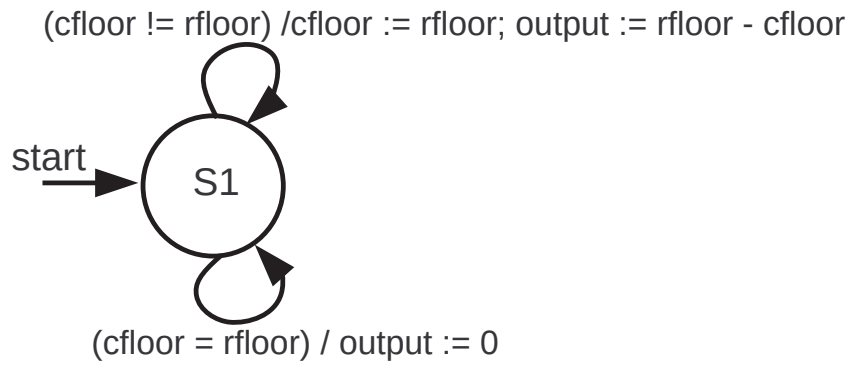

Figure 2.4: FSMD Model for an Elevator Controller

systems, since neither one explicitly supports concurrency and hierarchy. Without explicit support for concurrency, a complex system will certainly have an state-explosion. Consider, for example, a system consisting of two concurrent subsystems, each with 100 possible states. If trying to represent this system as a single FSM or FSMD, it must represent all possible states of the system, in which there are $100 \times 100 = 10,000$. At the same time, the lack of hierarchy would cause an increase in the number of arcs. For example, if there are 100 states, each requiring its own arc to transition to a specific state for a particular input value, it would need 100 arcs, as opposed to the single arc required by a model that can hierarchically group those 100 states into one state. The problem with such models, of course, is that once they reach several hundred states or arcs, they become incomprehensible to humans.

Hierarchical Concurrent Finite State Machine

The hierarchical concurrent finite-state machine (HCFSM) is essentially an extension of the FSM model, which adds support for hierarchy and concurrency, thus eliminating the potential for state and arc explosion that occurred when describing hierarchical and concurrent systems with FSM models.

Like the FSM, the HCFSM model consists of a set of states and a set of transitions. Unlike the FSM, however, in the HCFSM each state can be further decomposed into a set of sub-states, thus modeling hierarchy. Furthermore, each state can also be decomposed into concurrent sub-states, which execute in parallel and communicate through global variables.

One language that is particularly well-adapted to the HCFSM model is Statecharts [39], since it can easily support the notions of hierarchy, concurrency and communication between concurrent states. Statecharts uses unstructured transitions and a

broadcast communication mechanism, in which events emitted by any given state can be detected by all other states. Figure 2.5 shows an example of a system represented by means of Statecharts. In this figure, it can be seen that state Y is decomposed into two concurrent states, A and D; the former consisting of two further substates, B and C, while the latter comprises substates E, F, and G. The bold dots in the figure indicate the starting points of states. According to the Statecharts language, when event b occurs while in state C, A will transfer to state B. If, on the other hand, event a occurs while in state B, A will transfer to state C, but only if condition P holds at the instant of occurrence. During the transfer from B to C, the action c associated with the transition will be performed.

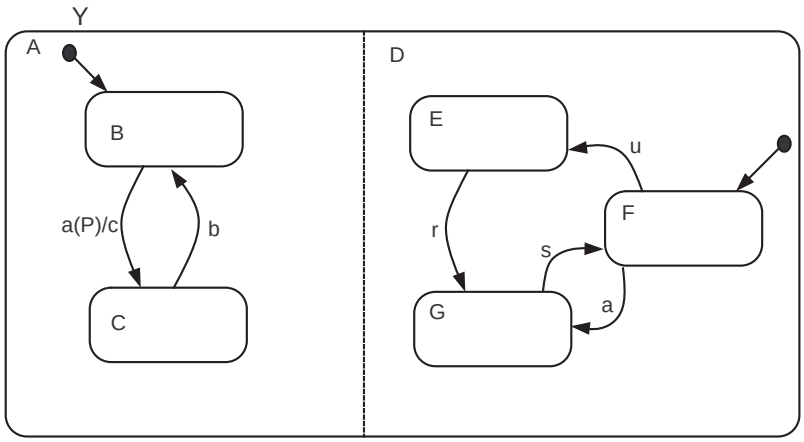

Figure 2.5: Statecharts: Hierarchical Concurrent States

Because of its hierarchy and concurrency constructs, the HCFSM model is well-suited to representing complex control systems. The problem with this model, however, is that, like any other state-oriented model, it concentrates exclusively on modeling control, which means that it can only associate very simple actions, such as assignments, with its transitions or states. As a result, the HCFSM is not suitable for modeling certain characteristics of complex systems, which may require complex data structures or may perform in each state an arbitrarily complex activity. For such systems, this model alone would probably not be sufficient.

Petri Nets

This section aims to present just an introduction about Petri nets. This subject will be returned in next chapter (Chapter 4).

Petri net model [70, 75, 79] is an example of heterogeneous model, specifically defined to model systems that comprise interacting concurrent tasks. A Petri net

model consists of a set of places, a set of transitions, and a set of tokens. Tokens reside in places, and circulate through the Petri net by being consumed and produced whenever a transition fires. More formally, a Petri net is a quintuple

$$\langle P, T, F, W, m \rangle$$

where $P = \{p_1, p_2, \ldots, p_m\}$ is a set of places, $T = \{t_1, t_2, \ldots, t_n\}$ is a set of transitions, and P and T are disjoint. Further, the relation function F, $F \subseteq (P \times T) \cup (T \times P)$, defines arcs between places to transitions and between transitions to places. $W : F \rightarrow \mathbb{N}$ represents the weight of the flow relation (F). Finally, the marking function $m : P \rightarrow \mathbb{N}$ defines the number of tokens in each place, where $\mathbb{N}$ is the set of nonnegative integers.

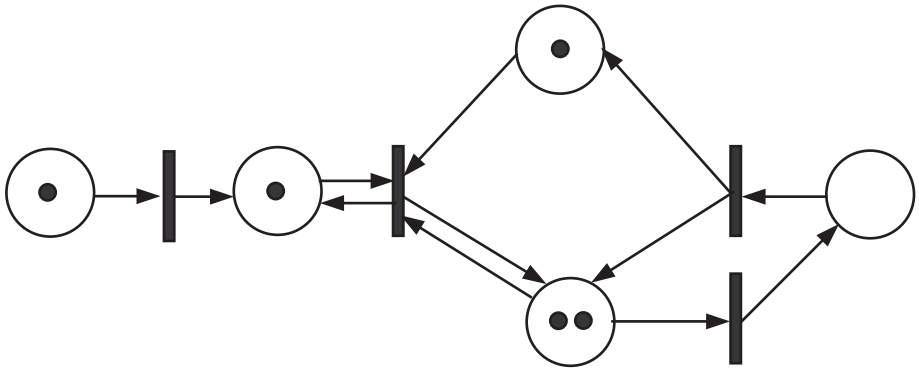

Figure 2.6: Petri Net Example

Figure 2.6 presents a graphic representation of a Petri net. Note that there are five places (graphically represented as circles) and four transitions (graphically represented as solid bars) in this Petri net. In this instance, the places p_2, p_3, and p_5 provide inputs to transition t_2, and p_3 and p_5 are the output places of t_2. The marking function m assigns one token to p_1, p_2 and p_5 and two tokens to p_3, as denoted by $m(p_1, p_2, p_3, p_4, p_5) = (1, 1, 2, 0, 1)$.

As mentioned above, a Petri net executes by means of firing transitions. A transition can fire only if it is enabled – that is, if each of its input places has sufficient tokens to fire. A transition is said to have fired when it has removed all of its enabling tokens from its input places, and then deposited tokens into each output place. In Figure 2.6, for example, after transition t_2 fires, the marking m will change to $(1, 0, 2, 0, 1)$.

Petri nets are useful because they can effectively model a variety of system characteristics, and may be used to check several useful properties. Chapter 4 presents Petri nets in more details. Although a Petri net does have many advantages in modeling and analyzing concurrent systems, it also has limitations that are similar to those of

an FSM, that is, it can quickly become incomprehensible with any increase in system complexity [35].

Program-State Machine

A program-state machine (PSM) [34] is another instance of a heterogeneous model that integrates an HCFSM with a programming language paradigm. This model basically consists of a hierarchy of program-states, in which each program-state represents a distinct mode of computation. At any given time, only a subset of program-states will be active.

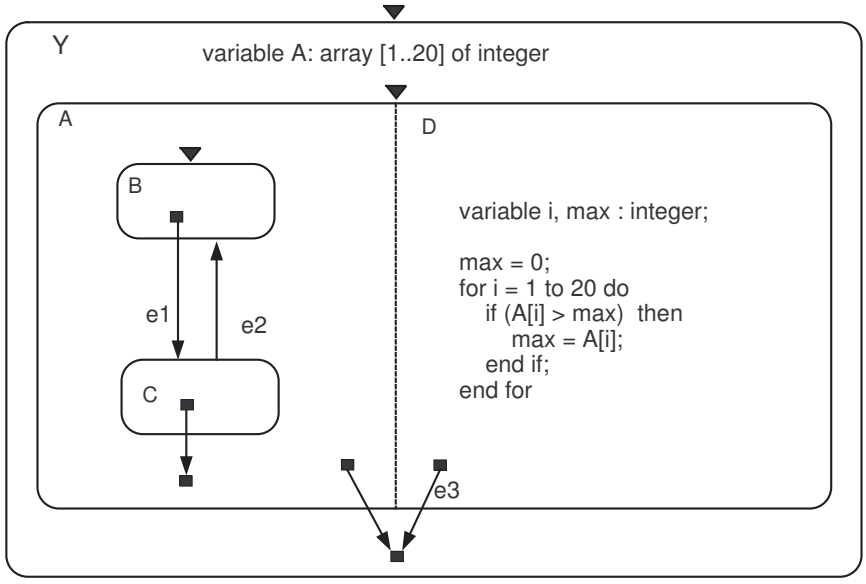

Figure 2.7: An Example of Program-State Machine

Figure 2.7 shows an example of a program-state machine, consisting of a root state Y, which comprises two concurrent substates, A and D itself. State A, in turn, contains two sequential substates, B and C. Note that states B, C, and D are leaf states, though the figure shows the program only for state D. According to the graphic symbols, the arcs labeled e_1 and e_3 are *transition-on-completion* arcs (will change state only when the source program-state has completed its computation and the associated arc condition evaluates to true), while the arc labeled e_2 is a *transition-immediately* (change state immediately whenever the arc condition becomes true, not considering if the source program-state has completed its computation) arc. The configuration of arcs would mean that when state B finishes and condition e_1 is true, control will transfer to state C. If, however, condition e_2 becomes true while in state C, control will transfer to state B regardless of whether C finishes or not.

Since PSMs can represent system states, data, and activities in a single model, they are more suitable than HCFSMs for modeling systems which have complex data and activities associated with each state. A PSM can also overcome the primary limitation of programming languages, since it can model states explicitly. It allows a modeler to specify a system using hierarchical statedecomposition until he/she feels comfortable using program constructs. The programming language model and HCFSM model are just two extremes of the PSM model. A program can be viewed as a PSM with only one leaf state containing language constructs. A HCFSM can be viewed as a PSM with all its leaf states containing no language constructs.

Process Algebra

The word "process" refers to behavior of a system. Usually, behavior comprises a set of events (or actions) that a system can perform, the order in which they can be executed and maybe other aspects of this execution such as timing or probabilities. The word "algebra" denotes taking an algebraic/axiomatic approach when talking about behavior [9]. So, process algebra is defined as an algebraic method for studying concurrent processes. Process algebra tools are algebraic languages for the specification of processes and the formulation of statements about them, together with calculi for the verification of these statements [98]. Process algebra is an example of activity-oriented model.

The main algebraic approaches to concurrency are:

(i) Milner's CCS *(Calculus of Communicating Systems)* [65];

(ii) Hoare's CSP *(Communicating Sequential Processes)* [41]; and

(iii) Bergstra and Klop's ACP *(Algebra of Communicating Processes)* [13].

The simplest model of behavior is to see behavior as an input/output function. A value or input is given at the beginning of the process, and at some moment there is a value as output. This model was instrumental in the development of (finite state) automata theory. In automata theory, a process is modeled as an automaton. An automaton has a number of states and a number of transitions, going from one state to another (or the same) state. A transition denotes the execution of an (elementary) action, the basic unit of behavior. Besides, there is an initial state (sometimes, more than one) and a number of final states. A behavior is a run, i.e. a path from initial state to final state.

When observing the automata theory, it can be seen that, basically, the notion of interaction is missing. That is, during the execution from initial state to final state, a system may not interact with another system. However, this is needed in order to describe parallel or distributed systems, or so-called reactive systems. Concurrency is the theory of interacting parallel and/or distributed systems. Process algebra is usually considered as an approach to concurrency theory, so it will usually (but not necessarily) have parallel composition as a basic operator.

Thus, process algebra is the study of the behavior of parallel or distributed systems by algebraic means. It offers means to describe or specify such systems, and thus it has means to talk about parallel composition. Besides this, it can usually also talk about alternative composition (choice) and sequential composition (sequencing). Moreover, by means of equational reasoning (argument using algebra), verification can be performed, i.e. establishment that a system satisfies a certain property.

The basic laws of process algebra are usually called structural or static laws. As basic operators, | denotes alternative composition, $\rightarrow$ denotes sequential composition and || denotes parallel composition. Some basic laws are the following (| weakest, $\rightarrow$ strongest):

- $x|y = y = |x$ (commutativity of alternative composition)
- $x|(y|z) = (x|y)|z$ (associativity of alternative composition)
- $x|x = x$ (idempotency of alternative composition)
- $(x|y) \rightarrow z = x \rightarrow z|y \rightarrow z$ (right distributivity of | over ;)
- $(x \rightarrow y) \rightarrow z = x \rightarrow (y \rightarrow z)$ (associativity of sequential composition)
- $x||y = y||x$ (commutativity of parallel composition)
- $(x||y)||z = x||(y||z)$ (associativity of parallel composition)

As an example of alternative (or choice) composition, suppose the description of a dispensing machine which dispenses hot coffee if the red button is pressed, and iced tea if the blue button is pressed. This is specified in the following way:

`DRINKS = ((red → coffee → DRINKS) | (blue → tea → DRINKS))`

Figure 2.8 depicts the graphical state machine description of the drinks dispenser.

The representation of parallel composition generates all possible interleavings of the traces of its constituent process. For example, the process:

`ITCH = (scratch → STOP)`

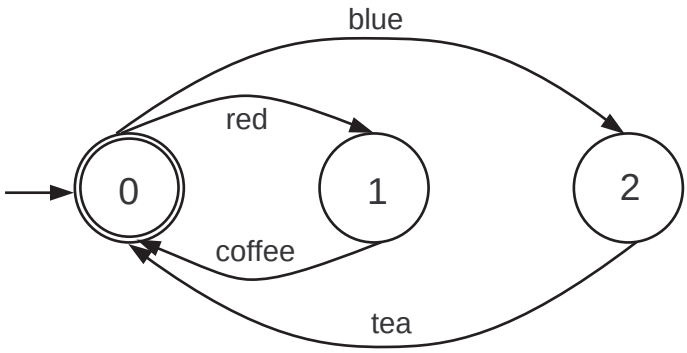

Figure 2.8: DRINKS state machine

has a single trace consisting of the action `scratch`. The process:

`CONVERSE = (think → talk → STOP)`

has the single trace `think → talk`. The parallel composition is:

`CONVERSE-ITCH = (ITCH || CONVERSE)`

which has the following traces:

`think → talk → scratch`

`think → scratch → talk`

`scratch → think → talk`

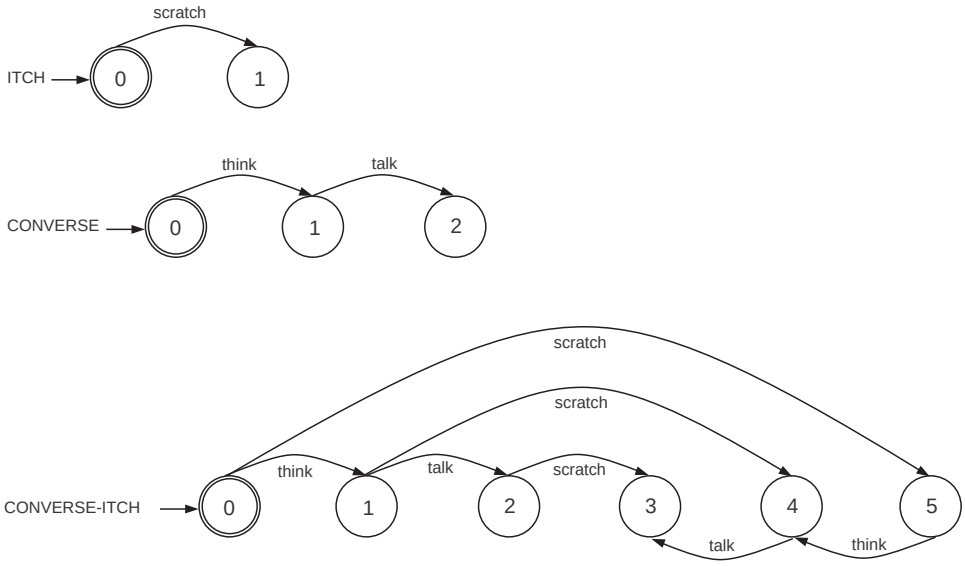

Figure 2.9: Composition CONVERSE-ITCH

Figure 2.9 depicts the graphical state machine description of the CONVERSE-ITCH parallel composition.

2.2 Embedded Systems

Embedded systems are everywhere, from home appliances to spaceships. Nowadays, the great majority of systems have an embedded system into them. Some products become possible thanks to the computational systems that are integrated into them, for instance, cellular phones, electronic fuel injection, ABS break, and so on. This phenomenon is somehow related to the technology advances (and reduced price) of micro-processed systems started in the 80's, which result in new conception of products.

This section provides an introduction to embedded systems. First, an overview is presented. Next, the three main design representation are shown. Later, design of embedded systems is depicted, where the two main methodologies are considered, namely, hardware-software co-design and platform-based design. Finally, the main problems with embedded software and challenges for embedded design methodologies are presented.

2.2.1 Overview

In general, an embedded system is a specialized digital system that executes a group of dedicated functions within a larger system in such a way that functionalities are added or optimized. Typically, embedded systems consist of off-the-shelf general-purpose processors, ASICs and/or FPGAs. They use a computer, but they are neither used nor perceived as a computer. Virtually all appliances that have a digital interface, such as watches, microwaves, VCRs, cars, etc, have an embedded system. Some embedded systems include an operating system kernel, but many are so specialized that the entire logic can be implemented as a single program.

Embedded systems and desktop computing application differ substantially on the design constraints. For instance, in addition to the CPU and memory hierarchy, there is a variety of interfaces that enable the system to measure, manipulate, and interact with external environment. Embedded systems typically have tight constraints on both functionality and implementation. In particular, they must guarantee reliability of the application, cost pressure, real-time requirements, small size, low weight, long-life cycle, low energy consumption, among others. Certainly, such restrictions make those systems difficult to be successfully designed through traditional computing methodologies. Another characteristic of embedded systems is that they usually do not operate in a strict controlled environment. Excessive heat is often a problem. Some systems need protection from vibration, shock, lightning, power supply fluctuations, water, corrosion, fire, and several others physical abnormalities [47].

As presented in Section 2.1 there are lots of models of computation available, which arise two alternatives for designing complex and heterogeneous systems: (i) using a single unified formalism; or (ii) mixing several models [35]. The first method is very complicated to attain, since the semantics of models may be suited for one domain, but not for another domain. Usually, the adopted solution is to use the second method. The key problem in the mixed method is to define the semantics of interactions among such models. This problem is not so easy as interfacing different languages, since the issue is on the respective semantics of each model. The main objective of the Ptolemy project [18] is to study the interaction semantics of mixed models of computation.

The software in embedded systems is much more constrained than in general-purpose computing. For instance, embedded software cannot use unconstrained dynamic memory allocation nor virtual memory. For some highly critical applications, even the use of a stack may be forbidden [50].

Designers have to deal with a dilemma, since embedded systems have increased complexity and, at the same time, market pressures have shortened the time-to-market. In order to cope with those stringent requirements, appropriated development methodologies play an important role.

2.2.2 Design Representations

Managing the design complexity and heterogeneity is the key problem in embedded systems development. Usually, embedded systems are developed in several levels of abstractions in such a way that the complexity is minimized. Embedded system design have established three different representations [35]:

- *Behavioral representation* views the design simply as a black box. It defines how the black box would respond to any combination of input values, but omits any indication on how to design such black box. In other words, this representation describes system functionality, and tells nothing about its implementation;

- *Structural representation* defines the black box in terms of a set of components and their inter-connections. This representation concentrates on specifying the product's implementation. However, the structural representation does not explicitly describe the whole functionality;

- *Physical representation* specifies the physical characteristics of the components described in the structural representation. For instance, a physical representation

would provide dimensions and location of each component as well as physical characteristics of the connections between them.

In general, the system design process starts from a behavioral representation, which is then translated to a structural one, and finally it is translated to a physical representation. This way, each translation adds implementation details in the design.

2.2.3 Design of Embedded Systems

Embedded systems are often used in life critical situations, where reliability and safety are more important criteria than performance. Usually, embedded systems are designed with an *ad hoc* method based on earlier experience of the designer. Development methodology is important since the major goal is to produce new systems in a reliable way. The methodology may be able to assess system requirements, sometimes develop an architecture, and synthesize the embedded system. These activities have to be made in such a way that predictable results may be obtained within an acceptable amount of time.

In accordance with Edwards *et al.* [29], the concurrent design process for mixed hardware/software embedded systems involves solving the following sub-problems: specification, validation, and synthesis. In that paper, the authors advocate a design process based on precise mathematical representations, in such a way that both verification and map from initial description to the various intermediate steps can be carried out with tools. The design process takes a model at a level of abstraction and refines it at lower ones. Moreover, the designer must ensure that the properties at each level of abstraction are verified, the constraints are satisfied, and performance is satisfactory.

The increasing embedded system design complexity combined with a very tight time-to-market has motivated research in embedded system development methodologies. The attempt to apply traditional computer design methodologies and tools to embedded applications is difficult to be successful, once embedded systems have several additional characteristics not found in traditional design.

There are some methodologies that have been used for embedded systems development. However, this section concentrates only on hardware-software co-design, and platform-based design.

Hardware-Software Codesign

Hardware-Software co-design can be defined as the cooperative design of hardware and software. Therefore, it certainly has to deal with heterogeneous environments. Taking

into account high complexity allied with short time-to-market, this methodology is used to maintain the abstraction level in such a way that the designer can be kept from design details, which are more suitable for automated tools.

The recent interest for hardware-software co-design can be justified by technological advances, and by the increasing complexity of applications. Hardware-software co-design is a design paradigm that comprises specification, design and synthesis of systems that mix hardware and software components, such as embedded systems. The growth of this paradigm owes mainly to hardware high-level synthesis development of the 80's. Therefore, the abstraction level has been raised, and hardware development is becoming more and more closer to software development.

However, this method should trade-off reconfigurability (solved by hardware to software migration) and performance (solved by software to hardware migration). Further, there is a constant effort for lowering costs and reducing time-to-market.

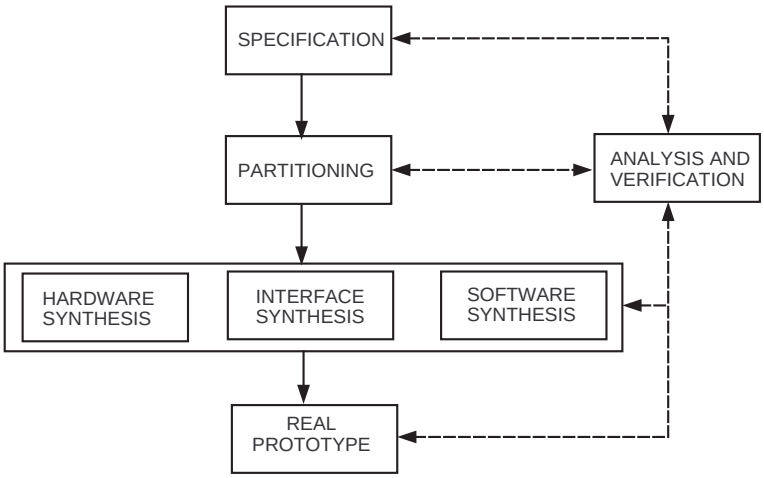

Figure 2.10: Main Phases of a Hardware-Software Codesign Methodology

Main Phases

The hardware-software codesign methodology is generally composed by four main phases (Figure 2.10), where each one enforces different aspects of the concurrent design. These phases are explained below:

a) *Specification.* This phase is related to high abstraction level requirement description of systems. The specification language should allow the description of functional and non-functional requirements. Examples of non-functional requirements can be: performance, energy consumption, cost, area, time-to-market, dependability, etc. There are several formalisms for system specification, such as:

Petri nets [74, 70], Finite State Machines [35], Statecharts [39], Process Algebra [41], and so on.

b) *Partitioning.* This phase decides which components should be implemented in hardware and those to be implemented in software. This process is carried out in three general ways: by hand, using automatic tools, or interactively (using a combination of the two previous methods). Since the partitioning is classified as good or bad according to estimated metrics, it is important to have good estimators. The most common metrics are: cost, execution time, silicon area, communication rate, power consumption, pin numbers, memory area, data area, program size, etc. Bad partitioning may affect the design time and cost, sometimes requiring re-implementation of the whole system. Designers are faced with a difficult choice, since hardware may provide better performance than software, but it is more expensive. On the other hand, software is cheaper than hardware, but it is slower. Several methods have been used for automatic partitioning, such as: hierarchical clustering [20], min-cut [49], simulated annealing [46], integer linear programming [52], and others.

c) *Co-Synthesis.* The output of the partitioning phase is a set of communicating modules, where some of them should be implemented in hardware and others in software. This set of modules is called virtual prototype. The next step is called co-synthesis, which is a method that allows the automatic mapping from the virtual prototype to the real prototype in such a way that all system constraints are satisfied. The synthesis decisions are considered in this phase. For instance, the 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, 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) *Analysis 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, its product 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. For instance, co-simulation makes possible program interaction with an ASIC without it has been implemented.

Since this methodology copes with heterogeneous environment, the design of embedded systems should support the complete hardware-software co-design phases, including partitioning, hardware, software, and interface synthesis.

Current Status and Trends

The main trends in hardware-software co-design are the subject of this section. In accordance with Rolf Ernst [30], the main status and trends in hardware-software co-design for embedded systems are:

- reusing components taken from previous designs or acquired from outside the design group is a main design goal to improve productivity and reduce design risk;
- hardware/software designers and system architects must synchronize their work progress to optimize and debug a system in a joint effort. The early discovery of design faults is a central requirement to that cooperation;
- the challenge is to support the migration of system functions between different technologies and between hardware and software without a redesign;
- a major problem in the design process is synchronization and integration of hardware and software design. This requires permanent control of consistency and correctness, which becomes more time consuming with increasing levels of details;
- executable specifications depend on the application domain, which are based on different models of computation;
- virtual prototypes do not cover most of the nonfunctional constraints and objectives, such as power consumption or safety;
- a considerable amount of assembly code in embedded systems is still observed. This is because the compilation is, in many cases, far less efficient than manual code generation. Even if compilation improved, the problem of generating efficient compilable code from abstract models is an important concern;
- guaranteed timing behavior of the generated code is another problem;

- interface synthesis has been neglected for a long time in commercial tools;
- many embedded systems consist of a complex, heterogeneous set of standard processors, ASIPs, co-processors, memories, and peripheral components. The designer typically preselects the architecture to reduce the design space;
- when considering ASIPs (Application-Specific Instruction-Set Processors), generally, compilers, libraries, operating system functions, and simulation/debugging environment have to be adapted;
- tools that cover a large variety of communication mechanism are still difficult to develop;
- efficient synthesis tools and compilers are generally not available for all target architectures;

Platform-Based Design

Platform-based design (PBD) is a powerful concept for dealing with the increased pressure on time-to-market, design and manufacturing costs [31, 63, 80]. These design problems are pressing companies toward designs that can be assembled quickly from pre-designed components versus full custom design methods. This implies in high-priority on design re-use, correct assembly of components, and fast, efficient compilation from specifications to implementations, correct-by-construction methodologies and fast/accurate verification. This idea has been exploited a long time ago in the design of personal computers, but now the method is generalized and formalized for the design of electronic systems that consist of software and hardware components and integrated circuits.

The establishment of an economically feasible electronic design flow requires a structured methodology in order to limit the space of exploration (in contrast with hardware-software co-design methodologies, for instance) with the aim of achieving very good results in the tight time-to-market constraints. This method has been very powerful in design for both integrated circuits and computer programs. For computer programs, the use of high-level programming languages has replaced for the most part assembly languages, for integrated circuits, regular structures such as gate arrays and standard cells have replaced transistors as a basic building block [80].

The concept of platform has been adopted for a long time. However, there are many definitions of "platform", where such definitions depend on the domain of application. In the integrated circuit (IC) domain, a platform is considered a flexible integrated

circuit where customization is achieved by programming components of the chip. In the personal computer (PC) domain, PC makers have been able to develop their products quickly and efficiently around a standard "platform" (such as x86 instruction set architecture, set of buses (ISA, PCI, USB), set of I/O devices, etc). Platform-based design is defined as the creation of a stable microprocessor-based architecture that can be rapidly extended, customized for a range of applications, and suitable for quick development.

In this context, each platform represents a layer in the design flow for which the underlying design-flow steps are abstracted, i.e., abstraction layers that hide the unnecessary details from lower level of abstractions. Often the combination of two consecutive layers and their inter-relations can be interpreted as a unique abstraction layer with an "upper" view, the top abstraction layer and a "lower" view, the bottom layer. Every pair of platforms, together with the tools and methods used to map the upper layer into the lower layer, is a *platform stack*.

The main issue of the application of such design principle is the careful definition of the platform layers. Platforms can be defined at several points of the design process. There are several distinct platforms. However, the two main platforms that need to be defined together with methods and tools necessary to link them are: architecture platform, and API platform.

Generally, integrated circuits used for embedded systems are developed as instances of a particular *architecture platform*. In other words, instead of being assembled from a collection of independently developed blocks of silicon, they are developed from specific micro-architectures that can be extended or reduced by the system developer. An *architecture platform instance* is derived from an architecture platform by choosing a set of components from the architecture platform library and/or by setting parameters of re-configurable components of that library. That is, the flexibility (or the ability of supporting different applications) is guaranteed by programmable components.

The concept of architecture platform is not enough to achieve the required level of software re-use. The architecture platform should abstract the layer called Application Program Interface (API) or Programmers Model. This layer involve the essential parts of the architecture platform: (i) programmable cores and memory subsystem via a Real Time Operating System (RTOS); (ii) I/O subsystem via the Device Drivers; and (iii) network connection via the network communication subsystem. Therefore, the API layer is a platform itself called the *API platform*. In the conceptual platform-based design framework, API platform is a unique abstract representation of the architecture platform. With an API so defined, the application software can be re-used for every

platform instance.

The *system platform-stack* is the combination of two platforms and the tools that map one abstraction into the other. A platform-stack is a single layer obtained by joining together both the top and bottom platforms, where the upper view is the API platform and the lower view is the architecture platform. The mapping of the application into the actual architecture may be carried out automatically if appropriate software tools (e.g., software synthesis, RTOS synthesis, device-driver synthesis) are available. It is clear that synthesis tools should consider architecture details as well as API features.

2.2.4 Embedded Software

The general aim of embedded system design is to implement a specific set of functions while satisfying constraints such as performance, cost, power consumption, size, and weight. These functions may be implemented as a hardware component or as software running on a programmable component.

Nowadays, the embedded systems functionalities have grown in number and complexity that development time has become difficult to predict and control. This complexity has forced designers to take into account flexible implementations. Furthermore, hardware-manufacturing cycles are more expensive and time-consuming. Hence, software-based implementation has become a feasible alternative solution.

In this case, the computational processor power increasing, and the corresponding processor size and const reductions have allowed moving more and more functionality to software. Moreover, software-based implementation provides higher degree of flexibility than hardware-based implementation, and so it is easier to meet time-to-market constraint. Recent market analysis indicates that software accounts for more than 80% of system development. Thus, in order to be competitive, companies have to have a powerful software development environment [81].

Problems with Embedded Software

Sangiovanni-Vincentelli and Martin [81] show several problems with embedded software development, which are summarized below.

Although software has several advantages when compared with hardware, it also has some disadvantages. One of these disadvantages is related to performance. It is clear that software has poorer performance than hardware. In order to overcome performance constraints, programmers usually use assembly or C. However, this policy

may affect the time-to-market, readability and maintainability of the resultant software. In general, embedded software also needs hardware support for debugging and performance evaluation, which are more important for embedded software than for standard software.

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

Another classical disadvantage is the increasing difficulty in verifying design correctness. This verification is critical due to safety considerations in several application domains. Additionally, little attention has been given on hard deadline constraints, low use of memory and power consumption of software.

Embedded Software Design Methodology

According with [82], a software development methodology for embedded systems has to consider challenges such as:

- 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 of parallel development via integration technology
- development of process standards and common workflows.

The embedded software design methodology proposed by [81, 82] is intended to have an optimized, semi-automated, transparent, verifiable, and mathematically correct flow from product specification to implementation of software-dominated products implemented with highly programmable platforms (this methodology is based on

platform-based design). The aim of this section is to depict the main stages of design (specification, refinement and decomposition, and analysis), implementation (target platform definition, mapping, and links to implementation) and verification. These stages are really associated with embedded systems design methodology. Nevertheless, they are analyzed in this section by the software project perspective.

Specification

In this context, specification is the entry point of the design process. It should contain:

- description of the system's functionality in such a way that it does not imply an implementation;
- a set of constraints on the system final implementation; and
- A set of design criteria.

A set of criterion is often more qualitative than quantitative characteristics, such as reliability, testability, maintainability, and manufacturability. The difference between constraints and criteria is that constraints, must be met, whereas you do your best to optimize criterion. For instance, a criterion may be higher autonomy, while constraint may be the maximum power dissipation permitted.

Refinement and Decomposition

After obtaining a specification, the design process should progress toward implementation through well-defined stages. The method manipulates the description by introducing additional details while preserving both functionality, properties and meeting the constraints. When steps are smaller, it is more easier to formally prove that constraints are met and properties are satisfied. This process, called successive refinement, is one of the main features of the proposed embedded software methodology of [81].

During successive refinement, it is often convenient to break parts of the design description into smaller parts so that optimization techniques have a better chance of producing interesting results. This is called decomposition. It must be determined whether the decomposed system satisfies the original specification.

Analysis

While going to the final implementation, designers sometimes take paths that lead to designs that do not satisfy some of the constraints. Hence, designers must have tools that evaluate intermediate results with respect to the constraints.

Target Platform Definition

Since most embedded systems are defined to map onto a target platform, it is necessary to find the right form and notation with which a target platform can be described. This description has to represent the full scope of its service (computation, communication, coordination, etc) and configurations. When a platform offers reconfigurable logic, new methods of describing the service and configuration are required.

Mapping

The mapping associates parts of the specification (usually they are already refined) with specific implementation components of the target platform.

Link to Implementation

In general, platform-based design uses reusable components offered by the platform together with the necessary configuration. However, resultant products often contain new or modified functionality. Thus, the methodology must support software, hardware, and interface synthesis to allow a comprehensive flow from specification to implementation.

Verification

This phase consists in verifying if the system is in accordance with the design criteria. If it is adopted a mathematical specification method, several properties are satisfied by construction. In addition, verifying whether an implementation satisfies the original specification can be made much easier if formal successive refinement techniques are used.

When the embedded software to be developed is simple, there is no need for a more sophisticated method. However, in complex embedded software applications, this rather primitive method has become the bottleneck. Most of the issues raised in this section is usually in *ad hoc* fashion. This is why Sangiovanni-Vincentelli and Martin [82] say that the way in which embedded software is developed today have to be changed radically.

2.3 Real-Time Systems

In real-time systems, not only the logical result of the computation is important, but also the time in which the result is obtained. In other words, real-time computing has to satisfy both logical and timing correctness. The logical correctness concerns the generated output by computation (correct result) and the internal state of the system (not to reach prohibited states). The timing correctness decides if a computation meets its timing constraints, such as completion time and deadlines.

Two different approaches for designing real-time systems can be identified: event-triggered and time-triggered. In *event-triggered*, a system activity (communication and/or processing) is initiated as a consequence of the occurrence of a significant event. In *time-triggered*, all activities are initiated at predetermined points in time [48].

2.3.1 Timing Constraints

The distinction between real-time computing and fast computing is important in this context. Fast computing aims to get results as quickly as possible while real-time computing aims to get the desired results within prescribed timing constraints. It is very common to see references to real-time systems when what is meant is just fast systems. Obviously, real-time is not necessarily synonymous with fast; that is, it is not the latency of the response *per se* that is an issue, but the fact that a bounded latency sufficient to solve the problem is guaranteed by the system.

Real-time systems are particularly interested in timing constraint satisfaction. Such constrains are generally specified by periods and deadlines. Periods denote execution of tasks in regular intervals. Deadlines, however, correspond to the maximum time, starting from the task arrival up to the task completion.

Other important timing constraints are: (i) *arrival time*, which is the time instant where the system knows about the arrival of a task; (ii) *start time*, which is the instant of the start of task processing in each activation; (iii) *execution time*, which is a sufficient time to complete execution of a task; (iv) *completion time*, which is the time instant of completion of task in each activation; (v) *release time*, which is the earliest start time allowed for a task in each activation.

Real-time systems can only guarantee that deadlines are satisfied if the worst-case execution times (WCET) of all application tasks are *a priori* known. The WCET of a task is an upper bound for the time between task activation and task completion. It must be valid for all possible input data and execution scenarios of the task.

2.3.2 Classes of Real-Time Systems

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

Hard real-time systems require absolutely guaranteed predictable responses and behaviors. These systems are often used to control life-critical operations in such a way that, any failure to meet timing constraints results in disastrous consequences, in some cases loss of human life. Furthermore, any lateness in execution of real-time tasks is not permitted under any circumstances. Such systems also have to employ a high degree of robustness and fault-tolerance. A good example is a robot that has to pick up something from a conveyor belt. The piece is moving, and the robot has a small window to pick up the object. If the robot is late, the piece will not be there anymore, and thus the job failed, even though the robot went to the right place. If the robot is early, the piece will not be there yet. Other examples of hard real-time systems are aircraft navigation, nuclear power plant control, health care equipments, automatic pilot, and so on.

In soft real-time systems, on the other hand, it is not catastrophic when deadlines are not met. In this case, timing constraints may occasionally not be reached, causing just degradation in the system behavior and such deadline missing can be tolerated. Usually, this kind of system has a trade-off between execution time and desired results accuracy. Moreover, results lateness may only increase system cost. A data acquisition and display application (in which readings are periodically taken and displayed) is an example of a soft real-time application. Although there may be a desired sample and display rate, there will not be an error if the sample rate is not accurately met. The worst that can happen is that samples are not displayed as quickly as desired. On-line transaction systems, telephone switches, electronic games, and airline reservation systems are other examples of soft real-time systems

2.3.3 Periodicity of Tasks Execution

Another important characteristic of real-time systems is based on activation regularity. The task model contain three kinds of tasks:

- Periodic tasks are those where activations occur once by regular interval called period.

- Aperiodic tasks are randomly activated.

- Sporadic tasks corresponds to a subset of the aperiodic tasks, but with a minimum interval between two activations.

Considering the predictability of periodic tasks, they are generally associated with hard deadlines. On the other hand, aperiodic tasks usually have soft deadlines. Sporadic tasks may have hard deadlines, since their definition guarantees a minimum interval between two activations of the same task.

2.3.4 Characteristics of Real-Time Systems

Real-time systems are characterized by their timely response to external stimuli, predictable behavior, dependability, accuracy of outputs, and concurrency of tasks. This section is based on [89].

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. A second 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. In other words, the system should not be executing the tasks in some non-deterministic fashion each time it runs.

Dependability. Predictability also implies dependability (or robustness) of the system. 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. Therefore, 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. This is often the hardest part of a real-time system and very difficult to ensure.

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 can have serious consequences. Sometimes it is impossible to compute accurate results in the given timing constraints. In such cases, a trade-off between computation time and accuracy results is very important. However, it is not easy to decide what is an acceptable level of accuracy and how much time should be spent in order to try to achieve it.

Concurrency. The least visible of all real-time characteristics is the inherent concurrency. Thus, viewing a real-time system as a collection of concurrent process is

actually quite common. The presence of such parallelism in real-time systems introduces some additional complexities, such as: (i) parallel process must be scheduled correctly to meet timing constraints. Conventional scheduling algorithms may not provide best solutions when considering multiprocessors; (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.

2.3.5 Specification and Verification of Real-Time Systems

The fundamental challenge in the specification and verification of real-time systems is how to incorporate the time. Methods (formal or informal) must be developed to incorporate these timing criteria into the specifications of a real-time system. Similarly, verification methods must make sure that these timing constraints are being met and that the system is robust, predictable and accurate. This 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 used because of the difficulty of using such formalisms. However, many accidents could have been avoided if formal methods were used. An example of such an accident is the delay in the first space shuttle flight due to an improbable race condition that went undetected during the multiple runs of the system [37]. Another problem was reported that during the Mars Pathfinder mission, the spacecraft experienced repeated total system resets, resulting in losses of data. The problem was reported to be caused by "priority inversion" [84] in the real-time systems kernel that used priority scheduling.

2.4 Scheduling

Some kind of scheduling is required by almost all software synthesis methods to sequence the execution of concurrent tasks. Although, concurrent tasks are an excellent specification mechanism they cannot be implemented as such on a standard CPU. The scheduling problem amounts to find a linear execution order of tasks, so that all timing constraints are satisfied.

Scheduling is a topic studied for many years. Although this section was restricted to consider only real-time scheduling theory, there are an enormous amount of research results. Therefore, this section presents just a summary of the main contributions concerning scheduling with timing constraints.

This section shows the scheduling complexity, three general methods for scheduling real-time systems, and a brief comparison between runtime and pre-runtime scheduling.

2.4.1 Scheduling Complexity

Garey and Johnson [36] have shown that the problem of finding a feasible schedule for a set of non-preemptable processes with release times and deadlines in a mono or multiprocessor architecture is NP-complete.

The scheduling problem can be solved in polynomial time when considering processes consisting of single segment that can be preempted by any other process (all process are independent), even if n processors are used [58].

Other works use heuristics, branch-and-bound, depth-first, and other techniques to find a feasible schedule (generally off-line) considering arbitrary precedence and exclusion relations. This problem is also NP-Complete [105]. However, any algorithm (using such techniques) for scheduling tasks in monoprocessors that takes into account exclusion relations has to consider a special case where all tasks mutually exclude each other. This is identical to the NP-complete problem studied in [36]. Thus, algorithms have to deal with the complexity involved in solving that special case.

The problem becomes more complex when multiprocessor architectures are considered. In this situation, process allocation and scheduling are NP-complete problems, even if the only goal is the minimization of the overall execution time.

2.4.2 Methods for Scheduling

In general, real-time scheduling policies are classified as:

- runtime (also called dynamic or on-line) scheduling; and
- pre-runtime (also called static or off-line) scheduling.
- hybrid scheduling.

Runtime policy is rigidly based on priorities, that is, the task to be chosen for execution (from the ready queue) is the one with highest priority. Therefore, the schedule is computed on-line when tasks arrive for execution. In pre-runtime policy, on the other hand, the schedule is computed entirely off-line, and tasks are executed in a fixed and predetermined order. In this case, this policy is not constrained by any such priority. Hybrid policy is a combination of both previous policies.

Let us take a look at runtime, pre-runtime, and hybrid methods in more depth.

Runtime Method

As stated before, the runtime method usually assumes that processes have priorities assigned to them. Priorities can be determined either statically, at design time, or dynamically at runtime. Moreover, a runtime scheduler may be *preemptive*, if a running task may be interrupted during its execution, or *non-preemptive*, otherwise.

This section presents the following runtime scheduling algorithms: (i) Rate Monotonic Scheduling, representing a simple static priority algorithm; (ii) Priority Ceiling Protocol, which is a static priority with access to shared variables; (iii) Deadline Monotonic, which is an static priority based on deadline, but deadlines can be less than or equal to the periods; and (iv) Earliest Deadline First, that represents the dynamic priority assignments. In the presentation that follows, let us assume that c_i is the execution time, d_i is the deadline, and p_i is the period of task τ_i.

Rate Monotonic Scheduling

Static priority runtime scheduling has received significant attention since the pioneering work of Liu and Layland [58], called Rate Monotonic Scheduling (RMS). RMS produces schedules at runtime through preemptive schedulers driven by fixed priorities. In spite of the theoretical results presented by Liu and Layland have improved the understanding of real-time scheduling, the method is valid only for very limited applications, since RMS premises define a very simple task model: (a) the tasks are periodic and independent; (b) deadlines are equal to periods; (c) execution times are known and constant; and (d) the context-switching time is assumed to be negligible.

In order to assign priorities, the adopted policy is based on the period of each task, that is, shortest period highest priority. Therefore, at any time, the task with highest priority, among all tasks ready to run, is assigned to the processor. Liu and Layland showed that RMS is optimal in the sense that if the RMS priority assignment is not feasible, a set of tasks is not schedulable.

The schedulability analysis (verification whether a given schedule satisfies all deadlines) is based on the utilization factor (U). Liu and Layland have found out the least upper bound on processor use in a static priority scheme. In this case, for n tasks reach timing constraints, the following test (sufficient condition) has to be satisfied:

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

In other words, if the utilization factor is less than $n(2^{1/n} - 1)$ that set of tasks is schedulable.

Priority Ceiling Protocol

The primary difficulty with the use of semaphores in real-time systems is that a high priority process can be blocked by lower priority processes an unbounded number of times. Consider for instance a high priority process H wishing to gain access to a critical section that is controlled by a semaphore. Assume at the time of H's request a low priority process, L, has locked the critical section. The process H is said to be blocked by L. This blocking is inevitable and is a direct consequence of providing resource integrity (i.e. mutual exclusion) [7]. This situation is known as priority inversion.

The Priority Ceiling Protocol (PCP) [84] makes the same assumptions as RMS, except that, in addition, processes may have critical sections guarded by semaphores, and a protocol is provided for handling them in order to avoid priority inversion. For each semaphore, it is assigned a *priority ceiling*, which is equal to the priority of the highest priority process that may use this semaphore. The process that has the highest priority among the processes which are ready to run, is assigned to the processor. Before any process p enters its critical section, it must first obtain the lock on the semaphore S guarding the critical section. If the priority of process p is not higher than the priority ceiling of all semaphores currently locked by processes other than p, then process p will be blocked and the lock on S denied. When process p blocks higher priority processes, p *inherits* the priority of the blocked processes by p. When p has a critical section, it resumes the priority it had at the point of entry into the critical section.

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.

Adopting PCP the following benefits are reached:

- A high priority process can be blocked at most once during its execution (per activation);
- Deadlocks are prevented; and
- Transitive blocks are prevented.

Deadline Monotonic

The deadline monotonic scheduling (DMS) [53] extends the RMS task model in the sense that the relative deadlines can be less than or equal to the periods. The priority

policy of the DMS defines a static assignment of priorities based on relative deadlines. The priorities are assigned in inverse order of the relative deadline values. Similarly to RMS, DMS is a static-priority and on-line scheduling algorithm. However, no schedulability tests were given by Leung and Whitehead [53]. The work of Audsley [8] provides schedulability tests for this scheme. One such schedulability test is given by:

$$\forall i : 1 \leq i \leq n : \frac{c_i}{d_i} + \frac{I_i}{d_i} \leq 1$$

where I_i is a measure of a higher priority processes interfering with the execution of τ_i:

$$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{d_i}{p_j} \right\rceil c_j$$

In other words, this sufficient tests states that for a process τ_i to be schedulable, the sum of its computation time and the interference that is imposed upon it by a higher priority processes, must be no more than d_i. Audsley [8] also shows an algorithm presenting a more accurate schedulability test applicable to any fixed priority process set, where process deadlines are no greater than periods, whatever the assignment rule used for priorities.

Earliest Deadline First

The earliest deadline first (EDF) scheduling [58] also produces a schedule at runtime based on priorities, but the priority policy is dynamic, not static as the RMS. The premises are the same as RMS. The priority policy is defined following the absolute tasks deadlines, that is, the task with earliest deadline (considering the current time) has highest priority. At each arrival of a task, the ready queue is re-sorted taking into account the new priorities distribution. In EDF, schedulability is also verified in design time, taking as base the processor utilization factor. According to EDF, a set of tasks is schedulable if and only if:

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

There are several examples where a set of tasks is not schedulable by RM, but it is by EDF. Besides the higher processor utilization factor, another difference is that EDF produces less preemptions than RM. In favor of RMS is simplicity and implementation facility.

Pre-runtime method

Pre-runtime method schedules processes off-line. This requires that the major characteristics of process to be known in advance. If it is the case, an optimal schedule for these processes can be scheduled beforehand. The advantage is that it allows the user to know in advance if all deadlines can be met.

It is only possible to use pre-runtime scheduling to schedule periodic tasks. This method computes off-line a schedule for the entire set of periodic processes occurring within a time period that is equal to the least common multiple (LCM) of the periods of the given set of processes [106]. Although most of hard real time systems (e.g. control systems) have a large number of periodic processes [105], it is possible to translate an sporadic process into an equivalent periodic process [67, 106], if the minimum time between two consecutive requests is known in advance, and the deadline is not very short. However, this strategy may impose significant overheads. Anyway, it is also possible to schedule such sporadic process using pre-runtime scheduling.

Several algorithms and techniques is shown at Section 3.1 at Chapter 3 (Related Works).

Hybrid Methods

Hybrid methods combine the two previous methods in order to obtain the best of both.

One way of integrating off-line/on-line scheduling is to consider strategies that mix hard and soft real-time systems. Thus, pre-runtime scheduling may be employed for hard real-time constraints, whereas runtime scheduling may be applied for soft real-time constraints.

In situations where the translation from sporadic to periodic is not suitable, for instance when the deadline is very short, another adopted strategy (e.g. [104, 108]) is to use the pre-runtime scheduling for periodic tasks, and runtime scheduling for sporadic tasks. Of course the solution has to be carefully adopted if sporadic tasks have hard deadlines.

Considering pre-runtime scheduling, several alternative schedules may be computed off-line; each such schedule corresponds to a different mode of operation. A small runtime scheduler can be used to switch among the alternative schedules in response to external or internal events.

Such strategies are suited when the use of pure strategies are very limited by using one of them alone. Mixed solutions may bring benefits of both methods.

2.4.3 Runtime versus Pre-runtime Scheduling

The schedule quality often depends on where processors spend most of their time. In runtime scheduling, processors may spend precious time in on-line computation of schedules (when a task arrives for execution). This overhead may not lead to a feasible schedule, although this schedule may exists. In general, a solution for this problem might be the use of faster processors, however, it certainly raises the cost. Pre-runtime scheduling requires almost no CPU time when executing. The time required to compute the schedule (before execution) may not be negligible, but it is required only once in a system's lifetime [12]. However, pre-runtime scheduling policy spends CPU time due to sporadic-to-periodic translation. Sporadic arrivals are expected to be ready at beginning of each period, but such constraint is not always satisfied. Nevertheless, in accordance with Xu and Parnas [106], most hard real-time applications have periodic processes as the main part of them. In summary, pre-runtime scheduling is best suited for rigid periodic tasks [107].

Another overhead source is related to several context-switchings due to the scheduling policy. Some authors do not take into account this overhead, but it is really a concern, mainly in embedded systems, where the resources are scarce.

Schedulability analysis is needed in a runtime method in order to guarantee that all tasks meet their deadlines. However, this is a hard problem to be solved, even if the task model is simple and the main characteristics of tasks are precisely known. In contrast, there is no need to perform any schedulability analysis when using pre-runtime scheduling, as the schedulability is guaranteed when a feasible schedule is found.

Schedulability analysis is usually difficult to consider precedence relations, exclusion relations, release time not equal to the beginning of their periods, low jitter requirements (in this context, jitter refers to the variation in time where a computed result is output), etc. The reason for this difficulty is that additional applications constraints are likely to conflict with priorities. In general, it is impractical to map application constraints into a fixed hierarchy of priorities. Pre-runtime scheduling is not constrained by any priority scheme. Thus, when compared with runtime method, schedules considering complex timing and resource constraints are more feasible to be found.

When processes are scheduled at runtime, the scheduling strategy must avoid deadlocks. Usually, deadlock avoidance at runtime requires a conservative synchronization mechanism, resulting in situations where a process is blocked by the synchronization mechanism, even though it could proceed without causing deadlock. This may reduce the level of processor utilization. In pre-runtime scheduling, on the other hand, there is no need to worry about deadlocks, as a feasible schedule is guaranteed to be

deadlock-free when it is found.

An inconvenience in the pre-runtime scheduling occurs when the process periods are relatively prime. This would make the LCM become very long. In accordance with [107], in practice the period length can be adjusted in order to obtain a satisfactory length of the LCM of process period. However, this adjustment may cause processor utilization reduction.

The behavior of runtime schedulers can be very difficult to analyze and predict accurately. For example, suppose that a fixed priority scheduling is implemented by priority queues, where tasks are moved between queues by a scheduler that runs at regular intervals by a timer interrupt. It may be observed that, as the timer interrupt handler has a priority which is greater than any application, even a high priority task could suffer long delays while lower priority tasks are moved from one queue to another. It is proved that the prediction of scheduler overhead is a very complicated task. When computing a schedule off-line, it is much easier to analyze and predict the runtime behavior of the system, since the system is highly predictable. Nevertheless, dynamically scheduled systems are much more difficult to tune and debug than statically scheduled systems, since execution times and execution order are largely unpredictable [12].

Another problem with runtime scheduling (static or dynamic priority based scheduling) is that it has less chance of finding a feasible schedule than an optimal pre-runtime scheduling algorithm. This situation is hardened when considering arbitrary precedence and exclusion relations. For instance, consider the task set consisting of five tasks, A, B, C, D, E, and the respective timing constraints (release, computation, and deadline): $A = (0, 30, 161)$; $B = (11, 30, 51)$; $C = (60, 30, 90)$; $D = (41, 10, 100)$; and $E = (90, 50, 140)$. This specification also considers that B `PRECEDES` D, A `EXCLUDES` B, and A `EXCLUDES` D. Figure 2.11(a) shows that a runtime method could not find a feasible schedule, since tasks B and E miss their deadlines. However, a pre-runtime method finds a feasible schedule (Figure 2.11(b)). It is worth observing that the processor must be left idle between time 0 and 11, even though A's release time is 0.

2.5 Summary

This chapter described the main concepts needed to the understanding of this thesis.

Firstly, it introduces the main formal models adopted in the design of embedded real-time systems. Usually, formal models describe the behavior of the system at a high level of abstraction in order to ensure safe and correct designs. This chapter presented a model taxonomy and introduced some representative models such as automata, Petri

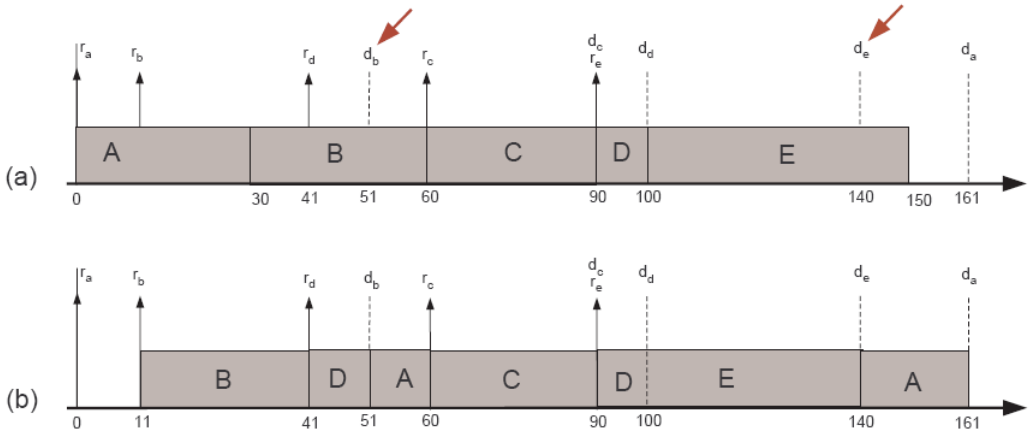

Figure 2.11: Comparison between runtime and pre-runtime scheduling

nets, program-state machine, and process algebra.

After that, this chapter detailed embedded systems. Particular attention was given to methodologies for design of embedded systems, where the two main methodologies were presented, namely, hardware-software co-design and platform-based design. The main problems with embedded software and challenges for embedded design methodologies are also considered.

In the following, real-time systems were introduced. A brief explanation about timing constraints, following by classes, periodicity, and some characteristics of real-time systems were described. Next, a briefly explanation about specification and verification of real-time systems was presented.

Finally, scheduling for real-time systems was explained. Scheduling complexity, three general approaches for scheduling real-time systems, and a brief comparison between runtime and pre-runtime scheduling methods were depicted.

Chapter 3

Related Works

This chapter shows a summary of the relevant related works. It is divided into four main sections: (i) pre-runtime scheduling; (ii) Petri nets in scheduling theory; (iii) integration between runtime and pre-runtime scheduling; and (iv) code generation.

3.1 Pre-runtime Scheduling

Although the literature presents several methods for scheduling, this section restricts the discussion to pre-runtime scheduling only.

Xu and Parnas [105] define the various terms and the exact scope of the problem of finding pre-runtime schedules. The authors divide each process into a set of continuous *segments*. Each segment has a release time, a fixed computation time and a deadline. All time intervals are measured in *time units*, which is defined as the smallest amount of time at which a segment can be in execution without being preempted. Exclusion, precedence and preemption relations can exist between segments. The *lateness of a segment* is calculated as the time at which the segment completes execution, subtracted by the deadline for that segment. The *lateness of a schedule* for a set of processes is the maximum lateness of any segment of any process in the set. A feasible schedule satisfies the deadlines of all processes and an optimal schedule is a feasible schedule with the minimum lateness. The algorithm proposed by Xu and Parnas considered a *branch-and-bound* technique, where a large number of possible schedules are analyzed in order to find the optimal solution.

The algorithm starts with the computation of an initial schedule, for the set of processes, using an EDF-based strategy. This initial schedule is the root of a search tree. Once the schedule has been computed, the segment with the maximum lateness, say j, is identified. The only way the lateness of j can be reduced is by making it

complete before one of the segments that currently finishes execution before it. Hence, two sets of segments $G1$ and $G2$ are identified such that the lateness of j can be improved by either scheduling it before a segment in $G1$ or by preempting a segment in $G2$. Finally, the lower bound for the delay is calculated. This value is the minimum lateness that any schedule generated from the current schedule can have. For each segment k in either $G1$ or $G2$ a child node is created. New relations are added to ensure that j precedes k, if k is in $G1$, and j preempts k, if k is in $G2$. The earliest-deadline-first schedule, the maximum lateness, and the lower bound (for that lateness) are again calculated, at each successor node. If the lateness of the schedule, at the child, is less than or equal to the minimum lower bound of any node, then the optimal solution has been found. If the lateness is equal to the lower bound (at a child), then this child is not further expanded, since a better schedule cannot be obtained in any successor node. If the minimum lateness among all other schedules is less than the lower bound at the child, then successor nodes are not generated, since the schedule computed at any of those nodes will be non-optimal. If, at this stage, the optimal solution has not been found, the node with the least lower bound among all nodes is selected, successor nodes are generated and the process continues.

Xu and Parnas presented the first attempt to formalize a method of pre-runtime schedules for real-time processes with arbitrary exclusion and precedence relations. Furthermore, the authors were able to devise a solution for an NP-hard problem that is applicable in most of real time applications. Although that work has proposed an algorithm which greatly reduced the time and possibility of errors, as compared when using ad-hoc methods, the authors did not present any real-world experimental results.

Shepard and Gagné [86] extended Xu and Parnas' work by proposing an implicit enumeration technique for dealing with multiprocessors. However, as pointed out in [3], the algorithm occasionally fails in finding existing feasible schedules, since it attempts to reduce schedule lateness by modifying only the schedule of the processor running the latest segment of a process. In general, if the latest process has predecessors on other processors it is possible to improve lateness by shifting these predecessors earlier in their schedules. However, this aspect was ignored by the authors.

Xu [102] presents a method for finding a feasible schedule considering a multiprocessor architecture. However, this solution is very limited since the author assumes that processors are identical, tasks are non-preemptable, can be resumed on any processor at no additional cost, and neglects the cost of intertask communication.

Abdelzaher and Shin [2] proposed an extension to Xu and Parnas' pre-run-time scheduling algorithm in order to deal with distributed real-time systems. This al-

gorithm takes into account delays and precedence relations imposed by interprocess communications. As in the Xu and Parnas work, an initial solution is computed using an EDF-based scheduling and then a branch and bound approach is employed to find an optimal solution. As the tasks are now scheduled on multiple processors and message passing is also involved, there are more ways of improving the lateness of a schedule and hence the algorithm should take into account more details than the original one proposed by Xu and Parnas.

This algorithm uses three alternatives, or branching rules, to reduce the lateness of a schedule, and three sets of child vertexes (L, M and N) are created. A set B_i is first defined, which consists of modules that execute just before the latest module and consequently affect its lateness. For the set L, the branching rule proposed by Xu and Parnas is used, i.e. the exclusion relations between two modules in B_i are replaced by precedence relations so that modules with late deadlines are scheduled after other modules which need to finish earlier. For the child vertexes in set M, the priority of a message to a module in B_i is set to the priority limit at the vertex. Further, the priority limit is decremented for all child vertexes to ensure that the relative priority of the current message remains unchanged in the future. For a child vertex, in the set N, the deadline of a remote predecessor of a module in B_i is decreased such that the lateness is equal to the schedule lateness.

Finally the child vertexes in all sets are created with the appropriate constraints, priority changes or deadline modifications. That work proves that using three branching rules an optimal schedule can be obtained for a set of real-time processes. The algorithm then calculates the lower bound of the lateness for each of the children and deletes those for which the lower bound is greater than the minimum schedule lateness observed up to then. The remaining children are added to the set of active vertexes and the parent vertex is removed. The vertex with the least lower bound is selected from among the active vertexes and child vertexes are created using the three branching rules. This process is continued until only one vertex is left for which the schedule has been computed.

The algorithm presented in [2] is very similar to the original one [105] and the major steps are the same. However, it does not consider the preemption relation used in the original algorithm but it extends the branching function by adding two more rules dealing with messages and remote predecessors. The calculation of the lower bound is also performed differently. Finally, the major contribution of that work is the integrated strategy for scheduling both tasks and messages in distributed real-time systems.

3.2 Integration Between Runtime and Pre-runtime Scheduling

There are several ways for integrating runtime with pre-runtime scheduling. One way is to consider strategies that mix hard and soft real-time systems. Thus, pre-runtime scheduling may be employed for hard real-time constraints, whereas runtime scheduling may be applied to soft real-time constraints. Another adopted strategy (e.g. [108]) is to use the pre-runtime scheduling for periodic tasks, and runtime scheduling for sporadic tasks. Such strategies are suited when pure strategies are very limited by using one of them alone. Therefore, mixed solutions may bring benefits from both approaches.

This section discusses multiple operational modes, which provide several alternative schedules, and hybrid scheduling.

3.2.1 Operational Mode Changes

Frequently, several alternative schedules may be computed pre-runtime for a given time period. Each such schedule may correspond to a partially or totally new set of tasks. A small runtime scheduler can be used to select among alternative schedules in response to internal or external events, where the changing from executing one schedule to another is referred to changing operational mode. An aircraft control system, for example, performs different tasks during take off, flight, and landing phases. By switching among a number of previously computed static schedules at runtime, a system may be able to adapt its behavior in environment changing. This solution gives some degree of *deterministic flexibility* to pre-runtime scheduling, considering that both schedules and mode changes are computed in advance.

Jahanian et al. [43] present Modechart, a language for constructing modes, which can be considered as control information imposing structure on the system operation. Modes are arranged hierarchically. Each mode is either primitive, parallel, or serial. A primitive mode has no internal structure. A parallel mode is constructed from a collection of child modes by parallel composition. The parallel relationship among several modes indicates that a system operates in all of these modes simultaneously. In comparison, a serial mode is constructed from a collection of child modes by serial composition, and the children are said to be in series. The serial relationship among several modes indicates that (when in the parent mode) the system operates in exactly one of these modes at any time. Modechart also allows at most one action to be associated with each mode. The action is executed upon mode entry.

Modechart also provides specification of transitions between modes. A transition is represented graphically by a directed edge between the two modes. A transition has three components, a source mode, a destination mode, and a condition. A transition between two modes represents a change in the control information of the system. A mode transition is an instantaneous event which takes zero time units.

Sha et al. [85] developed an algorithm to manage mode changes in the context of scheduling based on priority inheritance protocols. However, this approach views mode changes as adding, deleting, and changing of just a single task. The supported task model is limited to the use of semaphores as synchronization structure, and tasks have deadlines equal to periods. Furthermore, the algorithm is limited to single processors. Although the priority inheritance protocol has been extended to multiprocessors [77], its application to distributed systems is severely restricted, since they consider that communication delays are negligible, and they do not take into account overheads introduced in handling global semaphores.

Fohler [33] proposes mode changes in the context of distributed hard real-time systems. The implementation consists of a search algorithm based on precedence graph, which is described by an acyclic directed graph, where nodes and edges represent tasks and messages, respectively. This concept was extended to consider the modes of operations, that is, each edge of the precedence graph has associated modes. Therefore, the precedence constraints of a task τ in mode M are fulfilled, when all edges to τ in mode M are fulfilled.

The problem to be solved is finding a connection between start and goal states, which is nothing else than a search strategy. The search tree is designed as follows: (i) every node represents the decision to schedule a given set of tasks or messages at a given point in time; (ii) since nodes describe scheduling decisions, edges refer to choices of these decisions; (iii) the root node describes the initial situation of the problem (the empty schedule); (iv) the costs assigned to edges are the amount of CPU and bus slots, respectively.

The aim of this search is to find a feasible schedule, not necessarily an optimal one. This schedule corresponds to a path in the search tree, whose costs do not exceed the maximal response time and which fulfills the requirements given in the precedence graph. It traverses a search tree, with costs assigned to edges, and uses a heuristic function composed of two factors to guide the search: the costs of the path encountered so far, and an estimate of the remaining costs from the current node N to a goal node (called $h(N)$). This heuristic is used for pruning paths in the search tree. But, when considering multi mode scheduling construction, instead of traversing the precedence

graphs to search for a single schedule, the traversing is made at all graphs of all modes at the same time, trying to construct an individual schedule for each mode. If a task has to execute in more than one mode, it has to be scheduled at the same time in all modes. This enables switching between schedules at runtime without re-phasing.

Considering hard real-time systems, the deadline of a mode change has to be considered, where it is the time interval between the mode change request and the completion of the last task before the new mode is established.

3.2.2 Hybrid Scheduling

Hybrid scheduling is usually defined as a combination of runtime with pre-runtime scheduling in order to obtain the best from each of these methods.

Young and Shu [108] propose a hybrid scheduling technique, which makes use of pre-runtime scheduling for periodic tasks and runtime scheduling for sporadic tasks. The authors extend the pre-runtime Xu and Parnas' algorithm in order to generate cyclic schedules for the periodic tasks. They use a priority-based scheduling technique for runtime scheduling of sporadic tasks. In that work, the authors propose some modifications to the search strategy used in [105]. They implemented a depth-first search strategy, in such a way that the schedule that uses the first late segment helps to find a partial valid initial schedule instead of the complete initial schedule. Therefore, there is no need to reconstruct the complete schedule since rescheduling involves segments that have been scheduled before.

Wang, Mok, and Folhler [99] proposed a pre-scheduling method, which is a static scheduling without assuming a constant and completely predictable resource availability. They consider heterogeneous workloads consisting of event-driven as well as time-driven. In an effort to adapt to such changes, an integration of runtime with pre-runtime scheduling (in the paper called composition scheduling schemes) have been proposed. The time-driven workload (called *subject workload*) is scheduled statically, but it depends on a *resource supply contract*, which gives to critical time intervals the maximum of time reserved to the subject workload. This reserved time is specified by *supply constraints*, where the aggregate execution time of all jobs within the interval is upper bounded by the supply contract. The remaining time (critical time intervals) is used for the competitive workloads, where, in general, competitive workloads consist in sporadic and aperiodic tasks. At runtime, a *supply function* guarantees the contract is satisfied. Figure 3.1 shows the pre-scheduling framework.

Pre-scheduling aims to generate the execution order of jobs taking into account precedence constraints. Nevertheless, when the time interval between ready time and

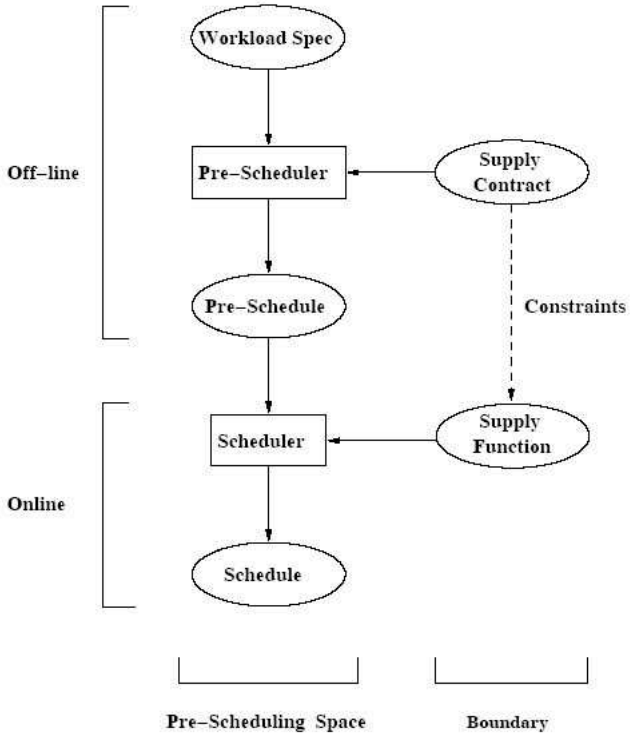

Figure 3.1: Pre-scheduling Framework

deadline (called *valid scopes*) of two jobs are overlapped, one of them is replaced by two jobs (in this case meaning a preemption). However, up to this phase, just the valid scopes are considered, not the execution time, where it is computed by a *linear programming* (LP) solver. Figure 3.2 shows the pre-scheduler (upper part) and two ways for generating schedules runtime. Black boxes in the row *supply functions*, indicate the time intervals in which the resource is not supplied to the pre-scheduling space. These time intervals are used to execute competitive workloads. Each schedule is shown as a sequence of gray boxes. Two valid schedules are generated according to two different valid supply functions, but the order of jobs defined by the pre-scheduler is always followed.

The drawbacks of this method are the following:

1. This method does not consider exclusion relations between tasks. However, the authors suggest (as future works) a combination between LP-based with search-based in order to solve this problem; and

2. That work does not present any real-world case study in order to show how their solution is applicable in an example having requirements that they propose to solve. Another problem is that the paper does not make clear if competitive workload might have hard or just soft deadlines.

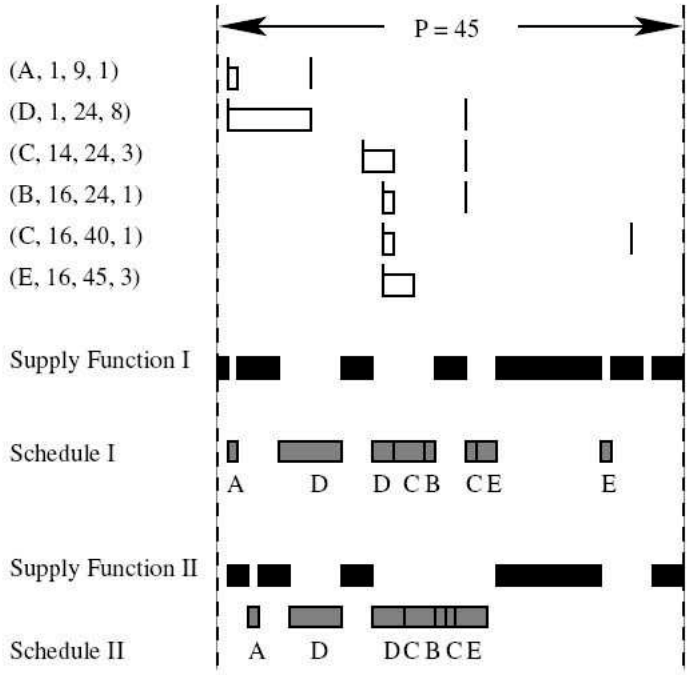

Figure 3.2: Pre-schedule and Online Generator

3.3 Petri Nets in the Scheduling Theory

Several authors also use Petri nets in scheduling theory. However, most of them are only concerned with schedulability analysis, e.g. [93, 101], that is, they are concerned with analysis whether firing sequences are schedulable or not.

Tsai et al. [93] propose the Timing Constraint Petri Net (TCPN), which is an extended Petri net by associating a minimum/maximum timing constraint with each transition and place, and a duration constraint for firing each transition. TCPNs use *weak firing rule* (it does not force an enabled transition to fire) instead of *strong firing rule* (transition is forced to fire if it remains enabled in its firing timing interval). TCPN can be more expressive, but it is certainly more difficult to use, since they propose different interpretations of timing constraints on net structures (such as synchronization and concurrency). As pointed out in Xu et.al. [101], their definitions of *earliest beginning firing time* (EBFT), and *latest fire ending time* (LFET) for week/strong firable transition are inconsistent with the meaning of the timing constraints. Xu et.al. [101] show examples of the problem and conclude stating that the complex timing constraints provide little help for modeling and analyzing real-time systems. Another problem is that Tsai et al. [93] assumes that resources are always available upon request.

Xu et al. [101] present a compositional approach to the schedulability analysis of complex real-time systems modeled using time Petri nets [64]. They propose an

alternative analysis technique, which separates the analysis of timing properties from the analysis of other non-timing behavioral properties. Therefore, the analysis can be conducted in two phases: reachability analysis without considering timing constraints, and timing analysis of task sequences. For instance, reachability may verify whether a transition sequence δ is an occurrence sequence reaching a certain marking M_n. Next, the δ is analyzed to verify whether δ is schedulable (or M_n is reachable) by means of δ with timing constraints. In that paper, an enabled transition t is said to be *schedulable* under marking M if t can be the first transition to fire, i.e., can fire before any other enabled transitions. Thus, a transition sequence $\delta = (t_1 \dots t_i \dots t_n)$ is said to be *schedulable* (or δ is a *schedule*) if all transitions in δ are schedulable in the given order. In other words, there exists markings $M_1, \dots, M_n$ such that $M_0 t_1 M_1 \dots t_i M_i \dots t_n M_n$ is a firing sequence in the underlying net, and $t_i (1 \leq i \leq n)$ is schedulable under M_{i-1}.

The main contributions of Xu et al [101] are twofold: (1) they propose an approach for determining whether a specific transition sequence is schedulable or not, calculate the time space of a schedulable task execution, or discover exactly non-schedulable transitions to help adjust timing constraints and correct design errors; and (2) a compositional approach to deal with complex task sequence, where a firing sequence is decomposed into a number of subsequences. The main restriction of [101] is that they just consider mono-processor architectures.

Bruno et. al. [17] present a schedulability analysis, using high-level Petri nets, based on PROTOB formalism (an object-oriented methodology). That work does not generate feasible schedules, but it relies on Xu and Parnas' algorithm in order to find them.

The scheduler synthesis proposed by Altisen et.al. [4] uses an extension of Petri nets called PND (*Petri Nets with Deadline*) as modeling language and TAD (*Timed Automata with Deadlines*) as semantic model. Their approach synthesizes all *dynamic runtime scheduling* satisfying a given property. In spite of their claim that using *synchronization modes* lower the complexity, they do not directly address the state explosion problem, stressed by the authors as a limitation of their approach.

3.4 Code Generation

There are several works that deal with code generation. However, code generation considering hard timing constraints is little explored.

Lin [55] proposes a software synthesis approach based on static compilation that generates ordinary C programs at compile-time without including (or generating) a

runtime scheduler. Lin considers mono-processor asynchronous process-based specifications written using a C-like programming language. The C language extension is based on CSP formalism, where it provides mechanisms not found in C for concurrency and communication. The specification is automatically translated into Petri nets by using compositions (sequential, parallel, choice, recursive) [24]. The author advocates that a key advantage for choosing Petri nets is that ordering relations across process boundaries are made explicit. Another advantage is that the C code generated can be easily re-targeted to different processors without requiring a runtime kernel. After the composition, all internal communication between nets disappear. The code is generated starting from a control-data-flow graph, which is constructed based on a traversal of an *expansion* (an acyclic Petri net fragment), but the Petri net firing rules is modified to consider the levels defined by a *pre-ordering*, such that transitions have levels (or priorities) called $\pi(t)$ in such a way that, if t_i precedes t_j, then $\pi(t_i) < \pi(t_j)$.

The main drawback of this approach is that it does not consider timing and resource constraints, which are key aspects in many embedded systems. Another problem is that this approach is based on the strong assumption that the Petri net is safe (places can only store at most one token). Safeness implies that algorithm always finishes, since it is impossible from one place to accumulate tokens, and, consequently, systems are always schedulable. Therefore, there is no need to check whether a specification is schedulable. However, this assumption may considerably limit the modeling.

Cornero et al. [23] propose a methodology for development of real-time information processing systems, which are systems that perform different kinds of functionalities, such as signal processing and control-intensive tasks. The specification is based on concurrent communicating processes with real-time constraints. Then, this specification is translated into a set of *program threads*, whose behavior generally starts with a non-deterministic operation, where its execution time is unknown at compile time, such as `wait` statement. The purpose of extracting program threads from concurrent processes is to isolate all the *uncertainties* related to the execution delay of a given program. However, by their definition of program threads, this isolation only occurs at the beginning of program threads. A program thread can be in one of four states: (i) *disabled*, indicating a state of inactivity; (ii) *enabled*, i.e. waiting for a specific event to occur; (iii) *active*, where it is ready to run; and (iv) *running*, that is, it is under execution. Using explicit commands, the designer can impose dependencies between threads.

Program threads are represented by a *constraint graph* model, such that where vertexes represent threads, and edges represent precedence relations and timing con-

straints. The consistency of the constraint graph, with respect to timing constraints, is introduced by the concept of *well-posedness*. Thus, a feasible timing constraint is well-posed if it can be satisfied for all values of the unbounded delays from non-deterministic operations. Specifically, a timing constraint is feasible, if it can be satisfied when the delays of all events (i.e. non-deterministic operations) in the constraint graph are set to zero. In the next phase on the proposed methodology, the initial constraint graphs are partitioned into disjoint *clusters of threads* (also called *thread frames*), where each cluster is triggered by a single event, in such a way that, analysis and synthesis can be readily executed. Next, static scheduling is performed for determining the relative ordering of threads in the same thread frame. This ordering is not changed anymore. Based on the imposed timing constraints and on the relative thread ordering within each frame, the time *slack* of each thread is determined (indicating the amount of time the end of a thread can be postponed). The static information is finally used at runtime by the *dynamic scheduler*, whose aim is to combine different thread frames according to runtime system evolution.

Cornero's methodology [23] can not be applied to safety time-critical systems, since it is not guaranteed that all timing constraints will be satisfied. This situation occurs because the approach considers that the arrival time of events is unknown at compile-time, where this is reflected in the specification model by associating a non-deterministic delay to event nodes. There are some situations in which the expected conditions are not satisfied, mainly due to system overloading. Moreover, schedulability analysis is not considered. Another issue concerns the assumption that events always occur just at the beginning of the threads' execution. This assumption may not be efficient for long execution time of threads, since valid orderings may be discarded because of the above approximation. Thus, this effect can only be reduced if the length of threads is limited.

Sgroi et. al. [83] proposed a method for software synthesis considering a quasi-static scheduling algorithm using a Petri net formalism. One restriction of this approach is that it is only applicable to a Petri net subclass, namely, free-choice Petri nets [25]. However, this subclass exhibits a clear distinction between concurrency and non-deterministic choices. The Petri net is partitioned in a number of tasks. Each task is generated considering one source transition with independent firing rate (representing the input from environment), which guarantees that the number of tasks is minimum. Moreover, that feature allows a runtime overhead reduction and, consequently, performance improvement.

A free-choice Petri net is considered schedulable if, for each possible choice, there

exists a cyclic finite sequence in such way that it always return to the initial marking. This is necessary to guarantee that no place will accumulate tokens arbitrarily, providing a way for estimating the amount of memory needed. If the net is schedulable, the quasi-static scheduling is generated through decomposition of the net in conflict-free components. Starting from a feasible schedule, a C code is generated by traversing the schedule and replacing transitions with the respective associated code.

This solution has some limitations. First, it does not consider timing constraints. This limitation reduces the applicability of this solution since most of embedded systems are time-sensitive. Second, they do not explicit how to deal with memory usage estimation using Petri nets. Finally, they propose that the generated tasks are to be scheduled by a real-time operating system. However, they do not take into account schedulability analysis.

Su and Hsiung [91] improved the quasi-static scheduling proposed by Sgroi et. al. [83], in the sense that they do not use free-choice Petri net, but a complex-choice Petri net. In this net, it is possible to have *confusions*, i.e., a mixing of conflict and concurrent transitions. The semi-static scheduling generation is almost the same as presented in [83], except by the fact that transition's firing may depend not only on one place. Transitions which participate on a confusion, are separated into a *complex choice set* (CCS). In order to analyze each CCS, an *exclusion table* is produced, where a transition mutually excludes another transition, if the firing of one disables the other one. Based on the exclusion table, a CCS is decomposed into two or more conflict-free sub-nets. The remaining of the algorithm is equal to the Sgroi's algorithm.

In the code generation phase, the authors added threads generation, where each transition source corresponds to a thread. Therefore, a thread is activated whenever there is an input event. The code generation algorithm for each thread visits all places and transitions, starting from the source transition. The authors use a semaphore to access the variable that controls a token, since two or more threads may be concurrently accessing the same variable.

The problems with this approach are as follows: (i) The same way as [83], they do not show how the threads are scheduled; (ii) The proposed algorithm uses semaphores in all places. However, if applications have only free-choice, there will be unnecessary overheads; (iii) Although the paper is related to code generation, they do not show any generated code; and (iv) Although mentioned that their approach could estimate the maximum memory usage, that work does not show how to obtain such memory limit.

Hsiung [42] presents a formal software synthesis based on Petri nets, mixing quasi-static scheduling (for dealing with task generation with limited memory), and dynamic

fixed-priority scheduling (for satisfying hard real-time constraints). The aim of Hsiung's work is to give a more complete vision of the software synthesis process not detaining only in the task generation, but also in the schedule of these tasks. The software is specified as a set of *time free-choice Petri nets* (TFCPN), which is a free-choice Petri nets (FCPN) extended with time. Therefore, confusions are not allowed. The time semantics is equal to the time Petri net [64]. The algorithm for quasi-static scheduling is the same as one presented in [83]. As in this scheduling each conflict-free component corresponds to a cyclic finite sequence, the execution time interval can be calculated by the sum of all EFT (earliest firing time) and LFT (latest firing time). For each TFCPN, the maximum LFT is chosen as the worst-case execution time to this TFCPN. In this way, a real-time scheduling algorithm, such as rate monotonic or deadline monotonic, may be used to schedule the TFCPN. In the code generation, a real-time process is created for each TFCPN. In each process, a task is created for each transition with independent firing rate.

The main drawbacks of this approach are: (i) the schedulability analysis is carried out manually; (ii) the limit of memory can be checked by observing the maximum number of tokens in each place. The problem is that it is assumed that each data has constant and fixed space, which is not always true; (iii) the case study presented is not a real-world example, instead it is a hypothetical example; and (iv) although applying dynamic scheduling, it is not shown how to add preemption in the proposed methodology.

Amnell et al. [5] present a framework for development of real-time embedded systems based on timed automata extended with real-time tasks, that is, a timed automata with annotated code. They describe how to compile the design model to executable programs with predictable behavior. Their solution is well suited for independent tasks, since it relies on a fixed-priority scheduling, where this policy may not reach feasible schedules when considering arbitrary intertask relations (such as precedence and exclusion relations).

3.5 Summary

This chapter summarized the main works related to scheduling and code generation. Scheduling is a topic studied for many years. Although restricting this chapter to only real-time scheduling, there are an enormous amount of works. Therefore, this chapter just presented a summary of pre-runtime scheduling methods. This chapter also shows related works with multiple operational modes and hybrid scheduling. Although there

are several works that propose software synthesis (scheduling + code generation) tools, none of them is intended for generating code for embedded hard real-time systems. So far, software synthesis for time-critical systems is a research area little explored.

Chapter 4

Petri Nets

The aim of this chapter is to introduce Petri nets and some extensions. This chapter is divided in five basic sections. First, Section 4.1 presents a brief introduction. This section depicts transition enabling and what follows after transition firing, what the elementary nets are, and the main subclasses of Petri nets. Section 4.2 deals with the main models for several situations often present in most systems, for instance, how to model parallel processing and mutual exclusion relations. Three important timing extensions are shown in Section 4.3, that is, time Petri nets, timed Petri nets, and stochastic Petri nets. In the following, Section 4.4 describes behavioral and structural properties, and presents the main methods for the analysis of such properties. The main techniques are reachability-based methods, structural methods, and reduction rules. Finally, Petri net synthesis is shown in Section 4.5 which overviews synthesis of large nets.

4.1 Introduction

Petri nets (PN) were introduced in 1962 by the PhD dissertation of Carl Adams Petri [76], at Technical University of Darmstandt, Germany. The original theory was developed as an approach to model and analyze communication systems. The simple Petri net has subsequently been adapted and extended in several directions. Many extensions to the simple Petri net model have been developed for various modeling and simulation purposes. The main such extensions are: inhibitor arcs, deterministic and stochastic timed nets, and high-level nets, such as object-oriented nets and colored nets. Nowadays, Petri net is a well-established formal description technique for concurrent systems. Several research groups in all around the world adopt such formalism in theoretical foundations as well as in practical development.

Petri net can be defined as a mathematical formalism that allows specification and verification of systems. It is possible to model systems, using Petri nets, that are: concurrent (with real parallelism or not), synchronous or asynchronous, and/or deterministic or non-deterministic.

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 [26].

A marked Place/Transition Petri net is a bipartite directed graph, usually represented by a quintuple $\mathcal{PN}= (P, T, F, W, m_0)$ 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;
- $W : F \to \mathbb{N}^+$ is a weight function for the set of arcs;
- m_0 is the initial marking.

This class of Petri net has two kinds of nodes, called *places* (P) and *transitions* (T), such that $P \cap T = \emptyset$. The set of arcs F is used to denote the places connected to a transition (and vice-versa). W is a weight function for the set of arcs. In this case, each arc is said to have *multiplicity* k, where k represents the respective weight of the arc.

A marked Petri net contains *tokens*, which reside in places, travel along arcs, and their flow through the net is regulated by transitions. 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 thesis 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 $m_i : P \to \mathbb{N}$, 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 thesis both notations are used interchangeably.

High-level Petri nets may have complex types of markings, for instance, the marks may be individualized by a color, or the mark may be a complex data structure. Examples of high-level Petri nets are: colored Petri nets, object-oriented Petri nets,

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

predicate/transition Petri nets, environment/relationship, and so on. However, high-level Petri nets are beyond the scope of this thesis.

The Petri net mathematical formalism may be graphically represented, which allows the visualization of processes, communication between them, and gaining an understanding of a particular model. Petri net graph uses circles and bars (or rectangles) to represent places and transitions, respectively. The arcs are represented by edges between the two types of nodes. The multiplicity of an arc is indicated by an integer adjacent to the arc. When the multiplicity is one, generally it is omitted in the graphical representation. Markings are represented graphically by dots.

Transitions are active components, whereas places are passive components. There are several interpretations for transitions and places. For instance, it can be adopted the concept of condition (place) and event (transition), where an event may have several pre- and post-conditions. Table 4.1, based on [70], shows other interpretations for places and transitions.

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:

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

4.1.1 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 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.

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'.

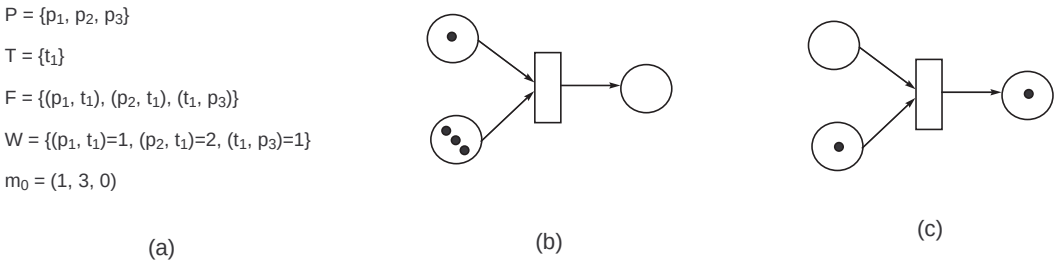

Figure 4.1: Petri net. (a) Mathematical formalism; (b) Graphical representation before firing of t_1; (c) Graphical representation after firing of t_1

Figure 4.1(a) shows a Petri net mathematical formalism for a model with three places and one transition. Figure 4.1(b) outlines its respective graphical representation, and 4.1(c) provides the same graphical representation after the firing of t_1.

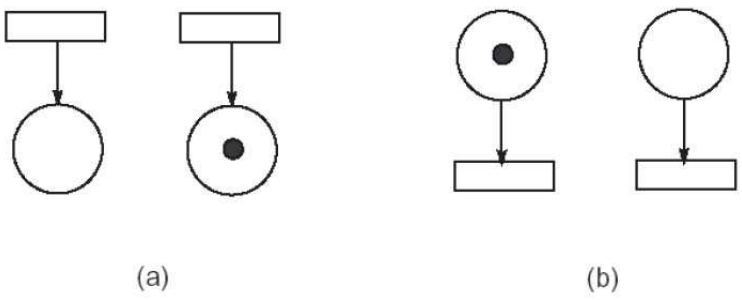

Figure 4.2: Source and sink transition before and after the firing

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. 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 4.2 shows source and sink transitions before and after the respective firing.

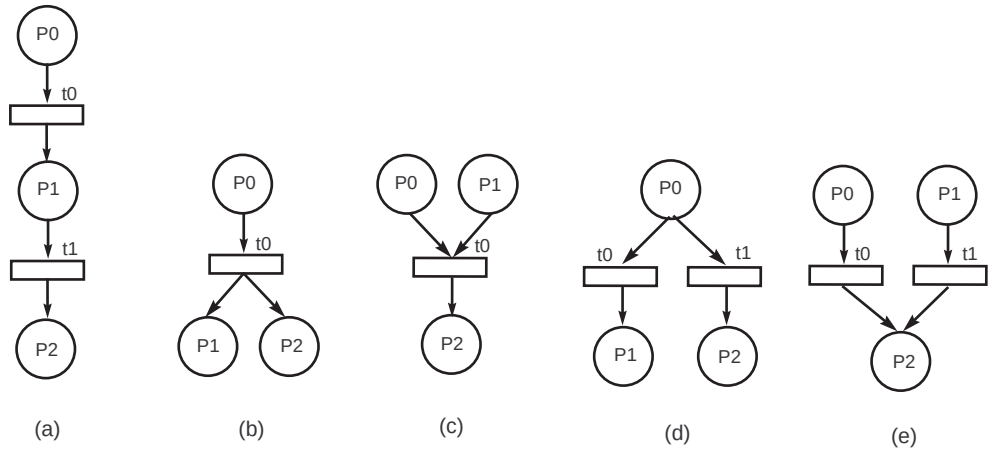

Figure 4.3: Elementary Structures

4.1.2 Elementary Nets

Elementary nets are used as building blocks in the specification of more complex applications. Figure 4.3 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 4.3(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 (see Figure 4.3(b)) allows the creation of parallel processes.

Synchronization (or Join)

Generally, concurrent activities need to synchronize with each other. This net (Figure 4.3(c)) combines two or more nets, allowing that another process continues this 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 4.3(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 4.3(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 allows the firing of another transition (not shown in the figure).

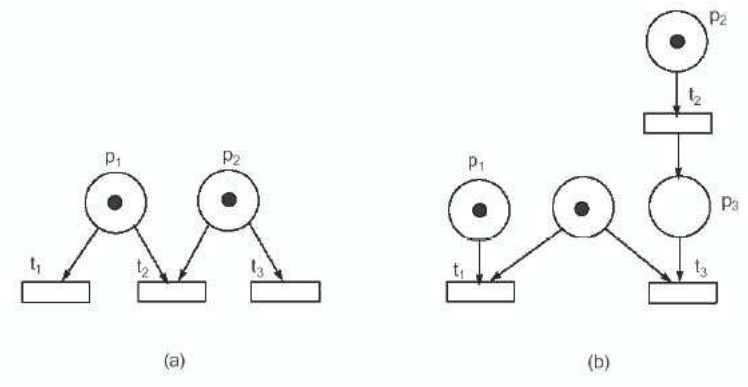

Figure 4.4: Confusions. (a) symmetric confusion; (b) asymmetric confusion

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 4.4: (a) *symmetric confusion*, where two transitions t_1 and t_3 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.

4.1.3 Petri Net Subclasses

Net subclasses is defined exclusively by introducing constraints on the structure of the nets [88]. 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 [70], let us introduce five important subclasses depicted in Figure 4.5.

State Machine

In this subclass (*state machine*-SM) (Fig. 4.5(a)) each transition has just one input and output arc, 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. 4.5(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. An important property of marked graphs is that the number of tokens in the net do not change with transition firing. It is easy to see whether the net is live or safe, and the reachability problem is decidable.

Free-Choice Petri Nets

The *free-choice* (FC)) (Fig. 4.5(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.,

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

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. 4.5(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.

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

Asymmetric Choice (or Simple Net)

An *asymmetric choice* (AC) (Fig. 4.5(e)) is a Petri net such that

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

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 [88].

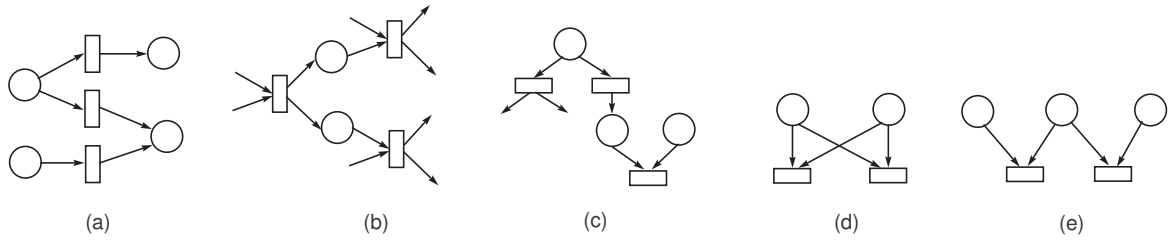

Figure 4.5: Five fundamental Petri net subclasses

In summary, SMs admit no synchronization, MGs admit no conflict, FCs admit no confusion, and ACs allow asymmetric confusion (Fig. 4.4(b)), but disallow symmetric confusion (Fig. 4.4(a)) [70].

4.2 Modeling with Petri Nets

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

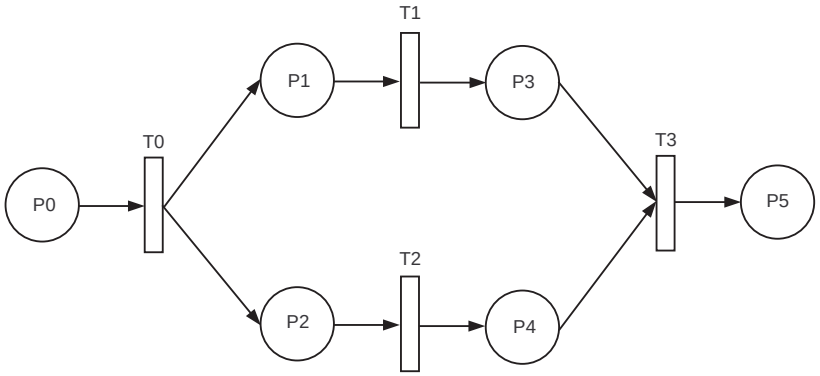

Figure 4.6: Transitions T_1 and T_2 represents parallel activities

4.2.1 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 4.6 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.

4.2.2 Mutual Exclusion

Some applications require sharing of resources and/or data. 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 4.7 shows an example of a machine accessed in a mutual exclusive way.

4.2.3 Dataflow Computation

Petri nets can be used to represent not only the control-flow but also the data-flow. The net shown in Figure 4.8 is a Petri net representation of a dataflow computation. A dataflow is characterized by the concurrent instruction execution (or transitions firing) as soon as the operands (pre-conditions) are available. In the Petri net representation,

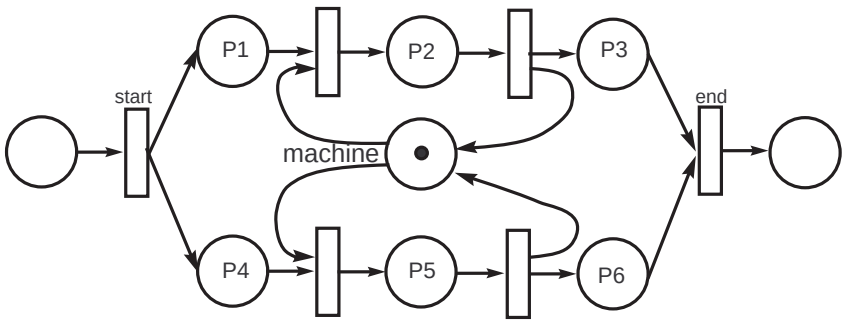

Figure 4.7: Mutual Exclusion

tokens may denote values of current data as well as the availability of data. The instructions are represented by transitions.

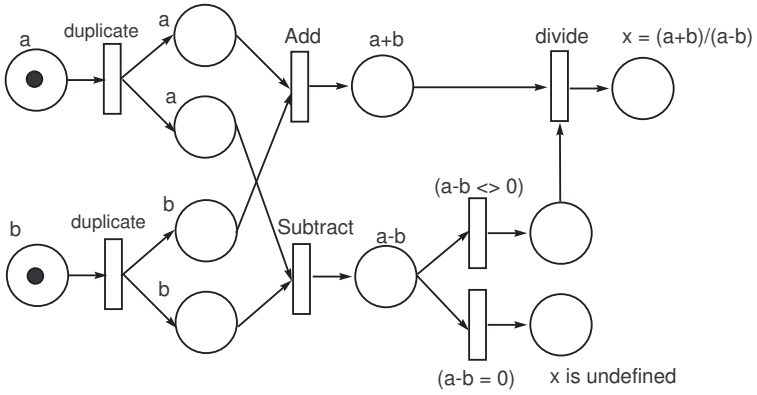

Figure 4.8: Dataflow Computation

4.2.4 Pipelined Systems

A very important point to be considered in the dataflow computation is the pipelined model. In this architecture, data is processed by successive stages, in such a way that, each stage is busy in each operation cycle.

Each stage manipulates only the data supplied in its input and provides this data transformed to the next stage. The dataflow is well-established, since the communication is restricted to neighbor stages. Furthermore, a pipelined system is composed by a number of stages which can be in simultaneous execution. Figure 4.9 shows a pipelined system consisting of two functional units.

Transitions are explained as follows: t_0 reads the input of stage A; t_1 represents the start of operations on stage A; t_2 represents the writing of results in the output of stage A; t_3 means the start of data transfer between stages; and t_4 represents the final

operation of data transfer between stages A and B.

The same way, places are as follows: p_0 is the system input; p_3 means stage A input is empty or not (marked or not); p_4 means stage A output is empty; p_1 means that the input is on stage A; p_2 means that the functional unit (stage A) is in operation; p_5 implies that stage A output has a data; p_6 finishes the operations related to data transfer from stage A to stage B. Similarly, the same operations are valid for stage B.

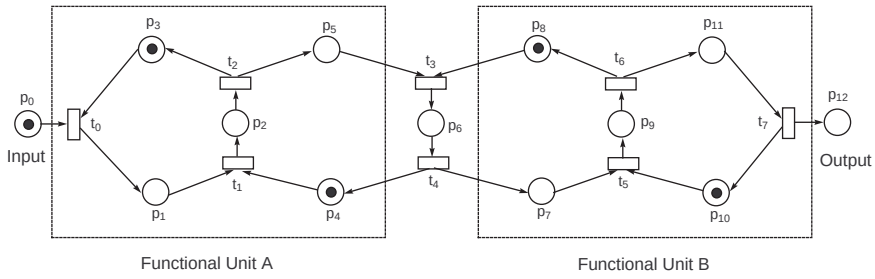

Figure 4.9: Pipeline of two stages

4.2.5 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)).

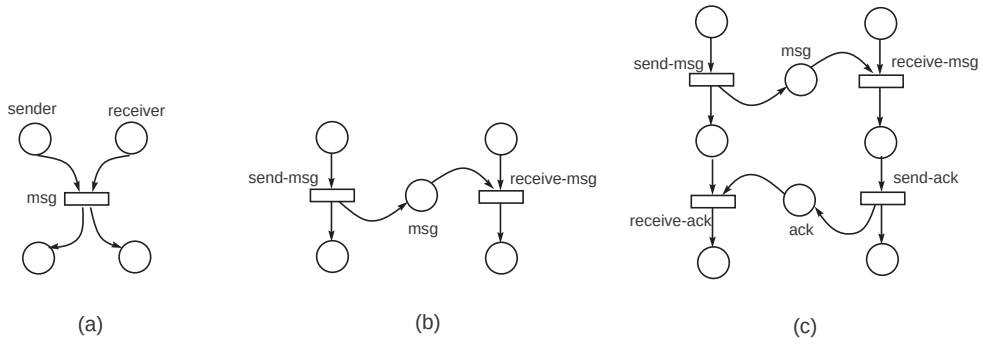

Figure 4.10: Communication Protocols

4.2.6 Producer-Consumer

Many other classical problems in concurrency have been modeled by Petri nets, among them the dining philosophers, readers-writers and producer-consumer problems. 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 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.

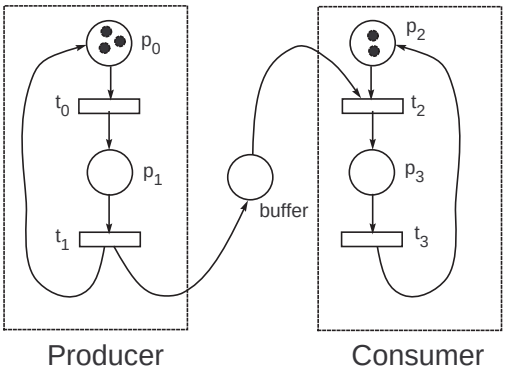

Figure 4.11: Producer/Consumer

4.3 Time Extensions

The original definition of Petri nets does not include any notion of time and is aimed to model only the logical behavior of systems by describing the causal relations between events. The introduction of timing specification is essential if we want to use this class of model to consider, for instance, performance, scheduling, real-time control, and so on. It is worth noting that some applications have requirements not only related to logical correctness, but also associated to the time at which results are produced. In several areas, such as hardware and architecture computer design, communication protocols, and software system analysis, timing is essential for assuring that systems are correct.

The time introduction in Petri nets should not modify the basic structure of the

underlying untimed model. It must however add mechanisms for computation of performance metrics. 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: a first instantaneous phase in which an enabled transition removes tokens from its input places, then a timed phase in which the transitions are working, and a final instantaneous phase in which tokens are deposited into the output places. Such time information is called *duration*;
- Atomic firing: Tokens remain in input places during the whole transition delay; 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.

Memory policies represent the way transitions are affected whenever a transition fires [62]:

- Resampling: the timers of all transitions are discarded (restart mechanism). New values of timers are reset for all enabled transitions at a new marking;
- Enabling memory: transitions that are still enabled in the new marking keeps the value of the timer; transitions that are not enabled have their timers reseted. The enabling time of a transition is measured since the last instant of time it became enabled;
- Age memory: the timer value is kept, even if the transition is not enabled in the new marking. Whenever this transition becomes enabled, the counting is resumed from the kept value.

Lets take a look at three important classes of timed extensions: time Petri nets, timed Petri nets, and stochastic Petri nets. However, many other time extensions have been proposed and adopted by the research community.

4.3.1 Time Petri Nets

Time Petri net [64] is defined by (PN, I), where PN is an underlying Petri net, and I is a time interval expressing timing constraints, where $I_i = (EFT_i, LFT_i)$ associated

with each transition t_i. EFT stands for *earliest firing time* and LFT stands for *latest firing time*. This non-negative interval express 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 weakest 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 *weakest 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.

4.3.2 Timed Petri Nets

A timed Petri net is a pair (PN, D) [78], where PN is a conventional Petri Net and D is a function which associates a non-negative real number to each transition t_i, known as the firing duration of transition t_i. Transitions in a timed Petri net are enabled by a marking m_i the same way as a conventional Petri Net. The firing of an enabled transition is in accordance with the *three-phase firing* presented above. Transitions in a timed Petri net must fire as soon as they are enabled. The memory policy of this model is enabling memory.

4.3.3 Stochastic Petri Nets

If performance evaluation is to be considered in system design, stochastic analysis must be used since the system behavior is not yet completely known in advance. State space of the stochastic process underlying a stochastic Petri net is defined by its tangible markings, and so, generation of the tangible reachability graph is a prerequisite for the quantitative analysis of a stochastic Petri net.

A Stochastic Petri net (SPN) [68] is a Petri net where each transition is associated with an exponential distributed random variable that express the delay from the enabling to the firing of the transition. In a case where several transitions are simultaneously enabled, the transition that has the shortest delay will fire first. Due to the memoryless property of the exponential distribution of firing delays, it has been shown that the reachability graph of a bounded SPN is isomorphic to a finite Markov Chain, that is, the Markov Chain of a SPN can be obtained from the reachability graph of the underlying Petri net.

The SPN models are complex to be analyzed due to the very large number of reachable markings and also because the presence in one SPN model of activities that take place on a much faster time scale than the one relating to the events that are critical to the overall performance. The result is linear equations that are difficult to solve with an acceptable degree of accuracy by means of the usual numerical techniques.

In 1984, Marsan et al. [60] introduced Generalized Petri Nets (GSPN). The GSPN comprises two type of transitions: (i) *timed transitions* (drawn as white boxes), where they have exponentially distributed firing delays, and have a weight associated which represents the parameter of the negative exponential probability density function of the transition firing delay; and (ii) *immediate transitions* (drawn as thin black bars), which are transitions with zero firing time, or zero delay, with priority over timed transitions and the weight is used for the computation of firing probabilities. GSPNs also permit the use of inhibitor arcs, priority functions, and random switches. *Inhibitor arcs* are used to prevent transitions from firing when certain conditions are true. *Priority functions* is defined for the marking in which both timed and immediate transitions are enabled. Immediate transitions have higher priority. *Random switch* is used to resolve conflicts between two or more immediate transitions. The random switch is basically a discrete probability distribution. These additional modeling capabilities do not destroy the equivalence with Markov chains. The authors showed that a stochastic Petri net with exponentially distributed firing times is isomorphic to a discrete space continuous-time Markov chain.

Deterministic and stochastic Petri nets (DSPN) [61] were introduced by Marsan and Chiola as an extension to GSPN. DSPN allows association of timed transition with either deterministic or an exponentially distributed firing delay. DSPN allows the association of a timed transition either with a deterministic or an exponentially distributed firing delay. Therefore, system features such as propagation delay, time-out and processor rebooting times, which are associated with constant delay can be represented in a DSPN by deterministic timed transitions.

4.4 Properties Analysis

Petri nets are a powerful description technique, which is able to model a large variety of problems present in most concurrent and real-time systems. However, Petri nets are not only restricted to design modeling, but also, for analyzing or verifying properties of the modeled system.

Four kinds of techniques have been used to test the properties of a given system: verification, proof, analysis, and validation. Verification is a deterministic algorithm for checking if a system have or not a given property. A proof technique is a formal argument for asserting or not if a model has or has not a given property. Analysis techniques provide a variety of information about properties. They are used as a basis for further arguments, generally for verification algorithms. Validation methods are a process for checking system reliability, that is, if the system has the expected behavior. Validation can be applied in real system (by testing) or models (by simulation).

Two types of properties have been considered in a Petri net model: behavioral and structural properties. Behavioral properties are those which depend on the initial marking. Structural properties, on the other hand, are those that are marking-independent.

This section presents some behavioral and structural properties.

4.4.1 Behavioral Properties

This section, based on [70], describes some behavioral properties, since such properties are very important when analyzing a given system.

Reachability

Reachability is a fundamental basis for studying the dynamics of any system. Given a Petri net PN and initial marking m_0, one would like to know if a specific marking m_i may be reached from the initial marking m_0. A marking m_i is said to be reachable from marking m_0, 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 $L(m_0)$.

Lipton [57] has shown 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.

The absence of *deadlock* is closely connected to the *liveness* concept. Actually, if a system is deadlock-free, it does not mean that system is live, although if a system is live it is certainly deadlock-free. Examples of deadlock-free non-live Petri nets are those that have not any dead state but they have at least one transition which is never fired.

Liveness is an ideal property for many real systems. However, it is very strong and too costly to verify. 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 $L(m_0)$, it is a dead transition.
- 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 and speed-independent asynchronous circuits.

Fairness

This concept is closely related to starvation. Such property is related to the possibility of a given part of a computational system to get the control (or be executed) forever. This is an important system property which should be carefully looked at in the qualitative analysis phase.

Petri net literature presents many different points of view about the fairness concept (e.g.[15, 95, 70]). 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.

Conservation

In a Petri net context, tokens can be used for resource modeling. In a conservative net resources are neither created nor destroyed. Nets in which any transition firing does not change the number of tokens within the net are said to be *strictly conservative*. For each transition in such nets, the number of input places are equal to the number of output places, since their structure does not allow for changes in the number of tokens.

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.

4.4.2 Structural Properties

Structural properties are independent of the initial marking. These properties are only dependent on the topological structure of the net. Therefore, such properties can often be described in terms of incidence matrix-based analysis methods. Thus, in this section, the nets are assumed to be pure. This section will not present details about any property. The interested reader is referred to [59, 70] for further information.

The main structural properties are:

- **Structural Boundedness**. As was seen in previous section, 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.

- **Structural Liveness**. A Petri net is structurally live if it is live for at least one initial marking.

- **Structural Conservativeness**. A particular kind of structural boundedness is called *structural conservativeness*. Such nets provide a constant weighted sum of tokens for any reachable marking when considering any initial marking.

- **Repetitiveness**. A net is classified as *repetitive* if there is an initial marking m_0 and an enabled firing sequence 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*.

4.4.3 Analysis Methods

Petri net analysis methods may be divided into three classes. The first method is graph-based and it builds on the reachability graph (reachability tree). The reachability graph is initial marking dependent and so it is used to analyze behavioral properties. The main problem in using a reachability tree is the high computational complexity, even if some interesting techniques are used [96], such as reduced graphs, graph symmetries, symbolic graph, etc.

The second method is based on state equations. The main advantage of this method, over the reachability graph, is the existence of simple linear algebraic equations that aid in determining net properties. However, it gives only necessary or sufficient conditions to the analysis of properties when it is applied to general Petri nets.

The third method is based on reduction laws. This method provides a set of transformation rules which reduces the size of models while preserves system's properties. However, it is possible that, for a given system and some set of rules, the reduction can not be completed.

Reachability Based Methods

The reachability tree (reachability graph) is a graphical representation of the reachable marking set ($R(m_0)$) for a given Petri net PN. If it is possible to compute all reachable markings, and their reachability relations, almost all qualitative behavioral properties could be analyzed.

The procedure to build this graph works in the following way: consider a Petri net and a given initial marking. The initial marking is represented as a node. Considering this marking, if two transitions (t_i and t_j) are enabled and by firing each transition, one will reach two new markings. These new markings are represented as new nodes and the arcs between markings are labeled by each transition fired. If this procedure is repeated

over and over, every reachable marking will eventually be produced. Moreover, consider a net in which a pair of transitions (t_i and t_j) is concurrent. In the reachability graph, t_i and t_j seems to be conflicting. This situation shows that concurrency and mutual exclusion on firings cannot be studied on the reachability tree alone.

A major problem of this approach arises with the analysis of systems in which the number of reachable markings is infinite (unbounded systems). Due to the infinite number of markings, such systems are not easily represented by enumeration. Hence, in order to deal with such systems, a finite representation of the reachable graph has been proposed. This graphical representation is called a coverability tree (coverability graph) [75].

To keep this tree finite, we introduce a special symbol ω, 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:

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".

An optimized version of the algorithm given above has been described in [32]. Although this approach allows for deciding about important properties such as coverability and boundedness, it does not permit one to deal with reachability, liveness and reversibility [75]. Of course if the symbol ω is absent from the tree, this tree is the reachability tree.

Structural Methods

The dynamic behavior studied in many systems in engineering can be described by differential equations or algebraic equations. The advantage over the coverability-tree analysis is the existence of simple linear-algebraic equations that aid in determining Petri net properties. It would be interesting if it was possible to model and analyze completely the dynamic behavior of Petri nets by equations. However, the non-deterministic nature inherent to Petri net models and the constraints of the solution as non-negative integers make the use of such an approach somewhat limited. The behavior of net models are non-linear, but the so-called state equation represents an interesting linear relaxation. Nevertheless, the state equation may provide *spurious solutions* [22], that is, a marking m^s that results from the state equation, but do not belong the reachability set ($m^s \notin R(m_0)$).

Because of spurious solutions, this approach usually leads to semi-decision algorithms, or rather, it only provides necessary or sufficient conditions for the analysis of such behavioral properties as reachability, boundedness, liveness and reversibility. Thus, for certain properties analysis it permits a fast diagnosis without enumeration. Spurious solutions can be removed using some other approaches, for instance the inclusion of *implicit places* [22]. A place is defined as implicit if it can be removed without changing the behavior of the net. Thus, the addition of implicit places generates a new model with identical behavior. The problem consists in where to insert the implicit places.

Whenever matrix equations are discussed, it is assumed that a Petri net is *pure* or is made pure by adding a dummy pair of a transition and a place.

Incidence Matrix

For a Petri net PN with n transitions and m places, the incidence matrix $A = [a_{ij}]$ is a $m \times n$ matrix of integers and its typical entry is given by: $a_{ij} = [a_{ij}^+ - a_{ij}^-]$, where $a_{ij}^+ = w(i,j)$ is the weight of the arc from transition i to its output place j, and $a_{ij}^- = w(j,i)$ is the weight of the arc to transition i from its input place j. Thus, a_{ij}^-, a_{ij}^+, and a_{ij} represent the number of tokens removed, added, and changed in place j when transition i fires once.

For instance, the incidence matrix A of the net in Figure 4.12 is as follows:

$$\begin{matrix} a & b & c & d & e & f \end{matrix}$$

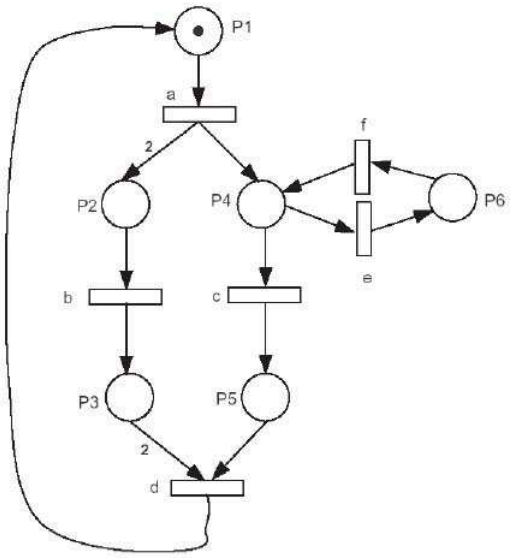

Figure 4.12: A Simple Petri net

$$A = \begin{array}{c|cccccc} p_1 & -1 & 0 & 0 & 1 & 0 & 0 \\ p_2 & 2 & -1 & 0 & 0 & 0 & 0 \\ p_3 & 0 & 1 & 0 & -2 & 0 & 0 \\ p_4 & 1 & 0 & -1 & 0 & -1 & 1 \\ p_5 & 0 & 0 & 1 & -1 & 0 & 0 \\ p_6 & 0 & 0 & 0 & 0 & 1 & -1 \end{array}$$

State Equation

A marking M_k is written as a $m \times 1$ column vector. The jth entry of M_k denotes the number of tokens in place j immediately after the kth firing in some firing sequence. The kth firing, u_k, is an $n \times 1$ column vector of $n-1$ 0's and one nonzero entry, a 1 in the ith position indicating that transition i fires at the kth firing. Since the ith row of the incidence matrix A denotes the change of the marking as the result of firing a transition i, the following state equation for a Petri net can be written [69]:

$$M_k = M_{k-1} + A^T u_k, \quad k = 1, 2, \ldots \tag{4.1}$$

Necessary Reachability Condition

Suppose that a destination marking M_d is reachable from M_0 through a firing sequence $\{u_1, u_2, \ldots, u_d\}$. Writing the state equation for $i = 1, 2, \ldots, d$ and summing them, it is obtained

$$M_d = M_0 + A^T u \tag{4.2}$$

which can be rewritten as

$$A^T u = \Delta M \tag{4.3}$$

where $\Delta M = M_d - M_0$ and $u = \sum_{k=1}^{d} u_k$. Here u is an $n \times 1$ column vector of nonnegative integers and is called the *firing count vector*. The ith entry of u denotes the number of times that transition i must fire to transform M_0 to M_d.

In fact, if a marking M_d is reachable from another initial marking M_0, then u is a vector of non-negative integers. Moreover, the converse is not necessarily true. If the state equation results in nonnegative integer solution u, M_d may or may not be reachable from M_0. Hence, this is a necessary but not sufficient condition for reachability. However, if no solution is found, then the desired marking is not reachable. This is one of the drawbacks of this method.

Invariants

An *P-invariant* or *S-invariant* is an $(m \times 1)$ nonnegative integer vector x satisfying:

$$x^T A = 0. \tag{4.4}$$

Combining 4.2 and 4.4 yields

$$x^T M_d = x^T M_0. \tag{4.5}$$

This equation implies that the total number of initial tokens in M_0 weighted by the P-invariant, is constant.

Similarly, a *T-invariant* is an $(n \times 1)$ nonnegative integer vector y satisfying:

$$Ay = 0. \tag{4.6}$$

Combining 4.2 and 4.6 yields

$$M_d = M_0, \tag{4.7}$$

with $y = u$. This implies that if the firing count vector is identical to a T-invariant, then the final marking is equal to the initial marking.

Traps and Siphons

In order to help in verifying properties, such as deadlock and mutual exclusion, two very useful set of places are considered: traps and siphons. Traps and siphons are a new kind of invariants. Differently from those associated with flows, the invariant laws associated with traps and siphons do not hold in every marking. However, once they become true they remain true forever.

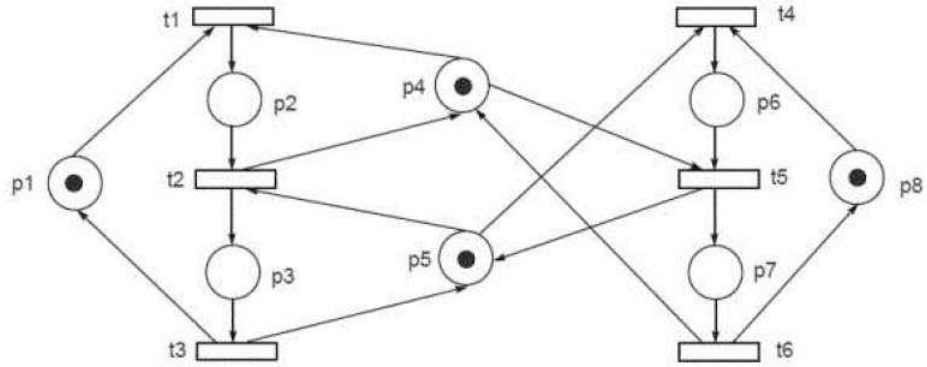

Figure 4.13: A net for illustrating traps and siphons

A subset S of places such that $S\bullet \subseteq \bullet S$ is called a trap. A trap is a set of places such that any output transition of S is also an input transition of S. So, once a place in a trap has a token, there will always be a token in at least one of the places in the trap. Hence, a trap having at least one token can never lose all of its tokens.

A subset S of places such that $\bullet S \subseteq S\bullet$ is called a siphon. A siphon is a set of places such that any input transition of S is also an output transition of S. So, once all places in a siphon have no token, there will never be a token in any place in the siphon. Hence, a siphon having lost all of its tokens can never obtain a token again.

Suppose the net of Figure 4.13. $S_1 = \{p_3, p_4, p_5, p_7\}$ is a siphon since $\bullet S_1 = S_1\bullet = \{t_2, t_3, t_5, t_6\}$. $S_2 = \{p_1, p_2, p_3\}$ is both a siphon and trap (or a P-invariant).

Simple Reduction Rules

Another very common and useful technique for qualitative analysis is the transformation based approach. Analysis of properties in large dimension nets is not trivial. Therefore, the availability of methods that allow for transforming models while preserving system properties has been studied. Normally, these transformations are reductions that are applied to the models in order to obtain smaller models preserving qualitative properties of the original ones. The reduction techniques are based on transformations of the original net into a more abstract model in such a way that properties such as liveness, boundedness and safeness are preserved in the models obtained by these reductions.

This section only presents simple reduction rules. For more detailed information on this topic, readers may refer to [14, 70]. The following rules transform the nets by applying fusion of places and transitions, and by elimination of loops. It is not difficult to see that the following six operations preserve the properties of liveness, safeness,

and boundedness:

1. Serial Places Fusion.
2. Serial Transitions Fusion.
3. Parallel Places Fusion.
4. Parallel Transitions Fusion.
5. Self-Loop Places Elimination.
6. Self-Loop Transition Elimination.

Figure 4.14 depicts these six transformations (reduction rules).

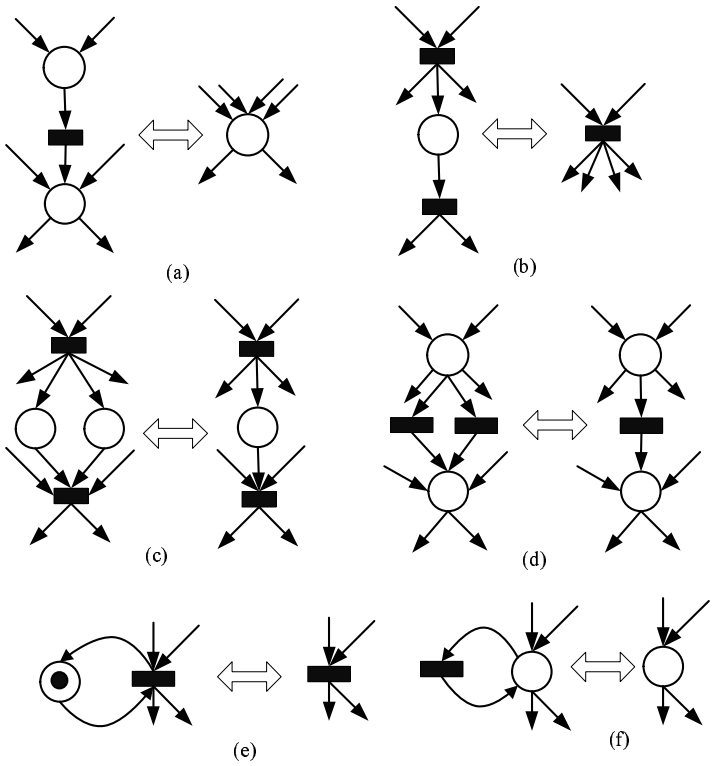

Figure 4.14: Six transformations preserving properties

4.5 Petri Net Synthesis

The aim of this section is to present an overview of Petri net synthesis including bottom-up, top-down, and hybrid techniques. Each sub-section also discusses the effect of the technique on net preservation of properties, such as liveness, boundedness, and

reversibility. This section is related to state-space generation methods and it is based on [27].

Petri nets have been applied for modeling several kinds of concurrent systems. However, problems arise when the system to be modeled is complex, yielding very large models, which are difficult to analyze. There are two general approaches for Petri net modeling. One is to model the system using a systematic procedure, and follow that by analysing if this model has the desired properties. However, the number of states may make the analysis practically impossible. In order to address this problem, transformation methods which reduce the size of the net while maintaining properties of interest have been developed. Thus, analysis can be performed on the reduced net. However, this method may not be sufficient. For instance, reductions are not so efficient for systems that have many shared resources. An alternative approach is to develop a systematic modeling method which guarantees the design properties. These synthesis methods may eliminate the need for analysis and avoid state space explosion problem.

4.5.1 Bottom-up Synthesis

The use of bottom-up or modular compositions methods is commonly used in methodologies for system design. Usually, this method involves the specification of subsystems (or modules) and some systematic procedure for combining such modules into an integrated system. These subsystems are usually very simple and easy to verify. Some interactions are represented by common places, transitions or paths in the individual subsystems. In each synthesis step, these interactions are considered, and the corresponding subsystems are combined through merging these places and/or transitions into a larger subsystem. Analysis of the combined net is usually performed immediately after each synthesis step, so when the final stage is reached, the analysis is greatly simplified. In this case, at the end of the synthesis steps, the final system and some of its important properties are obtained.

The first initiative in bottom-up techniques was proposed by Agerwala and Choed-Amphai [6]. They proposed a systematic bottom-up approach for synthesizing concurrent systems modeled by Petri nets. They suggest that synthesis can start with basic nets (or structures), which can be easily verified. At each synthesis step, subnets can be merged in such a way that a set of places, say P_δ, is merged into a new place. This is called 1-way merge. Figure 4.15 shows the net obtained by merging $P_\delta = \{p_3, p_6\}$ to a place named p'. Argewala and Choed-Amphai have provided a theorem which states that after every 1-way merge, the P-invariants of the resultant net can be known from the P-invariants of the subnets. In this case, if places in $P_\delta \subseteq PI$,

where PI is a P-invariant, the merged place p' belong to a new P-invariant PI', such that $PI' = (PI - P_\delta) \cup \{p'\}$.

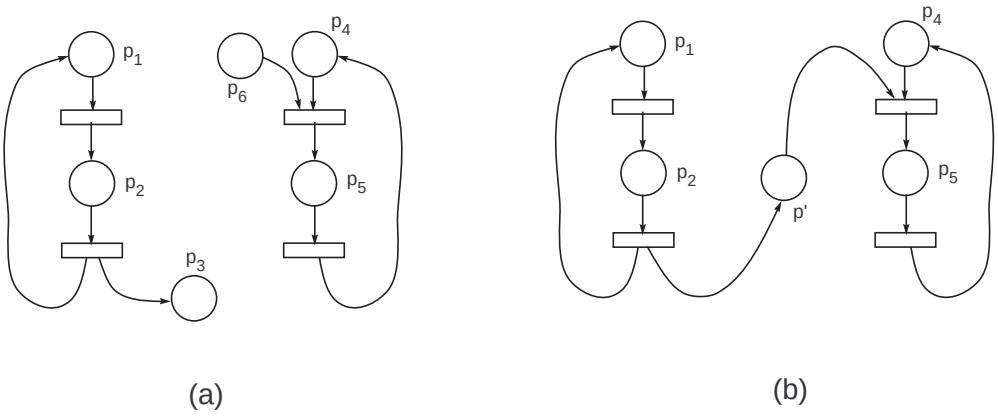

Figure 4.15: An example of 1-way merge

Narahari and Viswanadham [71] extended the work of Agerwala and Choed-Amphai in the sense that they allow places to be merged in more than one way at each synthesis step, that is, more than one set of places can be merged at one step, and the properties of the resultant net can be obtained. However, the basic principles are the same.

4.5.2 Top-down Synthesis

Top-down synthesis usually begins with an aggregate model of the system and neglects low-level detail. Then, refinement is done in a stepwise manner to incorporate more detail in the model. There are two commonly used schemes for refinements: expanding places and expanding transitions. The refinements continue until the level of detail satisfies the specification of the system. Top-down methods have the advantage of viewing the system globally from the beginning to the end of the synthesis. In addition, many researchers (e.g. Valette [94]) have made efforts to provide methods that guarantee that each synthesis step does not lose important properties of the system so that final analysis will not be necessary.

4.5.3 Hybrid Synthesis

Most systems are characterized by a high degree of concurrency, choice and shared resources. Problems arise when the complexity of a real-world system leads to a large Petri net which has many places and transitions. One way to construct such a large net is using bottom-up methods and merging subnets. However, it may be practically impossible to analyze it using reachability graph or invariant methods. Alternatively,

top-down methods are powerful when faced with a complex system. However, when confronted with detailed shared resources, the analysis problem again becomes practically impossible.

For overcoming limitations on bottom-up and top-down methods, hybrid synthesis has been proposed. For a formal presentation see [109, 110]. In this method, particular attention is given for dealing with shared resource in such a way that properties, such as liveness, boundedness and reversibility are preserved.

This design process is divided in two main phases: (a) the top-down phase where designers start with a first-level Petri net description, and use stepwise refinement to include more details up to the desired level is achieved; and (b) the bottom-up phase where the resource (in this case, places) are added to the net. In this way, the complexity of the detailed problem is reduced.

In order to avoid the qualitative analysis for a complex system, this method includes a set of mutual exclusion structures which are used in the proposed synthesis procedure. It is worth noting that resources may be divided in two kinds: resources whose number is either fixed or variable at design time. The number of the second kind of resources should be determined such that the system is neither deadlocked nor starved. This is done by finding the appropriate number of initial tokens in these resource places. Therefore, both net structure and initial marking are designed so that the desirable qualitative properties of the final Petri net are guaranteed.

DiCesare and Jeng [27] propose a complete hybrid synthesis procedure. Following this procedure, a bounded, live and reversible Petri net model is synthesized. However, the details about such procedure is beyond the scope of this section.

4.6 Summary

Petri nets are widespread used, since it provides a mathematical formalism, a graphical representation, simulation tools, and techniques for supporting specification, analysis, design, and code generation. This chapter introduced several concepts related to this subject. The expressiveness in modeling was highlighted by describing several classical problems and their respective Petri net models. Among such models it can be mentioned: parallel processing, mutual exclusion, communication protocols, pipelined systems, and dataflow computation. Special attention was given to properties and the main methods for the analysis of such properties. Finally, Petri net synthesis was described as a way for dealing with synthesis of large nets.

Chapter 5

Modeling Embedded Hard Real-Time Systems

This chapter describes the method for modeling embedded hard real-time systems. It is composed by four sections: formal model, specification, modeling the specification, and analysis and verification of the modeling.

The formal model syntax is given by a time Petri net [64], which is a Petri net extended with time, and its semantics is given by a timed labeled transition system.

Usually, the specification is composed by a set of tasks and their inter-relations, where such tasks are executed in one or more processors. This specification of tasks is divided into constraints specification, and behavioral specification. The specification of constraints comprises: (i) timing constraints, perhaps including inter-task communication time; (ii) inter-tasks relations, such as precedence and exclusion relations; (iii) scheduling method (preemptive, non-preemptive, or defined subtasks); and (iv) allocation of tasks to processors. The specification of behaviors are composed by: (i) source code of tasks; and (ii) communication pattern, whether adopted a multi-processor architecture.

Modeling is defined as a process of creating a representation of the objects of the specification. Usually, modeling is a simplified view of the system and just contains the characteristics of interest. In this work, the proposed modeling adopts a formal method to describe systems with timing constraints.

The model is used not only in representing the given specification, but also, for analysis, and verification of properties. The most important system property to be verified is its schedulability. However, this model has some interesting properties, such as, boundedness and deadlock-freedom. Other properties of interest are verified by using a model checking technique.

5.1 Proposed Formal Model

This section is divided into two subsections. The first one defines the computational model that enforce timing constraints. It defines time Petri nets, enabled transition set, implicit clocks for each enabled transition, states in a time Petri net, fireable transition set and its respective firing domain for each fireable transition, generation of new reachable states, and timed labeled transition systems. Another definition is on the feasible firing schedule, since one of the aim of this thesis is to find a schedule that satisfies all constraints. Later, time Petri net is extended in order to add code and priorities.

The second subsection extends the first one in order to define a computational model to deal with timing and energy constraints. Some previous definitions are redefined to precisely represent both constraints.

5.1.1 Computational Model for Timing Constraints

Definition 5.1 (Time Petri Net) *A time Petri net (TPN) is a bipartite directed graph represented by a tuple* $\mathcal{P} = (\mathcal{PN}, I)$*, where* $\mathcal{PN}$ *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* $EFT(t) \leq LFT(t)$*.*

A Petri net is defined in Section 4.1. This definition is an extension to the Petri nets concepts in order to add timing constraints. This extension is performed by introducing the static firing interval $I(t)$ associated with each transition $t \in T$. Therefore, I 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, on the other hand, denotes the maximum time during which the respective transition can be enabled without being fired.

Time can be modeled as either discrete or continuous. However, considering that in computers the time is not really continuous, since computers are always synchronized by a clock, this thesis adopts only the discrete case. Moreover, this definition is not a important issue, since discrete models may be acceptable depending on the time granularity. Thus, the definition of I implies in discrete time semantics.

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

- $P = \{p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$;
- $T = \{t_0, t_1, t_2, t_3, t_4\}$;

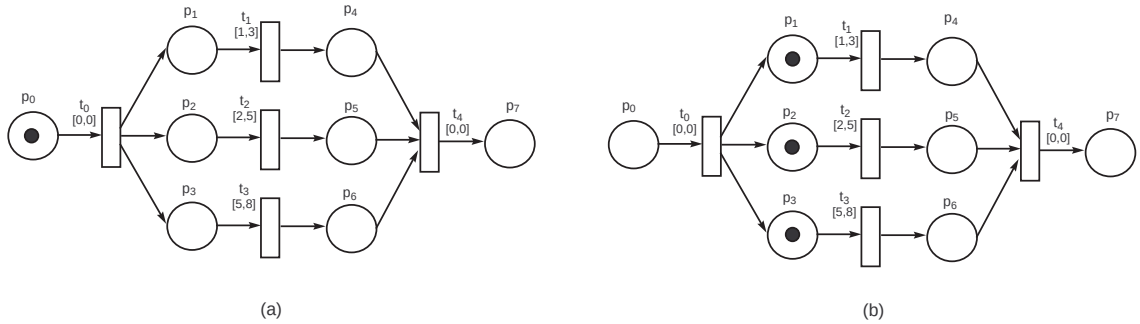

Figure 5.1: A Simple Example of Time Petri Net: (a) initial marking; (b) new marking after firing if t_0

- $F = \{(p_0, t_0),\ (t_0, p_1),\ (t_0, p_2),\ (t_0, p_3),\ (p_1, t_1),\ (p_2, t_2),\ (p_3, t_3),\ (t_1, p_4),\ (t_2, p_5),\ (t_3, p_6),\ (p_4, t_4),\ (p_5, t_4),\ (p_6, t_4),\ (t_4, p_7)\}$;
- $W(x, y) = 1,\ \forall (x, y) \in F$;
- $m_0(p_0) = 1,\ m_0(p_i) = 0,\ 1 \leq i \leq 7$; and
- $I = \{(0, 0), (1, 3), (2, 5), (5, 8), (0, 0)\}$.

The firing interval I of some transitions may be equal to zero, which means that these firings are instantaneous; all such transitions are called *immediate*, while the others are called *timed* transitions. The set of transitions (Figure 5.1) shows that transitions t_0 and t_4 are immediate transitions, while the remaining are timed transitions.

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:*

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

Definition 5.2 is more formal than the one presented at Section 4.1.1. In the net in Figure 5.1(a), only transition t_0 is enabled at the initial marking m_0.

Definition 5.3 (Clocks) *Let $\mathcal{P}$ be a time Petri net, and m_i a reachable marking. The 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.*

Each enabled transition has an implicit clock, which starts to count at the moment the transition is enabled. The clock function depends on the enabled transitions, which depends on the respective marking.

In order to not overload notations, it can be used interchangeably m_i for a marking function ($m_i : P \rightarrow \mathbb{N}$) as well as for the marking vector ($m_i \in \mathbb{N}^{|P|}$). The same notation is considered for c_i as the clock function ($c_i : ET(m_i) \rightarrow \mathbb{N}$), and as the clock vector ($c_i \in \mathbb{N}^{|ET(m_i)|}$).

In this model, in accordance to Merlin and Faber [64], tokens remain in places up to the firing of the transition. In this thesis, it is considered that this firing is instantaneous, that is, the firing takes no time. This means that, when firing a transition a new state is reached instantaneously. The firing semantics is *single server semantics with restart* [62], i.e., the firing strategy is to analyze the firing of a single transition *per* time. This semantics also implies that no transition may be fired more than once simultaneously, and its clock is reset to zero after the firing.

Considering the analysis of TPNs, it is necessary to distinguish between static and dynamic firing intervals associated with transitions. $I(t)$ is the static firing interval for transition t. The dynamic firing interval ($I_D(t) = (DLB(t), DUB(t))$), 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 does not fire. Initially, at the moment transition t becomes enabled, $I(t) = I_D(t)$.

As an example, suppose that at the time θ transition t_1 (Fig. 5.1(a)) is enabled. In this case, we have: $c(t_1) = 0$, $I_D(t_1) = [1, 3]$. At time $\theta + 1$, $c(t_1) = 1$ and $I_D(t_1) = [0, 2]$, t_1 is now fireable, but supposes that it does not fire. At time $\theta + 2$, $c(t_1) = 2$, $I_D(t_1) = [0, 1]$. If t_1 does not fire again, at time $\theta + 3$, $c(t_1) = 3$ and $I_D(t_1) = [0, 0]$, where, in this case, t_1 is *forced* to fire, since the strong firing semantics is assumed.

Definition 5.4 (States) *Let $\mathcal{P}$ be a time Petri net, M be the set of all reachable markings of $\mathcal{P}$, and C be the set of all clock vectors of $\mathcal{P}$. The set of states S of $\mathcal{P}$ is given by $S \subseteq (M \times C)$, that is, 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, a marking is not sufficient to describe a complete state of the system. Thus, the state must also include timing information. This is given by the clock function that, for each enabled transition, gives the amount of time that has elapsed since it has become enabled.

According to Definition 5.4, the state may change in two situations. The first is

related to *time elapsing*, not to transition firing. In this case, besides clock incrementation, there is no marking changing. The second is related to state change due to transition firings. In this case, changes occur in both marking and clock. Although, by definition, the first type represents a state change, this is not considered in this thesis. The following definition takes into account this aspect when defining fireable transitions.

In Figure 5.1(a), the initial state is $s_0 = ([1,0,0,0,0,0,0,0],[0])$, that is, the place m_0 is the only marked, and t_0 is the only fireable transition. Supposing that transition t_0 fires, the new reachable state is $s_1 = ([0,1,1,1,0,0,0,0],[0,0,0])$, that is, there are three places marked (p_1, p_2, and p_3), and three enabled transitions (t_1, t_2, and t_3), where all three clocks have value equal to zero. Supposing also that at time $\theta = 1$ transition t_1 fires. The new reachable state is $s_2 = ([0,0,1,1,1,0,0,0],[1,1])$, that is, there are three places marked (p_2, p_3, and p_4), and two enabled transitions (t_2, and t_3), where both clocks have value equal to one.

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 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 states the conditions that must be satisfied for the firing of a transition to be possible. This definition enforces the *strong firing semantics*, which establishes that an enabled transition t cannot fire before it has been enabled for $EFT(t)$ time units and no later than $LFT(t)$ time units. As it can be observed, an enabled transition is a necessary, but not sufficient condition for that transition to be fireable. It is easy to verify that $FT \subseteq ET \subseteq T$.

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:*

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

Fireable transition t at state s is only fireable in the interval expressed by $FD_s(t)$.

As an example, let us suppose that transition t_0 has fired in the net of Figure 5.1(a) resulting in the marking represented in Figure 5.1(b). In this situation, there are three enabled transitions, that is, t_1, t_2, and t_3. However, transition t_3 is not fireable, since $DLB(t_3)$ is greater than the minimum DUB of all enabled transitions. Furthermore, the firing domain for transition t_1 is [1,3], and for transition t_2 is [2,3], since the minimum DUB is 3.

Definition 5.7 (Reachable States) *Let $\mathcal{P}$ be a time Petri net, and $s_i = (m_i, c_i)$ 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;*
- $\forall t_k \in ET(m_j), C_j(t_k) = \begin{cases} 0, & if(t_k = t) \\ 0, & if(t_k \in ET(m_j) - ET(m_i)) \\ C_i(t_k) + \theta, & otherwise \end{cases}$

In a state s_i, the firing of a fireable transition t, at a specific time instant θ, leads to a new state s_j.

Continuing the previous example, analyzing the firing domain for each fireable transition (t_1 and t_2), it is observed that there are five firing possibilities (three for t_1 and two for t_2), which can lead to five different states.

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 $\mathcal{L}_{\mathcal{P}}$= $(S, \Sigma, \rightarrow, s_0)$, such that: (i) S is the set of states of $\mathcal{P}$; (ii) $\Sigma \subseteq (T \times \mathbb{N})$ is a set of activities labeled with (t_s^i, θ) corresponding to the firing of a fireable transition at a specific time value (θ) in the firing interval $FD_s(t_s^i)$, $\forall s \in S$; (iii) $\rightarrow \subseteq S \times \Sigma \times S$ is the transition relation; (iv) s_0 is the initial state of $\mathcal{P}$.

This definition states that the firing of a transition t_s^i, at a specific time θ_i at state (s_{i-1}) defines the next state (s_i). A state transition $\langle s, (t_s^i, \theta), s' \rangle$ in $\rightarrow$, is denoted by $s \xrightarrow{(t_s^i, \theta)} s'$, implying that the system can change its state from s to s' through activity represented by (t_s^i, θ).

One of the aims of this thesis is to provide a scheduling synthesis framework in such a way that all timing constraints are satisfied. Scheduling, considering a time Petri net model, imposes the existence of an additional control mechanism, called scheduler, for firing a sequence of fireable transitions. This firing sequence is feasible if the following definition is satisfied. As it is shown later (Section 6.1), one of the aims of this thesis is to find such feasible firing sequence (or schedule).

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. $s_0 \xrightarrow{(t_{k1},\theta_{k1})} s_1 \xrightarrow{(t_{k2},\theta_{k2})} s_2 \rightarrow \ldots \rightarrow s_{n-1} \xrightarrow{(t_{kn},\theta_{kn})} s_n$ 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 $\theta_{ki} \in FD_{s_i}(t_{ki})$.*

The automatic system modeling of the proposed methodology (Section 5.3) guarantees that the final marking M^F is well-known since it is explicitly modeled.

Considering that the desired final marking is a token in place p_7, a feasible firing schedule for the TPN model in Figure 5.1 might be:

$$s_0 \xrightarrow{(t_0,0)} s_1 \xrightarrow{(t_2,2)} s_2 \xrightarrow{(t_1,1)} s_3 \xrightarrow{(t_3,3)} s_4 \xrightarrow{(t_4,0)} s_5.$$

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

It is worth observing that $\mathcal{C}$ is a partial function, therefore, some transitions may have no associated source code.

5.1.2 Computational Model for Timing and Energy Consumption

The computational model considers also priorities and energy consumption values.

Definition 5.11 (CTPN with Priorities and Energy Consumption) *A code-labeled time Petri net with priorities and energy consumption (CTPNPE) is represented by $\mathcal{P}_\mathcal{E} = (\mathcal{P}_c, \pi, \mathcal{E})$ $\mathcal{P}_c$ is the underlying CTPN, $\pi : T \rightarrow \mathbb{N}$ is a priority function, and $\mathcal{E}$: $T \nrightarrow \mathbb{N}$ is a partial function that assigns transitions with energy consumption values.*

It is assumed that highest priority gets lowest number. Thus, zero represents the highest priority. In this work, when the priority is not specified, it is assumed that the default value is assigned ($\pi(t_i) = 0$). Considering that $\mathcal{E}$ is a partial function, this definition implies that some transitions may have no associated energy consumption value.

The definition of state in a time Petri net is also extended in order to consider the accumulated energy consumption.

Definition 5.12 (States with Energy Consumption) *Let $\mathcal{P}_\mathcal{E}$ be a CTPNPE, C be the set of all clock vectors in $\mathcal{P}_\mathcal{E}$, and M be the set of reachable markings of $\mathcal{P}_\mathcal{E}$. The set of states $S_\mathcal{E}$ of $\mathcal{P}_\mathcal{E}$ is given by $S_\mathcal{E} \subseteq (M \times \mathbb{N}^{|ET(M)|} \times \mathbb{N})$, that is, a single state is defined by a triple (m, c, e), where m is a marking, c is its respective clock vector for $ET(m)$, and e is the accumulated energy consumption up to this state. The initial state is $s_0 = (m_0, c_0, e_0)$, where $c_0(t) = 0,\ \forall t \in ET(m_0)$, and $e_0 = 0$.*

The definition of fireable transition set is extended to taking into account transitions that satisfy both priorities and energy constraints.

Definition 5.13 (Fireable Transitions with Priorities and Energy) *Let $\mathcal{P}_\mathcal{E}$ be a CTPNPE, $s = (m, c, e)$ be a state of $\mathcal{P}_\mathcal{E}$, and e_{max} the energy maximum value constraint. $FT_\mathcal{E}(s, e_{max})$ is the set of fireable transitions at state s defined by:*

$$FT_\mathcal{E}(s, e_{max}) = \{t_i \in ET(m) \mid (\mathcal{E}(t_i) + e \leq e_{max}) \wedge \pi(t_i) = \min(\pi(t_k)) \wedge (DLB(t_i) \leq min(DUB(t_k)))\ \forall t_k \in ET(m)\}.$$

In this new definition, an enabled transition (t_i) to be fireable must also have $\pi(t_i) = \min(\pi(t_k))$. Another constraint is related to the energy maximum value (e_{max}). It is worth observing that $FT_\mathcal{E} \subseteq FT \subseteq ET \subseteq T$. The definition of firing domain is the same as Definition 5.6.

The definition of reachable states is extended in the following way.

Definition 5.14 (Reachable States with Energy Constraint) *Let $\mathcal{P}_\mathcal{E}$ be a CTPNPE, and $s_i = (m_i, c_i, e_i)$ a reachable state, t a fireable transition ($t \in FT_\mathcal{E}(s_i, e_{max})$), $\mathcal{E}(t)$ the energy consumption related to transition t firing, and θ a specific time value in the firing domain of t ($\theta \in FD_{s_i}(t)$). $s_j = \texttt{fire}(s_i, (t, \theta))$ denotes that firing a transition t at time θ from the state s_i, a new state $s_j = (m_j, c_j, e_j)$ 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*
- $e_j = e_i + \mathcal{E}(t)$
- $\forall t_k \in ET(m_j),\quad c_j(t_k) = \begin{cases} 0, & if(t_k = t) \\ 0, & if(t_k \in ET(m_j) - ET(m_i)) \\ c_i(t_k) + \theta, & otherwise \end{cases}$

A new state s_j is reached from state s_i through the firing of a fireable transition ($t \in FT_\mathcal{E}(s_i, e_{max})$), at a specific time instant ($\theta \in FD_{s_i}(t)$). Additionally, Definition 5.14 is different from Definition 5.7 in the sense that it accumulates, on the new reachable state, the energy consumption value ($\mathcal{E}(t)$) of the fireable transition (t).

5.2 Specification Model

This section depicts the specification model considered in this thesis. As introduced in Section 1.5, one of the results of the user requirement analysis is the specification model. The specification model construction consists of the following steps:

1. defining the timing constraints of a set of cooperating sequential tasks;
2. identifying all critical sections, that is, code of sections that access shared resources, as well as any code sections of tasks that must be executed before some sections of other tasks;
3. dividing each task into subtasks such that appropriate exclusion and/or precedence relations can be defined in pairs of subtasks;
4. calculating the release time and deadline of each subtasks defined in previous item;
5. defining appropriate inter-tasks relations, such as exclusion and precedence relations;
6. translating each sporadic task into an equivalent periodic one;
7. choosing the scheduling method (preemptive or non-preemptive) for each task-subtask;
8. performing the allocation of tasks to processors;
9. providing the source code of each task/subtask;
10. analyzing the source code to obtain the communication pattern, whether adopted a multi-processor architecture.

In the following subsections, this specification model is detailed. Chapter 7 shows how this proposed specification model is integrated in the EZPetri environment [44].

5.2.1 Constraints Specification

This subsection aims to present details about the constraints of tasks. For each task, it shows the timing constraints, inter-task relations, scheduling method, and allocation of tasks to processors.

Task Constraints Specification

Let $\mathcal{T}$ be the set of tasks in a system. The proposed approach considers that tasks are periodic. The definition of constraints of periodic task is as follows.

Definition 5.15 (Periodic Task Constraints) *Let $\tau_i \in \mathcal{T}$ be a periodic task, and $\mathcal{P}$ is the set of processors. 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.*

A periodic task samples objects of interest at a fixed rate. The phase (ph_i) is the delay associated to the first time request of task τ_i after the system starting. Whenever not specified, it is considered that $ph_i = 0$. The periodicity in which τ_i is requested is denoted by the period p_i. In this context, period p_i is a number and not an interval. Release time r_i, WCET c_i, and deadline d_i, are time instants related to the beginning of a period. Thus, 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 a multi-processor architecture, the allocation of tasks to processors becomes necessary. This thesis proposes that this allocation have to be performed in advance by the designer. However, this allocation is beyond the scope of this thesis. Thus, task τ_i is allocated to processor $proc_i \in \mathcal{P}$, where $\mathcal{P}$ is the set of processors.

The definition of the phase time is important, since non schedulable system may become schedulable when a phase is specified. For instance, considering two tasks, τ_1 and τ_2, having equal timing constraints $(ph_1, c_1, d_1, p_1) = (ph_2, c_2, d_2, p_2) = (0, 5, 5, 10)$. As it can be seen, this system is not schedulable. However, if a phase time is specified, i.e. $ph_2 = 5$, the system becomes schedulable.

Particular attention is given to the WCET calculation, where although it is the worst-case, it must not be so pessimistic. The proposed approach considers that the instruction set architecture of the specific processor may help on this task. However, WCET calculation is beyond the scope of this thesis.

Without loss of generality, all timing constraints are expressed in task time units (TTUs), where each TTU has a correspondence with some multiple of a specific timing unit (millisecond, second, etc). For instance, suppose that one TTU corresponds to 10 milliseconds. Suppose also that a worst-case execution time is equal to 50 milliseconds, in this case, the same worst-case execution time is equal to 5 TTUs. If, on the other

hand, one TTU is equal to 5 milliseconds, the same worst-case execution time becomes equal to 10 TTUs. A TTU is the smallest indivisible granule of a task, during which a task cannot be preempted by any other task. A TTU is also called a preemption point. The granularity of the TTU is a designer choice. However, low granularity implies in high complexity, since low granularity may increase the number of objects to be analyzed.

Subtasks Definition

After identifying all critical sections and precedence constraints, usually a task has to be divided in two or more subtasks where appropriate exclusion and/or precedence relations can be defined on pairs of subtasks. This is performed in order to prevent simultaneous access to shared resources and ensure proper execution order. This is useful not only to guarantee mutual exclusion access to shared resources, and enforce the right execution order between tasks, but at the same time, to maximize the chances of finding a feasible schedule, since the relation is applicable only in part of the task and not in the entire task.

Suppose that a task τ_i with release time r_i, deadline d_i, and consisting of a sequence of subtasks $\tau_i^1, \tau_i^2, \cdots, \tau_i^j, \cdots, \tau_i^n$, with execution times $c_i^1, c_i^2, \cdots, c_i^j, \cdots, c_i^n$, respectively, the release time r_i^j and deadline d_i^j of each subtask τ_i^j can be calculated as follows:

$$r_i^j = r_i + \sum_{k=1}^{j} c_i^k \qquad\qquad d_i^j = d_i - \sum_{k=j+1}^{n} c_i^k$$

Suppose, for instance, that task τ_i has timing constraints defined by $(0, 0, 60, 120, 120)$. Suppose also that this task is divided into 3 subtasks, with all execution time equal to 20 TTUs. In this case, the release time and deadline of each subtask can be calculated as:

$$r_i^1 = r_i = 0; \quad r_i^2 = r_i + c_i^1 = 20; \quad r_i^3 = r_i + c_i^1 + c_i^2 = 40.$$

$$d_i^1 = d_i - (c_i^2 + c_i^3) = 80; \quad d_i^2 = d_i - c_i^3 = 100; \quad d_i^3 = d_i = 120.$$

In this thesis, each subtask is considered as if it is a complete task.

Inter-tasks Relations

The considered inter-tasks relations are precedence and exclusion relations.

A task τ_i `PRECEDES` task τ_j, if 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 `EXCLUDES` task τ_j, if no execution of τ_j can start while task τ_i is executing. If it is considered a single processor, then task τ_i could not be preempted by task τ_j. Exclusion relations may prevent simultaneous access to shared resources. In this work, it is considered that the exclusion relation is symmetric, that is, when A `EXCLUDES` B it implies that B `EXCLUDES` A.

Translation from Sporadic to Periodic

In real applications, there are some situations where the arrival of tasks is not periodic. These tasks are generally called aperiodic tasks, since they arrive randomly. However, there is a class of aperiodic tasks called sporadic tasks, where it is known the minimum period between two activations. Therefore, sporadic tasks can have hard deadlines, but aperiodic tasks cannot, once there is no guarantee that their deadlines will be met. The definition of sporadic tasks is as follows.

Definition 5.16 (Sporadic Task) *Let $\tau_k \in \mathcal{T}$ be a sporadic task defined by $\tau_k = (c_k,\ d_k,\ min_k,\ proc_k)$, 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.*

However, pre-runtime approaches may only schedule periodic tasks. In order to schedule sporadic tasks, each one should be translated into an equivalent periodic task. After this translation, a pre-runtime scheduling algorithm may be applied in the set of periodic tasks.

One technique, based on the Mok's work [66], was derived in order to consider such problem where each sporadic task $(c_s, d_s, min_s, proc_s)$ is translated into a corresponding periodic task $(ph_p,\ c_p,\ d_p,\ p_p,\ proc_p)$, satisfying the following conditions:

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. $proc_p = proc_k$.

As it can be seen, the choice of period and deadline is a trade off solution. Hence, larger deadline implies shorter period, and vice-versa. It is worth observing that the τ_p must be in accordance with Definition 5.15 and $c_i \leq d_i \leq p_i$.

For example, consider a sporadic task defined by $ph_s = 0$; $c_s = 2$; d_s= 9; and min_s = 10. The corresponding periodic process may be: $(0, 2, 2, 8)$, where $ph_p = 0$, $c_p = c_s = 2$, $d_p = c_s = 2$, and $p_p = \min(d_s - d_p + 1, min_s) = \min(8, 10) = 8$. In this case, periodic executions are scheduled to start at time 0, 8, 16, ..., and if the sporadic request are, for instance, 1, 11, and 30, then the start times of the sporadic tasks executions are 8, 16, and 32. As it can be noted, despite the arrival of sporadic tasks happen at random, they can be dealt with as periodic ones by buffering such events. Furthermore, the adopted translation from sporadic to periodic task allows d_s to be always met.

Figure 5.2 graphically shows how the behavior of the equivalent periodic tasks is related to the sporadic requests. In that figure, rs_i's are sporadic requests, and s_i's are actual sporadic executions. As it can be seen, all timing constraints for sporadic tasks are satisfied by the equivalent periodic task.

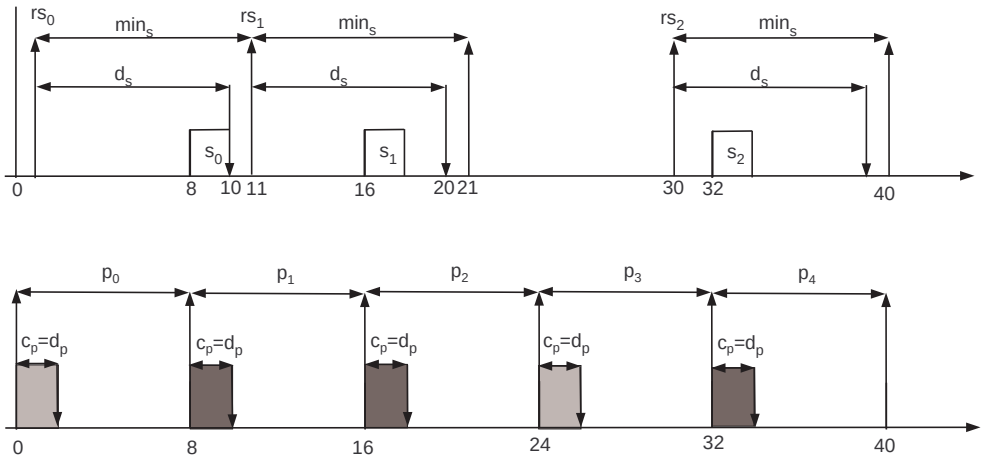

Figure 5.2: Translation from Sporadic to Periodic Task

As presented in Section 2.4.2, another alternative is to use a hybrid solution to schedule such sporadic tasks as shown in [104]. In this approach, a pre-runtime schedule is constructed to deal with periodic tasks. Using the information in this pre-runtime schedule, the sporadic tasks are scheduled by a runtime scheduler. In this case, a table of "safe start time intervals" is constructed in such a way that it is guaranteed that if a sporadic task is executed in such safe interval it will not interfere in the execution of any previously computed execution of periodic tasks.

For instance (from [107]), suppose that two tasks are defined, one sporadic and other periodic. The sporadic task is defined by $\tau_s = (3, 15, 15)$. The periodic task is defined by $\tau_p = (0, 3, 3, 8)$. Suppose also that τ_p is not allowed to preempt task τ_s. In this example, the safe start time interval for execution of task τ_s will be [3,5]. The worst-case response time happens when τ_s arrives at time 6, and it is delayed until τ_p has completed. In this case, τ_s completes its execution at time 14. So, this worst-case response time is $14 - 6 = 8$, which is less than its original deadline $d_{\tau_s} = 15$.

Scheduling Method

For each task (or subtask) the designer has to choose what scheduling method is best suited. The options are preemptive or non-preemptive. A task τ_i is said to be preemptive if its execution can be suspended by another tasks, excluding the tasks that τ_i excludes. A task τ_i is said to be non-preemptive if its execution cannot be suspended by any other task. In this case, τ_i runs up to completion.

It is worth observing that if the tasks are all non-preemptive the chance to find a feasible schedule is drastically minimized.

5.2.2 Behavioral Specification

The behavioral specification is divided into (i) source code of each task; and/or (ii) the communication pattern for multi-processor architecture expressed as a communication graph. However, as presented later in Section 6.2, this work calculates schedules, but not perform code generation considering multi-processor architectures.

The source code of each task is programmed using the C language augmented with communication constructs. These constructs are not found in standard C language, but in this specification they are useful for generating the communication pattern. Moreover, excluding the communication constructs, the code has to be in accordance with the respective compiler for the chosen processor.

In order to capture the communication pattern, the C code augmented with communication primitives is analyzed and a communication graph may be constructed. If the communication occurs between tasks in the same processor, it is treated as a precedence relation. If, on the other hand, the communication occurs between tasks in different processors, a new communication task is included. The communication task is formally defined in the following way.

Definition 5.17 (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 $proc_i \neq proc_j$.

This definition enforces the point-to-point communication, since it explicitly defines that communication can only occur between two tasks allocated in different processors.

The communication constructs are *SEND (channel, to, item)*, and *RECEIVE (channel, from, item)*. These constructs specify (i) the channel where the information is read/written; (ii) the other task from which the message will be sent to (`SEND construct`) or received from (`RECEIVE construct`); and the values read/written in this channel. It is worth noting that channel is a high level abstraction for several communication media, such as bus, serial/parallel ports, infrared, fiber optics, and so on.

5.2.3 Specification Example

Table 5.1 shows part of a specification responsible for describing task information. In this table, both periodic and communication tasks are specified. For periodic tasks, timing constraints (phase time, release time, worst-case execution time, deadline, and period), and the processor allocated are presented. For communication tasks, the worst-case communication time, sender and receiver tasks are specified. This table also shows inter-task relations, in this case, precedence and exclusion relations.

Another component of a specification can be seen in Figure 5.3. This table presents a C code template for each task. However, this table only concentrates on showing communication pattern between tasks. After associating code to each task, a communication graph (Figure 5.4) may be generated for facilitating the communication pattern presentation. This example does not consider energy constraints.

5.3 Modeling the Specification

The systems considered in this thesis are classified as embedded hard real-time systems. These kind of systems are those that besides their functional correctness, timeliness must be satisfied.

Time Petri net is a mathematical formalism that allows modeling of several features present in most concurrent and real-time systems, such as, precedence and exclusion relations, communication protocols, multiprocessing, synchronization mechanisms, and shared resources. Therefore, in this thesis, the modeling phase is based on time Petri net formalism.

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	4	3	30	30	proc1	-	-
D	0	0	2	20	30	proc1	-	-
E	0	2	3	30	30	proc1	-	-
F	0	0	2	10	30	proc2	-	-
G	0	2	3	30	30	proc2	-	-
H	0	3	3	30	30	proc2	-	-
I	0	5	2	30	30	proc2	-	-
J	0	0	3	10	30	proc3	-	-
K	0	2	3	30	30	proc3	-	-
L	0	3	2	30	30	proc3	-	-
M1	-	-	1	-	-	bus1	F	A
M2	-	-	1	-	-	bus1	F	J
M3	-	-	2	-	-	bus1	B	H
M4	-	-	2	-	-	bus1	L	H

Intertask Relations		
A PRECEDES B,	B PRECEDES C	
A EXCLUDES D,	D EXCLUDES A	
D PRECEDES B,	D PRECEDES E,	C PRECEDES E
F PRECEDES G,	G PRECEDES H,	H PRECEDES I
J PRECEDES K,	K PRECEDES L	
F PRECEDES M1,	M1 PRECEDES A	
F PRECEDES M2,	M2 PRECEDES J	
B PRECEDES M3,	M3 PRECEDES H	
L PRECEDES M4,	M4 PRECEDES H	

```
task A()
{
  .
  .
  .
  RECEIVE (bus, F, msgF);
  .
  .
  .
}

task B()
{
  .
  .
  .
  SEND (bus, H, msgBtoH);
  .
  .
  .
}

task C()
{
  .
  .
  .
}

task D()
{
  .
  .
  .
}

task E()
{
  .
  .
  .
}

task F()
{
  .
  .
  .
  SEND (bus, A, msgFtoA);
  .
  .
  .
  SEND (bus, J, msgFtoJ);
  .
  .
  .
}
```

```
task G()
{
  .
  .
  .
}

task H()
{
  .
  .
  .
  RECEIVE (bus, B, msgB);
  .
  .
  .
  RECEIVE (bus, L, msgL);
  .
  .
  .
}

task I()
{
  .
  .
  .
}

task J()
{
  .
  .
  .
  RECEIVE (bus, F, msgF);
  .
  .
  .
}

task K()
{
  .
  .
  .
}

task L()
{
  .
  .
  .
  SEND (bus, H, msgLtoH);
  .
  .
  .
}
```

Figure 5.3: Specification Behavior

The proposed modeling applies composition rules on building blocks models. These blocks are specific for the scheduling policy adopted, that is, pre-runtime scheduling policy. For instance, pre-runtime algorithm schedules tasks considering a schedule period that corresponds to the least common multiple between all periods in the task set. In this case, the modeling has to be adjusted to consider such slight intrinsic differences. The proposed building blocks are: (i) periodic task arrival; (i) task structure, which considers preemptive and/or non-preemptive task scheduling method; (ii) deadline checking, which uses elementary net structures; (iii) inter-task relations, such

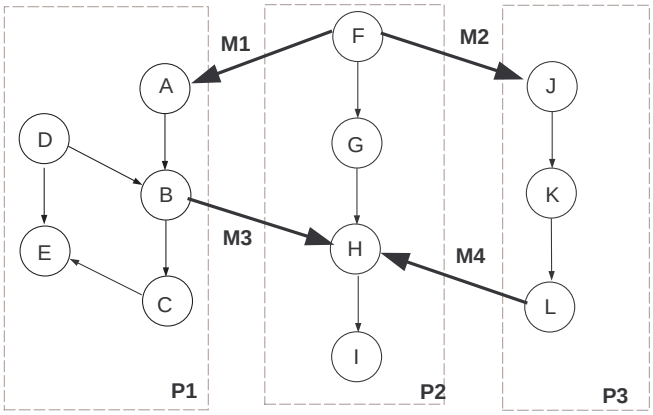

Figure 5.4: Communication Pattern

as precedence and exclusion relations, and (iv) inter-processor communication.

This section is divided into five subsections. The first one discuss about scheduling period. Next subsection introduces several ways to perform net compositions, such as place merging, addition and refinement, arc addition and removing, and net union. The third subsection deals with the modeling of tasks. It details all building blocks and latter shows two examples of composition of these blocks in order to model both a single task and two tasks sharing a single processor. After that, inter-task relation modeling is explained. This section finishes detailing how to model inter-processor communication.

5.3.1 Scheduling Period

Instead of computing a pre-runtime schedule considering an infinite period, the approach is to schedule the entire set of periodic tasks occurring within a time period that is equal to the least common multiple (LCM) among periods of the given set of tasks. The LCM is also called *schedule period* (P_S).

Within this new period, there are several *tasks instances* of the same task, where $\mathcal{N}(\tau_i) = P_S/p_i$ gives the instances of task τ_i. For example, consider the task set in Table 5.2. In this particular case, $P_S = 24$, implying that the two periodic tasks are replaced by seven new periodic tasks ($\mathcal{N}(\tau_1) = 3$, and $\mathcal{N}(\tau_2) = 4$), where the timing constraints of each task instance has to be transformed to consider that new period.

Table 5.3 depicts the modified timing constraints. Considering this new period, a periodic task τ_i has a finite number of *periodic task execution* $\tau_i^1, \tau_i^2, \cdots, \tau_i^j, \cdots, \tau_i^{N(\tau_i)}$, with one task execution for each period. Furthermore, for the jth 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 5.2: Timing Constraints for a Simple Task Set

task	r	c	d	p
τ_1	0	2	7	8
τ_2	2	2	6	6

Table 5.3: Modified Timing Constraints for a Simple Task Set

	τ_1^1	τ_1^2	τ_1^3	τ_2^1	τ_2^2	τ_2^3	τ_2^4
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 Net Composition Operators

The proposed modeling method is conducted by building block compositions in order to form larger nets. This section provides several operators for net compositions. These operators are: place merging, serial place refinement, place addition, arc addition, arc removing, and net union. The use of such operators is presented latter at Sections 5.3.3 and 5.3.4.

(1) Place Merging Operator. Combining nets by place merging is a simple and effective way to model communication between blocks. In Figure 5.5, the left-hand block produces tokens in its place *send-message*, and the right-hand block consumes such tokens from its place *recv-message*. By merging such places, communication between these blocks takes place. This work considers that place merging can only occur among two nets. In this context, places *send-message* and *recv-message* are called merging places, and place *message* is called merged place. This operator assumes that the marking of the merged place is the maximum between the original marking of the merging places. The formal definition of place merging operator is as follows.

Definition 5.18 (Place Merging) *Consider the following nets:*

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

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

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

where

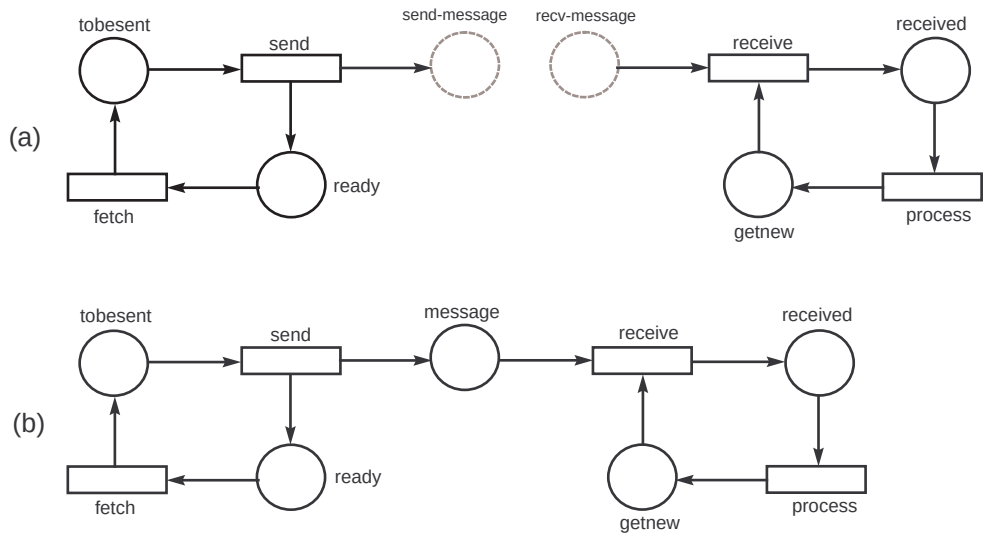

Figure 5.5: A Simple Example of Place Merging

$$P_1 = \{p_{1_1}, p_{1_2}, \ldots, p_{1_{n_1}}\}; \quad T_1 = \{t_{1_1}, t_{1_2}, \ldots, t_{1_{m_1}}\}$$
$$P_2 = \{p_{2_1}, p_{2_2}, \ldots, p_{2_{n_2}}\}; \quad T_2 = \{t_{2_1}, t_{2_2}, \ldots, t_{2_{m_2}}\}$$
$$P_c = \{p_{c_1}, p_{c_2}, \ldots, p_{c_{n_3}}\}; \quad T_c = \{t_{c_1}, t_{c_2}, \ldots, t_{c_{m_3}}\}.$$

Also consider the following three ordered set of places

$$\delta_1 = \langle p_1^1, p_1^2, \ldots p_1^i, \ldots, p_1^u \rangle \subseteq P_1$$
$$\delta_2 = \langle p_2^1, p_2^2, \ldots p_2^i, \ldots, p_2^u \rangle \subseteq P_2$$
$$\delta_c = \langle p_c^1, p_c^2, \ldots p_c^i, \ldots, p_c^u \rangle \subseteq P_c.$$

The composition by place merging is denoted by $N_c = \langle \texttt{Pmerg} \rangle\ (N_1,\ N_2,\ \delta_1,\ \delta_2,\ \delta_c)$*, where* N_1 *and* N_2 *are two merging nets,* δ_1 *is a set of merging places of* N_1*,* δ_2 *is a set of merging places of* N_2*, and* δ_c *of* N_c *is the set of merged places. The net* N_c *is composed in the following way:*

- $P_c = (P_1 \cup P_2 \cup \delta_c) - (\delta_1 \cup \delta_2).$
- $T_c = T_1 \cup T_2$
- $\forall t \in T_c :\ I_c(t) = \begin{cases} I_1(t), & if\ t \in T_1 \\ I_2(t), & if\ t \in T_2 \end{cases}$
- $F_c = \{F_1 - (\delta_1 \times T_1 \cup T_1 \times \delta_1)\}\ \cup$
  $\{F_2 - (\delta_2 \times T_2 \cup T_2 \times \delta_2)\}\ \cup$
  $\{(p_c^i, t_{1_j}) \mid t_{1_j} \in T_1\ \wedge\ (p_1^i, t_{1_j}) \in F_1,\ 1 \le i \le u,\ 1 \le j \le m_1\}\ \cup$
  $\{(t_{1_j}, p_c^i) \mid t_{1_j} \in T_1\ \wedge\ (t_{1_j}, p_1^i) \in F_1,\ 1 \le i \le u,\ 1 \le j \le m_1\}\ \cup$
  $\{(p_c^i, t_{2_k}) \mid t_{2_k} \in T_2\ \wedge\ (p_2^i, t_{2_k}) \in F_2,\ 1 \le i \le u,\ 1 \le k \le m_2\}\ \cup$
  $\{(t_{2_k}, p_c^i) \mid t_{2_k} \in T_2\ \wedge (t_{2_k}, p_2^i) \in F_2,\ 1 \le i \le u,\ 1 \le k \le m_2\}$

$$\star\ \forall p \in P_c:\ M_{0c}(p) = \begin{cases} M_{01}(p) & if\ p \in P_1, P_c \\ M_{02}(p) & if\ p \in P_2, P_c \\ \max\left(M_{01}(p_1^i), M_{02}(p_2^i)\right) & if \exists\ p_c^i = p,\ 1 \leq i \leq u. \end{cases}$$

$$\star\ \forall f \in F_c : W_c(f) = \begin{cases} W_1(f) & if\ f \in F_1 \\ W_2(f) & if\ f \in F_2 \\ W_1(p_1^i, t_{1_j}) & if\ f = (p_c^i, t_{1_j}),\ p_c^i \in \delta_c,\ t_{1_j} \in T_1 \\ W_1(t_{1_j}, p_1^i) & if\ f = (t_{1_j}, p_c^i),\ p_c^i \in \delta_c,\ t_{1_j} \in T_1 \\ W_2(p_2^i, t_{2_k}) & if\ f = (p_c^i, t_{2_k}),\ p_c^i \in \delta_c,\ t_{2_k} \in T_2 \\ W_2(t_{2_k}, p_2^i) & if\ f = (t_{2_k}, p_c^i),\ p_c^i \in \delta_c,\ t_{2_k} \in T_2 \\ \multicolumn{2}{l}{such\ that,\ 1 \leq i \leq u,\ 1 \leq j \leq m_1,\ 1 \leq k \leq m_2.} \end{cases}$$

The new set of places (P_c) is the union of the two sets of places (P_1 and P_2) of the merging nets (N_1 and N_2), increased by the set δ_c and decreased by sets δ_1 and δ_2. The new set of transitions T_c is just the union of the set of transitions (T_1 and T_2) of the merging nets. In the same way, the timing interval I_c is the union of the timing interval (I_1 and I_2) of the merging nets. The new arcs (F_c) is defined by: (i) removing arcs with places of δ_1, and δ_2; (ii) including arcs where places in δ_1 (in pre or post-conditions) are replaced by the respective places in δ_c; and (iii) including arcs where places in δ_2 (in pre or post-conditions) are replaced by the respective places in δ_c. As it is assumed that merging places (δ_1 and δ_2) are with no marking, the merged places (δ_c) is also with no marking.

In order to explain the new weight of arcs (W_c) consider the Figure 5.6. Suppose that places p_2 of net N_1 and p_3 of net N_2 are merged into the place p_m on the new net N_c.

The new weight of arcs is defined in the following way.

(a) If the arc f is from F_1, then the weight is $W_1(f)$. As an example, see $W_c(p_0, t_0) = W_1(p_0, t_0) = 1$.

(b) If the arc f is from F_2, then the weight is $W_2(f)$. See, for instance, $W_c(p_1, t_1) = W_2(p_1, t_1) = 1$.

(c) If the arc $f \in F_c$ has a place from δ_c and output transition from T_1, then $W_c(f)$ is equal to weight of the arc between the corresponding place in δ_1 and the same output transition. See $W_c(p_m, t_2) = W_1(p_2, t_2) = 4$.

(d) If the arc $f \in F_c$ has a place from δ_c and input transition from T_1, then $W_c(f)$ is equal to weight of the arc between the same input transition and the corresponding place in δ_1. Refer to $W_c(t_0, p_m) = W_1(t_0, p_2) = 2$.

(e) If the arc $f \in F_c$ has a place from δ_c and output transition from T_2, then $W_c(f)$ is equal to weight of the arc between the corresponding place in δ_2 and the same output transition. For instance, see $W_c(p_m, t_3) = W_2(p_3, t_3) = 6$.

(f) Finally, if the arc $f \in F_c$ has a place from δ_c and input transition from T_2, then $W_c(f)$ is equal to weight of the arc between the same input transition and the corresponding place in δ_2. Refer to $W_c(t_1, p_m) = W_c(t_1, p_3) = 3$.

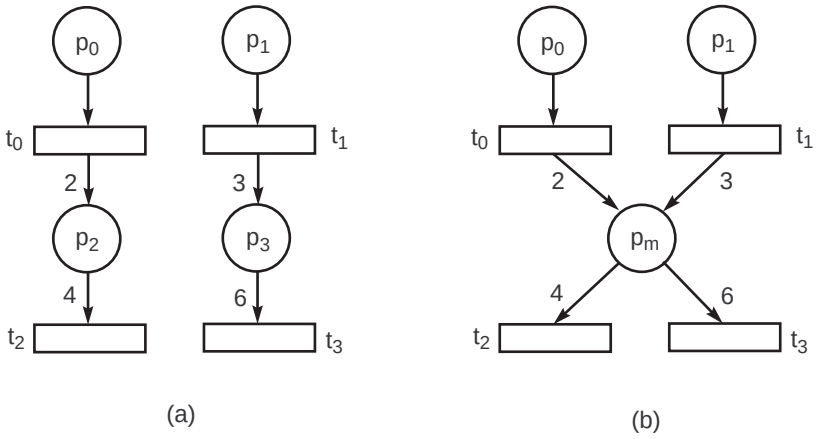

Figure 5.6: An Example of Place Merging: (a) Before Place Merging; (b) After Place Merging

(2) Serial Place Refinement Operator

In this work, a serial place refinement can be seen as a replacement of a single place (p_δ) by a sequence of one place (p_σ), one transition (t_σ), and another place (p'_δ). In this context, the place p_δ is called refining place, and places p_σ and p'_δ are called refined places. In the same way, transition t_σ is called refined transition. Figure 5.7 depicts such serial place refinement. In order to maintain behavioral and timing properties, it is worth observing the weight of the input (α) and output (β) arcs of the refining place (p_δ) is the same as the second refined place, in this case, p'_δ. Without loss of generality, this place refinement assumes that the timing interval for the refined transition t_σ is always $[0, 0]$.

The formal definition of serial place refinement is as follows.

Definition 5.19 (Serial Place Refinement) *Considering the following time Petri*

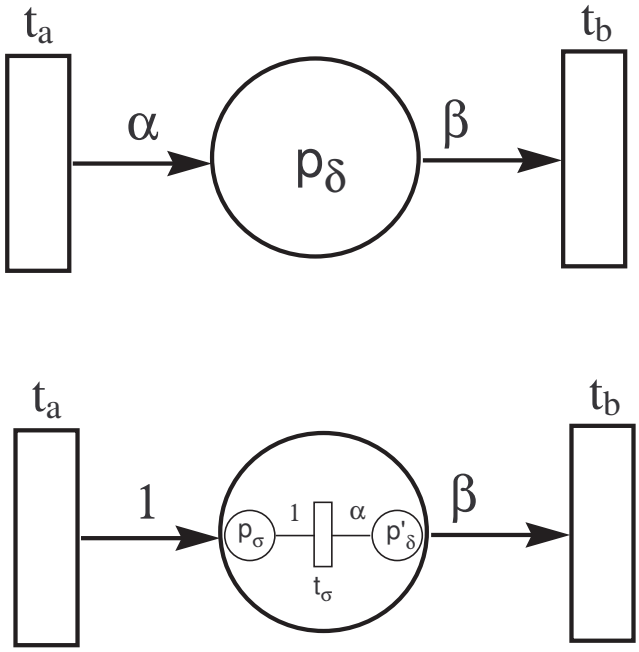

Figure 5.7: Place Refinement

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

Informally, the net obtained from the serial place refinement is composed as follows.

The new set of places (after refinement) is increased by two places p_σ and p'_δ and, at the same time, the refining place p_δ is removed. The new set of transitions is increased by t_σ. The aim of this refinement is replacing the refining place (p_δ) by the following sequence: a place (p_σ), a transition (t_σ), and another place (p'_δ).

In the definition of the flow relation (F_c), F^1 removes the arcs from and to the place p_δ. F^2 is used for adding pre and post-conditions for place p'_δ, which are the same as pre and post-conditions of the refining place p_δ. F^3 is responsible for adding arcs from p_σ to t_σ, and from t_σ to p'_δ.

The two refined places are assumed to have no initial marking, i.e., $M_0(p_\sigma) = M_0(p'_\delta) = 0$. The remaining places will continue with their respective markings.

This proposed refinement assumes that timing interval for firing the refined transition t_σ is always $[0, 0]$.

After the refinement, the arc weight of $\bullet p_\sigma$ is assumed to be unitary. The same occurs if the arc is from p_σ to t_σ. As shown at Figure 5.7, the weight of the pre and post-conditions of place p_δ (refining place) is the same as the weight of the pre and post-conditions of place p'_δ. As presented before, this refinement is performed this way for maintaining both behavioral and timing properties. All other arcs that do not refer to any refining element (places or transition) or the place to be refined are the same as in the original net.

(3) Arc Addition Operator

Arc addition is an operator that adds an arc from a place to a transition or from a transition to a place. Usually, arc addition must be used with care. If not, some properties may not be satisfied. However, this work proposes that the modeling method is performed automatically. Thus, the user does not have direct participation in the modeling. The formal definition of arc addition is as follows.

Definition 5.20 (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* $N_c = \langle \texttt{Aadd} \rangle (N, (x, y), w)$*, 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:*

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

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

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

The resultant net N_c is obtained by just adding a single arc (x, y) with weight w.

(4) Arc Removing Operator

Arc removing is an operator that removes an arc from a place to a transition or from a transition to a place. In the same way as arc addition operator, arc removing must be used with care. The formal definition of arc removing is as follows.

Definition 5.21 (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:*

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

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

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

The resultant net N_c is obtained by just removing a single arc (x, y).

(5) Place Addition Operator

Place addition is another operator that adds a single place to a net. Hence, after the place addition, the added place is disconnected from any other element (place or transition). Therefore, the place addition has to be used with other operators in order to obtain the desired modeling. In the same way as arc addition operator, the place addition has to be used with care. However, as presented before, the modeling is performed automatically, and all the operators usage is conducted carefully. The formal definition of place addition is as follows.

Definition 5.22 (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:*

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

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

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

The output net N_c is obtained by just adding a single place p_δ with marking m_δ.

(6) Net Union Operator

Net union operator simply unifies two nets producing another net. Hence, after using such operator, certainly the net is disconnected. The model should take care in using such operator. The same considerations about automatic modeling made in previous operators are also valid for this one.

Definition 5.23 (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* $N_c = N_1 \sqcup N_2$. *The resultant net* N_c *is computed in the following way:*

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

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

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

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

This operator simply joins two nets into a new net.

5.3.3 Modeling of Tasks

This section aims to describe how to represent the specification of tasks by adopting a suitable formal model, in this case, a time Petri net model. In order to depict the method for modeling of tasks and inter-tasks relations, this section considers the task timing specification presented in Table 5.4.

Table 5.4: A Simple Example of Task Timing Specification and Inter-task Relations

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

The proposed modeling of tasks is performed by building block compositions. The considered building blocks are: (i) Periodic Task Arrival; (ii) Task Structure; (iii) Deadline Checking; (iv) Inter-processor sending message; (v) Resources, such as processors and buses; (vi) Fork; and (vii) Join. These blocks are detailed below.

When presenting these blocks, it is worth observing that places and transitions are expressed with indexes. These indexes are used for instantiation purpose, or in other words, the same block may have slight differences when applied to different tasks. For instance, the timing interval for a transition that represents the execution of a task may have different values when considering different tasks.

(i) Periodic Task Arrival Block

The Periodic Task Arrival Block (Figure 5.8) models the periodic invocation for all task instances in the schedule period (P_S). A transition t_{ph_i} models the initial phase of the task first instance. Similarly, transition t_{a_i} models the periodic arrival (after the initial phase) for the remaining instances. It is worth noting the weight (α_i) of the arc (t_{ph_i}, p_{wa_i}), where this weight models the invocation of all remaining instances after the first task instance.

The building block periodic task arrival is a TPN $N_a = (P_a, T_a, F_a, W_a, M_{0_a}, I_a)$, such that:

⋆ $P_a = \{p_{st_i}, p_{wa_i}, p_{wd_i}, p_{wr_i}\}$. These places model the following conditions:

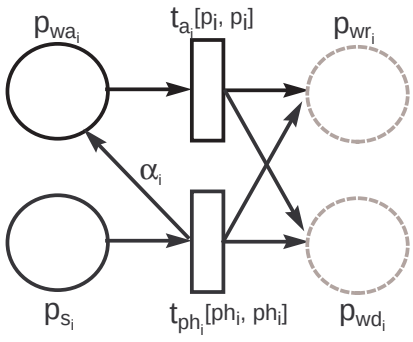

Figure 5.8: Building Block Arrival

p_{st_i}: starting of task;

p_{wa_i}: waiting for the arrival of another task instance;

p_{wd_i}: waiting for deadline missing; and

p_{wr_i}: waiting for release time.

⋆ $T_a = \{t_{a_i}, t_{ph_i}\}$. These transitions model the following actions:

t_{a_i}: arriving of a new task instance; and

t_{ph_i}: elapsing of the task initial phase.

⋆ Pre and post-conditions of the transitions are:

$$\bullet t_{a_i} = \{p_{wa_i}\}$$
$$t_{a_i}\bullet = \{p_{wr_i}, p_{wd_i}\}$$
$$\bullet t_{ph_i} = \{p_{st_i}\}$$
$$t_{ph_i}\bullet = \{p_{wa_i}, p_{wr_i}, p_{wd_i}\}$$

⋆ $W_a(m,n) = \begin{cases} \alpha_i, & if\ (m = t_{ph_i} \wedge n = p_{wa_i}),\ \alpha_i \in \mathbb{N} \\ 1, & otherwise. \end{cases}$

⋆ $M_{0_a}(p) = 0\ \forall p \in P$.

⋆ $I_a(t_{ph_i}) = [ph_i, ph_i]$; and $I(t_{a_i}) = [p_i, p_i]$.

The timing intervals of transitions t_{a_i} and t_{ph_i} are fulfilled by the timing constraints specification, in this case, ph_i (phase) and p_i (period) of task τ_i.

Figure 5.9 explains graphically the application of this block into the task T_0 in Table 5.4. Note that $P_S = 750$, hence, in this case the number of task instances for T_0 is equal to $\mathcal{N}(T_0) = 750/250 = 3$. In this specific situation, $\alpha = \mathcal{N}(T_0) - 1 = 2$

(ii) Task Structure Block

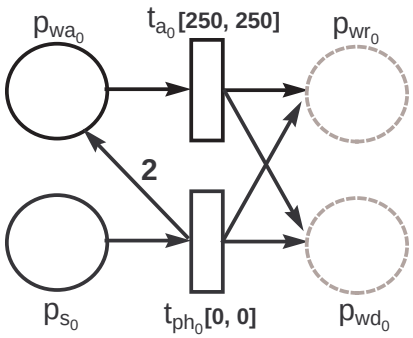

Figure 5.9: Building Block Arrival for Task T_0

The building block task structure models: release time, processor granting, computation, and processor releasing. Although release time is a time instant related to the beginning of the period, this is modeled by an interval $[r_i, d_i - c_i]$. This interval is adopted since there are situations where a system has to be left idle in order to reach all timing constraints (see Section 2.4.3 for more details). Processor granting and releasing is needed in order to access this resource in mutual exclusion. Computation can be modeled either as preemptive or non-preemptive policy. In both policies, there is a specific transition for processor granting. Nevertheless, processor releasing is performed by the respective computation transition.

In the following, it is shown how to model tasks that require either preemptive or non-preemptive scheduling methods.

Preemptive Task Structure Block

This scheduling method implies that tasks are implicitly split into all possible subtasks, where the computation time of each subtask is exactly equal to one task time unit (TTU). Before computation, the processor is granted to the respective task, and after computation the processor is released. This method allows the running of another conflicting task, in this case, meaning that one task preempts another task. Figure 5.10 presents the structure of the preemptive method. As presented before (Section 5.2.1), a TTU is the smallest indivisible granule of a task. Thus, the preemption points are equal to one TTU. This is modeled by the time interval of computation transitions ([1,1]), and the entire computation is modeled through the arc weights. Hence, c_i tokens are put in place p_{wg_i}, and

the same amount of tokens is needed for firing of transition t_{f_i}. Depending on the TTU granularity, this schedule method may generate much more states.

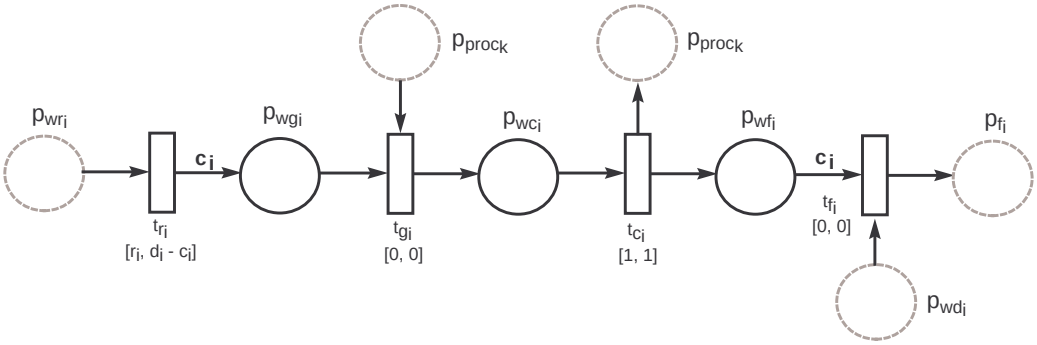

Figure 5.10: Building Block Preemptive Task Structure

The building block preemptive task structure is a TPN $N_p = (P_p, T_p, F_p, W_p, M_{0_p}, I_p)$, such that:

- $P_p = \{p_{wr_i}, p_{wg_i}, p_{wc_i}, p_{wf_i}, p_{f_i}, p_{dm_i}, p_{proc_k}\}$. These places model the following conditions:

  p_{wr_i}: waiting for releasing;

  p_{wg_i}: waiting for processor granting;

  p_{wc_i}: waiting for task computation;

  p_{wf_i}: waiting for task instance end;

  p_{f_i}: end of a task instance;

  p_{wd_i}: waiting for deadline missing.

  p_{proc_k}: processor.

- $T_p = \{t_{r_i}, t_{g_i}, t_{c_i}, t_{f_i}\}$. These transitions model the following actions:

  t_{r_i}: task releasing;

  t_{g_i}: processor granting;

  t_{c_i}: executing one task unit, and processor releasing; and

  t_{f_i}: concluding the task computation;

- Pre and post-conditions of the transitions are:

  $\bullet t_{r_i} = \{p_{wr_i}\}; \quad t_{r_i}\bullet = \{p_{wg_i}\};$

  $\bullet t_{g_i} = \{p_{wg_i}, p_{proc_k}\}; \quad t_{g_i}\bullet = \{p_{wc_i}\};$

  $\bullet t_{c_i} = \{p_{wc_i}\}; \quad t_{c_i}\bullet = \{p_{wf_i}, p_{proc_k}\};$

  $\bullet t_{f_i} = \{p_{wf_i}, p_{wd_i}\}; \quad t_{f_i}\bullet = \{p_{f_i}\}.$

- ⋆ $W_p(p_{wr_i}, t_{r_i}) = W_p(p_{wg_i}, t_{g_i}) = W_p(p_{proc_k}, t_{g_i}) = W_p(t_{g_i}, p_{wc_i} = W_p(p_{wc_i}, t_{c_i}) = W_p\ (t_{c_i}, p_{wf_i}) = W_p(t_{c_i}, p_{proc_k}) = W_p(t_{f_i}, p_{f_i}) = 1; W_p(t_{r_i}, p_{wg_i}) = W_p(p_{wf_i}, t_{f_i}) = c_i$.
- ⋆ $M_{0_p}(p_{proc_k}) = \beta,\ \beta \in \mathbb{N}^+; \quad M_{0_p}(p) = 0\ \forall p \in P \wedge p \neq p_{proc_k}$.
- ⋆ $I_p(t_{r_i}) = [r_i, d_i - c_i]$; $I_p(t_{g_i}) = [0, 0]$; $I_p(t_{c_i}) = [1, 1]$; and $I_p(t_{f_i}) = [0, 0]$.

The timing interval of transition t_{r_i} is fulfilled by the timing constraints specification, in this case, r_i (release time) of task τ_i. All remaining timing intervals are constant. The same way, the arc weight (from t_{r_i} to p_{wg_i} and p_{wf_i} to t_{f_i}) comes from the c_i (execution time) of task τ_i. The initial marking of the p_{proc_k} is a non-zero integer.

Figure 5.11 shows the TPN of this block applied into the task T_0. This figure considers a preemptive scheduling method for T_0.

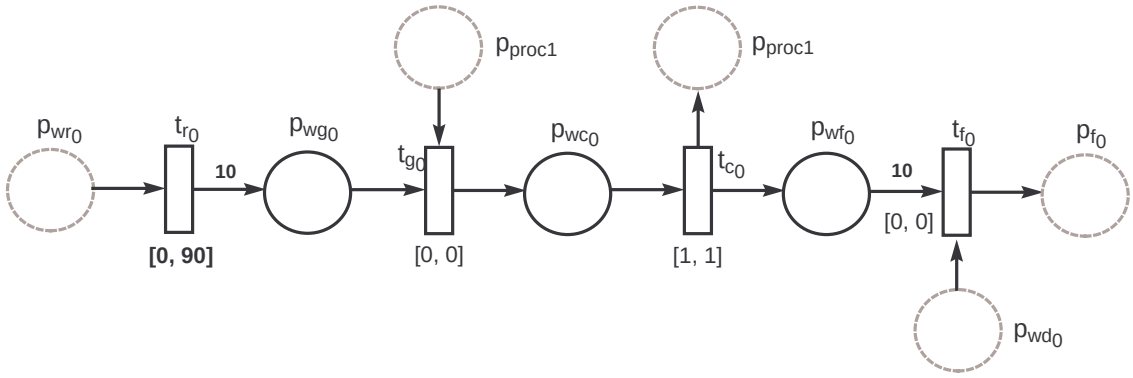

Figure 5.11: Building Block Preemptive Task Structure for T_0

Non-Preemptive Task Structure Block

Considering a non-preemptive scheduling method, the processor is just released after the entire computation to be finished. Figure 5.12 shows that time interval of computation transition has bounds equal to the task computation time (i.e., $[c_i, c_i]$).

The building block non-preemptive task structure is a TPN $N_{np} = (P_{np}, T_{np}, F_{np}, W_{np}, M_{0_{np}}, I_{np})$, such that:

- ⋆ $P_{np} = \{p_{wr_i}, p_{wg_i}, p_{wc_i}, p_{wf_i}, p_{f_i}, p_{wd_i}, p_{proc_k}\}$. These places model the following conditions:

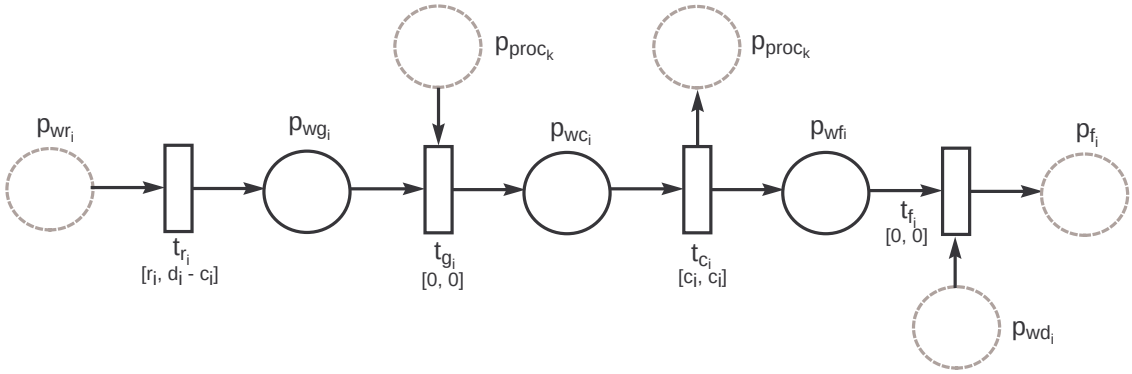

Figure 5.12: Building Block Non-Preemptive Task Structure

p_{wr_i}: waiting for releasing;

p_{wg_i}: waiting for processor granting;

p_{wc_i}: waiting for task computation;

p_{wf_i}: waiting for task instance end;

p_{f_i}: end of a task instance;

p_{wd_i}: waiting for deadline missing; and

p_{proc_k}: processor.

⋆ $T_{np} = \{t_{r_i}, t_{g_i}, t_{c_i}, t_{f_i}\}$. These transitions model the following actions:

t_{r_i}: task releasing;

t_{g_i}: processor granting;

t_{c_i}: executing a task, and processor releasing; and

t_{f_i}: concluding the task computation.

⋆ Pre and post-conditions of the transitions are:

$$\bullet t_{r_i} = \{p_{wr_i}\}; \quad t_{r_i}\bullet = \{p_{wg_i}\}$$
$$\bullet t_{g_i} = \{p_{wg_i}, p_{proc_k}\}; \quad t_{g_i}\bullet = \{p_{wc_i}\}$$
$$\bullet t_{c_i} = \{p_{wc_i}, p_{wd_i}\}; \quad t_{c_i}\bullet = \{p_{f_i}, p_{proc_k}\}$$
$$\bullet t_{f_i} = \{p_{wf_i}\}; \quad t_{f_i}\bullet = \{p_{f_i}\}.$$

⋆ $W_{np}(x, y) = 1 \; \forall (x, y) \in F$.

⋆ $M_{0_{np}}(p_{proc_k}) = \beta, \; \beta \in \mathbb{N}^+; \quad M_{0_{np}}(p) = 0 \; \forall p \in P \wedge p \neq p_{proc_k}$.

⋆ $I_{np}(t_r) = [r_i, d_i - c_i]$; $I_{np}(t_g) = [0, 0]$; and $I_{np}(t_c) = [c_i, c_i]$.

The timing intervals of transitions t_{r_i} and t_{c_i} are fulfilled by the timing constraints specification, in this case, r_i (release) and c_i (execution time) of task τ_i. The

timing interval of transition t_{g_i} is constant. The initial marking of the p_{proc_k} is a non-zero integer.

As an example, Figure 5.13 depicts the application of the block non-preemptive task structure into the task T_0 of Table 5.4.

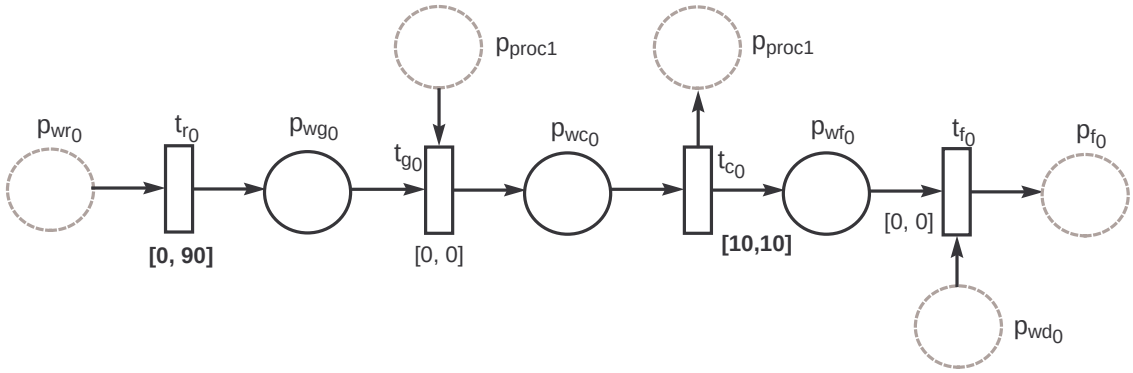

Figure 5.13: Building Block Non-Preemptive Task Structure for Task T_0

(iii) Deadline Checking Block

Some works (e.g. [4]) extended the Petri net model for dealing with deadline checking. The proposed modeling method uses elementary net structures to capture deadline missing. Obviously, deadline missing is an undesirable situation when considering hard real-time systems. Therefore, the scheduling algorithm (Section 6.1) must eliminate states that represent undesirable situations like this one.

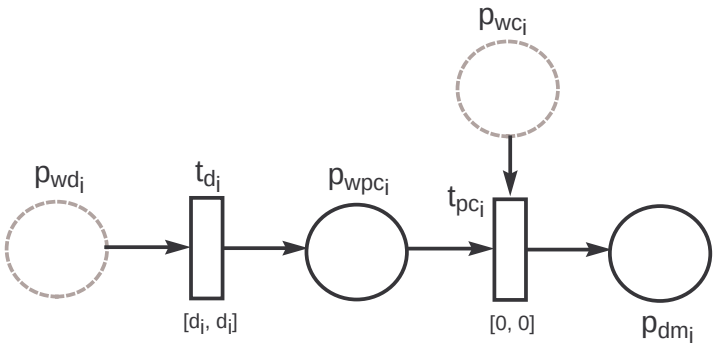

Figure 5.14: Building Block Deadline Checking

The building block deadline-checking is a TPN $N_d = (P_d, T_d, F_d, W_d, M_{0_d}, I_d)$, such that:

- $P_d = \{p_{wd_i}, p_{wpc_i}, p_{dm_i}, p_{wc_i}\}$. These places model the following situations:

  p_{wd_i}: waiting for deadline missing;

  p_{wpc_i}: waiting for computation removing;

  p_{dm_i}: deadline missed; and

  p_{wc_i}: waiting for task computation.

- $T_d = \{t_{d_i}, t_{rc_i}\}$. These transitions model the following actions:

  t_{d_i}: deadline missing; and

  t_{pc_i}: computation removing.

- Pre and post-conditions of the transitions are:

  $\bullet t_{d_i} = \{p_{wd_i}\}; \quad t_{d_i}\bullet = \{p_{wrc_i}\};$

  $\bullet t_{rc_i} = \{p_{wrc_i}, p_{wc_i}\}; \quad t_{rc_i}\bullet = \{p_{dm_i}\}.$

- $W_d(x, y) = 1 \; \forall (x, y) \in F.$

- $M_{0_d}(p) = 0 \; \forall p \in P.$

- $I_d(t_{d_i}) = [d_i, d_i]; \quad I_d(t_{pc_i} = [0, 0])$

The timing interval of transition t_{d_i} is fulfilled by the timing constraints specification, in this case, d_i (deadline) of task τ_i. The timing interval of transition t_{pc_i} is constant.

Figure 5.15 shows the block deadline checking for the task T_0 of Table 5.4.

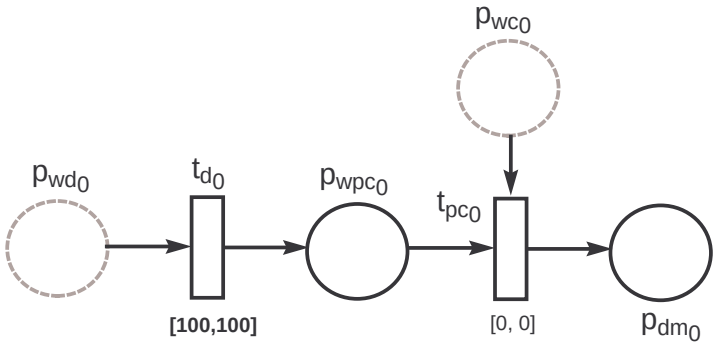

Figure 5.15: Building Block Deadline Checking for Task T_0

(iv) Inter-processor Sending Message Block

As introduced before in Section 5.2.2 (Behavioral Specification), the specification considers that all inter-processor communication are dealt with as a new communication task. Table 5.4 shows M_1 and M_2 communication tasks.

This section aims to present a specific block for modeling inter-processor message sending. Figure 5.16 depicts such building block.

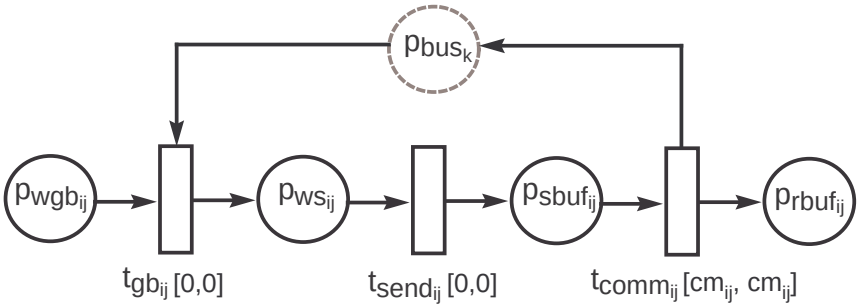

Figure 5.16: Building Block Send

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

- $P_{sm} = \{p_{wgb_{ij}}, p_{ws_{ij}}, p_{sbuf_{ij}}, p_{rbuf_{ij}}\}$. These places model the following situations:

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

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

  $p_{sbuf_{ij}}$: sending buffer; and

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

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

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

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

  $t_{comm_{i,j}}$: communication.

- Pre and post-conditions of the transitions are:

  $\bullet t_{gb_{ij}} = \{p_{wgb_{ij}}\}; \quad t_{gb_{ij}}\bullet = \{p_{ws_{ij}}\};$

  $\bullet t_{send_{i,j}} = \{p_{ws_{ij}}\}; \quad t_{send_{i,j}}\bullet = \{p_{sbuf_{ij}}\}.$

  $\bullet t_{comm_{i,j}} = \{p_{sbuf_{ij}}\}; \quad t_{comm_{i,j}}\bullet = \{p_{rbuf_{ij}}\}.$

- $W_{sm}(x, y) = 1 \; \forall (x, y) \in F_{sm}$.

- $M_{0_{sm}}(p_{bus_k}) = \beta, \; \beta \in \mathbb{N}^+; \quad M_{0_{sm}}(p) = 0 \; \forall p \in P \wedge p \neq p_{bus_k}$.

- $I_{sm}(t_{comm_{ij}}) = [ct_m, ct_m]; \quad I_{sm}(t_{gb_{ij}}) = I_{sm}(t_{send_{ij}}) = [0, 0])$

The timing interval of transition $t_{comm_{ij}}$ 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 $t_{send_{ij}}$ are constant.

It is worthwhile to point out that inter-processor receiving message is modeled by place refinement not by block composition. Section 5.3.5 presents an example of modeling inter-processor communication (message sending and receiving).

(v) Resource Block

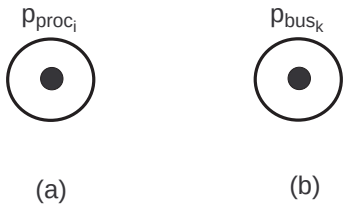

Figure 5.17: Modeling of Resources: (a) Processor; (b) Bus

The only explicitly modeled resources considered in this work are processors and buses.

As this work considers that the allocation task-to-processor is performed in advance by the designer, and task migration is not allowed, so each processor has to be explicitly modeled. The processor modeling (Fig. 5.17(a)) consists of a single place p_{proc_i}, where its marking states how many processors are available. If a place representing a processor is modeled having more than one marking, it represents a multiprocessor architecture with unified memory architecture (UMA) [92].

Buses (Fig. 5.17(b)), on the other hand, are communication channels used for providing communication between tasks from different processors. In the same way as processors, a bus is modeled by a single place p_{bus_k}, where it is assumed that this place must have exactly one marking.

In order to compose resources (processor or bus) with a single task model, both place and arc addition operators are adopted.

(vi) Fork Block

Let us suppose that the system has n tasks. The building block fork (Fig. 5.18) is responsible for starting all tasks instances occurring in the schedule period (or hyper-period). This block consists of the creation of n concurrent process starting from a single parent process.

The fork block is modeled by a TPN $N_f = (P_f, T_f, F_f, W_f, M_{0_f}, I_f)$, where:

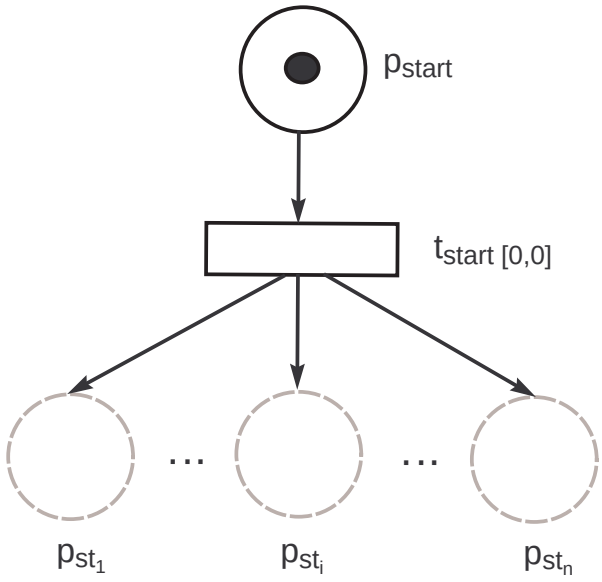

Figure 5.18: Building Block Fork

- $P_f = \{p_{start}, p_{st_1}, \cdots, p_{st_i}, \cdots, p_{st_n}\}$. These places model the following situations:

  p_{start}: waiting for system starting.

  p_{ts_i}: starting of the i^{th} task, $1 \leq i \leq n$.

- $T_f = \{t_{start}\}$. This transition model the following action:

  t_{start}: starting of all tasks of the system.

- Pre and post-conditions of the transition are:

  $\bullet t_{start} = \{p_{start}\}$

  $t_{start}\bullet = \{p_{st_1}, \cdots, p_{st_i}, \cdots, p_{st_n}\}$

- $W_f(x, y) = 1,\ \forall (x, y) \in F$.

- $M_{0_f}(p_{start}) = 1;\ M_0(p) = 0,\ \forall p \in P \wedge p \neq p_{start}$.

- $I_f(t_{start}) = [0, 0]$

The timing interval of transition t_{start} is constant. Figure 5.19 illustrates the application of the fork block in the task set of Table 5.4.

(vii) Join Block

Usually, concurrent activities need to synchronize with each other. The join block execution states that all tasks in the system have concluded their execution in the schedule period. Figure 5.20 presents the join block.

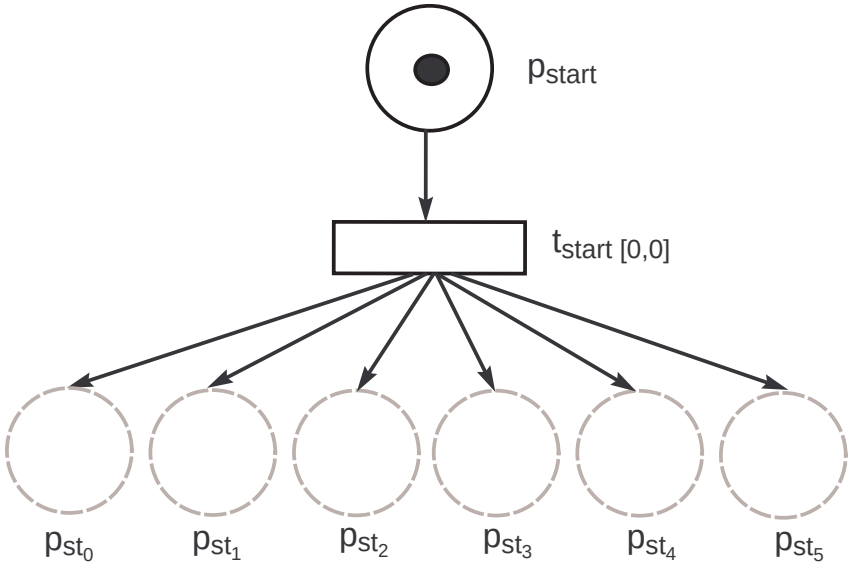

Figure 5.19: Building Block Fork for Task Set in Table 5.4

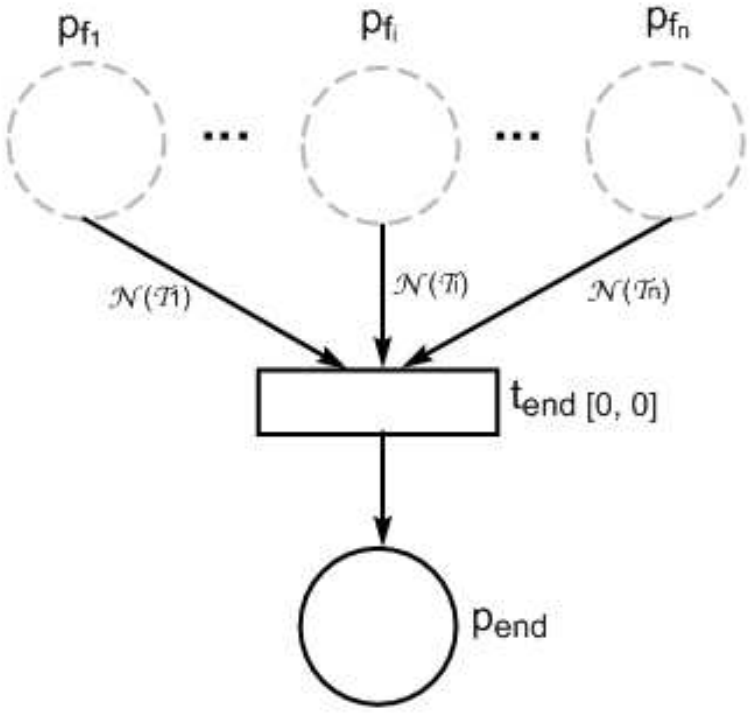

Figure 5.20: Building Block Join

This block is modeled by a TPN $N_j = (P_j, T_j, F_j, W_j, M_{0_j}, I_j)$, where:

- $P_j = \{p_{f_1}, \cdots, p_{f_i}, \cdots, p_{f_n}, p_{end}\}$. These places model the following situations:

p_{f_i}: end of the i^{th} task, $1 \leq i \leq n$.

p_{end}: end of the system.

⋆ $T_j = \{t_{end}\}$. This transition model the following action:

t_{end}: end of tasks in the system.

⋆ Pre and post-conditions of the transition are:

$\bullet t_{end} = \{p_{f_1}, \cdots, p_{f_i}, \cdots, p_{f_n}\}$

$t_{end}\bullet = \{p_{end}\}$

⋆ $W_j(x, y) = \alpha,\ \forall (x, y) \in F,\ \alpha \in \mathbb{N}$.

⋆ $M_{0_j}(p) = 0,\ \forall p \in P$.

⋆ $I_j(t_{end}) = [0, 0]$

The timing interval of transition t_{end} is constant. It is worth remembering 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 firing schedule (Definition 5.9) was found.

As an example, Figure 5.21 depicts the application of the join block in the task set of Table 5.4.

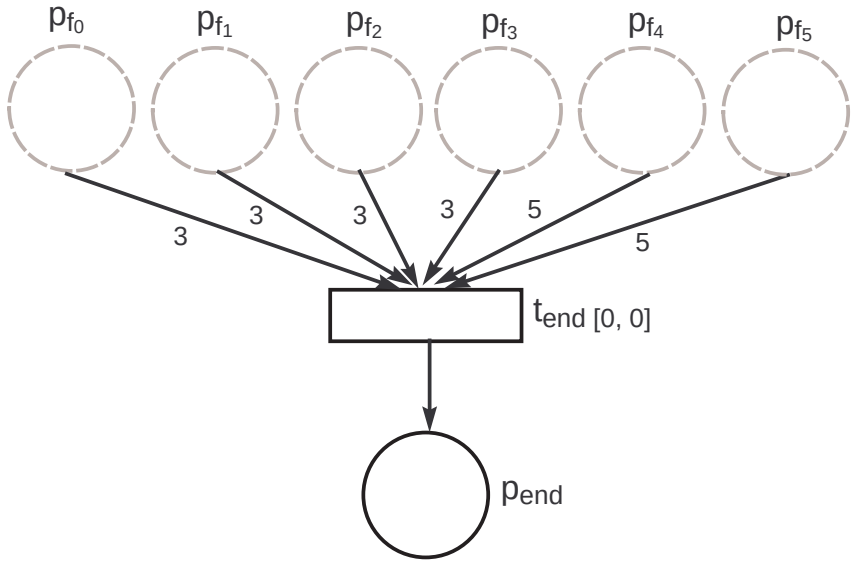

Figure 5.21: Building Block Join for Task Set in Table 5.4

(viii) Composition of a Single Task

Lets consider task T_0 on Table 5.4 as a non-preemptive task. For generating such task model the method is based on composition of building blocks. In this case,

the blocks are: its task structure (N_{np}), periodic task arrival (N_a) and deadline checking (N_d) blocks. In order to instantiate such blocks some information are needed, for instance, the period (for the periodic task arrival block), deadline (for the deadline checking block), and releasing, deadline and execution time (for the task structure block). After instantiating these blocks, they are named: N_{a_0}, N_{np_0}, and N_{d_0}.

For composing this single task, the following suppositions are considered:

$\delta_1 = \langle p_{wr_0} \rangle \in P_{a_0}$ (from N_{a_0});

$\delta_2 = \langle p_{wr_0} \rangle \in P_{np_0}$ (from N_{np_0});

$\delta_{m_1} = \langle p_{wr_0} \rangle \in P_0$ (from N_0);

$\delta_3 = \langle p_{wc_0}, p_{wd_0} \rangle \in P_0$ (from N_0);

$\delta_4 = \langle p_{wc_0}, p_{wd_0} \rangle \in P_{d_0}$ (from N_{d_0});

$\delta_{m_2} = \langle p_{wc_0}, p_{wd_0} \rangle \in P_0$ (from N_0);

The net $N_0 = (P_0, T_0, F_0, W_0, M_{0_0}, I_0)$, representing T_0, is defined by:

(a) $N_0 = \langle \texttt{Pmerg} \rangle (N_a, N_{np}, \delta_1, \delta_2, \delta_{m_1})$

(b) $N_0 = \langle \texttt{Pmerg} \rangle (N_0, N_d, \delta_3, \delta_4, \delta_{m_2})$

Figure 5.22 shows the composition of task T_0 using building blocks models. This composition is performed in two steps: First (Step 0a), merging places p_{wr_0} (from net N_{a_0}) and p_{wr_0} (from net N_{np_0}) generating the place p_{wr_0} on net N_0. This merging is depicted in Figure 5.22(a). Second (Step 0b), merging places p_{wc_0} and p_{wd_0} (from net N_0) and p_{wc_0} and p_{wd_0} (from net N_{d_0}) and obtaining places p_{wc_0} and p_{wd_0} on the same net N_0. This merging is depicted in Figure 5.22(b). These merged places, in Figure 5.22, are gray colored in order to highlight them.

If, instead of non-preemptive, task T_0 is preemptive, the first composition should consider N_{p_0} instead of N_{np_0}.

(ix) Composing Two Tasks and a Single Processor

The system is composed by two tasks T_0 and T_1 of Table 5.4. This example considers that these two tasks share a single processor. The net representing this system is called $N_{aux} = (P_{aux}, T_{aux}, F_{aux}, W_{aux}, M_{0_aux}, I_{aux})$. The two nets ($N_0$ and N_1) representing both tasks (T_0 and T_1) are already defined, as depicted

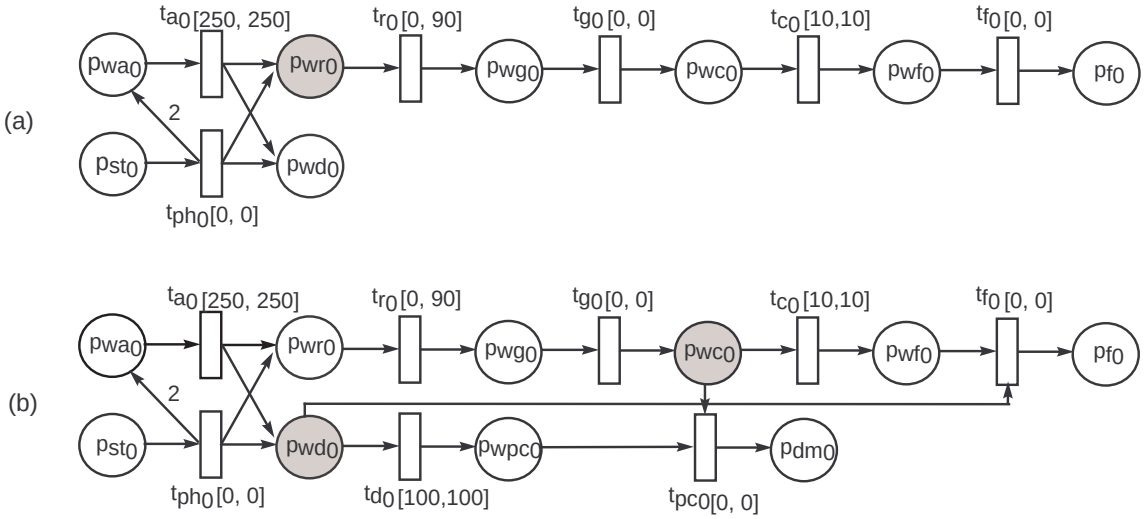

Figure 5.22: Complete Model for Task T_0

previously. 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 composing two tasks sharing a single processor, the following suppositions are considered:

$\delta_{st} = \langle p_{st_0}, p_{st_1} \rangle \in P_{aux}$;

$\delta_f = \langle p_{st_1}, p_{st_2} \rangle \in P_f$ (from N'_f);

$\delta_{m_1} = \langle p_{st_0}, p_{st_1} \rangle \in P_{aux}$;

$\delta_{end} = \langle p_{f_0}, p_{f_1} \rangle \in P_{aux}$;

$\delta_j = \langle p_{f_1}, p_{f_2} \rangle \in P_j$ (from N'_j);

$\delta_{m_2} = \langle p_{f_0}, p_{f_1} \rangle \in P_{aux}$;

p_{proc} is a single place, where $M(p_{proc}) = 1$.

In the proposed modeling method, the TPN representing this system (N_{aux}) is modeled as follows:

(a) $N_{aux} = (N_0 \sqcup N_1)$;

(b) $N_{aux} = \langle \texttt{Padd} \rangle (N_{aux}, p_{proc}, 1)$

(c) $N_{aux} = \langle \texttt{Aadd} \rangle (N_{aux}, (p_{proc}, t_{g0}), 1)$

(d) $N_{aux} = \langle \texttt{Aadd} \rangle (N_{aux}, (p_{proc}, t_{g1}), 1)$

(e) $N_{aux} = \langle \texttt{Aadd} \rangle (N_{aux}, (t_{c0}, p_{proc}), 1)$

(f) $N_{aux} = \langle \texttt{Aadd} \rangle \, (N_{aux}, (t_{c1}, p_{proc}), 1)$

(g) $N_{aux} = \langle \texttt{Pmerg} \rangle \, (N_{aux}, N'_f, \delta_{st}, \delta_f, \delta_{m_1})$

(h) $N_{aux} = \langle \texttt{Pmerg} \rangle \, (N_{aux}, N'_j, \delta_{end}, \delta_j, \delta_{m_2})$

Initially, the two nets N_0 and N_1, representing the two tasks, are joined in a new net called N_{aux} (Step 0a). N_0 and N_1 may be either preemptive or non-preemptive. After that, place p_{proc} is added into the N_{aux} net (Step 0b). This place has just one marking representing a single processor. The next four steps (Steps from 0c to 0f) 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 0g) and join (Step 0h) nets.

Figure 5.23 shows a TPN representing both T_0 and T_1 as non-preemptive task. If, however, both tasks are preemptive, the minor changes can be seen in Figure 5.24.

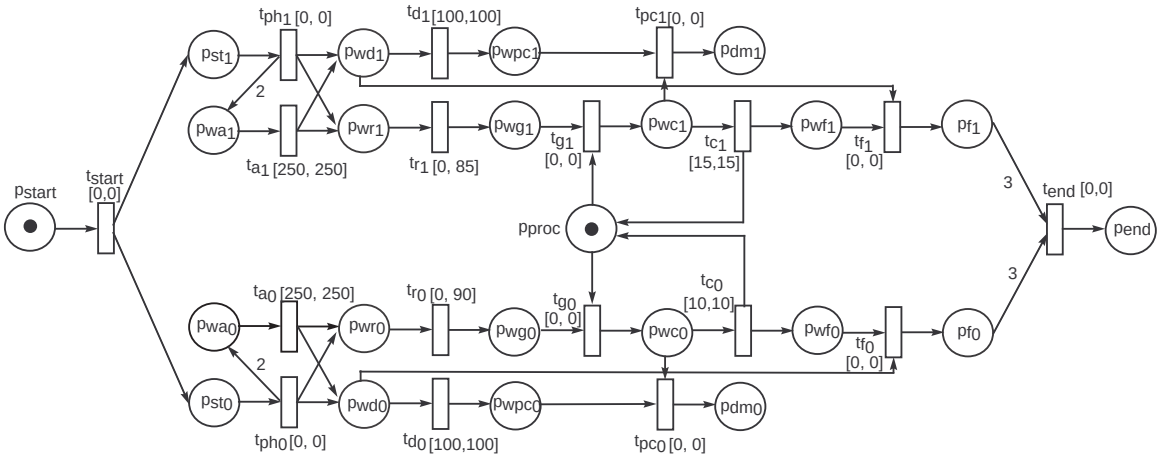

Figure 5.23: Complete Model for T_0 and T_1 Non-preemptive Tasks

5.3.4 Inter-task Relations Modeling

This section presents how to model inter-task relations considered in this work, namely, precedence and exclusion relations. These two inter-tasks relations are described below.

Precedence Relation

Precedence relations are defined between pairs of tasks, such that one task can only start executing after the other has been finished. Considering that τ_i `PRECEDES` τ_j is

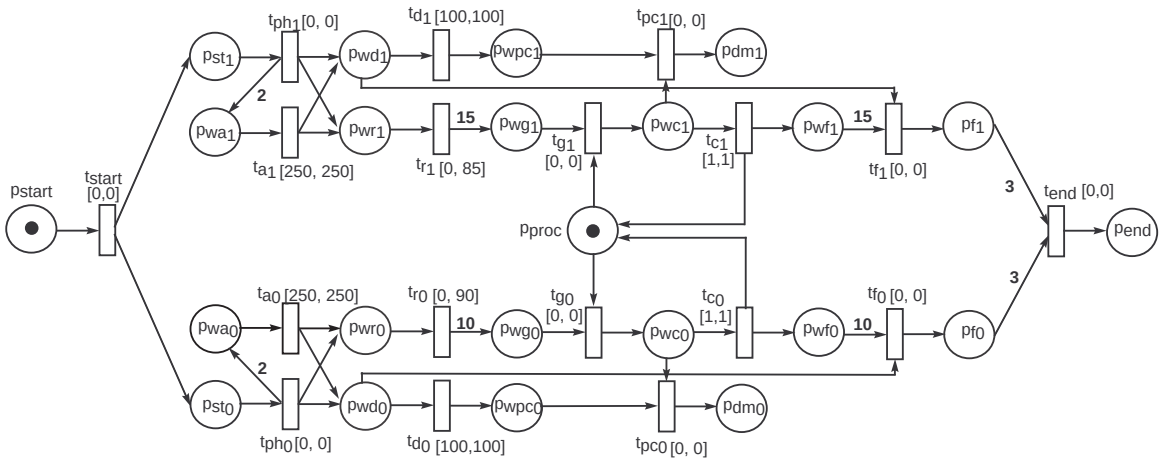

Figure 5.24: Complete Model for T_0 and T_1 Preemptive Tasks

specified, and supposing that both tasks are represented by the N_{aux} net, the following steps are performed in order to model such precedence relation:

1. $N_{aux} = \langle \texttt{Padd} \rangle \, (N_{aux}, p_{prec_{ij}}, 0)$

2. for the preceding task τ_i:

   (a) $N_{aux} = \langle \texttt{Aadd} \rangle \, (N_{aux}, t_{f_i}, p_{prec_{ij}})$

3. for the preceded task τ_j:

   (a) $N_{aux} = \langle \texttt{Pref} \rangle \, (N_{aux}, p_{wg_j}, p_{wp_{ij}}, t_{prec_{ij}}, p_{wg_j})$.

   (b) $N_{aux} = \langle \texttt{Aadd} \rangle \, (N_{aux}, t_{prec_{ij}}, p_{prec_{ij}}, 1)$

A single place $p_{prec_{ij}}$ is included for each defined precedence relation (Step 1). In the preceding task, an arc is included (Step 2a) for stating that the included place $p_{prec_{ij}}$ is the post-condition of transition t_{f_i} in net N_i (representing task τ_i). In the net that represents the preceded task (in this case, N_j), the place p_{wg_j} is refined (Step 3a) in such a way that this place is replaced by the following sequence: $p_{wp_{ij}}$, $t_{prec_{ij}}$, and p_{wg_j}. Finally, an arc is added (Step 3b) stating that place $p_{prec_{ij}}$ is pre-condition for firing transition $t_{prec_{ij}}$ on net N_j.

Figure 5.25 shows the TPN model for tasks T_1 and T_2, and the T_1 `PRECEDES` T_2 inter-task relation. It worth noting that task T_2 can only proceed after task T_1 has finished its execution.

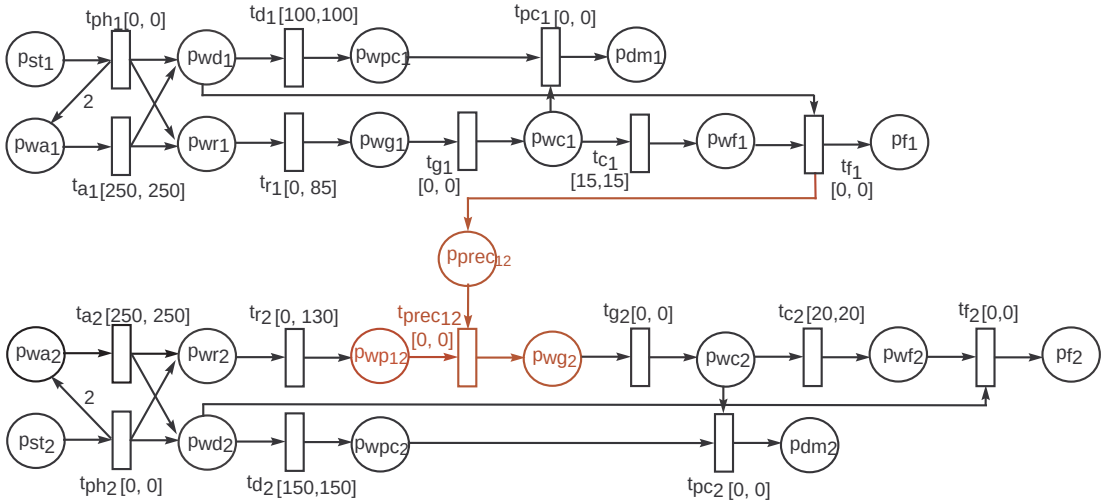

Figure 5.25: Precedence Relation Model for tasks T_1 and T_2

Exclusion Relation

Exclusion relations are also defined between pairs of tasks. Supposing that τ_i `EXCLUDES` τ_j represents a situation in which two tasks cannot be concurrently executing at the same time. In other words, if task τ_i starts executing, task τ_j has to wait up to task τ_i finishes its execution. Usually, this relation is well-suited for tasks that access the same critical region.

The proposed modeling method adds a single place (with one marking), which is pre-condition for executing both tasks. Therefore, just one of both tasks is executing at the same time. It is worth observing that the proposed exclusion relation is symmetrical, that is, if task τ_i excludes task τ_j, it implies that task τ_j also excludes task τ_i.

After modeling the two tasks (τ_i and τ_j), represented by the N_{aux} net, the following actions are performed in order to model this exclusion relation:

1. $N_{aux} = \langle \texttt{Padd} \rangle \, (N_{aux}, p_{excl_{ij}}, 1)$;
2. In task τ_i, do:
   (a) $N_{aux} = \langle \texttt{Pref} \rangle \, (N_{aux}, p_{wg_i}, p_{wexcl_{ij}}, t_{excl_{ij}}, p_{wg_i})$;
   (b) $N_{aux} = \langle \texttt{Aadd} \rangle \, (N_{aux}, p_{excl_{ij}}, t_{excl_{ij}})$;
   (c) $N_{aux} = \langle \texttt{Aadd} \rangle \, (N_{aux}, t_{f_i}, p_{excl_{ij}})$;
3. In task τ_j, do:
   (a) $N_{aux} = \langle \texttt{Pref} \rangle \, (N_{aux}, p_{wg_j}, p_{wexcl_{ji}}, t_{excl_{ji}}, p_{wg_j})$;

(b) $N_{aux} = \langle \texttt{Aadd} \rangle (N_{aux}, p_{excl_{ij}}, t_{excl_{ji}})$;

(c) $N_{aux} = \langle \texttt{Aadd} \rangle (N_{aux}, t_{f_j}, p_{excl_{ij}})$;

Firstly, a new place $p_{excl_{ij}}$ is inserted (Step 1) into the net. This place has one token and it is responsible for guaranteeing the mutual access to the critical section. Both nets, which represent the pair of tasks, are serially refined (Steps 2a and 3a) in such a way that:

- ⋆ the sequence $(p_{wexcl_{ij}}, t_{excl_{ij}}, p_{wg_i})$ substitutes the single place p_{wg_i} (from τ_i);
- ⋆ the sequence $(p_{wexcl_{ji}}, t_{excl_{ji}}, p_{wg_j})$ substitutes the single place p_{wg_j} (from τ_j).

After that, four arc additions (Steps 2b, 2c, 3b, 3c) are performed:

- ⋆ $(p_{excl_{ij}}, t_{excl_{ij}})$ and $(t_{f_i}, p_{excl_{ij}})$ (from τ_i); and
- ⋆ $(p_{excl_{ij}}, t_{excl_{ji}})$ and $(t_{f_j}, p_{excl_{ij}})$ (from τ_j).

Figure 5.26 shows the TPN model for both tasks T_0 and T_2, and the T_0 `EXCLUDES` T_2 inter-task relation. This figure considers that both tasks are preemptive.

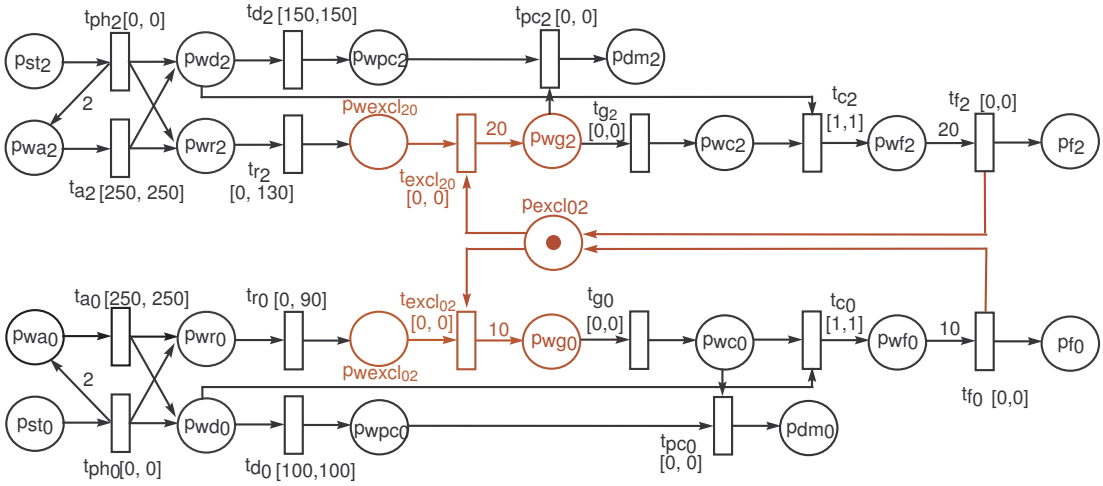

Figure 5.26: Exclusion Relation Model for Preemptive Tasks T_0 and T_2

5.3.5 Modeling Inter-processor Communication

This work supposes that communication time between tasks allocated to the same processor is negligible, 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 time is not negligible.

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:

- ⋆ non-blocking message sending, implying that after sending a message the task may continue its work;
- ⋆ message receiving is blocking, that is, the task can only continue after receiving the complete message;
- ⋆ point-to-point communication (or unicasting);
- ⋆ buses are reliable;
- ⋆ before communication takes place, the specific bus has to be granted to the respective task;
- ⋆ communication time is represented by the respective communication transition.

Communication tasks (Definition. 5.17) 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 5.27 applies the building block send message for modeling the sending task τ_i as well as Figure 5.28 presents the serial place refinement for modeling the receiving task τ_j.

Taking into account that: (i) buses are specified (such as place p_{bus_m} represents the bus bus_m); (ii) nets N_i and N_j are defined, such that these nets represent tasks τ_i to τ_j, respectively; (iii) there is just one transition in the pre-set of place p_{f_i}. Formally, inter-processor communication model is obtained by carrying out the following steps:

1. instantiate the block inter-processor message sending (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) $t_{prev} = {}^\bullet p_{f_i}$
   (b) remove the arc from t_{prev} to p_{f_i} $\quad (N_{aux} = \langle \texttt{Arem} \rangle (N_{aux}, t_{prev}, p_{f_i}))$;
   (c) add an arc from t_{prev} to $p_{wgb_{ij}}$ $\quad (N_{aux} = \langle \texttt{Aadd} \rangle (N_{aux}, t_{prev}, p_{wgb_{ij}}))$.

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

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

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

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

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

(b) add an arc from $p_{rbuf_{ij}}$ to $t_{recv_{ij}}$ $(N_{aux} = \langle \texttt{Aadd} \rangle (N_{aux}, p_{rbuf_{ij}}, t_{recv_{ij}}))$

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 message sending block (Step 2). Considering the modeling of the sending task, one arc is removed (Step 3b) and four arcs are added (Steps from 3c to 3f). Moreover, for the receiving task, the place p_{wc_j} is refined (Step 4a), and one arc is added (Step 4b).

Supposing 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 of the second message sending block from task τ_i is shown in Figure 5.29.

The adoption of $t_{prev} = \bullet p_{f_i}$ (Step 3a) is due to the fact that the same task may send messages to more than one task, as presented previously. In Figure 5.27, $t_{prev} = \bullet p_{f_i} = t_{f_i}$, however, in case of Figure 5.29, $t_{prev} = \bullet p_{f_i} = t_{send_{ij}}$.

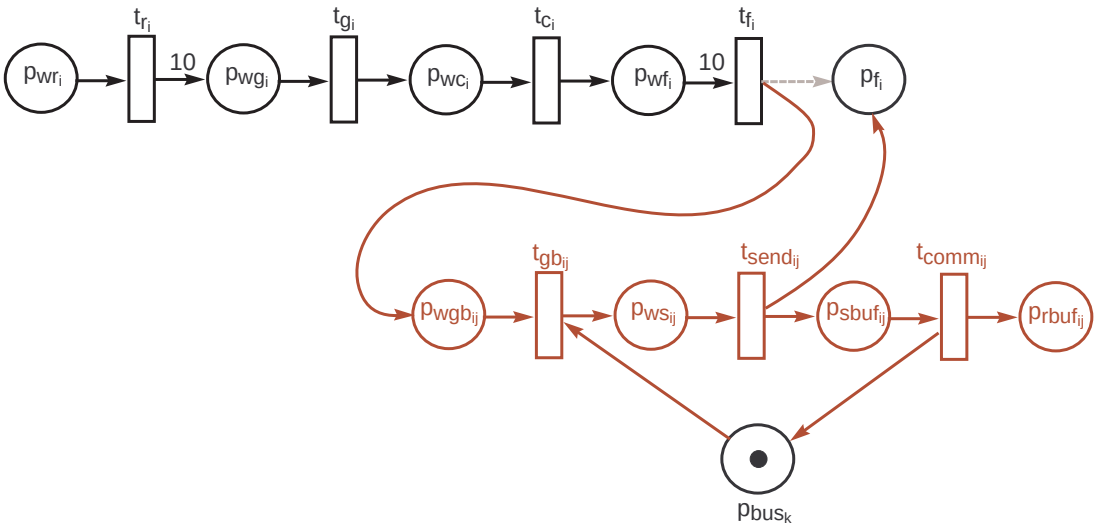

Figure 5.27: Modeling of the Sending Task from τ_i to τ_k

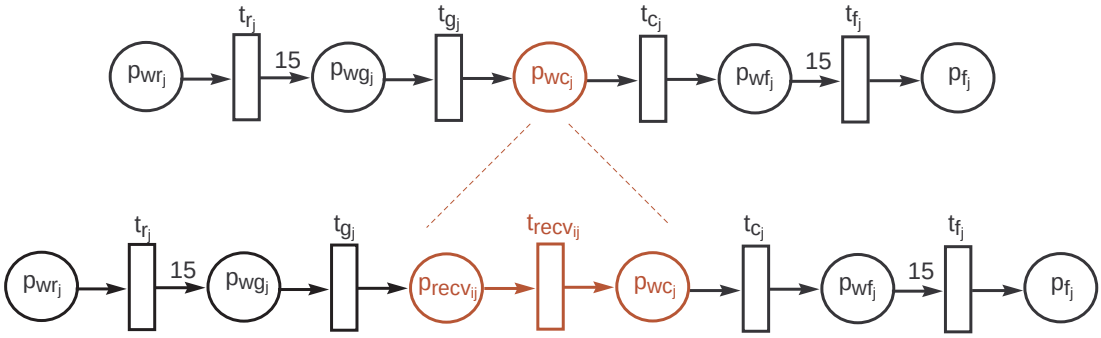

Figure 5.28: Modeling of the Receiving Task

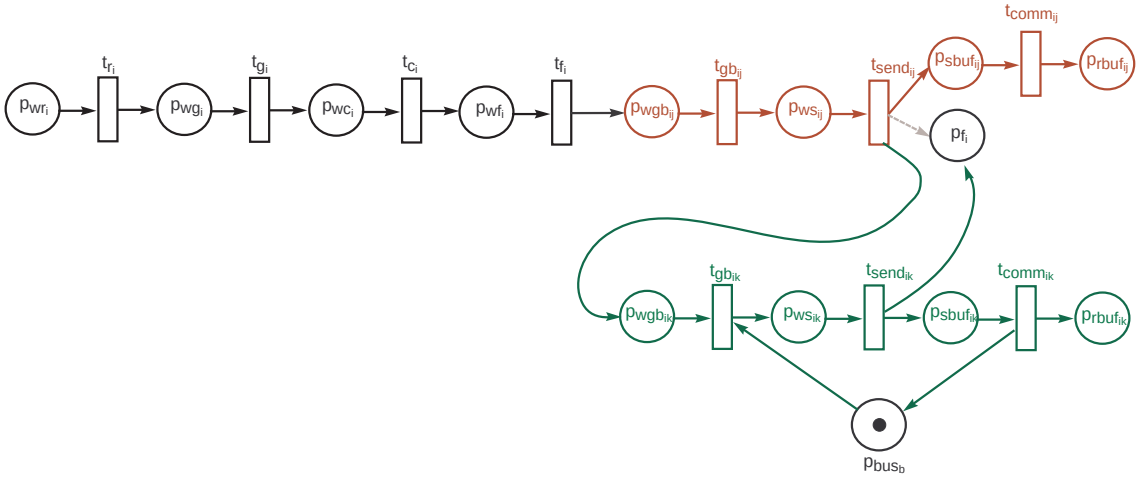

Figure 5.29: Modeling the Second Message from τ_i to τ_k

Figure 5.30 presents a communication graph that describes a communication pattern between five tasks located in two processors. This communication pattern is represented in the net of Figure 5.31. For sake of readability, this net not shows the processor. It is worth observing that the communication between tasks in the same processor is dealt with as precedence relation.

5.3.6 Modeling Dispatcher Overheads

An often neglected situation in software synthesis research is the dispatcher and timer interrupt handler overheads. If the computation time of tasks is small, and the respective deadline is short, this shortcoming may not reach all timing constraints. One solution, adopted in several works, for instance [103], considers that the WCET of tasks already includes this overhead. This solution is rather pessimistic, since it is not known how many preemptions will occur in each task before a schedule has been found.

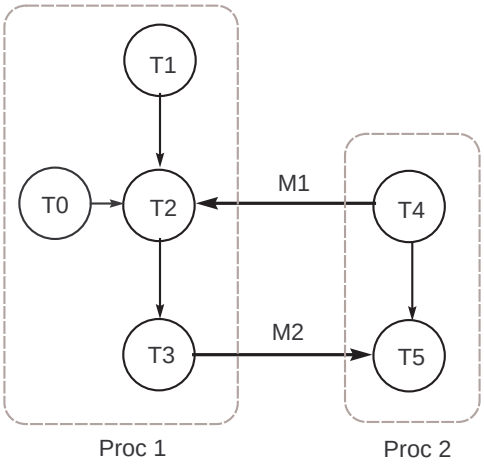

Figure 5.30: Communication Graph

In this work, both dispatcher and timer interrupt handler overheads are simply called dispatcher overhead.

On the other hand, the solution adopted in this work explicitly models the WCET of the dispatcher. In this case, the overhead is considered during the schedule generation, but only when needed, leading to a more realistic estimation for the system behavior.

This section shows the formal definition of the new task structure (preemptive and non-preemptive), this time including the modeling of the dispatcher overheads.

The aim of the dispatcher and interrupt handler is made clear in Section 6.2.

Dispatcher Overhead Block

The dispatcher overhead is captured in the grant-processor transition. When the task is non-preemptive, the timing interval of the grant-processor transition corresponds to the WCET of the dispatcher.

When the task is preemptive, the model is slightly more complex. In this case, the proposed modeling adopts the TPN with priorities.

The proposed model considers two grant-processor transitions: grant-processor-with-overhead (t_{gw_i}) and grant-processor-without-overhead (t_{gwo_i}). As it can be seen in Figure 5.32, the timing interval ($[\alpha, \alpha]$) for transition t_{gw_i} models such timing overhead.

Place $p_{proc_k T_i}$ states that task τ_i was last executed task by the processor $proc_k$. Transition t_{gwo_i} has as one of its pre-condition the place $p_{proc_k T_i}$.

The dispatcher overhead is considered in two situations: (1) when the next task to use the processor is different from the task that used the processor before; or (2) when a task instance ends its execution. The first situation is represented by the place

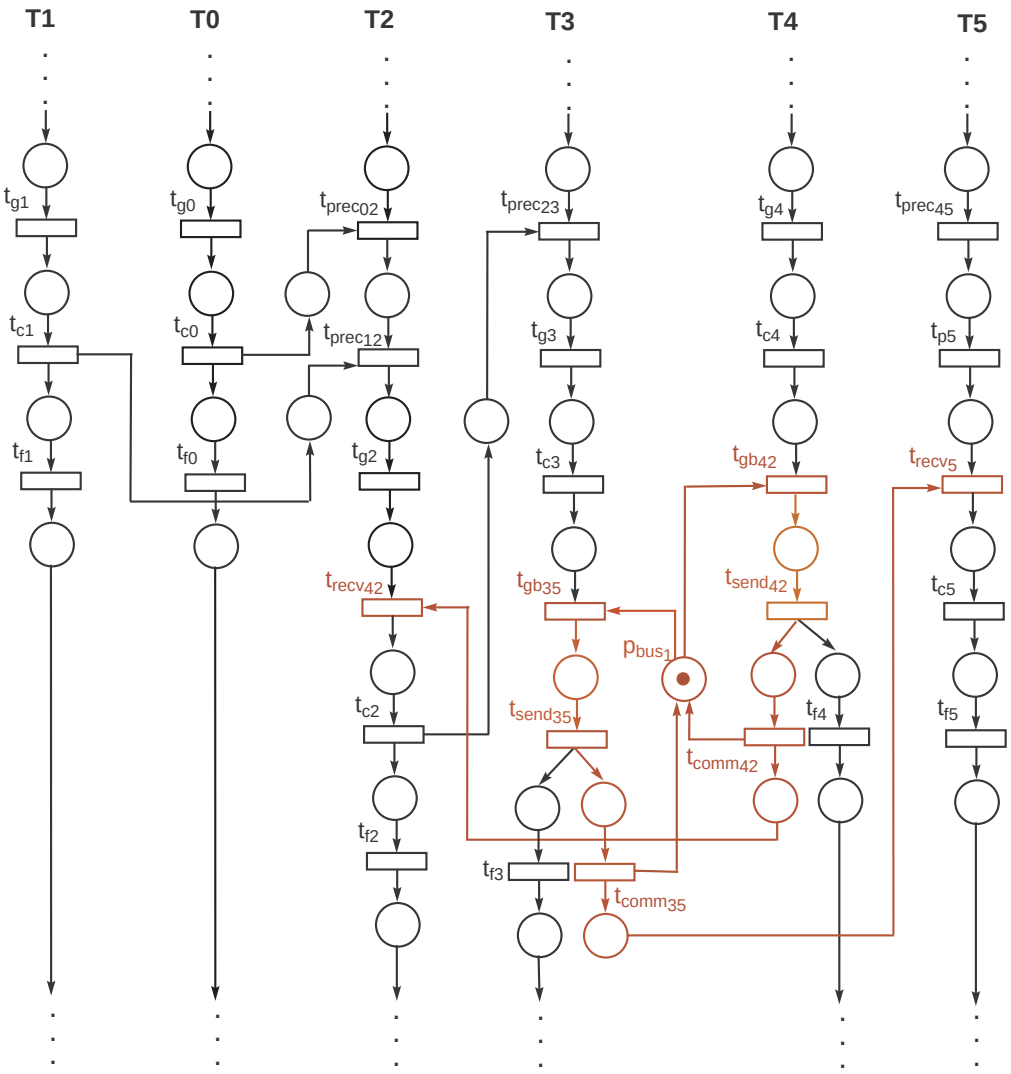

Figure 5.31: A Simple Example of Inter-processor Communication

$p_{proc_k T_i}$, where if such place is marked, it implies that the processor was lastly allocated to task τ_i. However, the second situation deserves an explanation. Supposing that a task instance i of task τ_j ends its execution, and the following task to be executed is the task instance $i+1$ of the same task τ_j. In this case, although the two instances are from the same task, the dispatcher calling is mandatory. In order to solve this problem the model considers two final transitions: one removes the marking in place $p_{proc_k T_i}$ (highest priority); and the other does not (lowest priority).

In spite of this block may seem complicated, it is worth remembering that this model is performed automatically. In other words, the user does not have to deal with such complex modeling.

All places $p_{proc_k T_j}$ are mutually exclusive marked, that is, just one must be marked

at a time. In order to assure this constraint, the model includes, for each task, n_k computation transitions $t_{c_{i_j}}$, $1 \leq j \leq n_k$, where n_k is the number of tasks that share the same processor $proc_k$. These n_k computation transitions are included in order to guarantee that the only place marked is $p_{proc_k T_i}$. Initially, no one of these places are marked.

In this proposed model, transition t_{gwo_i} has priority equal to zero (the highest) and transition t_{gw_i} has priority equal to one (lower than). In the same way, transitions $t_{c_{i_j}}$, $1 \leq j \leq n_k$ $i \neq j$, for each task τ_i, has highest priority (value equal to zero) related to transitions $t_{c_{i_i}}$.

For sake of readability, Figure 5.32 does not show computation transitions having an arc from them to place p_{proc_k} (meaning processor releasing).

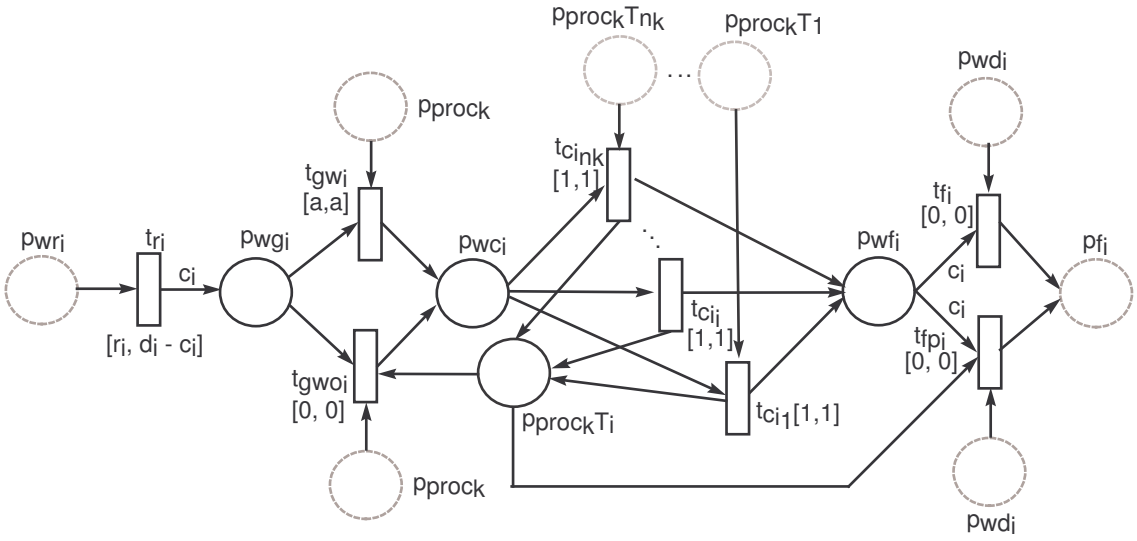

Figure 5.32: Building Block Dispatcher Overhead

The building block dispatcher overhead is a TPN $N_o = (P_o, T_o, F_o, W_o, M_{0_o}, I_o)$, such that:

⋆ $P_o = \{p_{wr_i}, p_{wg_i}, p_{wc_i}, p_{proc_k T_1}, \ldots, p_{proc_k T_{n_k}}, p_{wf_i}, p_{f_i}, p_{wd_i}, p_{proc_k}\}$. These places model the following:

p_{wr_i}: waiting for release time;

p_{wg_i}: waiting for processor granting;

p_{wc_i}: waiting for task computation;

$p_{proc_k T_i}$: states that task τ_i was the last to be allocated to the processor $proc_k$, for $1 \leq i \leq n_k$;

p_{wf_i}: waiting for task instance end;

p_{f_i}: end of a task instance;

p_{wd_i}: waiting for deadline missing.

p_{proc_k}: processor $proc_k$.

⋆ $T_o = \{t_{r_i}, t_{gw_i}, t_{gwo_i}, t_{c_{i_1}}, \ldots, t_{c_{i_{n^k}}}, t_{fp_i}, t_{f_i}\}$. These transitions model the following actions:

t_{r_i}: task releasing;

t_{gw_i}: processor granting with dispatcher overhead (lowest priority);

t_{gwo_i}: processor granting without dispatcher overhead (highest priority);

$t_{c_{i_j}}$: executes one task unit, removes the marking in place $p_{proc_k T_j}$ $(1 \leq j \leq n_k,\ j \neq i)$, releases the processor, and marks place $p_{proc_k T_i}$ (highest priority);

$t_{c_{i_i}}$: executes one task unit, releases the processor, and marks place $p_{proc_k T_i}$ (lowest priority);

t_{fp_i}: concluding the task computation, and removes the marking $p_{proc_k T_i}$ (higher priority than t_{f_i})

t_{f_i}: concluding the task computation (lower priority than t_{fp_i}).

⋆ Pre and post-conditions of transitions are:

$\bullet t_{r_i} = \{p_{wr_i}\};\quad t_{r_i}\bullet = \{p_{wg_i}\};$

$\bullet t_{gw_i} = \{p_{wg_i}, p_{proc_k}\};\quad t_{gw_i}\bullet = \{p_{wc_i}\};$

$\bullet t_{gwo_i} = \{p_{wg_i}, p_{proc_k T_i}, p_{proc_k}\};\quad t_{gwo_i}\bullet = \{p_{wc_i}\};$

$\bullet t_{c_{i_j}} = \{p_{wc_i}, p_{proc_k T_j}\};\quad t_{c_{i_j}}\bullet = \{p_{proc_k T_i}, p_{wf_i}, p_{proc_k}\};$

$\bullet t_{c_{i_i}} = \{p_{wc_i}\};\quad t_{c_{i_i}}\bullet = \{p_{proc_k T_i}, p_{wf_i}, p_{proc_k}\};$

$\bullet t_{fp_i} = \{p_{wf_i}, p_{wd_i}, p_{proc_k T_i}\};\quad t_{fp_i}\bullet = \{p_{f_i}\}.$

$\bullet t_{f_i} = \{p_{wf_i}, p_{wd_i}\};\quad t_{f_i}\bullet = \{p_{f_i}\}.$

⋆ $\forall f \in F_o:\ W_o(f) = \begin{cases} c_i & \text{if } f = (t_{r_i}, p_{wg_i}) \text{ or } f = (p_{wf_i}, t_{f_i}) \text{ or } f = (p_{wf_i}, t_{fp_i}) \\ 1 & otherwise \end{cases}$

⋆ $M_{0_o}(p_{proc_k}) = \beta;\quad M_{0_p}(p) = 0\ \forall p \in P \wedge p \neq p_{proc_k}.$

⋆ $I_o(t_{r_i}) = [r_i, d_i - c_i]$;
$I_o(t_{gwo_i}) = I_o(t_{f_i}) = I_o(t_{fp_i}) = [0, 0]$;
$I_o(t_{gw_i}) = [\alpha, \alpha]\ \alpha \in \mathbb{N}$;
$I_o(t_{c_{i_j}}) = [1, 1],\ 1 \leq j \leq n_k$.

The release time of task τ_i is assigned to the timing interval of transition t_{r_i}. The timing interval $[\alpha, \alpha]$ of transition t_{gw_i} comes from the WCET of the dispatcher and timer interrupt handler. All remaining timing intervals are constants. The same way, the arc weights ($(t_{r_i},\ p_{wg_i})$, $(p_{wf_i},\ t_{fp_i})$, and $(p_{wf_i},\ t_{f_i})$) come from the c_i (execution time) of task τ_i.

Figure 5.33 shows the TPN of tasks T_0 and T_1 considering the modeling of the dispatcher overhead. This figure shows that the dispatcher overhead is equal to two TTUs. It is worth noting that the dispatcher overhead block is just applicable in tasks that share the same processor.

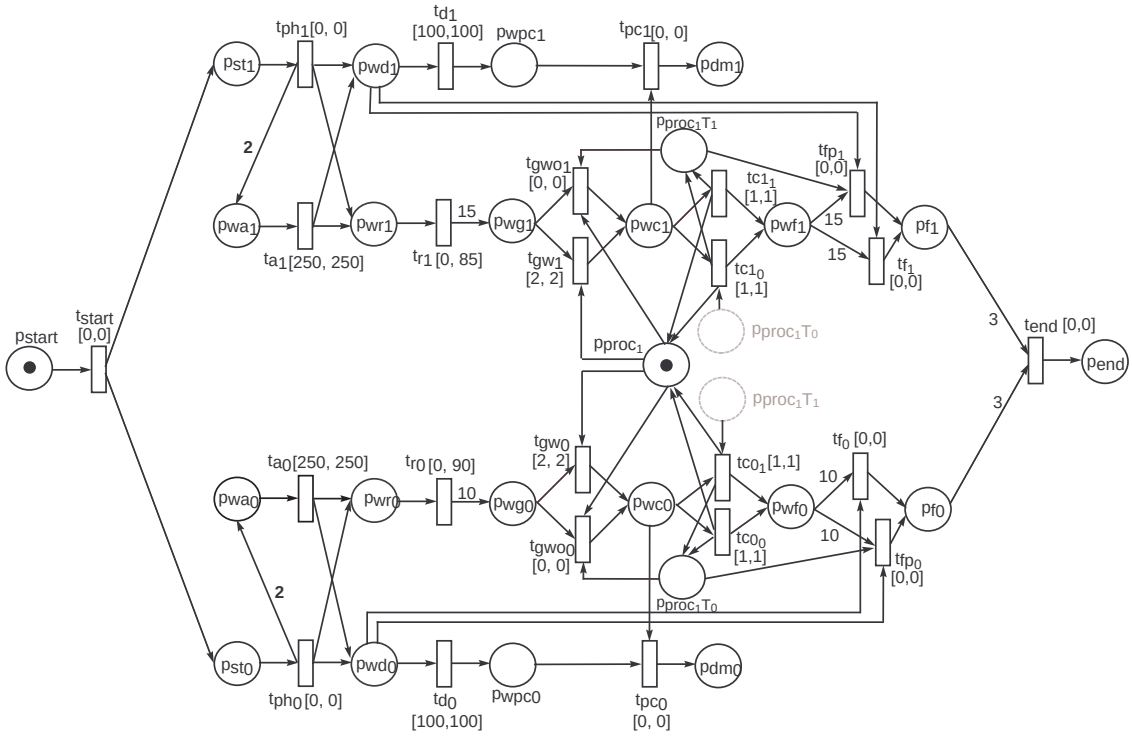

Figure 5.33: Tasks T_0 and T_1 modeled with dispatcher overhead

5.4 Analysis and Verification of the Model

This section provides qualitative analysis and properties verification of the model.

Verification is usually associated to a deterministic algorithm for checking if a model has a given property. Analysis, on the other hand, does not look for properties verification, instead, the analysis provides information about several properties. Furthermore, the results of the analysis may be used as a basis for verification algorithms.

5.4.1 Qualitative Analysis

INA [90] is a tool devoted to modeling, analysis and verification of place/transition and several classes of timed Petri nets. This tool was adopted for carrying out qualitative analysis as well as for properties verification. This section presents qualitative analysis result obtained for a basic model, that is, a model without considering inter-task relations provided by designers. A basic model does not consider the deadline checking block. This is a weak limitation, since the deadline checking block represents undesirable states.

The result of the qualitative analysis is as follows:

- **Bounded**. As presented before in Section 4.4, a net is bounded if no place is allowed to accumulate an infinite number of tokens. Therefore, this property indicates a finite state space. This property is important since it guarantees that the search-based algorithm (that traverses the state space) will stop.

- **Deadlock-freedom**. A net satisfies the deadlock-freedom property, if the maximal trap (where in this case it is not a proper subset of a trap) in each siphon is sufficiently marked. A set of places is sufficiently marked, if it contains a place with sufficiently many tokens to enable all its post-transitions. This property is essential in concurrent systems, where deadlock is a situation not desired. This property is also important to guarantee that the model does not introduce any deadlock. Of course if the user specifies a system with precedence relations that form a cycle, certainly a deadlock will occur. However, such deadlock is introduced by the user, not from the proposed model. As presented before in this subsection, the basic model does not consider the deadline checking block, since the structure of this block, *per se*, is a deadlock.

- **Liveness**. Liveness guarantees the absence of deadlocks. Moreover, it is a stronger condition than deadlock-freedom. A net may be deadlock-free but not live. The converse is always true. A transition t is live if for all reachable marking, there exists a transition firing sequence σ leading to a marking m' in which

t is enabled. A net is live if all transitions are live. The same remark about the deadline checking block is valid for this property.

One of the properties not found in the proposed model is *structurally boundedness*. A net is structurally bounded, if it is bounded for every initial marking. Furthermore, this property is stronger than boundedness.

A single model representing two tasks (T_0 and T_1) was developed for investigating both boundedness and structural boundedness properties. Figure 5.34 presents such model. First of all, structural properties do not consider priority, initial marking, and time. Observing the behavior of the net (without such features), it has been found out that there is a possibility of tokens to be accumulated into places $p_{proc_kT_0}$ and $p_{proc_kT_1}$. This situation occurs if transitions t_{c_0} and t_{c_1} is chosen for firing instead of transitions $t_{c_{0_1}}$ and $t_{c_{1_0}}$. In this case, this net systems cannot be structurally bounded.

As presented in subsection 5.4.2, the initial marking will stabilize at $m(p_{proc_kT_0}) + m(p_{proc_kT_1}) = 1$ when considering priorities. However, it is not structurally guaranteed, but only when analyzing behavioral properties, since such places are not covered by P-invariants. Thus, this net system is bounded but not structurally bounded.

The model presented in Figure 5.34 has no inter-task relations. When introducing such relations, there is no guarantee of liveness and deadlock-freedom preservation. For instance, if the user specifies a cyclic precedence relation, a deadlock is introduced. In the same way, this may occur with inter-processor communication. Therefore, a model verification is required.

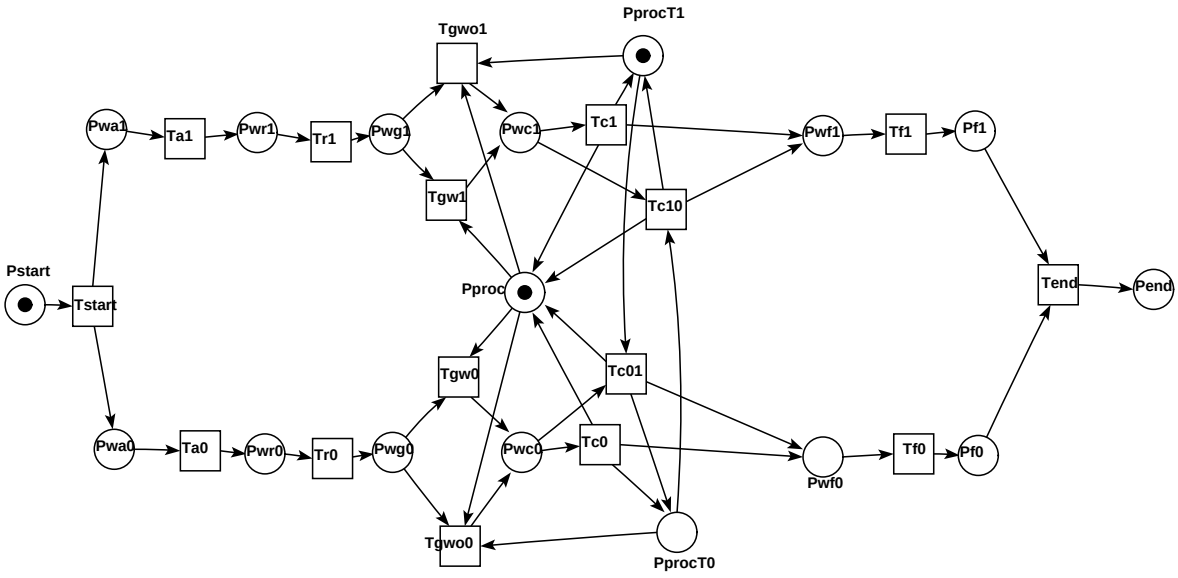

Figure 5.34: Model for Verifying Mutual Exclusive Marking

5.4.2 Modeling Verification by Model Checking

This section presents several verifications using the model checking method. For more information about model checking, the interested reader is referred to Appendix A.

This work verifies several important properties, such as:

- verifying that most recent allocation of processor to tasks are mutually exclusive. The verification of this property states that no more than one task was the last to be allocated to the processor;
- verifying that the processor is used by one task at a time;
- verifying that exclusive access (of tasks) to critical regions is guaranteed;
- verifying that the preceded task can only start its execution after the preceding task had finished.

The verifications of such properties are depicted below.

Mutual Exclusive Marking

As presented at Section 5.3.6, the set of places $\{p_{proc_k T_i},\ 1 \leq i \leq n_k\}$ is mutual exclusive marked, that is, whenever one of these places has a token the other ones are not marked. In order to verify this property, model checking is adopted.

Figure 5.34 shows a net system representing two tasks: T_0, and T_1. Using CTL (Computation Tree Logic), the formula representing this property is

$$AG\neg(p_{procT_0} \wedge p_{procT_1})$$

This formula states that for all paths (A - Always) the property holds in every state (G - Globally). In other words, this property should be satisfied for all system execution. The property states that places p_{procT_0}, and p_{procT_1} are not simultaneously marked.

This formula was verified using the INA tool, and the result was `TRUE`, that is, the model preserves the mutual exclusive condition between the set of places $\{p_{proc_k T_i},\ 1 \leq i \leq 2\}$. Since the model is automatically generated, it may be argued that this property is satisfied for all places $\{p_{proc_k T_i},\ 1 \leq i \leq n_k\}$.

Section B.1 (Appendix B) presents the steps performed by the INA tool for checking this property.

If the model consists of three tasks, the formula would be

$$AG(((p_{procT_0} \wedge \neg(p_{procT_1} \vee p_{procT_2})) \vee (p_{procT_1} \wedge \neg(p_{procT_0} \vee p_{procT_2})) \vee (p_{procT_2} \wedge \neg(p_{procT_0} \vee p_{procT_1}))$$

Processor Utilization

The aim of this verification is to check if the model correctly represents that a specific processor (represented by place p_{proc_k}) is used by just one task at a time. The solution adopted checks if all places immediately after the grant-processor transitions are mutually-exclusive marked, i.e., just one of these places is marked at a time. This set of places is $\{p_{wc_i},\ 1 \leq i \leq n_k\}$.

Considering the model in Figure 5.34, the formula representing this property is

$$AG\neg(p_{wc_0} \wedge p_{wc_1})$$

This formula was also verified using the INA tool, and the result was `TRUE`, that is, the model preserves the property of exclusive access to the respective processor. This verification is performed considering two tasks. However, as the modeling is performed automatically, it may be argued that this property is satisfied for all places $p_{wc_i},\ 1 \leq i \leq n_k$.

Section B.2 (Appendix B) presents the steps performed by the INA tool for this verification.

If the model consists of three tasks, the formula would be

$$AG(\neg(p_{wc_0} \wedge p_{wc_1}) \wedge \neg(p_{wc_0} \wedge p_{wc_2}) \wedge \neg(p_{wc_1} \wedge p_{wc_2}))$$

Precedence Relation

Precedence relations are defined between pair of tasks. So, let us suppose that the net system represents the T_1 PRECEDES T_0 precedence relation. Figure 5.35 presents a simplified model for verifying the precedence relation.

The aim of this verification is to check if after defining a precedence relation both tasks are not executing at the same time.

The solution adopted checks if all places immediately before the task computation transitions are mutually-exclusive marked, namely, just one of these places is marked at a time. In this modeling specific situation, this set of places is $\{p_{wc_i},\ 1 \leq i \leq n_k\}$.

Using CTL, the formula representing this property is

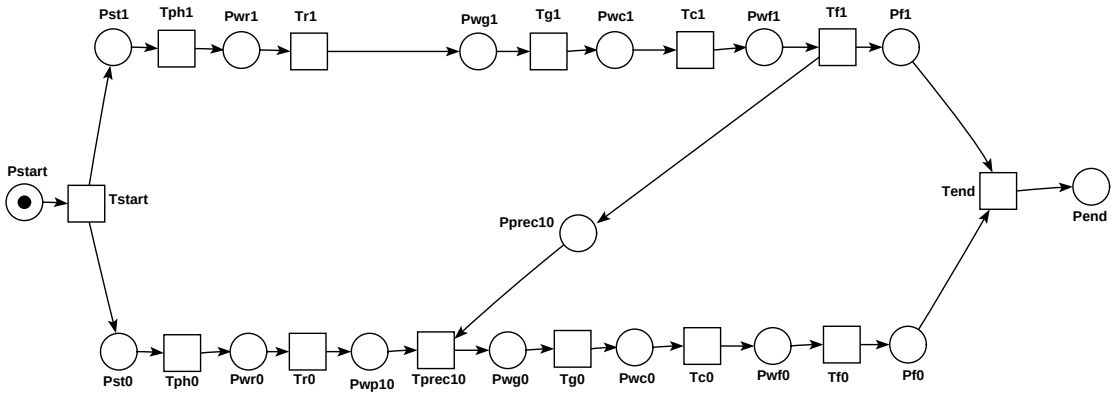

Figure 5.35: Model for Verifying Precedence Relation

$$AG\neg(p_{wc_0} \wedge p_{wc_1})$$

This formula was also verified using the INA tool, and the result was `TRUE`.

Another way to perform the same verification is by using another formula. This time, the aim is to verify if there is no path where p_{wg_0}, p_{wc_0}, and p_{wf_0} are marked and p_{f_1} is not marked. The formula is

$$AG\neg(p_{wg_0} \wedge p_{wc_0} \wedge p_{wf_0} \wedge \neg p_{f_1})$$

The result of this formula is `TRUE`, that is, the model preserves the property of precedence relation in all paths starting from the initial state. Section B.3 (Appendix B) shows the steps performed by the INA tool for this verification.

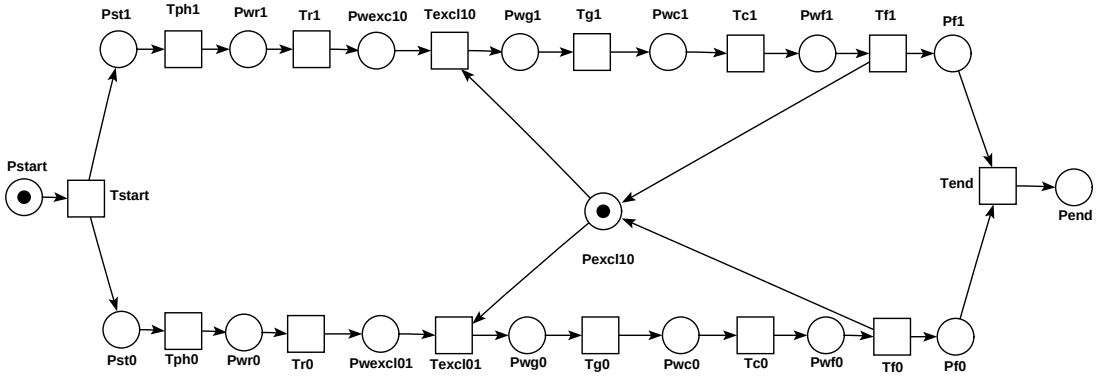

Figure 5.36: Model for Verifying Exclusion Relation

Exclusion Relation

Exclusion relations are also defined between pair of tasks. Thus, supposing that T_1 EXCLUDES T_0 exclusion relation is defined, Figure 5.36 presents a simplified model for verifying the exclusion relation.

In order to verify if both tasks are not simultaneously executed, the solution adopted checks if places p_{wc_0} and p_{wc_1} are not simultaneously marked at the same time.

Using CTL, the formula representing this property is

$$AG\neg(p_{wc_0} \wedge p_{wc_1})$$

The result of applying such formula to the INA tool was `TRUE`, that is, the model satisfies the property of exclusion relation in all paths starting from the initial state. Section B.4 (Appendix B) explains the steps performed by the INA tool for this verification.

5.5 Summary

This chapter introduced three important aspects of modeling embedded hard real-time systems, namely, formal model, specification, and the process for obtaining the model from the specification.

The first section has shown the formal model syntax and semantics. The syntax is based on an extension of time Petri nets considering priorities, source task code, and energy consumption. The model semantics is described by enabling and timing rules and is represented by a timed labeled transition systems. This section presented a formal definition of time Petri net as well as the set of enabled transitions, clocks (which are implicit for each enabled transition), states of a time Petri net, the set of fireable transitions with its respective firing domain, reachable states, timed labeled transition system, and feasible firing schedule (which is a timed labeled transition system where the final state is specified and well-known). A source code transition labeling has also been considered. Additionally, priorities have been defined for extending the basic timed model as well as energy consumptions related to executions of activities. These extensions required the redefinition of state, fireable transition set, and reachable states.

The second section described the specification model, explaining in detail all components of this model. The specification model is divided in two parts: (i) specification of constraints; and behavioral specification. Periodic task timing constraints, inter-task relations, and allocation of tasks to processors were defined in the specification of con-

straints. This section also explained how to translate a sporadic task into an equivalent periodic one, how to deal with subtasks, and scheduling methods (preemptive or non-preemptive). In the behavioral specification, the source code of each task is specified in C language augmented with communicating constructs. The syntax and semantics of these communication constructs were detailed. If such communication occurs in the same processor, it is considered as a precedence relation. However, if communicating tasks are in different processors, a special communication task is added, and proper precedence relations are also included. Finally, this section showed the components of a complete specification using a simple example.

The modeling of embedded hard real-time systems using time Petri net formalism was the main aim of the third section. The solution adopted is based on the composition of building blocks. The building blocks are (i) periodic task arrival; (ii) task structure, where either preemptive or non-preemptive scheduling methods are considered; (iii) deadline checking; (iv) inter-processor sending message; (v) resources, such as processors and buses; (vi) fork; and (vii) join. These blocks are composed by applying several operators, such as merging, addition and refinement of places, addition and removing of arcs, and net union. This section also has shown how to model inter-task relations, in this case, precedence and exclusion relations. It also explained how to model inter-processor communication, that is, message sending and message receiving considering that tasks are allocated to different processors. Next, in order to consider the dispatcher and timer interrupt handler overheads, this section provided a specific building block for modeling such overhead so that it may lead to a more realistic estimation of the system behavior. Finally, this section presented some analysis and properties verification considering the proposed model. The model was analyzed and it was found out that the model has interesting properties. A set of properties of interest were also verified: (i) if conditions that assert last allocation of processor to tasks are mutually exclusive; (ii) if the processor is used by one task at a time; (iii) if the exclusive access of tasks to critical regions is guaranteed; and (iv) if the preceded task can only start its execution after the preceding task finished. In this case, all such properties are satisfied by the model.

Chapter 6

Software Synthesis

Software synthesis aims to generate the source program code with lower overheads and satisfying all constraints. This chapter is divided in two sections: scheduling, and code generation.

Embedded hard real-time systems have stringent timing constraints that must be satisfied. Additionally, when considering safety or timing-critical systems, predictability is one of the main concerns. Scheduling plays an important role for attaining such constraints in a predictable way.

Starting from the found feasible pre-runtime schedule, a C-code is generated. There are two ways for this code generation, namely, with and without multiple operational modes. Each mode represents an alternative schedule, which is activated by a specific condition.

Although the scheduling synthesis framework provides schedules considering multiprocessors, the code generation for this architecture is beyond the scope of this work.

6.1 Scheduling Synthesis

Starting from the time Petri net model, the proposed scheduling synthesis framework generates and analyzes the timed labeled transition system (TLTS), resultant of this model. It also analyzes the TLTS in order to find a pre-runtime schedule, provided that such schedule exists.

This work uses *state space exploration* for automatic verification of finite-state systems [38]. It consists of recursively checking all successor states, starting from a given initial state, by executing all enabled transitions at each state. In spite of the fact that a scheduling can be found using this strategy, it may be limited by the excessive size of

its state space. This problem comes up due to the analysis based on the interleaving of concurrent activities. This exponential growth is known as the *state explosion problem* [38, 97]. The proposed approach tackles this problem by applying techniques for state space reduction, and a depth-first search algorithm.

The proposed scheduling policy is *pre-runtime scheduling*, where schedules are computed entirely off-line. As presented previously, this strategy has advantages over others, mainly when adopting arbitrary precedence and exclusion relations.

This section starts by showing how to minimize the state space size by modeling dependencies between tasks, applying a partial-order reduction technique, and removing undesirable states. Next, it presents a depth-first search algorithm for finding, if one exists, a feasible firing schedule. Finally, the proposed algorithm is applied to a simple time Petri net model.

6.1.1 Minimizing State Space Size

The analysis of n concurrent events, using state space exploration, needs verifying all $n!$ interleaving possibilities of these events, unless there exist dependencies between these events. The proposed method models explicitly such dependencies. For instance, resources (including processors) with resource granting and releasing, precedence and mutual exclusion relations, markings representing properties to be avoided or verified, and synchronizations. The modeling methodology itself aids in minimizing the state space size. Besides, this section presents two other ways of minimizing the state space size, namely, partial-order reduction and elimination of undesirable states.

Partial-Order Reduction

When generating a timed labeled transition system of a time Petri net, the interleaving of activities is the fundamental point to be considered in the analysis of the state space explosion problem. However, if activities can be executed in any order, in which the system always reaches the same state, these activities are *independent*. In other words, it does not matter in which order the activities are executed. Partial-order reduction methods are based on the independence of activities. The reduction is obtained by throwing away one of these interleaving, i.e., executing a subset of the enabled activities, usually called a *persistent set*. The interested reader is referred to [38] for a good overview.

The correctness of partial-order methods is based on the *diamond property* (Figure 6.1(a)). Here, if two activities are independent, the order between them does not

matter. These techniques have been studied within the context of *untimed* systems, whereas in the context of timed systems little progress has been made. The main problem, in accordance with [54] is the global nature of time, which makes all clocks in the system dependent on each other. The standard semantics for time Petri nets implicitly stores the firing order of transitions in the timing constraints.

In case of Figure 6.1, the standard semantics defines the state as the composite between marking and clock. Analyzing the time Petri net (Figure 6.1(b)) it can be seen that transitions t_1 and t_2 are really independent. However, suppose that both transitions have been enabled at different time instants. Although they are independent, the final state will not be the same, since the clocks are different. If it happens, the state space will form a tree (Figure 6.1(c)) and the diamond property is never reached.

Lilius [54] studied such partial-order techniques and derived a semantics for time Petri nets that does not store the firing order into timing constraints. Therefore, it becomes possible to directly apply the theory of partial-order reductions to time Petri nets. He defined state class, which is a pair (m, I), where $m \subseteq P$, and I is a set of constraints over T. A state class describes the constraints on the possible firing times of the enabled transitions in a specific marking. I can be represented as matrices, where each matrix entry represents an inequation.

Starting from these definition, Lilius shows that independent transitions in time Petri nets may be defined in the following way. Two transitions t_1 and t_2 are independent, iff for all states s of the state space:

(i) if t_1 is fireable in s and $s \xrightarrow{t_1} s'$, then t_2 is fireable in s iff t_2 is also fireable in s'; and

(ii) if t_1 and t_2 are fireable in s, then there is a unique state s' such that $s \xrightarrow{t_1;t_2} s'$, and $s \xrightarrow{t_2;t_1} s'$.

In the example shown at Figure 6.1(b), the firing of t_1, t_2 or t_2, t_1 leads to the same state class.

In the same way as Lilius [54], this work considers that two transitions are independent if one is not in conflict with the other, i.e., when one is fired it does not disable the other. Analyzing the specific transitions of the models generated by this work, it was found out that arrival transitions are independent on other arrival transitions. The same occurs with release, precedence, computation, final, send-message, and receive-message transitions. Thus, when one of these *class of transitions* is fired the other continues fireable. On the other hand, processor granting, and exclusion are examples of *dependent* transitions.

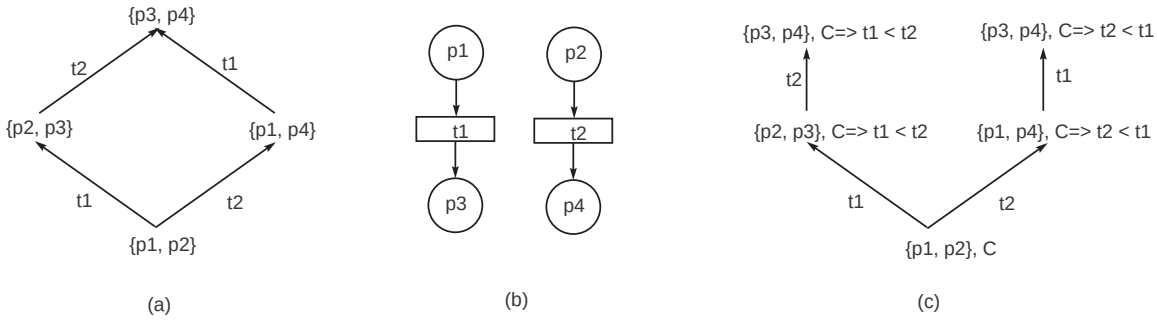

Figure 6.1: Standard semantics of timed systems: (a) diamond property; (b) a time Petri net model; (c) a reachability tree

As it can be observed, there is a causal relationship between release and arrival, processor-granting and release, computation and processor-granting, and so on. In order to define the *persistent set* for each state, and considering the specific adopted model, the causal relationship, the independent, and dependent class of transitions, this method proposes to give each class of transitions a different *choice-priority* level. In this case, the independent class of transitions have the highest choice-priorities. The dependent class of transitions, such as processor-granting, and exclusion, have lowest choice-priority. Therefore, when changing from one state to another, it is sufficient to analyze the class with the highest choice-priority and pruning the other ones, in such a way that the exploration occurs only in part of the state space. When all such independent transitions are executed, the state will certainly be the same, since the order between them does not matter. The priority-choice is detailed in Table 6.1. In this table, transitions like deadline-checking and deadline-missing are not considered since these transitions are undesirable and will not be fired.

Table 6.1: Choice-priorities for each transition class

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

This reduction method is not general, rather it is specific for the proposed model. This reduction is important due to two reasons: (i) it reduces the amount of storage (the

```
1 scheduling-synthesis(S,M^F,TPN, V_max)
2 {
3   if (S.M = M^F) return TRUE;
4   tag(S);
5   PT = remove-undesirable(partial-order(firable(S,V_max)));
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,V_max)){
10      add-in-trans-system (S,S',t,θ);
11      return TRUE;
12    }
13  }
14  return FALSE;
15 }
```

Figure 6.2: Scheduling Synthesis Algorithm (Timing and Energy Constraints)

amount of main memory is the major limiting factor of most state space exploration algorithms); and (ii) when the system does not have a feasible schedule, it returns more rapidly.

Undesirable States

Section 5.3.3 presents how to model undesirable states, for instance, states that represent missed deadlines. The proposed method is interested in schedules that do not reach any of these undesirable states. Therefore, undesirable states are simply discarded. As all transitions are annotated with their class, it is easy to check the transitions which their firing are leading to an undesirable state. For instance, all deadline-checking transitions should not fire unless deadlines are not met.

6.1.2 Pre-Runtime Scheduling Algorithm

The proposed algorithm (Figure 6.2) is a depth-first search method that traverses a TLTS. The *stop criterion* is obtained whenever the desirable final marking M^F is reached. The state space is not completely generated, because of the search method adopted and the state space size minimization techniques.

Considering that the Petri net model is bounded (Section 5.4), and the timing constraints are discrete, this implies that the TLTS is finite and thus the proposed algorithm always finishes.

When the algorithm reaches the desired final marking (M^F), it implies that a feasible schedule satisfying both timing and energy constraints was found (line 3). It is worth observing that the fireable function (line 5) takes into account timing, energy, and priority. The state space generation is modified (also line 5) to incorporate the state space pruning (partial-order and undesirable states pruning). `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 reduced state space is visited only when the system does not have a feasible schedule (line 14), where the algorithm returns `FALSE`.

The tagging scheme is needed to visit a state not more than once. This test (line 9) is needed so that termination can be guaranteed in reasonable time. In order to verify such situation, an experiment was performed considering an unmanned ground vehicle (Chapter 8) without such tagging scheme. The result is that a feasible schedule was found after analyzing more than 2 billion states in more than 48 hours. Using the tagging scheme, the same schedule was found in just 2.5 seconds after analyzing 14,761 states. However, this test may cause two problems. The first problem concerns the (new) search that has to be made in the set of all previously visited states. This may be very time consuming. The second problem is related to the set of visited states, which may also be very large.

For minimizing the first problem, a binary-tree search was adopted. The first attempt was to apply a hashing table technique. However, this solution caused lots of collision. Thus, in this specific situation, this solution was not so efficient. The second problem is beyond the scope of this work.

This algorithm has an important characteristic, namely, it is deadlock and starvation-free. The aim of this algorithm is to start from the initial state and find the desirable final state. Therefore, it is impossible to find such final state with either deadlock or starvation. When considering a single processor architecture, this deadlock/starvation-free is also guaranteed by the modeling based on compositions of building blocks. Since the modeling is automatic, there is no way of introducing deadlocks. However, when considering a multi-processor architecture, the communication between tasks in different processors may introduce deadlocks. Nevertheless, a technique for deadlock avoidance is not implemented in this work. See more information about this subject on Section 9.3 at Chapter 9.

6.1.3 Application of the Algorithm

The simple task set presented at Table 5.2 (Section 5.3.1 at Chapter 5) produces the TPN model outlined in Figure 6.3. Table 6.2 shows the application of the proposed algorithm in such TPN model. For illustrative purpose, it is worth noting that this

Table 6.2: Illustrative Example (Timing and Energy Constraints)

#	st	ET	C	FT	PT	trans+time	Energy
1	0	{tstart}	{0}	{tstart}	{tstart}	{tstart,0}	0
2	1	{tph1,tph2}	{0,0}	{tph1,tph2}	{tph1,tph2}	{tph1,0}	0
3	2	{tph2,tr1,ta1,td1}	{0,0,0,0}	{tph2,tr1}	{tph2}	{tph2,0}	0
4	3	{tr1,ta1,ta2,td1,td2}	{0,0,0,0,0}	{tr1}	{tr1}	{tr1,0}	0
5	4	{tp1,ta1,ta2,td1,td2}	{0,0,0,0,0}	{tp1}	{tp1}	{tp1,0}	0
6	5	{tr2,tc1,ta1,ta2,td1,td2}	{0,0,0,0,0,0}	{tr2,tc1}	{tr2}	{tr2,2}	0
7	6	{tc1,ta1,ta2,td1,td2}	{2,2,2,2,2}	{tc1}	{tc1}	{tc1,0}	2
8	7	{tf1,ta1,ta2,td1,td2}	{0,2,2,2,2,2}	{tf1}	{tf1}	{tf1,0}	2
9	8	{tp2,ta1,ta2,td2}	{0,2,2,2}	{tp2}	{tp2}	{tp2,0}	2
10	9	{tc2,ta1,ta2,td2}	{0,2,2,2}	{tc2}	{tc2}	{tc2,2}	4
11	10	{tf2,ta1,ta2,td2}	{0,5,5,5}	{tf2}	{tf2}	{tf2,0}	4
12	11	{ta1,ta2}	{5,5}	{ta2}	{ta2}	{ta2,2}	4
13	12	{ta1,ta2,tr2,td2,}	{6,0,0,0}	{ta1,tr2}	{ta1}	{ta1,2}	4
14	13	{ta1,ta2,tr1,tr2,td1,td2}	{0,2,0,2,0,2}	{tr1,tr2}	{tr1,tr2}	{tr1,0}	4
15	14	{ta1,ta2,tr2,td1,td2,tp1}	{0,2,2,0,2,0}	{tr2,tp1}	{tr2}	{tr2,0}	4
16	**15**	**{ta1,ta2,td1,td2,tp1,tp2}**	**{0,2,0,2,0,0}**	**{tp1,tp2}**	**{tp1,tp2}**	**{tp1,0}**	4
17	16	{ta1,ta2,td1,td2,tc1}	{0,2,0,2,0}	{tc1}	{tc1}	{tc1,2}	6
18	17	{ta1,ta2,td1,td2,tf1}	{0,2,0,2,0}	{tf1}	{tf1}	{tf1,0}	6
19	18	{ta1,ta2,td2,tp2}	{2,4,4,0}	{tp2}	{tp2}	{tp2,0}	6
20	19	{ta1,ta2,td2,tc2}	{2,4,4,0}	{ta2,td2}	{ta2}	{ta2,2}	6
21	**20**	**{ta1,ta2,td2,tr2}**	**{4,0,6,2}**	**{td2}**	**{td2}**	**{td2,0}**	6
22	**15**	**{ta1,ta2,td1,td2,tp1,tp2}**	**{0,2,0,2,0,0}**	**{tp1,tp2}**	**{tp2}**	**{tp2,0}**	4
23	16	{ta1,ta2,td1,td2,tc2}	{0,2,0,2,0}	{tc2}	{tc2}	{tc2,2}	6
24	17	{ta1,ta2,td1,td2,tf2}	{0,2,0,2,0}	{tf2}	{tf2}	{tf2,0}	6
25	18	{ta1,ta2,td1,tp1}	{3,5,3,0}	{tp1}	{tp1}	{tp1,0}	6
26	19	{ta1,ta2,td1,tc1}	{3,5,3,0}	{ta2}	{ta2}	{ta2,2}	6
27	20	{ta1,ta2,td1,tc1,tr2}	{4,0,4,1,0}	{tc1}	{tc1}	{tc1,1}	8
28	21	{ta1,ta2,td1,tr2,tf1}	{4,0,4,0,0}	{tf1}	{tf1}	{tf1,0}	8
29	22	{ta1,ta2,tr2}	{4,1,1}	{tr2}	{tr2}	{tr2,1}	8
30	23	{ta1,ta2,tp2}	{5,2,0}	{tp2}	{tp2}	{tp2,0}	8
31	24	{ta1,ta2,tc2}	{5,2,0}	{tc2}	{tc2}	{tc2,2}	10
32	25	{ta1,ta2,tf2}	{5,2,0}	{tf2}	{tf2}	{tf2,0}	10
33	26	{ta1,ta2}	{8,5}	{ta1}	{ta1}	{ta1,1}	10
34	27	{ta2,td1,tr1}	{5,0,0}	{tr1}	{tr1}	{tr1,0}	10
35	28	{ta2,td1,tp1}	{5,0,0}	{tp1}	{tp1}	{tp1,0}	10
36	29	{ta2,td1,tc1}	{5,0,0}	{ta2}	{ta2}	{ta2,1}	10
37	30	{td1,tc1,td2,tr2}	{1,1,0,0}	{tc1}	{tc1}	{tc1,1}	12
38	31	{tf1,td1,td2,tr2}	{0,1,0,0}	{tf1}	{tf1}	{tf1,0}	12
39	32	{td2,tr2}	{1,1}	{tr2}	{tr2}	{tr2,1}	12
40	33	{td2,tp2}	{2,0}	{tp2}	{tp2}	{tp2,0}	12
41	34	{td2,tc2}	{2,0}	{tc2}	{tc2}	{tc2,2}	14
42	35	{td2,tf2}	{2,0}	{tf2}	{tf2}	{tf2,0}	14
43	**36**	**{tend}**	**{0}**	**{tend}**	**{tend}**	**{tend,0}**	14

model does not consider dispatcher overhead.

This table shows the number of states searched, the respective state visited (which can be returned by backtracking as we can see at state 15), the enabled transition set, the clock set, the fireable transition set, the post-pruned fireable transition set, the transition to be fired and its respective time, and the energy accumulated up to this state. It is considered that both tasks consume $2nJ$.

In this table, at state 13, two processor-granting transitions (tp_1 and tp_2) are fireable. The possible execution of task $T1$ (choosing tp_1 for firing) is a wrong choice since, after that, task $T2$ misses its deadline (state 17). The algorithm *backtracks* to state 13 and tries another alternative, that is, to grant the processor to the task $T2$ (firing tp_2). This new decision leads to a feasible schedule, since in the state 29 the firing of transition t_{end} reaches the desired final marking (M^F). Thus, a feasible scheduling is found after analyzing 35 states, where the minimum number of states is 30.

Therefore, the feasible firing schedule found to the TPN model of Figure 6.3, ex-

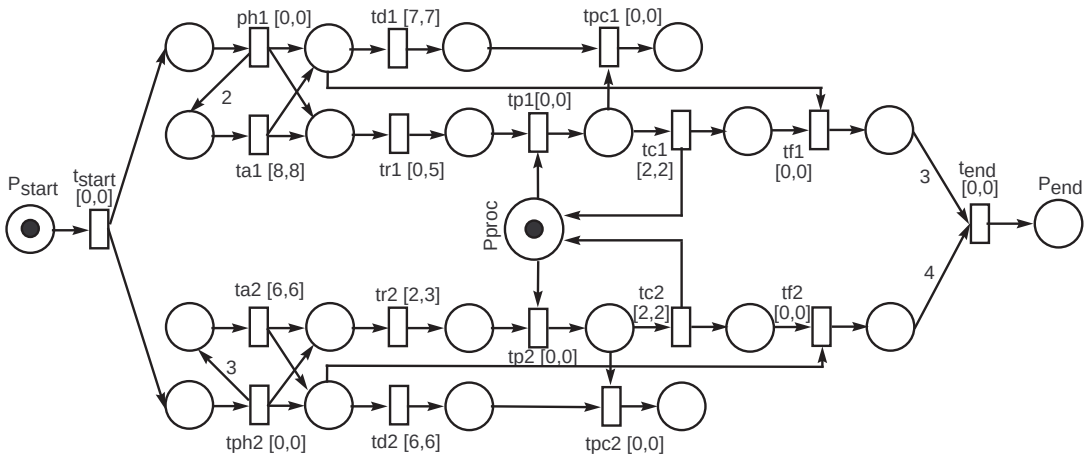

Figure 6.3: TPN for the task set in Table 5.2

pressed as a TLTS (Definition 5.9), is: $s_0 \xrightarrow{(t_{start},0)} s_1 \xrightarrow{(t_{ph1},0)} s_2 \xrightarrow{(t_{ph2},0)} s_3 \xrightarrow{(t_{r1},0)} s_4 \xrightarrow{(t_{p1},0)} s_5 \xrightarrow{(t_{r2},2)} s_6 \xrightarrow{(t_{c1},0)} s_7 \xrightarrow{(t_{f1},0)} s_8 \xrightarrow{(t_{p2},0)} s_9 \xrightarrow{(t_{c2},2)} s_{10} \xrightarrow{(t_{f2},0)} s_{11} \xrightarrow{(t_{a2},2)} s_{12} \xrightarrow{(t_{a1},2)} s_{13} \xrightarrow{(t_{r1},0)} s_{14} \xrightarrow{(t_{r2},0)} s_{15} \xrightarrow{(t_{p2},0)} s_{16} \xrightarrow{(t_{c2},2)} s_{17} \xrightarrow{(t_{f2},0)} s_{18} \xrightarrow{(t_{p1},0)} s_{19} \xrightarrow{(t_{a2},2)} s_{20} \xrightarrow{(t_{c1},1)} s_{21} \xrightarrow{(t_{f1},0)} s_{22} \xrightarrow{(t_{r2},1)} s_{23} \xrightarrow{(t_{p2},0)} s_{24} \xrightarrow{(t_{c2},2)} s_{25} \xrightarrow{(t_{f2},0)} s_{26} \xrightarrow{(t_{a1},1)} s_{27} \xrightarrow{(t_{r1},0)} s_{28} \xrightarrow{(t_{p1},0)} s_{29} \xrightarrow{(t_{a2},1)} s_{30} \xrightarrow{(t_{c1},1)} s_{31} \xrightarrow{(t_{f1},0)} s_{32} \xrightarrow{(t_{r2},1)} s_{33} \xrightarrow{(t_{p2},0)} s_{34} \xrightarrow{(t_{c2},2)} s_{35} \xrightarrow{(t_{f2},0)} s_{36} \xrightarrow{(t_{end},0)} s_{37}$.

Another two ways to see this TLTS is by a timing diagram, and an energy chart. These new views of the TLTS are explained in Chapter 7.

6.2 Scheduled Code Generator Framework

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 where 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 deal with 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.

This section is divided in two sections: Scheduled Code Generation, and Scheduled Code Generation with Multiple Operational Modes. The first section shows how to generate code considering just one schedule. The second section describes how to generate code considering several alternative schedules, which are activated by a specific condition.

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 of performing either context saving or restoring, and calling the specific task. It is worthwhile to remind that, as presented before (Section 5.2.1), the proposed method considers that the timer is always programmed by a multiple of the TTU.

This code generation framework may be applied to several processor platforms. It is sufficient to make the dispatcher and timer interrupt handler available for the respective platform. In the experiments conducted in this work, the considered platform is 8051-based family of micro-controllers.

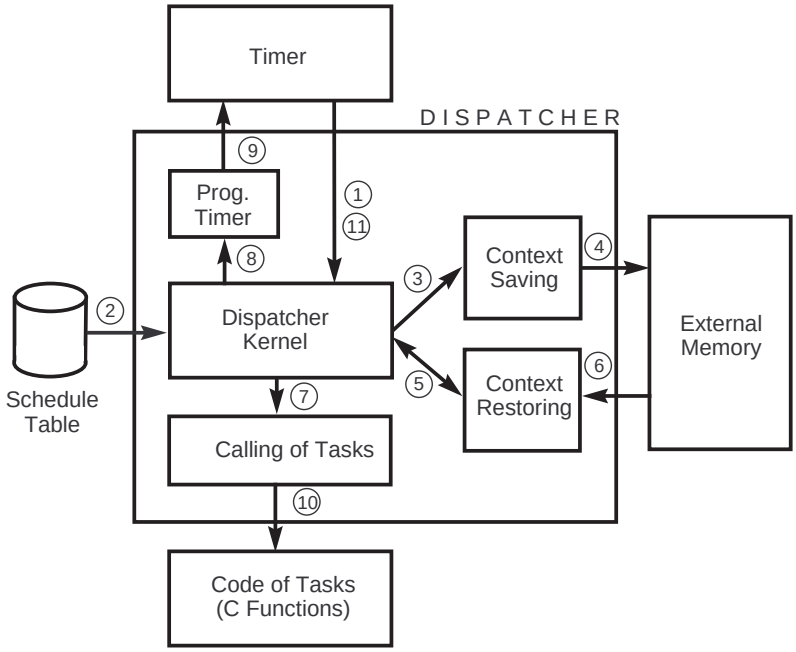

Figure 6.4: Proposed Code Generator Overview

Figure 6.4 overviews the proposed code generator framework, where the dispatcher is the main component. Figure 6.5 shows a simplified version of the proposed dispatcher function. Using Figure 6.4, the description of the code generator framework can be summarized as follows.

1. Considering that the system starts and that the clock value is equal to zero, the timer interrupt handler is forced to be called, and the control is transferred to the dispatcher kernel. This dispatcher kernel uses the current clock (line 4 of Figure 6.5) 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 6.5);

3. The dispatcher kernel saves the context of current task (line 7 -Figure 6.5) 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 6.5). 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 6.5), 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 6.5) 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 6.5). It is worth observing that `scheduleIndex` was incremented at line 13 of Figure 6.5.

9. The timer is activated (line 15 of Figure 6.5);

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.

```
 1 void dispatcher()
 2 {
 3   struct ScheduleItem newTaskInfo = scheduleTable[scheduleIndex];
 4   globalClock = newTaskInfo.clock;
 5
 6   if(existTaskInExecution) {                          // current task is preempted
 7     // context saving
 8   }
 9   if(newTaskInfo.isPreemptionReturn) {
10     // context restoring
11   }
12   taskFunction = newTaskInfo.functionPointer;         // Store current function
13   scheduleIndex = ((++scheduleIndex) % SCHEDULE_SIZE); // Information of new task called
14   programmingTimer(scheduleTable[scheduleIndex].clock);// Timer programmed for next task
15   activateTimer();                                    // Timer activated
16 }
```

Figure 6.5: Simplified Version of the Dispatcher

As presented in Section 5.3.6, in order to lead to a more realistic estimation of the system behavior, the proposed methodology considers the WCET of the dispatcher and timer interrupt handler. In this case, the time reserved for execution of a task already includes this time overhead.

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 6.6 shows the schedule table for a preemptive example that contains 7 task instances and 4 preemptions. Thus, the array has 11 entries. Figure 6.7 presents the respective timing diagram.

```
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 6.6: Example of a Schedule Table

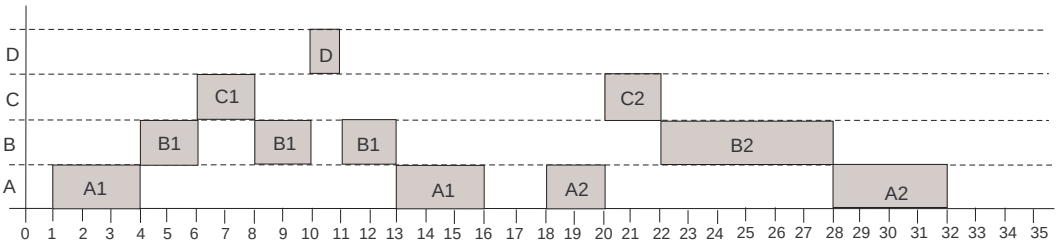

Figure 6.7: Timing Diagram for Schedule Table in Figure 6.6

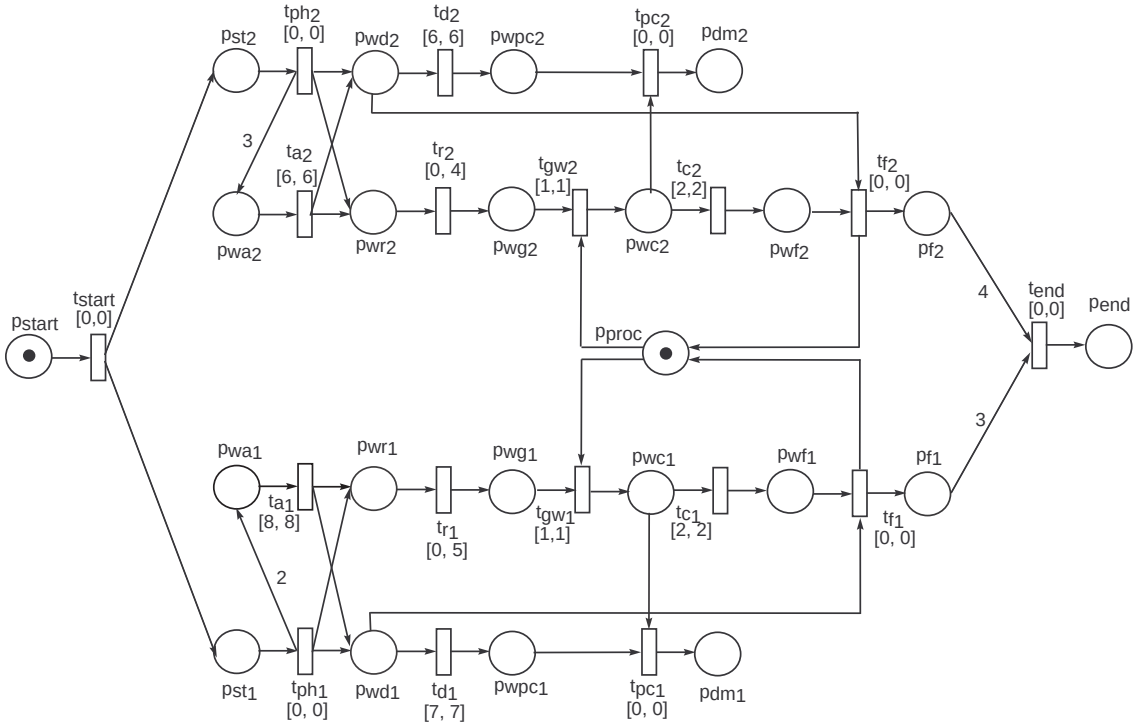

Figure 6.8: TPN model for two non-preemptive tasks with dispatcher overheads depicted in Table 5.2

The generated code has a set of global variables. Figure 6.5 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`).

Let us take a look at how to apply the proposed code generation method in the simple specification depicted at Section 5.3.1. This specification is automatically translated into the TPN model of Figure 6.8. A TLTS is generated by applying the proposed algorithm (Figure 6.2) on this model. In this example, it is considered that the interrupt and dispatcher overhead is equal to 1 TTUs. Analyzing the TLTS, it has been found out that task $T1$ is to be executed three times, for clock values equal to 0, 11, and 17. The same way, task $T2$ is executed four times, when clock values are equal to 3,

8, 14, and 20. Figure 6.9 shows the C code generated for this example, and Figure 6.10 presents the respective timing diagram that depicts the dispatcher overhead.

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

void taskT2()
{
  ...
}

#define SCHEDULE_SIZE 7

struct ScheduleItem schedule[SCHEDULE_SIZE] =
{
  { 0, false, 1, (int *)taskT1},
  { 3, false, 2, (int *)taskT2},
  { 8, false, 2, (int *)taskT2},
  {11, false, 1, (int *)taskT1},
  {14, false, 2, (int *)taskT2},
  {17, false, 1, (int *)taskT1},
  {20, false, 2, (int *)taskT2}
};
```

Figure 6.9: Generated code for a simple example

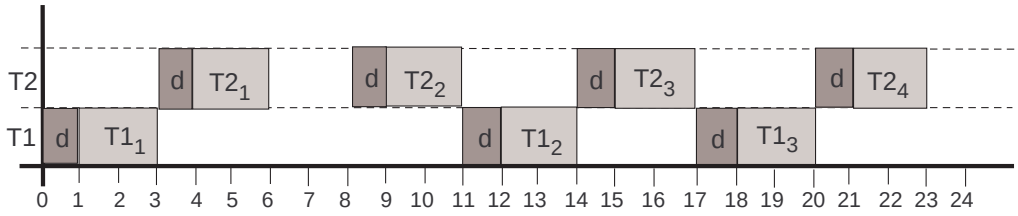

Figure 6.10: Timing Diagram for the Simple Example

6.2.2 Scheduled Code Generation with Multiple Modes

This section explains how to generate code considering multiple operational modes. Usually, pre-runtime method lacks flexibility, since a pre-runtime schedule solely cannot be adapted for handling environment changes at runtime. Therefore, new tasks cannot be removed or added, unless the system is stopped and a new schedule replaces the previous schedule.

A very common solution is to compute a pre-runtime schedule considering all possible tasks. However, there are situations where not all tasks need to be executed. An airplane system, for instance, performs different tasks during the take off, flight, and landing. Creating a pre-runtime schedule with all tasks will certainly result in an

enormous overhead during system's execution, or worse, a powerful processor will be necessary in order to accommodate all such tasks.

One solution for this kind of problem is to adopt a multiple operational mode technique. In this method, more than one mode is defined. Each mode represents an alternative schedule, which is activated by a specific condition. Such conditions usually use variables that are modified by events (internal or external). A problem may occur if two or more conditions become true at the same time. It is said that these conditions are in conflict. However, this work considers that all conditions are conflict-free. Multiple operational modes may reduce the problems caused by an unique pre-runtime schedule, and, additionally, increases flexibility.

Only one operational mode can be in execution at the same time. However, the same task may exist in more than one mode. Besides that, the same task may have different timing constraints in these modes.

As an example, consider the information of tasks in Table 6.3. In this table, it can be seen the name of the operational mode, the condition for its execution, and for each mode, the set of tasks timing constraints. This specification example does not have any inter-task relations, and energy constraints are not considered.

This table shows two operational modes with different tasks. For each mode, a pre-runtime schedule is generated using the proposed algorithm (Section 6.1.2). Mode 1 is defined as the starting mode. The dispatcher controls the mode switching based on the specified conditions. In this case, if temperature becomes higher than or equal to 100^oC, a mode switching from *mode1* to *mode2* occurs. In the same way, if the temperature becomes lower than 100^oC, a mode-switching from *mode2* to *mode1* is carried out. These conditions are defined with the use of a shared variable (`temp`), which is modified by an external event.

Table 6.3: Task Timing Specification

Oper.Mode	Condition	Tasks	Release	Comp.	Deadline	Period
mode1 (starting)	`(temp < 100)`	T_1	0	2	7	8
		T_2	2	2	6	6
mode2	`(temp ≥ 100)`	T_3	0	4	7	8
		T_4	1	1	9	10

The dispatcher is modified to control such mode switching. However, a mode switching may only be performed if there is no preempted task to be resumed. If a mode switching is requested and there are preempted tasks, new tasks instances of the current mode are not created. Thus, the required operational mode is set as the current

mode only after termination of all preempted tasks. This waiting for the conclusion of all preempted tasks is needed to avoid undesirable inconsistences.

The new dispatcher can be seen in Figure 6.11. Small modifications were made from the version depicted in Figure 6.5. The first one is the update of the pre-runtime schedule table, which now depends on the current mode (line 1). Two global variables are added for keeping the information of the current operational mode:

- `currentSchSize` (at line 2) contains the size of the current schedule table;
- `currentModeId` (at line 3) identifies the actual operational mode.

Such variables may be modified in the beginning of dispatcher execution by a new added function, namely `checkModeSwitching()`, which is depicted in Figure 6.12. When this function is called, firstly, it verifies whether exist preempted tasks in the current operational mode. If there are such tasks, this function simply returns without checking any pre-conditions. This verification is important in order to guarantee that the system will not become in a inconsistent state. After that, provided that there are no preempted tasks, the following statements verify if a specific mode switching can be performed. Such verification is done with the use of pre-condition functions (e.g. `checkMode1Switching()`), which represent the pre-condition for each operational mode switching.

In case of a pre-condition is satisfied and the current mode is different from the new one, some actions are performed:

1. the current schedule table is updated with the new operational mode schedule table (lines 8 and 16);
2. the size of the current schedule is also changed (lines 9 and 17);
3. the current mode id is set for the corresponding new mode id (lines 10 and 18);
4. the global clock (lines 11 and 19) and the table index (lines 12 and 20) are reset.

Section 8.4 provides an experiment showing the application of multiple operational modes for a pulse-oximeter case study.

6.3 Summary

This chapter described the proposed software synthesis framework. It discussed about the proposed scheduling and code generation phases.

```
 1 struct ScheduleItem *scheduleTable = scheduleTableMode1;
 2 unsigned int currentSchSize = SCHEDULE_SIZE_MODE1;
 3 unsigned int currentModeId = MODE1_ID;
 4
 5 void dispatcher()
 6 {
 7   struct ScheduleItem newTaskInfo;
 8
 9   checkModeSwitching()
10
11   newTaskInfo = scheduleTable[scheduleIndex];
12   globalClock = newTaskInfo.clock;
13
14   if(currentTaskPreempted) {
15     // context saving
16   }
17   if(newTaskInfo.isPreemptionReturn) {
18     // context restoring
19   }
20   else if(permitsNewTasksInstance) {
21     taskFunction = newTaskInfo.functionPointer;          // Store current task
22   }
23
24   scheduleIndex = ((++scheduleIndex) % currentSchSize);  // New task information
25   programmingTimer(scheduleTable[scheduleIndex].clock);  // Timer programming
26   activateTimer();                                       // Timer activated
27 }
```

Figure 6.11: Simplified Version of the Dispatcher for Multiple Operational Modes

```
 1 void checkModeSwitching()
 2 {
 3   if(existPreemptedTasks) {
 4     return;
 5   }
 6
 7   if(currentModeId != MODO1_ID && checkMode1Switching()) {
 8     *scheduleTable = scheduleTableMode1;
 9     currentSchSize = SCHEDULE_SIZE_MODE1;
10     currentModeId = MODE1_ID;
11     globalClock = 0;
12     scheduleIndex = 0;
13   }
14
15   if (currentModeId != MODE2_ID && checkMode2Switching()) {
16     *scheduleTable = scheduleTabelMode2;
17     currentSchSize = SCHEDULE_SIZE_MODE2;
18     currentModeId = MODE2_ID;
19     globalClock = 0;
20     scheduleIndex = 0;
21   }
22 }
```

Figure 6.12: `checkModeSwitching` Function

Firstly, it presented how to find a feasible pre-runtime schedule starting from the formal model. It showed the way that this work is maintaining the state space size under control by adopting a partial-order technique. It also explained about the proposed algorithm, which is a depth-first search method on a reduced state space. The proposed algorithm is interesting since it is deadlock- and starvation-free, where both are undesirable situations often present in concurrent systems. However, two problems

may arise in this algorithm. The first one is related to the search in the set of states already visited, which may cause inefficiencies in the whole algorithm execution. The second problem is concerned with the size of the set of visited states, which can be much larger. The first problem was minimized by adoption of a binary-tree search, and for the second problem data compression was used. Finally, it depicted the execution of the algorithm in an simple example that has been conducted by this work.

The code generation method presented in this chapter was proposed in such a way that the overheads were minimized. In order to attain such requirement, it was proposed the addition of a dispatcher and a timer interrupt handler. The dispatcher performs several controls needed to the execution of tasks, such as timer programming, context saving, context restoring, and tasks calling. Another key feature of the proposed solution is that the overheads of the dispatcher and timer interrupt handler are considered before the code generation. This implies a more accurate estimation for the system behavior. In order to provide more flexibility to the proposed scheduling method, it was also shown the multiple operational mode solution. In this method there are alternative pre-runtime schedules that may be switched, depending on whether the respective pre-condition is satisfied. It is supposed that these pre-conditions are not in conflict. The proposed code generation framework may be applied to several processor platforms. It is sufficient to make the dispatcher and timer interrupt handler available for the respective platform.

At the present moment, we do not know a similar work that generates timely and predictable scheduled code, starting from a formal model, and considering arbitrary precedence and exclusion relations.

Chapter 7

Tools

This chapter describes several tools for assisting the designer when using the proposed methodology. First of all, this chapter describes the EZPetri environment, which is used as the integration tool. In the following, each section explains specific tools that are applied to each phase of the methodology, such as, specification, modeling, schedule synthesis, and code generation.

7.1 EZPetri Environment

EZPetri [44] is an environment for integrating Petri net tools based on Petri Net Markup Language (PNML) [100] and Eclipse Platform [56].

The number of Petri net classes and tools has significantly increased over the last four decades. Such diversity represents an advance for the Petri net community. However, due to the use of specific file formats, these tools are usually incompatible. The problem occurs even in tools supporting the same type of Petri nets. The lack of integration among these tools imposes serious limitations on productivity. Therefore, functions for importing/exporting Petri nets from/to other tools are an important requirement nowadays. In order to solve this drawback, PNML has been proposed as an XML-based interchange format for Petri nets.

Eclipse Platform is designed for building integrated development environments (IDEs), where each IDE can support the construction of a variety of tools for application development. Such platform also provides useful building blocks and frameworks that facilitate the developing of new tools as well as the integration of existent ones.

EZPetri is an extendable Eclipse-based tool suite that supports editing Petri nets, as well as importing/exporting Petri nets from/to different Petri net tools, such as INA [90] and PEPTOOL [16]. It takes advantage of the plug-in technology of Eclipse to couple existing Petri net tools and to implement new functionalities.

PNML forms the kernel of EZPetri. It means that any Petri net type may be

represented through the PNML format in the EZPetri environment. Therefore, it glues together the integration facilities provided by Eclipse with the PNML interchange format.

EZPetri also contributes for reducing the gap between members of the Petri net community who use different Petri net classes, tools and file formats. Moreover, EZPetri improves productivity in the development of new products by offering several functionalities on a single development platform.

In this work, EZPetri is adopted for integrating several provided tools. Tasks (and their interrelations) are specified, translated into a time Petri net model, which is used for finding a feasible pre-runtime schedule, and generating a scheduled code. The schedule can be visualized as a timing diagram. Another view is an energy chart, which shows the energy consumption in the schedule. Moreover, all formal activities, from the specification up to the final result, are hidden from the final user.

7.2 Specification Editor

This section describes the proposed *specification editor* tool for entering the specification. This editor was plugged into EZPetri platform. This editor is composed by an environment for specifying tasks and their attributes, inter-task relations, and inter-task communications.

Figure 7.1 shows that the specification editor is a tree-based editor, where it is easy to see the interrelation between tasks. For instance, task T_4 precedes task T_5 and message M_1. Message M_1, on the other hand, precedes task T_2. In this case, it implies that task T_4 sends a message to task T_2, and both tasks (T_4 and T_2) are executed in different processors.

The use of the specification editor is as follows:

1. with a right-click on mouse, new tasks or messages may be created (Figure 7.2);
2. using the property view, the attributes of a task may be updated (Figure 7.3);
3. inter-task relations and inter-task communication are defined by a *drag-and-drop* feature;
4. the resultant specification is represented as a XML file (Figure 7.4).

The property-view shows the attributes of a task, such as, name, timing information (phase, release, WCET, deadline, and period), code, scheduling method (preemptive or non-preemptive), processor, and energy consumption.

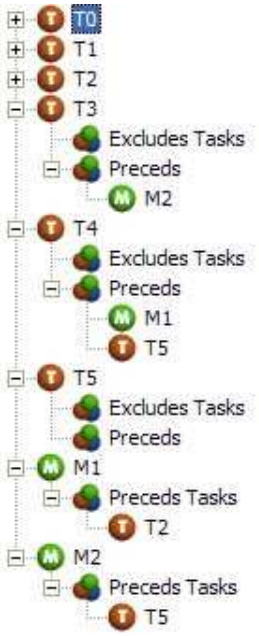

Figure 7.1: Tree-based Specification Editor

Figure 7.2: New Task/Message

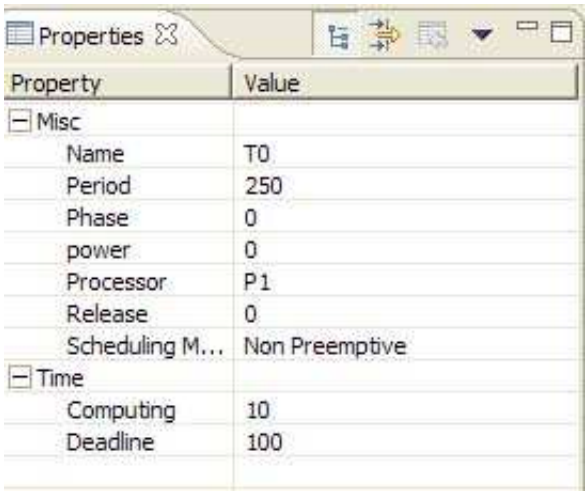

Figure 7.3: Properties View

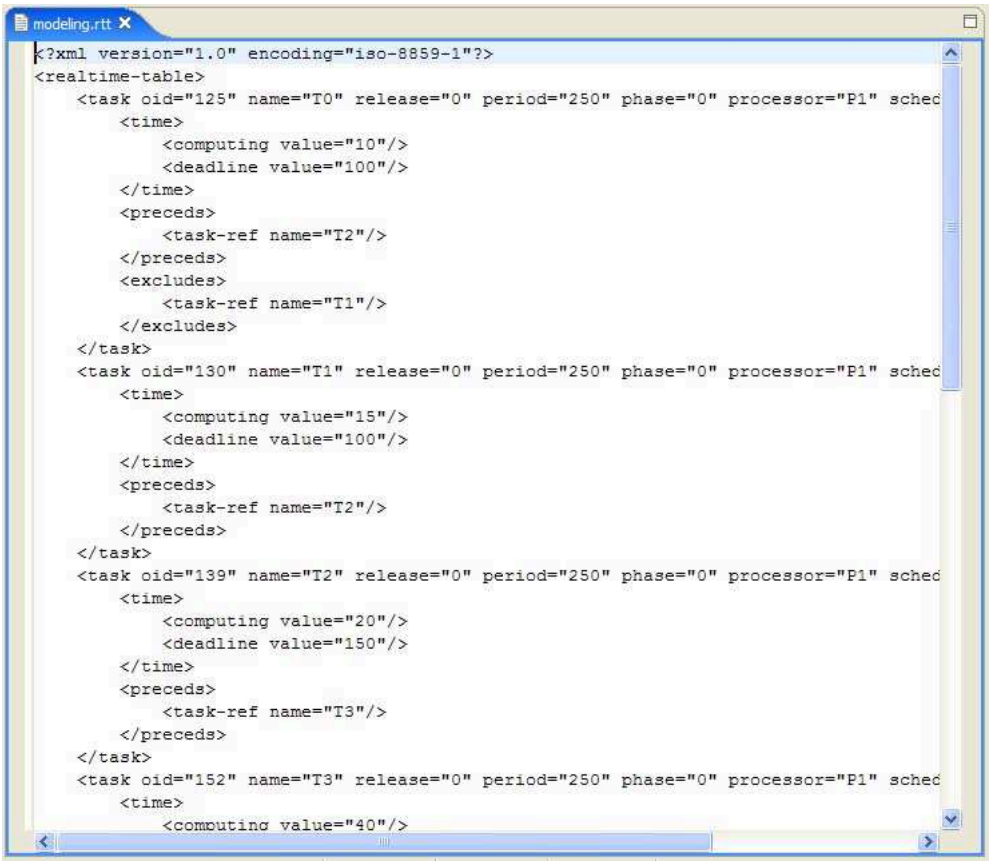

Figure 7.4: Specification represented as a XML file

7.3 Automatic Model Generation

Starting from the specification (XML file), the next phase is the automatic generation of a time Petri net that represents such specification [72]. This translator is implemented using the Java programming language. As presented before, the modeling phase is based on the composition of building blocks. In order to automatically generate a model from the specification, the following steps should be taken into account:

1. generate a model for arrival, deadline, and task structure blocks in each task;
2. generate each precedence relation;
3. generate each exclusion relation;
4. generate each inter-tasks communication;
5. generate each processor and/or bus;
6. generate the fork block;
7. generate the join block.

During this procedure, the time Petri net model is stored in main memory. After that, two files are generated: (i) a PNML file; and (ii) a specific file format for the schedule generator. The PNML file (Figure 7.5) may be used for interchanging between tools. The specific file format (Figure 7.6) represents the same time Petri net model, but in a format suitable for reading by the schedule generator. Section 9.3 proposes an extension to consider just the PNML format.

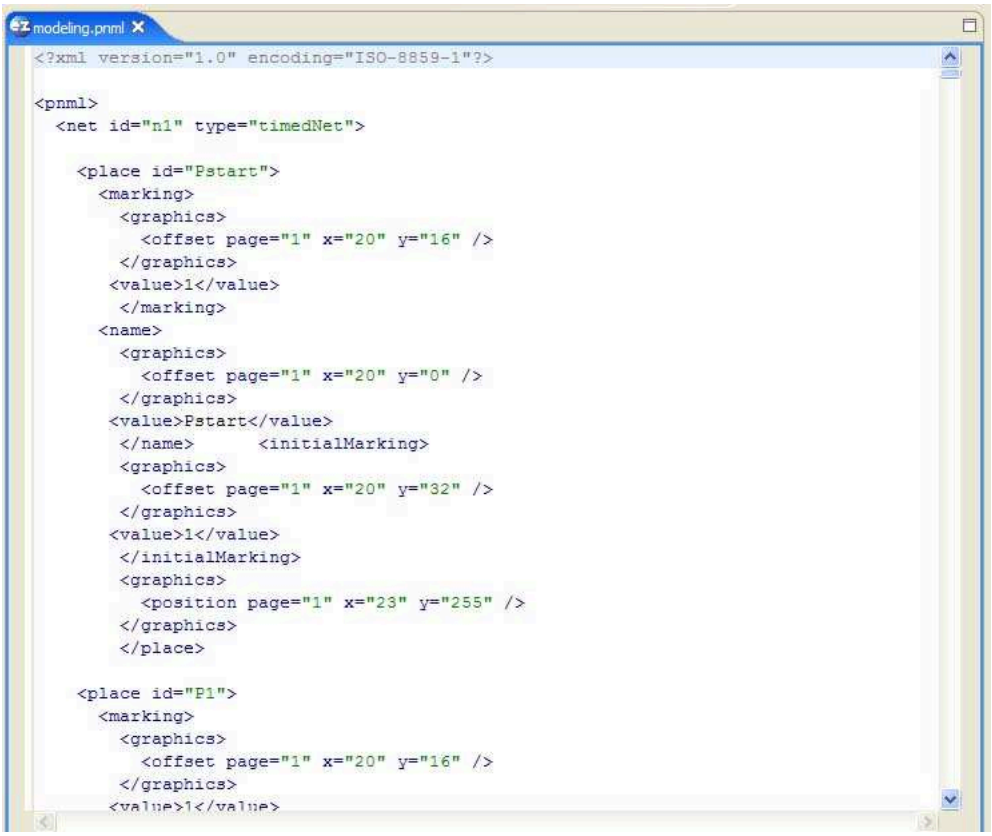

Figure 7.5: A TPN represented by a PNML file

There are several Java classes for implementing this translator. For instance, arc, place and transition, are examples of such classes. There are also classes that models the building blocks, such as, arrival, deadline, task structure (preemptive and non-preemptive), send and receive message, precedence and exclusion relations. Another important class is the one that performs the translation from PNML to the schedule generator specific format.

7.4 Schedule Generator

The proposed algorithm depicted in Section 6.1.2 is implemented using the C programming language. However, in order to reduce the amount of memory, the recursive algorithm is converted into a iterative program.

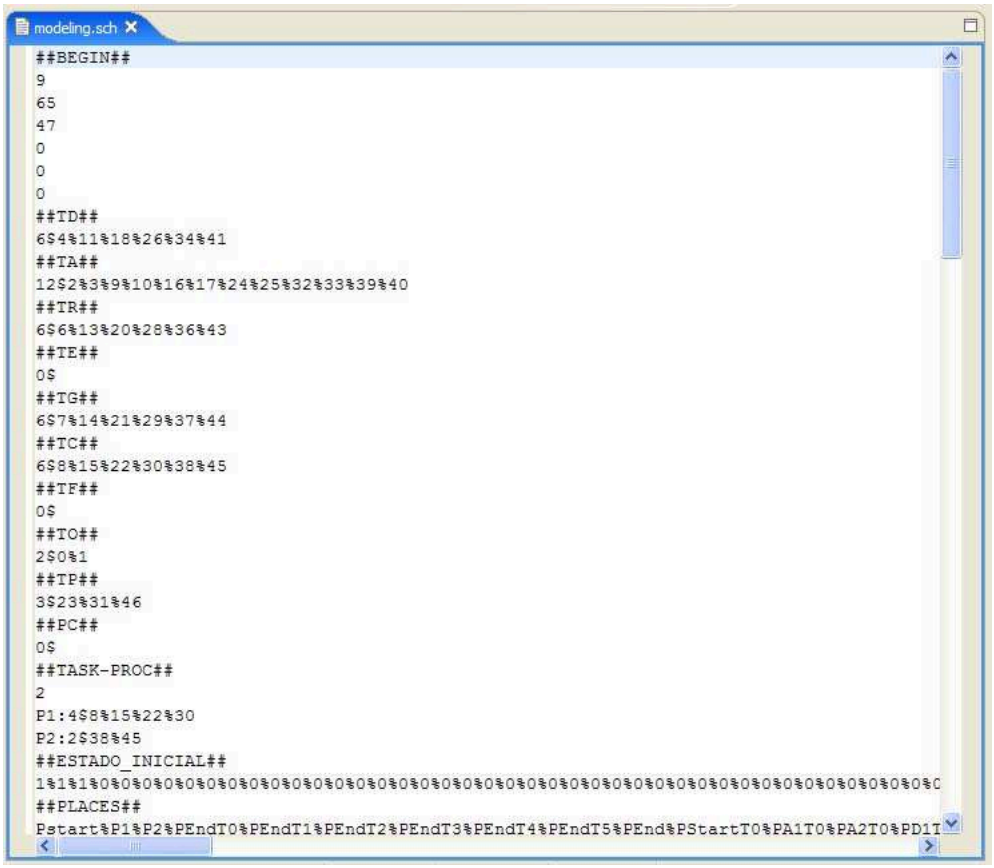

Figure 7.6: The same TPN represented by a specific file format for the schedule generator

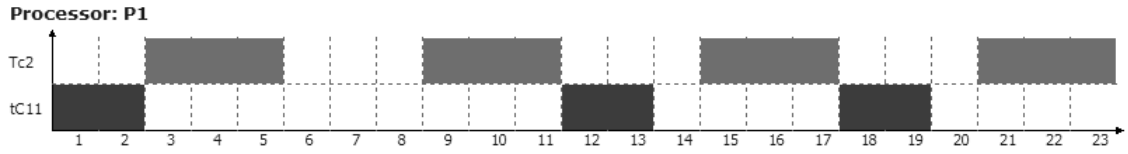

Figure 7.7: Timing Diagram

In the actual implementation, before choosing a computation transition to be analyzed, the post-pruning fireable transitions set (PT) is sorted based on the deadline. Hence, the task with earliest deadline is preferred to be analyzed first. The argument for this kind of sorting is that, in most cases, this kind of approach allows finding a feasible schedule more quickly.

The dispatcher consumes both time and energy. This overhead may be increased if the dispatcher also performs context-switching. The method used to minimize the number of context-switching is storing the last executed task by the processor. If the same task (that had used the processor) is again concurrent to the processor, that specific task is preferred to be analyzed first. This way, the context-switching occurs only when it is strictly necessary.

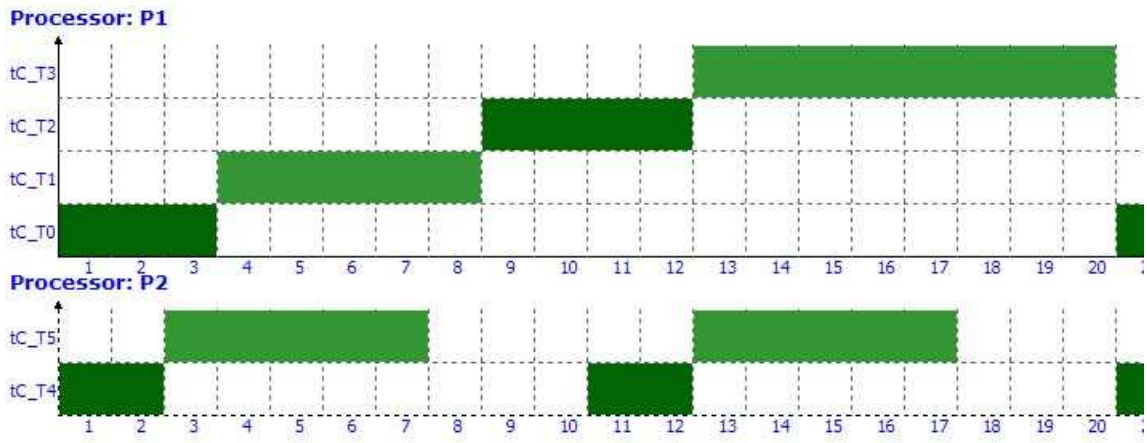

Figure 7.8: Timing Diagram for 2-Processors

7.5 Timing Diagram and Energy Chart

When successful, the schedule generator produces a timed labeled transition system, which is a sequence of transitions to be fired at a specific time instant. Obviously, this description is not easy to have a global view of the schedule. Therefore, only for visualization purposes, the schedule (represented by a TLTS) may be seen as a timing diagram (Figure 7.7).

For generating the timing diagram, the proposed method identifies all computation transitions and the time instant where such transition starts execution. This information is automatically rendered as a timing diagram. Figure 7.8 presents another timing diagram, this way considering two processors.

Another way to visualize the schedule is by an energy chart (Figure 7.9). This chart exhibits the accumulated energy consumption in each task time unit. The information comes from the same TLTS, which also stores the accumulated energy consumed at each state.

7.6 Code Generator Engine

A code generator engine was developed in order to automate the code generation process. Such engine uses an open source framework named Velocity [1], which is a Java code generator utility based on template files. Figure 7.10 depicts a visual representation of Velocity execution. Each template file have a set of rules, which are executed in accordance with a set of data informed by the user. A rule may verify whether or not a block of code should be generated. Figure 7.11 shows a high level example of a dispatcher template. The rules begin with the character "#". This template shows that `hasPreemption` is needed information.

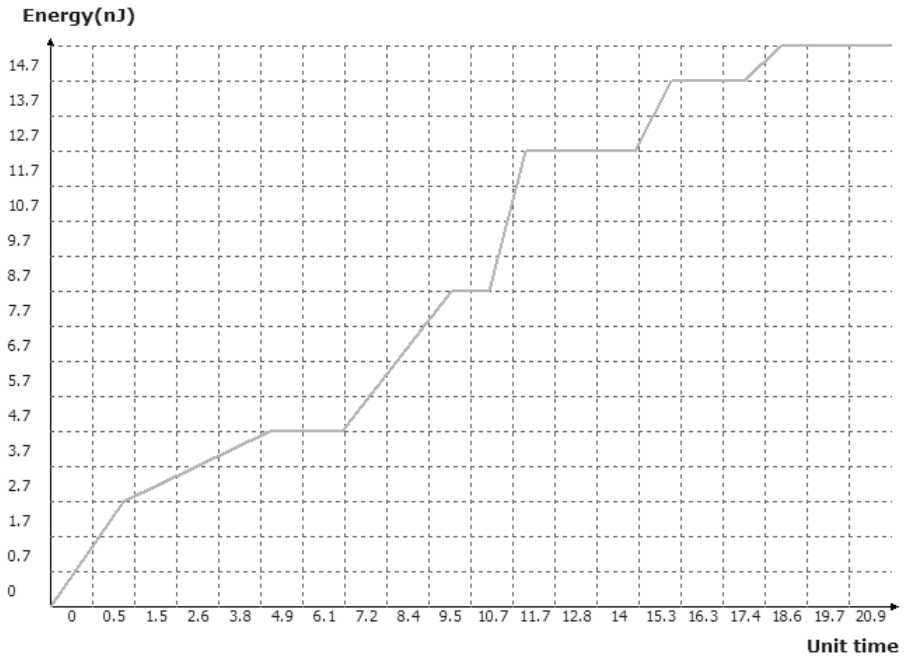

Figure 7.9: Energy Chart

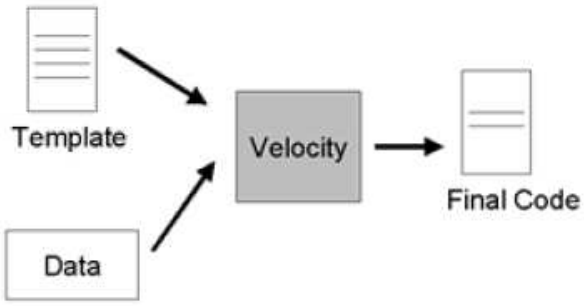

Figure 7.10: Velocity Framework

The engine consists in a set of six pre-defined templates. There is a set of templates for each specific processor architecture. Such templates are:

1. tasks generation;
2. schedule tables;
3. dispatcher;
4. interrupt handler;
5. system constants; and
6. global variables and types (e.g. a struct that represents a task context).

Before starting the code generation, the engine automatically generates a Java class that represents the data to be merged with the templates. This Java class is called

`CodeSpecification`, and contains the following data: (i) the specific architecture; (ii) the pre-runtime schedule table; (iii) the body code of each task; and (iv) the task time unit.

In order to generate the code, the engine checks the availability of the templates for a specific architecture chosen by the user. If the templates are available, this engine generates:

1. codes for each task;
2. schedule table;
3. the dispatcher file;
4. interrupt handler file;
5. a file with the system constants (for instance, the task time unit); and
6. a file containing global variables and types (e.g. task context).

```
void dispatcher()
{
  struct SchItem item;
  item = sch[schIndex];
  globalClock = item.starttime;
  #if($hasPreemption)
  if(currentTaskPreempted) {
    // context saving
  }
  if(item.isPreemptionReturn) {
    // context restoring
  }
  else
  #end
  taskFunction = item.functionPointer;
  schIndex = ((++schIndex)%currentSchSize);
  progrTimer(sch[schIndex].starttime);
  activateTimer();
}
```

Figure 7.11: Dispatcher Template

7.7 Summary

This chapter described several tools to assist the designer in adopting the proposed methodology. Using the EZPetri environment is easier for integrating several tools. In this work, the user specifies the tasks of a system using a specification editor. The result of this phase is a XML file that is input for automatically translating this

specification into a time Petri net model. This model is used for finding a feasible pre-runtime schedule. If successful, the timed labeled transition system, which represents the feasible schedule found, may be visualized as a timing diagram. Another possible view is the energy consumption chart, which shows the energy consumption in the schedule. With such tools, all formal activities, from the specification up to the final scheduled code, are hidden from the final user.

Chapter 8

Experiments

This work has conducted several experiments, which are summarized in Table 8.1. In this table, *instances* represent the number of task instances; *state-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, where most of the time is spent due to the tag/untag schema; and *method* states the chosen (preemptive (P) or non-preemptive (nP)) scheduling method. The presented results were obtained in order to find the first feasible schedule. All experiments were performed on a Duron 900 Mhz, 256 MB RAM, OS Linux, and compiler GCC 3.3.2.

Table 8.1: Experimental Results Summary

Example	instances	state-min	found	time (s)	method
Simple Control Application	**28**	**50**	**50**	**0.001**	**nP**
Robotic Arm	37	150	150	0.014	nP
Pulse-oximeter (mode2)	**19**	**78**	**78**	**0.110**	**nP**
Pulse-oximeter (mode1)	**19**	**78**	**78**	**0.100**	**nP**
Xu&Parnas (example 3)	4	171	1558	0.120	P
Xu&Parnas (figure 9)	5	281	2406	0.220	P
Pulse-Oximeter	**178**	**850**	**850**	**0.256**	**nP**
Mine Pump Control	782	3130	3255	0.462	nP
Heated-Humidifier	**1505**	**6022**	**6022**	**0.486**	**nP**
Unmanned Ground Vehicle	433	4701	14761	2.571	P

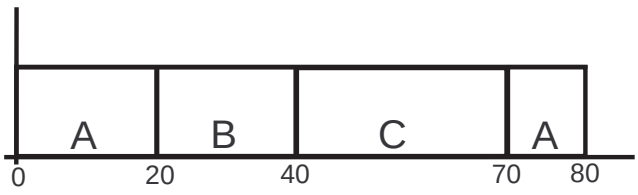

Figure 8.1: Timing Diagram of the Xu-Parnas Example 3

In Table 8.1 some case studies are expressed in boldface. The aim of this Chapter

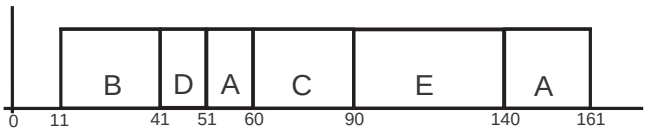

Figure 8.2: Timing Diagram of the Xu-Parnas Figure 9

is investigate such examples in more detail. In the following, the other examples are briefly commented.

- **Robotic Arm**. This case study is a real application and comes from [4]. It is a robotic arm programmed to take objects from a conveyor belt, store them in a buffer shelf, and to put them eventually into a basket. This arm is controlled by four critical tasks. This case study has stringent timing constraints, since the CPU utilization factor is 81.7%. A feasible schedule was found after examining 150 states, which is the minimum, in 14 ms, for 37 instances of tasks.

- **Xu&Parnas-example3**. This example is the third presented in [105] that shows that static or dynamic priority-driven scheduling algorithms (for instance, earliest deadline first, or deadline monotonic) can fail in finding a feasible schedule, even when such schedule exist. In general, this situation often occurs when the task model imposes inter-task relations. The specification model considers three tasks with execution times equal to 30, 20, and 30, respectively. Additionally, an exclusion relation is defined. As it considers a preemptive scheduling method, the execution of tasks are split in 80 (30+20+30) parts. The proposed scheduling algorithm visited 1558 states, in 120 ms, for finding a feasible schedule. The minimum number of states is 171 states. This case study shows the increased complexity when adopting a completely preemptive solution. Figure 8.1 presents a time diagram of the schedule found.

- **Xu&Parnas-figure9**. In accordance with [106], priority-driven scheduling policies are only capable of producing a very limited subset of the possible schedules for a given set of tasks. For example, there are situations where, in order to satisfy all given timing constraints, it is necessary to let the processor idle for a certain time interval, even if there are tasks ready for execution. This situation is presented in the Xu&Parnas-figure9 case study. The timing constraints require that the processor must be left idle between the interval [0,11]. Neither static nor dynamic priority-driven schemes can deal properly with such situation. The scheduling method considered was the preemptive method. The specification

model considers five tasks with execution times equal to 30, 30, 10, 10 and 50, respectively. In this case, the execution of tasks are split in 130 (30+30+10+10+50) parts. The proposed scheduling algorithm visited 2406 states, in 220 ms. The minimum number of states is 281 states. Figure 8.2 shows a time diagram of the schedule found.

- **Mine Pump Control**. This example is another real-world application known as mine drainage system. Detailed specification for this example can be found in [19]. This problem has not tight timing constraints in general, but the schedule for this problem is interesting because it has 10 tasks, implying 782 tasks' instances and, at the beginning, all 10 tasks arrive at the same time. Our solution searched 3268 states (where minimum number of states is 3130), in this case having an overhead of 138 states (4.4%), which is very low considering the complexity of this example. The time performance is 462 ms, using a non-preemptive (NP) method.

- **Unmanned Ground Vehicle**. This kind of vehicle is designed to traverse hazardous ground for collecting various kinds of data (data, images, etc). A semi-autonomous capability for making local path decisions is triggered when it encounters unforeseen hazards, e.g., debris, rubble, land miles, etc. An UGV provides two main services: its own mobility, and collecting information of interest for the controllers. The first one includes functions such as steering, braking, and speed control as well as planning local autonomous movement. The second service includes the capture of data sensors, such as infrared, microwave, radar, and others. In accordance with [87], the tasks are all independent ones. This task model has a processor utilization factor of about 61%, which is not very low. This case study has 14 tasks, where four of then are sporadic and were translated into periodic ones. The amount of tasks' instances is 433. The proposed algorithm finds a feasible schedule after analyzing 14761 states, where the minimum number of states is 4701, in 2,5 seconds. It is worth noting the scheduling method is preemptive, which certainly increases time and space complexity.

For depicting the practical usability of the proposed framework for scheduling synthesis (with timing constraints), scheduling synthesis (with timing and energy constraints), and code generation (with and without multiple modes) in more details, four of these examples are considered, namely, a simple control application, two examples based on a pulse-oximeter, and a heated-humidifier, respectively.

8.1 Simple Control Application

This case study exemplifies the application of the scheduling synthesis framework considering a multi-processor architecture.

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 [28], and later used in [2]. Four processors are connected by a single bus. The model consists of 6 tasks split into 22 subtasks, which exchanges 10 messages, 6 of them are sent across processor boundaries.

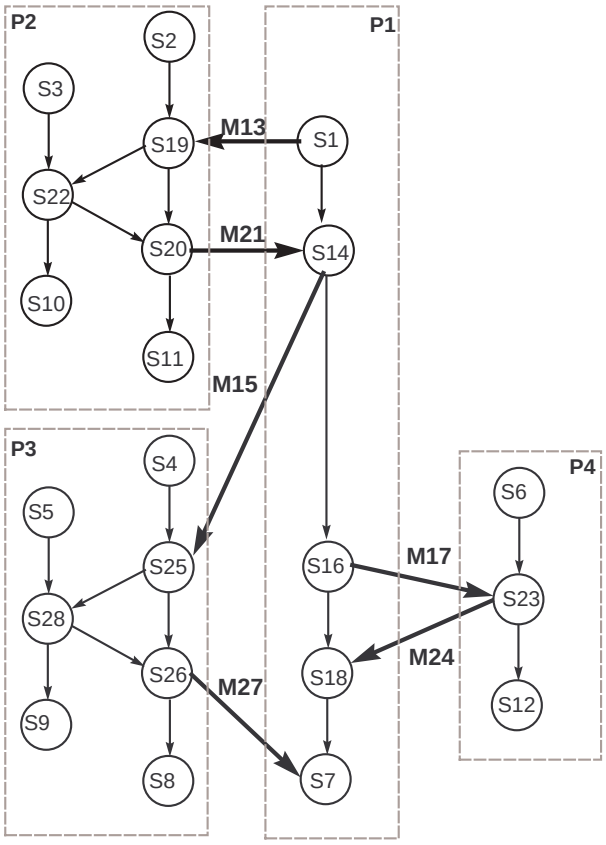

Figure 8.3: The Simple Control Application Graph

Figure 8.3 shows the communication graph for this application, presenting the subtasks allocated to processors, and its communication pattern, where the interprocessor communications are labeled with "M" in the figure. Table 8.2 gives the worst-case execution time and deadline for each subtask as well as the worst-case communication time for each inter-processor message. Figure 8.4 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 deadline checking, the

Table 8.2: Task Set for the Simple Control Application

Tarefa	C_i	D_i	Tarefa	C_i	D_i	Tarefa	C_i	D_i	Tarefa	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

bus, and the shared processors (P2 and P3).

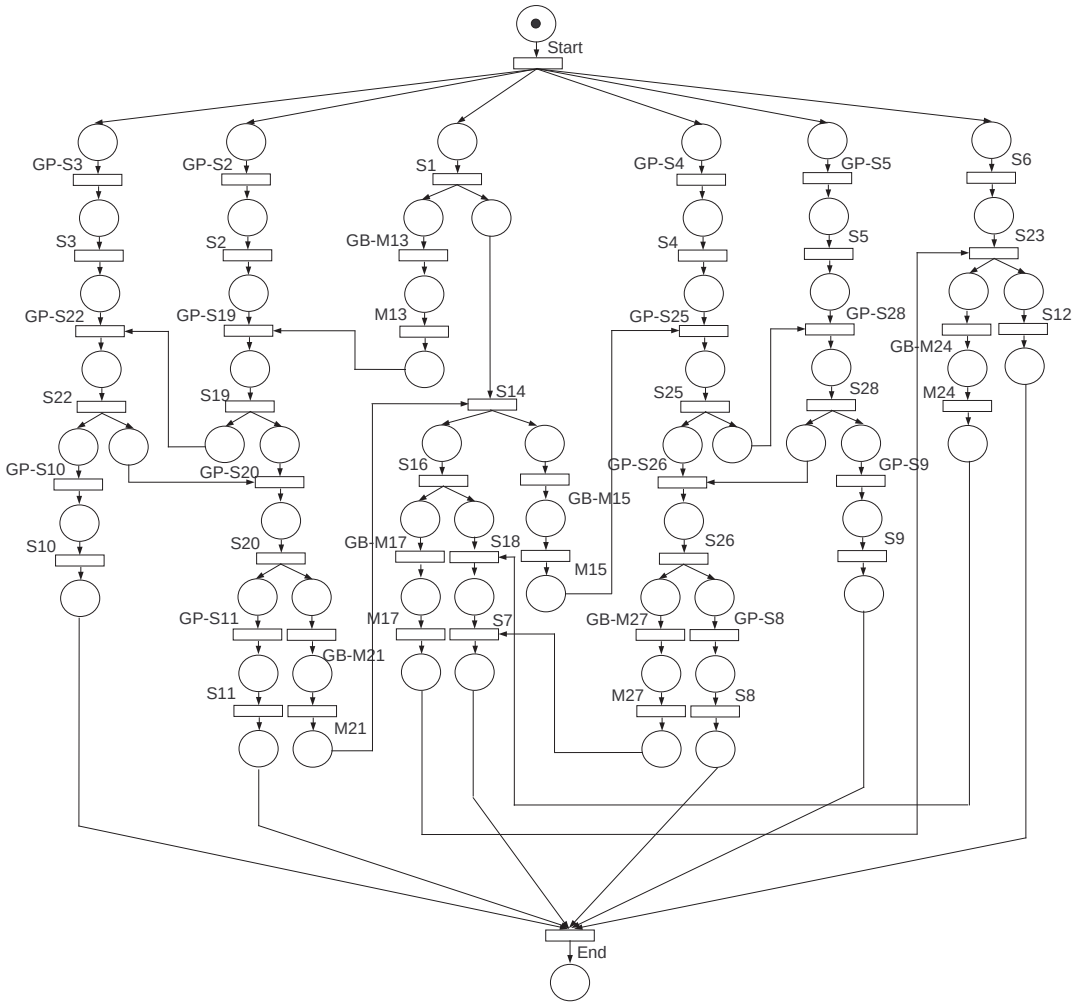

Figure 8.4: Simplified Simple Control Application Time Petri Net Model

The proposed algorithm found a feasible scheduling after examining the minimum number of states (in this case 50 states) in just one millisecond. Table 8.3 presents the

algorithm execution results. The transitions' labels are: `gp` means processor granting, `gb` means bus granting, `s` means subtasks, and `m` means interprocessor messages. The firable transitions are underlined. As it can be noted in this particular example, the number of transitions to be evaluated are very often reduced compared with the firable transition sets. As an example, in state six there are four firable transitions but only one is to be evaluated, since it was not cut by the pruning.

Table 8.3: Execution Results for the Simple Control Application

state	ET	C	PT	trans+time
0	{start}	{0}	{start}	{start,0}
1	{gp-s3, gp-s2, s1, gp-s4, gp-s5, s6}	{0,0,0,0,0,0}	{gp-s3, gp-s2, gp-s4, gp-s5}	{gp-s3,0}
2	{s3, s1, gp-s4, gp-s5, s6}	{0,0,0,0,0}	{gp-s4,gp-s5}	{gp-s4,0}
3	{s3, s1, s4, s6}	{0,0,0,0}	{s3,s1,s4,s6}	{s3,3}
4	{gp-s2, s1, s4, s6}	{0,3,3,3}	{s1,s4,s6}	{s1,0}
5	{gp-s2, gb-m13, s4, s6}	{0,0,3,3}	{s4,s6}	{s4,0}
6	{gp-s2, gb-m13, gp-s5, s6}	{0,0,0,3}	{s6}	{s6,0}
7	{gp-s2, gb-m13, gp-s5}	{0,0,0}	{gb-m13}	{gb-m13,0}
8	{gp-s2, m13, gp-s5}	{0,0,0}	{gp-s2,gp-s5}	{gp-s2,0}
9	{s2, m13, gp-s5}	{0,0,0}	{gp-s5}	{gp-s5,0}
10	{s2, m13, s5}	{0,0,0}	{m13}	{m13,1}
11	{s2, s5}	{1,1}	{s2,s5}	{s2,2}
12	{gp-s19, s5}	{0,3}	{s5}	{s5,0}
13	{gp-s19}	{0}	{gp-s19}	{gp-s19,0}
14	{s19}	{0}	{s19}	{s19,5}
15	{gp-s22}	{0}	{gp-s22}	{gp-s22,0}
16	{s22}	{0}	{s22}	{s22,6}
17	{gp-s10, gp-s20}	{0,0}	{gp-s10,gp-s20}	{gp-s10,0}
18	{s10}	{0}	{s10}	{s10,2}
19	{gp-s20}	{0}	{gp-s20}	{gp-s20,0}
20	{s20}	{0}	{s20}	{s20,7}
21	{gp-s11, gb-m21}	{0,0}	{gb-m21}	{gb-m21,0}
22	{gp-s11, m21}	{0,0}	{gp-s11}	{gp-s11,0}
23	{s11, m21}	{0,0}	{m21}	{m21,1}
24	{s11, s14}	{1,0}	{s11}	{s11,1}
25	{s14}	{1}	{s14}	{s14,14}
26	{s16, gb-m15}	{0,0}	{gb-m15}	{gb-m15,0}
27	{s16, m15}	{0,0}	{m15}	{m15,1}
28	{s16, gp-s25}	{1,0}	{gp-s25}	{gp-s25,0}
29	{s16, s25}	{1,0}	{s25}	{s25,2}
30	{s16, gp-s28}	{3,0}	{gp-s28}	{gp-s28,0}
31	{s16, s28}	{3,0}	{s16,s28}	{s16,7}
32	{gb-m17, s28}	{0,7}	{s28}	{s28,0}
33	{gb-m17, gp-s26, gp-s9}	{0,0,0}	{gb-m17}	{gb-m17,0}
34	{m17, gp-s26, gp-s9}	{0,0,0}	{gp-s26,gp-s9}	{gp-s26,0}
35	{m17, s26}	{0,0}	{s26}	{s26,1}
36	{m17, gp-s8, gp-s9}	{1,0,0}	{m17}	{m17,0}
37	{gp-s8, gp-s9, gb-m27, s23}	{0,0,0,0}	{gb-m27}	{gb-m27,0}
38	{gp-s8, gp-s9, m27, s23}	{0,0,0,0}	{gp-s8,gp-s9}	{gp-s8,0}
39	{s8, m27, s23}	{0,0,0}	{m27}	{m27,1}
40	{s8, s23}	{1,1}	{s8}	{s8,1}
41	{gp-s9, s23}	{0,2}	{gp-s9}	{gp-s9,0}
42	{s9, s23}	{0,2}	{s9}	{s9,2}
43	{s23}	{4}	{s23}	{s23,6}
44	{s12, gb-m24}	{0,0}	{gb-m24}	{gb-m24,0}
45	{s12, m24}	{0,0}	{m24}	{m24,1}
46	{s18, s12}	{0,1}	{s18,s12}	{s18,1}
47	{s7, s12}	{0,2}	{s12}	{s12,0}
48	{s7}	{0}	{s7}	{s7,2}
49	{end}	{0}	{end}	{end,0}

8.2 Pulse Oximeter

This case study is considered for applying the scheduling synthesis with timing and energy constraints.

The pulse oximeter [45] 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, when a patient is sedated with anesthetics for a surgical procedure. This equipment is widely used in center care units (CCU) in hospitals.

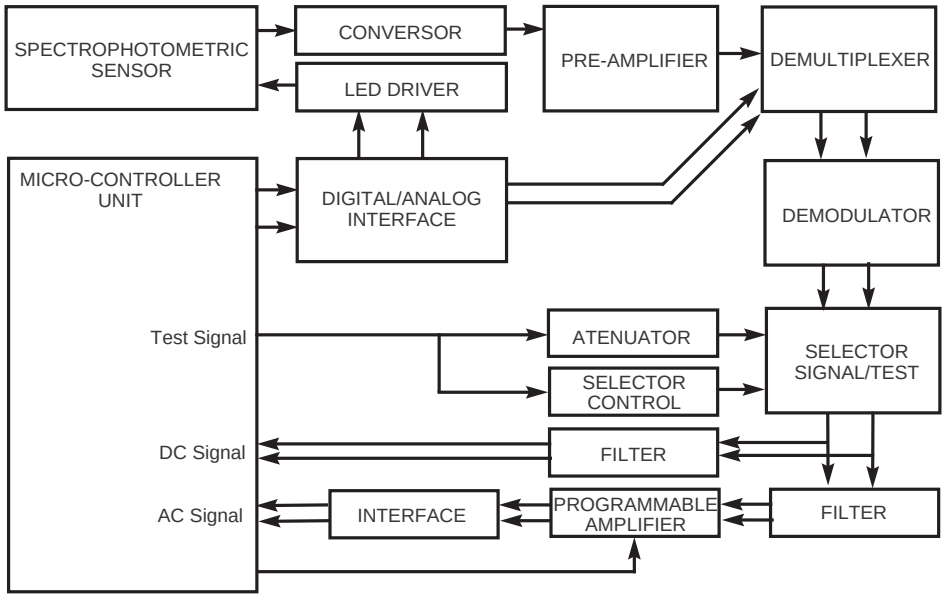

Figure 8.5: Pulse Oximeter Architecture

The architecture of this equipment can be seen in Figure 8.5. The architecture consists of a micro-controller unit, a spectrophotometric sensor (which is compounded by 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.

The micro-controller controls the synchronization and amplitude of the led driver, which 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 micro-controller. Lastly, the micro-controller performs the calculation related to oxygen saturation level based on data received, and shows the result on a display.

The code was downloaded to an AT89S8252 micro-controller single board. The energy measurement was obtained by inserting probes (commands before and after a task) to synchronize the measurement with task events. It was measured instantaneous current drawn by the processor during the execution of each task. For data acquisition it was used a TDS220 digital oscilloscope linked to PC desktop computer by serial port.

Table 8.4 shows the pulse oximeter task set. In this work, for sake of simplicity, the task set of the oximeter was based only on two general processes: an excitation (PA), and an acquisition-control process (PB). The excitation process (PA) is responsible to dispatch stream pulses to the leds in order to generate radiation pulses. The acquisition-control process (PB) captures radiations crossing patient's finger, and realizes the calculation of oxygen saturation level. Both processes are divided in threads, where each thread represents a task. It is considered that a thread of a process cannot be interrupted by any other thread of the same process. In this case, context saving and restoring are not performed between tasks of the same processes, but only between threads (tasks) of different processes.

Using the proposed approach, a feasible schedule is found in 256 milliseconds. The amount of visited states was the minimum, that is, 850 states. The energy consumed by the system considering the generated schedule is:

32 TA1 instances x 8576,69 nJ +
32 TA2 instances x 52,35 nJ +
32 TA3 instances x 8576,69 nJ +
32 TA4 instances x 52,35 nJ +
5 TB1 instances x 55,58 nJ +
5 TB2 instances x 222,32 nJ +
5 TB3 instances x 55,58 nJ +
5 TB4 instances x 222,32 nJ +
5 TB5 instances x 55,58 nJ +
5 TB6 instances x 222,32 nJ +
5 TB7 instances x 55,58 nJ +
5 TB8 instances x 222,32 nJ +
5 TB9 instances x 430 nJ +
5 TB10 instances x 7089 nJ +
24 context-switching x 400 nJ,

resulting in 605,011.56 nJ.

Table 8.4: Task Set for the Pulse Oximeter

TaskID	Task Name	r	c	d	p	energy(nJ)
TA1	SetExcitationLedRed	0	41	1000	2500	8576
TA2	ResetExcitationLedRed	371	41	1000	2500	52
TA3	SetExcitationLedInfra	576	41	1000	2500	8576
TA4	ResetExcitationLedInfra	947	41	1000	2500	52
TB1	StartChannelACRed	0	41	5000	16000	55
TB2	ReadChannelACRed	141	50	5000	16000	222
TB3	StartChannelACInfra	191	41	5000	16000	55
TB4	ReadChannelACInfra	323	50	5000	16000	222
TB5	StartChannelADRed	382	41	5000	16000	55
TB6	ReadChannelADRed	523	50	5000	16000	222
TB7	StartChannelADInfra	573	41	5000	16000	55
TB8	ReadChannelADInfra	714	50	5000	16000	222
TB9	StoreDataArrays	764	60	5000	16000	430
TB10	Control	0	90	10000	16000	7089

Intertask Relations
TA1 PRECEDES TA2, TA2 PRECEDES TA3, TA3 PRECEDES TA4, TB1 PRECEDES TB2, TB2 PRECEDES TB3, TB3 PRECEDES TB4, TB4 PRECEDES TB5, TB5 PRECEDES TB6, TB6 PRECEDES TB7, TB7 PRECEDES TB8, TB8 PRECEDES TB9

8.3 Heated-Humidifier

This case study explains the application of the code generator framework.

Heated-humidifier is a control system that aims to insert water vapor in the gaseous mixture used in a sort of electro-medical systems. For maintaining such vapor, the system must warm up the water in a recipient and maintain the water temperature in a prescribed value. This equipment is also very useful in critical care units (CCU).

Table 8.5: Specification for the Heated-Humidifier

Tasks	r	c	d	p
A (temp-sensor-start)	0	1	1,500	10,000
B (temp-sensor-handler)	11	1	1,500	10,000
C (PWM)	0	8	1,500	10,000
D (pulse-generator)	0	4	4	50
E (temp-adjust-part1)	0	1	5,000	10,000
F (temp-adjust-part2)	1501	2	5,000	10,000

Inter-Task Relations
A PRECEDES B
B PRECEDES C
E PRECEDES F

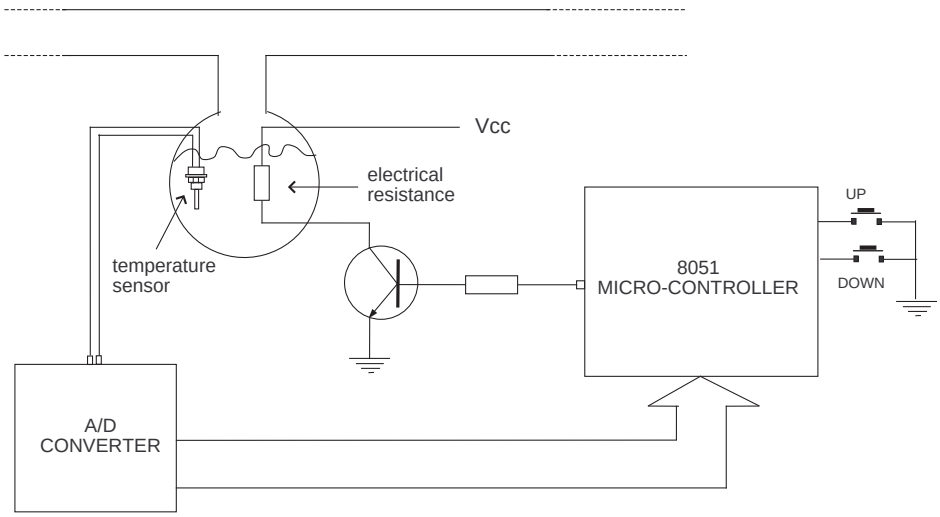

Figure 8.6: Heated-Humidifier Architecture

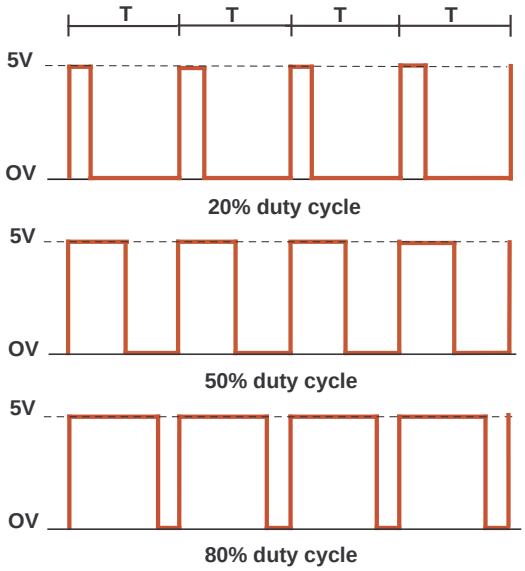

Figure 8.7: PWM Control

The architecture of this equipment can be seen in Figure 8.6. This architecture consists of a micro-controller (8051), two keys for adjustment of the desired temperature, a temperature sensor, and an electric resistance (in a water recipient). Water warming is controlled by pulse width modulation (PWM) technique, which switches the supplied energy on and off. In this case, the DC voltage is converted to a square-wave signal, alternating between fully on and zero. PWM control consists of changing the pulse duty cycle (Figure 8.7) of a square wave in order to change its average value.

Table 8.5 shows the task set considering a 8051-family architecture. The values are expressed in TTUs (task time units), where each time unit is equal to 10μs. Considering

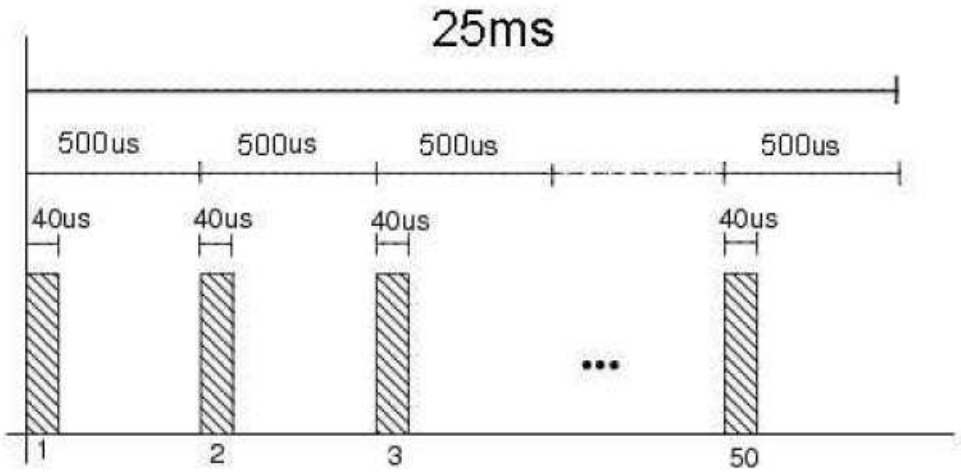

Figure 8.8: Pulse Generator Slot Time

such architecture, the overhead of the interrupt and dispatcher is equal to 200 μs (20 TTUs). Except task D, the period of all tasks are equal to 100ms (10,000 TTUs), since this period is sufficient for meeting the specification constraints. The deadline of tasks A and B takes into account that, after A/D conversion, the sensor has to be read at most in 15ms. Tasks E and F consider that keys are kept pressed up to 50ms.

PWM task is responsible for increasing or decreasing the pulse duty cycle. Initially it starts with duty cycle equal to 50%, and it is adjusted in conformity with the measured water temperature. In this case, if the temperature is lower than the desired temperature, the PWM task increases the duty cycle. On the other hand, PWM task decreases the duty cycle.

The pulse-generator task generates a cyclic square wave with the duty cycle controlled by PWM task. The pulse frequency considered is 40 Hz, which leads to a period (T) equal to 25ms. The PWM resolution is 500μs, i.e., the duty cycle is increased or decreased at 500μs steps. In this work, the computation time of the pulse-generator task is equal to 40μs, remaining 460μs for executing other tasks. It is worth remembering that the dispatcher is called before each task, which consumes 200 μs of time in the worst-case. Figure 8.8 shows the pulse generator slot time and its relationship with the PWM period.

Temperature adjustment is divided into two tasks (temperature-adjust-part1 and temperature-adjust-part2) in order to avoid the key bouncing. If temperature-adjust-part1 indicates that a key is pressed, after a specific minimal time (generally 15ms), the temperature-adjust-part2 task must confirm such pressing. It is worth noting that its release time is equal to 1501 time units (15,000μs (key bouncing) + 10μs (execution

of task E)). Such timing constraints allow other tasks to be executed avoiding time wasting due to key bouncing procedure.

The same solution is applied for reading the temperature sensor. The first task (temperature-sensor-start) is responsible for starting the A/D conversion. Since this conversion takes time, the processor may execute another tasks in the meantime. After elapsing a specific time (generally 100μs), the second task (temperature-sensor-handler) may start reading the temperature and updating the respective shared variable, which is read by the PWM task. Note that the release time of the task temperature-sensor-handler is equal to 11 TTUs (100μs (A/D conversion) + 10μs (execution of task A)).

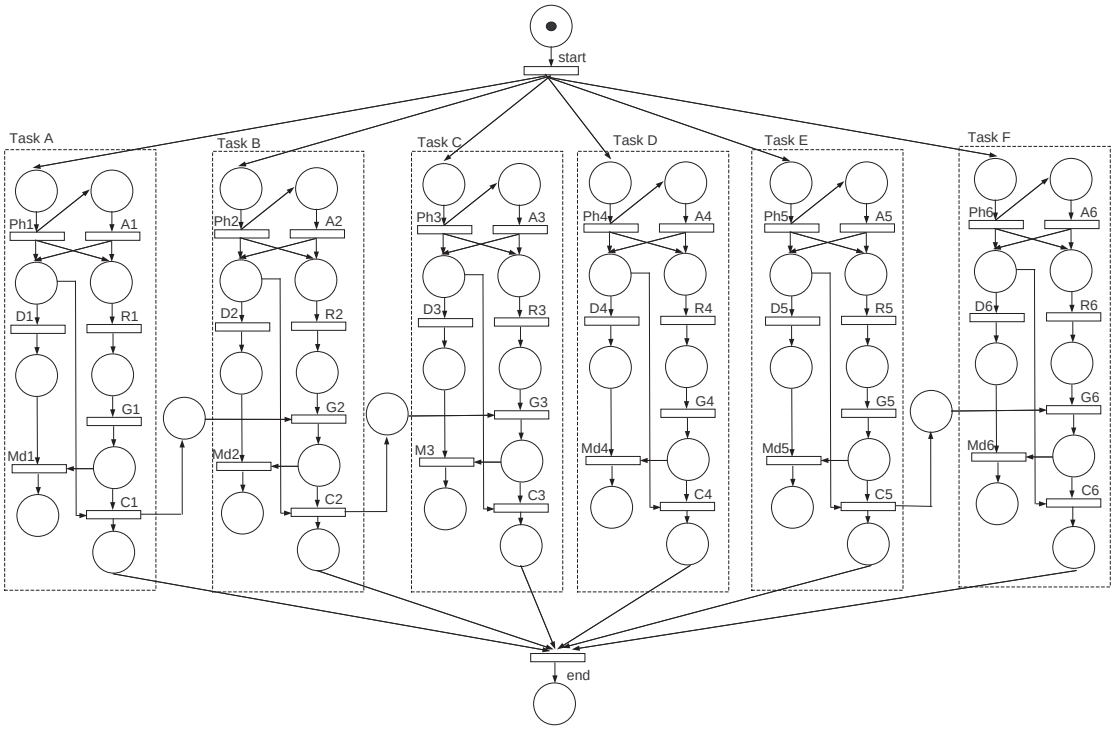

Figure 8.9: Heated-Humidifier Time Petri Net Model

All communication between tasks is carried out by shared memory. For instance, the temperature-sensor-handler task communicates the measured temperature to the PWM task; the same way, the PWM task communicates the duty cycle rate to the pulse-generator task; and the temperature-adjust-part2 sends the desired temperature to the PWM task.

Figure 8.9 presents a simplified time Petri net model for this task set, where a non-preemptive scheduling method is used. For sake of simplicity, the processor is not modeled in this figure. The next step of the methodology searches for a feasible scheduling using the TPN model. This schedule was found in 486 ms, verifying 6022 states, which is the minimum number of states to be verified.

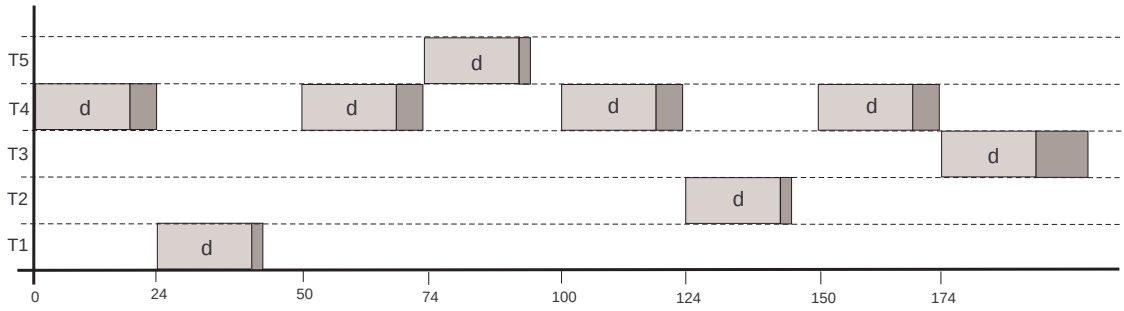

Figure 8.10: Heated-Humidifier Timing Diagram

Figure 8.10 depicts part of a timing diagram that shows the dispatcher and interrupt handler overheads.

As presented in Section 6.2, C code is generated by traversing the feasible firing schedule returned by the scheduling synthesis framework. Figure 8.11 shows parts of the generated code, where constants, tasks, and the schedule table are defined.

```
// Constants
#define SCHEDULE_SIZE 505

// Tasks
void taskT1() {...}
void taskT2() {...}
void taskT3() {...}
void taskT4() {...}
void taskT5() {...}
void taskT6() {...}

// Schedule Table
struct SchItem sch[SCHEDULE_SIZE] =
{
  {0,  false, 4, (int *)taskT4},
  {24, false, 1, (int *)taskT1},
  {50, false, 4, (int *)taskT4},
  {74, false, 5, (int *)taskT5},
  {100,false, 4, (int *)taskT4},
  {124,false, 2, (int *)taskT2},
  {150,false, 4, (int *)taskT4},
  {174,false, 3, (int *)taskT3},
  {200,false, 4, (int *)taskT4},
      .
      .
      .
}
```

Figure 8.11: Heated-Humidifier Generated Code

8.4 Pulse Oximeter with Multiple Modes

The pulse oximeter is used to show how to generate code considering multiple operational modes.

The considered pulse-oximeter has two operational modes: (i) executing; and (ii) programming mode. Executing mode represents the normal functioning of the equipment. This mode has tasks that perform the acquisition, control, calculation, analysis and presentation of the results on a display. Programming mode is responsible for providing a different graphical interface, which is manipulated by an user for modifying the acceptable levels of the arterial oxygen saturation, cardiac beats, and alarm volume. The alarm is triggered when the oxygen saturation is beyond acceptable levels. Moreover, the programming mode is comprised by tasks that perform the acquisition, control, presentation and supervision of the programming interface. Table 8.6 depicts the oximeter task timing specification. Furthermore, the intertask relations are:
T_1 `PRECEDES` T_2, T_2 `PRECEDES` T_3, T_3 `PRECEDES` T_4, T_5 `PRECEDES` T_6, T_6 `PRECEDES` T_7, T_7 `PRECEDES` T_8, T_8 `PRECEDES` T_9, T_{10} `PRECEDES` T_{11}, T_{11} `PRECEDES` T_{12}, T_{12} `PRECEDES` T_{13}, T_{15} `PRECEDES` T_{16}, T_{17} `PRECEDES` T_{18}.

The mode changing is triggered when an user presses the programming button on the oximeter panel. The tasks *KeyPressChecking* and *KeyPressConfirmation*, included in both operational modes, are responsible to periodically verify the button state. Two tasks are needed in order to avoid the key bouncing. Task *KeyPressChecking* indicates if the key is pressed. After a specific minimal time, task *KeyPressConfirmation* confirms such pressing, and, consequently, puts such state in a shared variable called `isButtonProgPushed`. In the next dispatcher execution, the buttons state is verified using a pre-condition function. If the programming button is pressed and there are no preempted tasks, the mode changing is realized. The programming mode also has two tasks responsible for verifying the state of the button OK. When this button is pressed, the programming mode is replaced by the executing mode in the next dispatcher execution, only if there are no preempted tasks. Table 8.7 depicts the specification of operational mode pre-conditions for the pulse-oximeter.

The schedule for the programming mode was obtained in 0.100 seconds. In addition, the schedule for the executing mode was found in 0.110 seconds. As presented before, C code is generated by traversing the feasible firing schedule of each operational mode.

Figure 8.12 shows parts of the generated code for the pulse-oximeter. This figure presents definition of shared variables, conditions for mode switching, constants, tasks, and schedule table of each mode.

Table 8.6: Oximeter Task Timing Specification

Oper. Mode	ID	Task Name	Release	Comp.	Deadline	Period
Executing	T_1	SetExcitationLedRed	0	41	1000	80000
	T_2	ResetExcitationLedRed	371	41	1000	80000
	T_3	SetExcitationLedInfra	576	41	1000	80000
	T_4	ResetExcitationLedInfra	947	41	1000	80000
	T_5	StartChannelACRed	0	41	5000	80000
	T_6	ReadChannelACRed	141	50	5000	80000
	T_7	StartChannelACInfra	191	41	5000	80000
	T_8	ReadChannelACInfra	323	50	5000	80000
	T_9	StartChannelADRed	382	41	5000	80000
	T_{10}	ReadChannelADRed	523	50	5000	80000
	T_{11}	StartChannelADInfra	573	41	5000	80000
	T_{12}	ReadChannelADInfra	714	50	5000	80000
	T_{13}	StoreDataArrays	764	60	5000	80000
	T_{14}	Control	0	90	10000	80000
	T_{15}	LevelsCalculation	5001	30000	80000	80000
	T_{16}	ResultAnalisys	5001	3000	45000	80000
	T_{17}	KeyPressChecking	5001	1000	40000	80000
	T_{18}	KeyPressConfirmation	6502	1000	40000	80000
	T_{19}	ReseultsDisplaying	45000	32000	80000	80000
Progr.	T_1	SetExcitationLedRed	0	41	1000	70000
	T_2	ResetExcitationLedRed	371	41	1000	70000
	T_3	SetExcitationLedInfra	576	41	1000	70000
	T_4	ResetExcitationLedInfra	947	41	1000	70000
	T_5	StartChannelACRed	0	41	5000	70000
	T_6	ReadChannelACRed	141	50	5000	70000
	T_7	StartChannelACInfra	191	41	5000	70000
	T_8	ReadChannelACInfra	323	50	5000	70000
	T_9	StartChannelADRed	382	41	5000	70000
	T_{10}	ReadChannelADRed	523	50	5000	70000
	T_{11}	StartChannelADInfra	573	41	5000	70000
	T_{12}	ReadChannelADInfra	714	50	5000	70000
	T_{13}	StoreDataArrays	764	60	5000	70000
	T_{14}	Control	0	90	10000	70000
	T_{20}	PrintMenuDisplaying	11000	25000	65000	70000
	T_{17}	KeyPressingChecking	5001	1000	40000	70000
	T_{18}	KeyPressingConfirmation	6502	1000	40000	70000
	T_{21}	MenuHandler	45000	2500	60000	70000
	T_{22}	LevelsAdjustment	50000	20000	70000	70000

Table 8.7: Oximeter Operational Modes Pre-Condition Specification

Operational Mode	Pre-Condition
Operating Mode (initial)	`(isButtonProgPressed)`
Programming Mode	`(isButtonOKPressed)`

8.5 Summary

In order to illustrate the practical usability of the proposed software synthesis method, this chapter has presented in details four experimental results. The first one deals with

the method for finding a feasible schedule considering a multi-processor architecture. In this case, a simple control application was used, which runs on 4-processors. This case study is used for explaining how to find a feasible schedule considering timing and resource constraints.

The second experiment was conducted in 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. This case study was adopted for explaining the proposed method for finding feasible schedules considering timing, resource and energy constraints.

The third experiment was also performed in order to illustrate the code generation phase in the proposed methodology. It was used a heated-humidifier case study, which is a control system that aims to insert water vapor in the gaseous mixture used in several electro-medical equipments.

The last experiment was applied for describing the proposed method of code generation considering multiple operational modes. The adopted case study was again the pulse-oximeter. However, this time the specification have two separate modes.

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

```
//Shared variables
bit isButtonProgPressed;
bit isButtonOKPressed;

// Constants
#define SCHEDULE_SIZE_MODE1 19
#define SCHEDULE_SIZE_MODE2 19

// Conditions
bit checkProgModeChanging() {
     return (isButtonOKPressed);
}
bit checkOperModeChanging() {
     return (isButtonOKPressed);
}

// Tasks
void taskT1() {...}
void taskT2() {...}
...
void taskT22() {...}


// Schedule Tables
struct SchItem sch_mode1[SCHEDULE_SIZE_MODE1] =
{{0, false, 1, (int *)taskT1},
  {41, false, 5, (int *)taskT5},
  {82, false, 14, (int *)taskT14},
  {371, false, 2, (int *)taskT2},
  {412, false, 6, (int *)taskT6},
  {462, false, 7, (int *)taskT7},
  {503, false, 8, (int *)taskT8},
  {553, false, 9, (int *)taskT9},
  {594, false, 3, (int *)taskT3},
  {635, false, 10, (int *)taskT10},
  {685, false, 11, (int *)taskT11},
  {947, false, 4, (int *)taskT4},
  {988, false, 12, (int *)taskT12},
  {1038, false, 13, (int *)taskT13},
  {5001, false, 15, (int *)taskT15},
  {35001, false, 17, (int *)taskT17},
  {36001, false, 18, (int *)taskT18},
  {37001, false, 16, (int *)taskT16},
  {45000, false, 19, (int *)taskT19}
};

struct SchItem sch_mode2[SCHEDULE_SIZE_MODE2] =
{{0, false, 1, (int *)taskT1},
  {41, false, 5, (int *)taskT5},
  {82, false, 14, (int *)taskT14},
  {371, false, 2, (int *)taskT2},
  {412, false, 6, (int *)taskT6},
  {462, false, 7, (int *)taskT7},
  {503, false, 8, (int *)taskT8},
  {553, false, 9, (int *)taskT9},
  {594, false, 3, (int *)taskT3},
  {635, false, 10, (int *)taskT10},
  {685, false, 11, (int *)taskT11},
  {947, false, 4, (int *)taskT4},
  {988, false, 12, (int *)taskT12},
  {1038, false, 13, (int *)taskT13},
  {11000, false, 17, (int *)taskT17},
  {12000, false, 18, (int *)taskT18},
  {13000, false, 20, (int *)taskT20},
  {45000, false, 21, (int *)taskT21},
  {50000, false, 22, (int *)taskT22}
  };
```

Figure 8.12: Generated Code for the Pulse Oximeter Considering Multiple Modes

Chapter 9

Conclusions

Embedded system designers have to deal with a dilemma, since those systems have increased complexity and, at the same time, market pressures have shortened the time-to-market. The processors computational power increasing on the one hand, and the size and cost reduction on the other hand, have allowed moving more and more functionality to software. Nowadays, the software is usually responsible for more than 80% of the functions in embedded systems. However, due to the increasing complexity and diversity of requirements, embedded software has become much harder to design. Correctness and timeliness verification are issues that must be concerned, since several applications demand safety properties.

For coping with those stringent requirements, embedded software development methodologies play an important role. Formal methods are an alternative to deal with the inherent complexity of embedded systems. In order to improve the degree of confidence of critical systems, formal methods allow precise specification, verification and/or analysis of qualitative as well as quantitative properties. However, for effective use of formalisms, the availability of automated tools to assist the design of embedded software is an important matter.

The goal of this thesis has been the development of a methodology for generating predictable source code in a suitable programming language, where such code satisfies timing, energy, and resource constraints.

The application domain is embedded hard real-time systems, where correct behavior depends not only on the integrity of the results, but also on the time when results are produced. Therefore, later results may have serious consequences, including resource damages or risk of human life.

As it is considered time-critical systems, predictability is an important concern. In order to guarantee that all critical tasks meet their deadlines, it was used the pre-runtime scheduling, since the runtime scheduling may constrain the possibility of finding a feasible schedule, even if such schedule exists, mainly when considering arbitrary

precedence and exclusion relations.

In order to solve this problem, the proposed approach started from a specification model, which is automatically translated to a formal model, in this case a TPN model, where such formal model is analyzed for finding a feasible schedule. After that, the code generation phase is performed. In the literature, the problem solved by this thesis is commonly called software synthesis, which is defined as the task of translating a complex specification into a source code such that functionality is attended, and the typical runtime support is provided. However, software synthesis taking into account time-critical systems is little explored in the research community.

This work applied the proposed methodology into several case studies, but only four of them were detailed. It was presented examples for scheduling computation considering timing and resource constraints on multi-processors, scheduling computation considering timing, energy and resource constraints on a single processor, code generation for single and multiple operational modes.

This chapter summarizes this thesis, depicting the main contributions, limitations, and future directions on this research.

9.1 Contributions

This work faced three main problems, namely, modeling, scheduling, and code generation; and contributed proposing alternatives for dealing with embedded software system design.

Modeling

The starting point for all results produced by this thesis was the modeling. The modeling phase described how to model embedded hard real-time systems using time Petri net formalism.

Time Petri net is a mathematical formalism that allows specification, properties verification, and modeling of several features present in most concurrent and real-time systems, such as, stringent timing constraints, precedence and exclusion relations, communication protocols, multiprocessing, synchronization mechanisms, and shared resources.

The proposed modeling phase is based on composition of building blocks. The building blocks considered in this thesis are: (i) periodic task arrival; (ii) task structure (preemptive or non-preemptive), where dispatcher overheads might be considered or not; (iii) deadline checking; (iv) inter-processor sending message; (v) resources, such as processors and buses; (vi) fork; and (vii) join. These blocks are composed by application

of several operators, such as place merging, addition and refinement, arc addition and removing, and net union. This modeling method also shows how to model inter-tasks relations, in this case, precedence and exclusion relations. Finally, it explained how to model inter-processor communication, that is, message sending and message receiving considering that tasks are allocated to different processors.

Scheduling

Embedded hard real-time systems have stringent constraints that must be satisfied. Hence, when considering safety or timing-critical systems, predictability should be provided. Therefore, scheduling plays an important role for attaining such constraints in a predictable way.

Starting from the time Petri net model, the proposed scheduling synthesis framework analyzes the timed labeled transition system (resultant from this model) in order to find a pre-runtime schedule, provided that such schedule exists.

This work uses state space exploration, which consists in recursively checking all successor states, starting in a given initial state, by executing all enabled action in each state. In spite of the fact that a scheduling can be found using this strategy, it may be limited by the excessive size of the state space. This problem comes up due to the analysis based on the interleaving of concurrent activities. This exponential growth is known as the state explosion problem. This work starts by showing how to minimize the state space size. It was used three approaches: (i) modeling explicitly dependencies between tasks; (ii) applying a partial-order reduction technique, which is specific for the proposed modeling, where it was identified a simple way to define persistent sets; and (iii) removing undesirable states.

The proposed scheduling policy is *pre-runtime scheduling*, where schedules are computed entirely off-line. As presented previously, this strategy has advantages over others, mainly when adopting arbitrary precedence and exclusion relations.

The proposed algorithm for finding a feasible schedule is a depth-first search method on the reduced state space. This algorithm is interesting since it is deadlock and starvation-free, where both are undesirable situations often present in concurrent systems. However, one problem may arise in this algorithm: the search in the set of states already visited, which may cause inefficiencies in the whole algorithm execution. Nevertheless, this problem was minimized by adoption of a binary-tree search.

Code Generation

The code generation method presented in this thesis was proposed in such a way that the main overheads were minimized. In order to attain such requirement, it was provided a dispatcher and a timer interrupt handler. The dispatcher performs several controls needed to execution of tasks, such as timer programming, context saving, context restoring, and tasks calling. The timer is programmed by the dispatcher to interrupt at the time where the next task instance must be executed (or resumed). It is worth observing that just one timer is needed since the generated code is already scheduled. This solution eliminates the "busy-waiting", which is very often adopted strategy in practice.

Another key feature of the proposed solution is that overheads of the dispatcher and timer interrupt handler are considered in the modeling phase, that is, before the schedule computation. This overhead is often neglected in several research papers. However, if it is not considered, this overhead may affect the deadline of tasks. One usually adopted solution considers that the WCET of tasks already includes this overhead. This solution is rather pessimistic, since it is unknown how many preemptions will occur in each task before a schedule has been found. On the other hand, the solution adopted in this thesis explicitly models the WCET of the dispatcher and timer interrupt handler. In this case, the overhead is considered during the schedule generation, but only when needed, leading to a more realistic estimation for the system behavior.

In order to give more flexibility to the pre-runtime method, it was also shown the multiple operational mode solution. In this method, there are several alternative pre-runtime schedules that may be switched, depending on whether the respective pre-condition is satisfied.

The proposed code generation framework may be applied to several processor platforms. It is sufficient to make the dispatcher and timer interrupt handler available for the respective platform.

At the best of our present knowledge, there is no similar work that generates timely and predictable scheduled code, starting from a formal model, and considering arbitrary precedence and exclusion relations.

9.2 Limitations

This section aims to describe the limitations of the proposed scheduling strategy.

As shown previously, pre-runtime scheduling is often the only means for providing

predictability in complex real-time systems. However, this approach has some drawbacks:

1. pre-runtime scheduling is based on the assumption that all information about the system is known before runtime. Nevertheless, such information may not be always available in advance. However, if this information is not known, it is not possible to guarantee that all hard deadlines will be met.

2. another issue concerns the "periodic world" assumed in this methodology. Some designers may have difficulties in fitting the problem in this paradigm. Although having an elegant way for transforming a sporadic task into a periodic one, this strategy may impose significant overheads. However, it is the price to be paid for the predictability requirement.

3. pre-runtime schedule is computed considering a period equal to the least common multiple (LCM) among all periods in the task set. If periods of a task set are different prime numbers, the LCM might be very large. This problem can only be reduced if the designer is able to change the periods of tasks.

9.3 Future Works

Software synthesis for embedded hard real-time systems remains a relatively new topic in systems research. Consequently, this topic has several opportunities for further improvement. This section presents future directions into specification, modeling, scheduling, and code generation.

This thesis has not directly addressed the possibility of deadlock in both precedence relation, and inter-processor communication. However, deadlock-detection may be performed by a cycle search in the graph that represents the precedence relation and/or the communication pattern.

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 was not considered in this thesis. However, certainly the complexity will decrease and, at the same time, properties of interest will be preserved. This is a further work to be investigated.

Another interesting work would be formally proof that the TPN model faithfully represents the specification. The idea is to prove that all building blocks really model

the desired behavior. After that, it is necessary to prove that all compositions maintain the aggregate behavior.

So far, two problems may arise in the proposed scheduling algorithm related to the proposed tagging scheme, that is, the scheme that stores a set of visited states for avoiding the analysis of them more than once. The first problem is related to the search in the set of states already visited, which may cause inefficiencies in the whole algorithm execution. This problem was minimized by the adoption of a binary-tree search. The second problem, which may compromise the whole solution, is the size of the set of visited states. This size can be huge. For minimizing this problem, a solution might apply methods of compression to reduce the size of this set of states.

As stated before, the state space explosion problems is hardened by the interleaving semantics when considering concurrent activities. It is well-known, however, that Petri nets are widely used as a model of concurrency, which allows representing the occurrence of independent events. Thus, Petri net models can also be a model of parallelism. Therefore, the simultaneity of the events is only considered when adopting the *step semantics*. In this semantics, the execution is represented by a sequence of steps, each of them being the simultaneous (or parallel) firing of enabled transitions. Obviously, step semantics reduces the size of the state space to be analyzed, since, instead of analyzing a single transition, this semantics analyzes a set of transitions. However, a key problem in applying such semantics may be how much time is needed to find how many transitions belongs to a single step, mainly when considering larger models. This is a point to be investigated.

In this work, it is assumed that the scheduling algorithm always finds a feasible schedule, provided that at least one schedule exists. However, no formal proof asserting its correctness is given. One way to prove it is to use mathematical induction.

As presented at Section 7.3, the automatic modeling generates the same time Petri net model in two file formats: (i) a PNML file format; and (ii) a specific file format for the schedule generator. Another extension is to consider just the PNML format.

Although the scheduling synthesis may generate schedules considering a multiprocessor architecture, the code generator proposed in this thesis has been only focused on uniprocessor architectures. Thus, a possible extension would be the addition of this functionality into the code generator. Basically, synthesis of communication constructs will be the major addition.

The aim of any synthesis method is to implement the specification with minimum overhead. This is why the proposed code generation phase adopted a small dispatcher for improving the management of execution of tasks. Another solution for minimizing

the overhead due to tasks calling is the code concatenation approach. For instance, if two (or more) tasks always are executed in a chain (or sequence), these two codes are candidate to be concatenated. After the concatenation, both tasks can be seen as being one single task. Obviously, this solution reduces the amount of dispatcher calls.

In order to add some flexibility in the proposed methodology, the code generation method may generate code considering multiple operational modes. In this case, the dispatcher looks for conditions that allow a mode to be switched, where a mode is just an alternative pre-runtime schedule. However, such alternative schedules have to be previously added in the system. In this case, another possible extension is to propose a solution for adding, at runtime, new operational modes and conditions for mode-switching. A little modification on this extension could be the addition of new tasks.

9.4 Closing Remarks

The high complexity of embedded systems increases the difficulty in verifying design correctness. This verification is critical due to safety considerations in several application domains. In particular, this thesis proposed a formal approach for software synthesis in embedded hard real-time systems. As these systems need high predictability, this thesis proposed a code generator for guaranteeing that specified constraints are satisfied and, at the same time, increasing software quality and productivity.

Software synthesis for time-critical systems is a fertile research area since it has been little explored by the research community. New frontiers for automatic software synthesis based on formal timed models is opened up.

Bibliography

[1] Apache Jakarta Project. Version 1.4. January 2005. http://jakarta.apache.org/velocity/.

[2] 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.

[3] T. F. Abdelzaher and K. G. Shin. Comments on a pre-run-time scheduling algorithm for hard real-time systems. *IEEE Trans. Soft. Engineering*, 23(9):599–600, September 1997.

[4] K. Altisen, G. Göbler, A. Pnueli, J. Sifakis, S. Tripakis, and S. Yovine. A framework for scheduler synthesis. *IEEE Real-Time System Symposium*, pages 154–163, December 1999.

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

[6] T. Argewala and Y. Choed-Amphai. A synthesis rule for concurrent systems. *Design Automation Conference (DAC'78)*, pages 305–311, June 1978.

[7] N. Audsley and A. Burns. Real-time systems scheduling. Technical report, ycs 134, Department of Computer Science. University of York, 1990.

[8] N. C. Audsley. Deadline monotonic scheduling. Technical report, ycs 146, Department of Computer Science. University of York, 1990.

[9] J.C.M. Baeten. A brief history of process algebra. Technical Report CSR 04-02, Vakgroep Informatica, Technische Universiteit Eindhoven, 2004.

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

[11] 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.

[12] F. Balarin, L. Lavagno, P. Murthy, and A. Sangiovanni-Vincentelli. Scheduling for embedded real-time systems. *IEEE Design and Test of Computers*, pages 71–82, Jan-Mar 1998.

[13] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, January 1985.

[14] G. Berthelot. Checking Properties of Nets Using Transformations. In G. Rozenberg, editor, *Advances in Petri Nets*, volume 222 of *Lecture Notes in Computer Science*, pages 19–40. Springer-Verlag, 1986.

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

[16] E. Best and B. Grahlmann. Pep - more than a petri net tool. In *LCNS*, volume 1055, pages 397–401. Springer-Verlag, 1996.

[17] G. Bruno, A. Castella, G. Macario, and M. Pescarmona. Scheduling hard real time systems using high-level petri nets. In *Lecture Notes in Computer Science; 13th International Conference on Application and Theory of Petri Nets 1992, Sheffield, UK*, volume 616, pages 93–112. Springer-Verlag, June 1992.

[18] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation, special issue on "Simulation Software Development"*, 4:155–182, April 1994.

[19] A. Burns and A. Wellings. HRT-HOOD: A structured design method for hard real-time systems. *Real-Time Systems Journal*, 6(1):73–114, 1994.

[20] R. Camposano and R. Brayton. Partitioning before logic synthesis. *International Conference on Computer Aided Design*, 1987.

[21] C. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, 1999.

[22] J. M. Colom, E. Teruel, M. Silva, and S. Haddad. Structural methods. In C. Girault and R. Valk, editors, *Petri Nets for Systems Engineering: A Guide to Modeling, Verification and Applications*, chapter 15, pages 277–316. Springer, 2003.

[23] 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.

[24] G. de Jong and B. Lin. A communicating Petri net model for the design of concurrent asynchronous modules. *Design Automation Conference (DAC'94)*, 1994.

[25] J. Desel and J. Esparza. *Free Choice Petri Nets*. Cambridge University Press, January 1995.

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

[27] F. DiCesare and M. D. Jeng. Synthesis for manufacturing integration. In F. DiCesare, G.Harhalakis, J. M. Proth, M. Silva, and F. B. Vernadat, editors, *Practice of Petri Nets in Manufacturing*, chapter 3. Chapman & Hall, 1993.

[28] 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.

[29] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):336–390, March 1997.

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

[31] A. Ferrari and A. Sangiovanni-Vincentelli. System design: Traditional concepts and new paradigms. In *Proceedings of the International Conference on Computer Design (ICCD'99)*, pages 1–12. Austin, Texas, October 1999.

[32] A. Finkel. The Minimal Coverability Graph for Petri Nets. In G. Rozenberg, editor, *Advances in Petri Nets*, volume 674 of *Lecture Notes in Computer Science*, pages 210–243. Springer-Verlag, 1993.

[33] G. Fohler. *Flexibility in Statically Scheduled Hard Real-Time Systems.* PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. Vienna, Austria, 1994.

[34] D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems.* Prentice-Hall, New Jersey, 1994.

[35] D. Gajski, J. Zhu, and R. Domer. Essential issues in codesign. Technical Report ICS-97-26, Department of Information and Computer Science. University of California at Irvine, June 1997.

[36] M. Garey and D. Johnson. *Computer and Intractability: a Guide to the Theory of the NP-Completeness.* W. H. Freeman and Company, 1979.

[37] J. R. Garman. The bug heard round the world. *ACM SIGSOFT Software Engineering Notes*, 1981.

[38] 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.

[39] D. Harel. Statecharts: A visual formalism for complex systems. *Science for Computer Programming*, 1987.

[40] K. Havelund, M. Lowry, and J. Penix. Formal analysis of a space-craft controller using spin. *IEEE Transactions on Software Engineering*, 27(8):749–765, August 2001.

[41] C. Hoare. *Communicating Sequential Process.* Prentice-Hall, 1985.

[42] 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.

[43] F. Jahanian and A. Mok. Modechart: A specification language for real-time systems. *IEEE Transactions on Software Engineering*, 20(12):933–947, December 1994.

[44] A. Arcoverde Jr, G. Alves Jr, R. Lima, P. Maciel, M. Oliveira Jr, and R. Barreto. Ezpetri: A petri net interchange framework for eclipse based on PNML. In *Proceedings of the 1st International Symposium on Leveraging Applications of Formal Methods ISOLA'04.* October 30 - November 2 2004.

[45] 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.

[46] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, 220(4589):671–680, 1983.

[47] P. Koopman. Embedded system design issues: The rest of the story. *Proceedings of the International Conference on Computer Design*, Austin, October 7-9 1996.

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

[49] B. Krishnamurthy. An improved min-cut algorithm for partitioning VLSI networks. *IEEE Transactions on Computers*, 1984.

[50] 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.

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

[52] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley and Sons, England, 1990.

[53] 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.

[54] J. Lilius. Efficient state space search for time petri nets. In *Electronic Notes in Theoretical Computer Science*, volume 18. Elsevier Science, 1998.

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

[56] D. Lipcoll, D. Lawrie, and A. Sameh. *Eclipse Platform Technical Overview*. Object Technology International Inc., July 2001.

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

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

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

[60] A. Marsan, G. Balso, and G. Conte. A class of generalized stochastic petri nets for the performance analysis of multiprocessor systems. In *ACM Transactions on Computing Systems*, volume 2, pages 93–122. ACM, 1984.

[61] A. Marsan and G. Chiola. On petri nets with deterministic and exponentially distributed firing times. In G. Rozenberg, editor, *Advances in Petri Nets*, volume 266 of *Lecture Notes in Computer Science*, pages 132–145. Springer-Verlag, 1987.

[62] 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.

[63] G. Martin, H. Chang, and et al. *Surviving the SOC Revolution: A Guide to Platform Based Design*. Kluwer Academic Publishers, September 1999.

[64] 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.

[65] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1982.

[66] A. K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment*. PhD Thesis, MIT, May 1983.

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

[68] M. K. Molloy. *On the Integration of Delay and Throughput Measures in Distributed Processing Model*. PhD Thesis, UCLA, Los Angeles, CA, 1981.

[69] T. Murata. State equation, controllability, and maximal matchings of Petri nets. *IEEE Transactions on Automatic Control*, 22(3):412–416, June 1977.

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

[71] Y. Narahari and N. Viswanadham. A Petri net approach to the modelling and analysis of flexible manufacturing systems. *Annals of Operations Research*, 3:449–472, 1985.

[72] M. L. Neves. *Geração Automática de Modelos Temporizados para Geração Off-line de Escalas (in portuguese)*. Graduation Final Project, Centro de Informática. Universidade Federal de Pernambuco, August 2004.

[73] G. Palshikar. An introduction to model-checking. *Embedded Systems Programming*, December 2004. http://www.embedded.com/showArticle.jhtml?articleID=17603352.

[74] J. L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–252, September 1977.

[75] J. L. Peterson. *Petri Nets: An Introduction*. Prentice-Hall, 1981.

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

[77] R. Rajkumar. Synchronizations in real-time systems: A priority inheritance approach. 1991.

[78] C. Ramchandani. *Analysis of Asynchronous Concurrent Systems by Petri Nets*. PhD Thesis, MIT, Cambridge, USA, February 1974.

[79] W. Reisig. *A Primer in Petri Net Design*. Springer-Verlag, New York, 1992.

[80] A. Sangiovanni-Vincentelli. Defining platform-based design. *EEdesign*, 2001. http://www.eedesign.com/.

[81] 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.

[82] 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.

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

[84] 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.

[85] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham. Mode change protocols for priority-driven preemptive scheduling. *Real-Time Systems*, 1(3):243–265, December 1989.

[86] 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.

[87] L. Sieh, P. Haniak, and P. Richardson. Implementing transient fault tolerance in embedded real-time systems. *IEEE Electronics and Information Technology Conference*, 2001.

[88] 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.

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

[90] P. Starke and S. Roch. *INA - Integrated Net Analyzer - Version 2.2.* Humbolt Universität zu Berlin - Institut für Informatik, 1999.

[91] 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.

[92] A. Tanenbaum. *Structured Computer Organization.* Prentice-Hall, 2001.

[93] J. Tsai, S. Yang, and Y.-H. Chang. Timing constraint petri nets and their application to schedulability analysis of real-time system specifications. *IEEE Trans. Software Enginenring*, 21(1):32–49, January 1995.

[94] R. Valette. Analysis of Petri nets by stepwise refinement. *Journal of Computer Systems Science*, 18:35–46, 1979.

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

[96] A. Valmari. Compositional state space generation. *LNCS: Advances in Petri Nets*, 674:427–457, 1993.

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

[98] R. J. van Glabbeek. Bounded nondeterminism and the approximation induction principle in process algebra. In F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *Proceedings 4th Annual Symposium on Theoretical Aspects of Computer Science (STACS 87)*, pages 336–347. Passau, Germany, LNCS 247, Springer-Verlag, February 1987.

[99] W. Wang, A. Mok, and G. Fohler. Pre-scheduling. In *IEEE Transactions on Computers*. IEEE, 2004.

[100] M. Weber and E. Kindler. The petri net markup language. *Petri net Technology Communication Systems. Advances in Petri Nets.*, 2002.

[101] D. Xu, X. He, and Y. Deng. Compositional schedulability analysis of real-time systems using time petri nets. *IEEE Trans. Soft. Engineering*, 28(10):984–996, October 2002.

[102] J. Xu. Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations. *IEEE Trans. Soft. Engineering*, 19(2):139–154, February 1993.

[103] J. Xu. On inspection and verification of software with timing requirements. *IEEE Transactions on Software Engineering*, 29(8):705–720, August 2003.

[104] J. Xu and K. Lam. Integrating run-time scheduling and pre-run-time scheduling of real-time processes. In *23rd IFAC/IFIP Workshop on Real-Time Programming*. Shantou, China, june 1998.

[105] 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.

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

[107] 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.

[108] M. Young and L-C. Shu. Hybrid online/offline scheduling for hard real-time systems. *2nd International Symposium on Real-Time and Media Systems*, pages 231–240, July 1996.

[109] M. Zhou. *A Theory for the Synthesis and Augmentation of Petri Nets in Automation.* PhD Thesis, Rensselaer Polytechnic Institute, May 1990.

[110] M. Zhou and F. DiCesare. *Parallel and sequential mutual exclusions for Petri net modeling of manufacturing systems with shared resources.* IEEE Transactions on Robotics and Automation, 1991.

Appendix A
Model Checking

Usually bad design requirements lead to high cost of maintenance. The cost of errors in requirements are often high, requiring at least rework and maintenance. If you implement the incorrect requirements as they are, it may lead to incorrect system behavior in the field and high costs, such as loss of life and property, particularly in real-time, embedded safety-critical systems. Similar problems exist in ensuring the quality of system design.

In the last decade, the computer science research community has made tremendous progress in developing tools and techniques for verifying requirements and design. One of the most successful technique that has emerged is called model checking. When combined with strict use of a formal modeling language, it may automate the verification process fairly well. The aim of this appendix is to introduce model checking and show how it works. This appendix is based on [73].

A.1 Challenge

When checking design requirements, usually the designer is seeking for answers to a series of questions. Here are the general questions when checking requirements:

- Do they accurately reflect the users' requirements?
- Does everything stated match what the users want and have you included everything the users have requested?
- Are the requirements clearly written and unambiguous?
- Are they flexible and realizable for the engineers?
- Can the requirements be used to easily define acceptance test cases to check the conformance of the implementation against the requirements?

- Are the requirements written in an abstract and high-level manner, away from design, implementation, technology platforms and so on, so as to give enough freedom to the designer and developers to implement them efficiently?

Finding the answers to these questions is difficult and there is no easy way to do it.

Despite some help from modeling tools, the problem of ensuring the quality of requirements remains. The process is heavily manual and time consuming, involving reviews and sometimes partial prototyping. Using multiple notations introduces additional problems:

- which notation to use for what requirements?
- how to ensure that the descriptions in different notations are consistent with each other?

One way for improving the quality of requirements and design is to use automated tools to check the quality of various aspects of the requirements and design. But what tools? Building tools to check requirements or design written in natural language (Portuguese, English, etc) is clearly extremely difficult. It is necessary to enforce a clear, rigorous, and unambiguous formal language for stating the requirements. If the language for writing requirements and design has well-defined semantics, it may be feasible to develop tools to analyze the statements written in that language. This basic idea using a rigorous language for writing requirements or design is now acknowledged as a foundation for system verification.

A.2 Model Checking

Model checking is one of the most successful method for verifying requirements. The essential idea behind model checking is shown in Figure A.1. A model-checking tool accepts system requirements or design (called models) and a property (called specification) that the final system is expected to satisfy. The tool then outputs yes if the given model satisfies given specifications and generates a counterexample otherwise. The counterexample details why the model does not satisfy the specification. By studying the counterexample, you can find the source of the error in the model, correct the model, and try again. The idea is that by ensuring that the model satisfies enough system properties, the confidence is increased in the correctness of the model. The system requirements are called models because they represent requirements or design.

But what formal language works for defining models? There is no single answer, since requirements (or design) for systems in different application domains vary greatly.

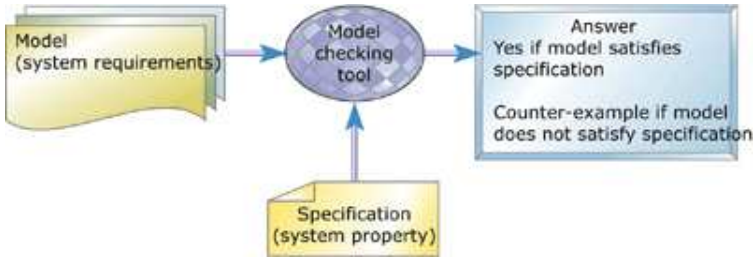

Figure A.1: The Model Checking Approach

For instance, requirements of a banking system and an aerospace system differ in size, structure, complexity, nature of system data, and operations performed. In contrast, most real-time embedded or safety-critical systems are control-oriented rather than data-orientedmeaning that dynamic behavior is much more important than business logic (the structure of and operations on the internal data maintained by the system). Such control-oriented systems occur in a wide variety of domains: aerospace, avionics, automotive, biomedical instrumentation, industrial automation and process control, railways, nuclear power plants, and so forth. Even communication and security protocols in digital hardware systems can be thought of as control oriented.

Section 2.1 presented a model taxonomy and some representative models of each class of such taxonomy, such as automata and extensions, Petri nets, program-state machine, and process algebras.

A.3 A Simple System Model

How model checking can be used for verifying properties of a simple embedded system? In order to answer this question, the symbolic model verifier (SMV) model-checking tool from Carnegie-Mellon University will be adopted. Of course, this model can also be written in other model-checking tools.

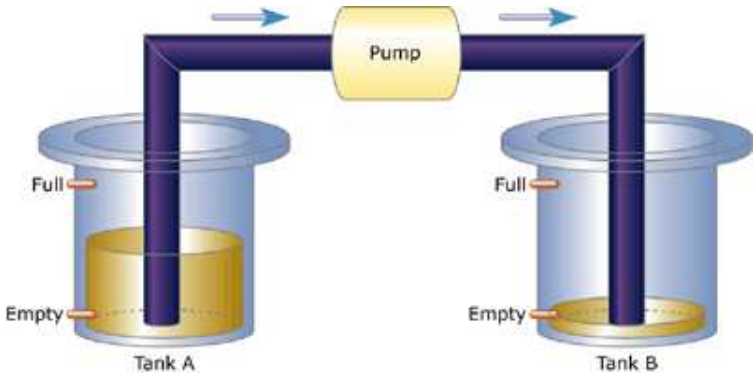

Figure A.2: A simple two tank pumping system

Consider a simple pumping control system that transfers water from a source tank

A into another sink tank B using a pump P, as shown in Figure A.2. Each tank has two level-meters: one to detect whether its level is empty and the other to detect whether its level is full. The tank level is ok if it is neither empty nor full; in other words, if it is above the empty mark but below the full mark.

Initially, both tanks are empty. The pump is to be switched on as soon as the water level in tank A reaches ok (from empty), provided that tank B is not full. The pump remains on as long as tank A is not empty and as long as tank B is not full. The pump is to be switched off as soon as either tank A becomes empty or tank B becomes full. The system should not attempt to switch the pump off (on) if it is already off (on). While this example may appear trivial, it easily extends to a controller for a complex network of pumps and pipes to control multiple source and sink tanks, such as those in water treatment facilities or chemical production plants.

```
MODULE main
VAR
   level_a : {empty, ok, full}; -- lower tank
   level_b : {empty, ok, full}; -- upper tank
   pump : {on, off};
ASSIGN
   next(level_a) := case
       level_a = empty : {empty, ok};
       level_a = ok & pump = off : {ok, full};
       level_a = ok & pump = on : {ok, empty, full};
       level_a = full & pump = off : full;
       level_a = full & pump = on : {ok, full};
       1 : {ok, empty, full};
   esac;
   next(level_b) := case
       level_b = empty & pump = off : empty;
       level_b = empty & pump = on : {empty, ok};
       level_b = ok & pump = off : {ok, empty};
       level_b = ok & pump = on : {ok, empty, full};
       level_b = full & pump = off : {ok, full};
       level_b = full & pump = on : {ok, full};
       1 : {ok, empty, full};
   esac;
   next(pump) := case
       pump = off & (level_a = ok | level_a = full) &
       (level_b = empty | level_b = ok) : on;
       pump = on & (level_a = empty | level_b = full) : off;
       1 : pump; -- keep pump status as it is
   esac;
INIT
   (pump = off)
SPEC
   -- pump if always off if ground tank is empty or up tank is full
   -- AG AF (pump = off -> (level_a = empty | level_b = full))
   -- it is always possible to reach a state when the up tank is ok or full
   AG (EF (level_b = ok | level_b = full))
```

Figure A.3: An SMV model description and requirements list

The model of this system in SMV is as follows and is shown in Figure A.3. The first VAR section declares that the system has three state variables. Variables level_a and level_b record the current level of the upper and lower tank respectively; at each "instant" these variables take a value, which can be either empty, ok, or full. Variable pump records whether the pump is on or off.

A system state is defined by a tuple of values for each of these three variables. For example, (level_a = empty, level_b=ok, pump=off) and (level_a = empty, level_b=full, pump=on) are possible system states. The INIT section, near the end, defines initial values for the variables (here, initially the pump is assumed to be off but the other two variables can have any value).

The ASSIGN section defines how the system changes from one state to another. Each next statement defines how the value of a particular variable changes. All these assignment statements are assumed to work in parallel; the next state is defined as the net result of executing all the assignment statements in this section. The lower tank can go from empty to the empty or ok state; from ok to either empty or full or remain ok if the pump is on; from ok to either ok or full if the pump is off; from full cannot change state if the pump is off; from full to ok or full if the pump is on. Similar changes are defined for the upper tank.

A.4 Paths and Specifications

Initially, the system could be in any of the nine states where there are no restrictions on the water level in A or B but the pump is assumed to be off. Let us denote a state by an ordered tuple <A,B,P> where A and B denote the current water level in tank A and B, and P denotes the current pump status. To illustrate, let us assume the initial state to be <empty,empty,off>. The next state from this state could be any of the <empty,empty,off>, <ok,empty,on>. From <ok,empty,on> the next state could be either of <ok,empty,on>, <ok,ok,on>, <full,empty,on>, <full,ok,on>, <empty,empty,off>, or <empty,ok,off>. For each of these states, the next possible states can be calculated.

The states can be arranged in the form of an infinite execution (or computation) tree, where the root is labeled with our chosen initial state and the children of any state denote the next possible states. A system execution is a path in this execution tree. In general, the system has infinitely many such execution paths. The aim of model checking is to examine whether or not the execution tree satisfies a user-given property specification.

The question now is how do we specify properties of paths (and states in the paths) of an execution tree? Computation tree logic (CTL) technically a branching time temporal logic is a simple and intuitive notation suitable for this purpose. CTL is an extension of the usual Boolean propositional logic (which includes the logical connectives such as and, or, not, implies) where additional temporal connectives are available.

Table A.1: Some temporal connectives in CTL

EX φ	true in current state if formula φ is true in at least one of the next states
EF φ	true in current state if there exists some state in some path beginning in current state that satisfies the formula φ
EG φ	true in current state if every state in some path beginning in current state satisfies the formula φ
AX φ	true in current state if formula φ is true in every one of the next states
AF φ	true in current state if there exists some state in every path beginning in current state that satisfies the formula φ
AG φ	true in current state if every state in every path beginning in current state satisfies the formula φ

Table A.1 and Figure A.4 illustrate the intuitive meaning of some of the basic temporal connectives in CTL. Basically, E (for some path) and A (for all paths) are path quantifiers for paths beginning from a state. F (for some state) and G (for all states) are state quantifiers for states in a path.

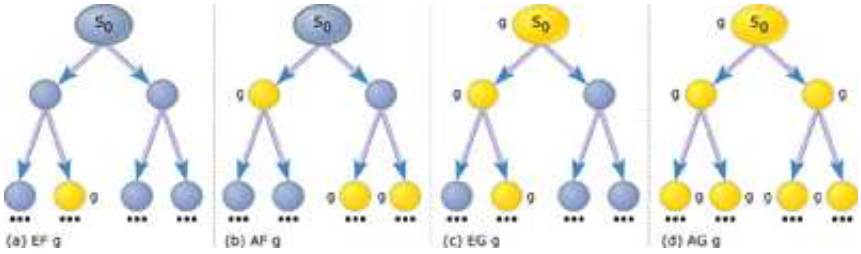

Figure A.4: Intuition for CTL formulae which are satisfied at state s0

Given a property and a (possibly infinite) computation tree T corresponding to the system model, a model-checking algorithm essentially examines T to check if T satisfies the property. For example, consider a property $AF\ g$ where g is a propositional formula not involving any CTL connectives. Figure A.4(b) shows an example of a computation tree T. The property $AF\ g$ is true for this T if it is true at the root state s_0 in other words, if there is some state in every path in T starting at s_0 such that the formula g is true in that state.

Figure A.4(b) shows that g is true at the root of the left subtree (indicated by the filled circle). Hence all paths from s_0 to left child and further down in the left subtree satisfy the property. Now suppose g is not true at the right child of s_0; hence the property is recursively checked for all its children. Figure A.4(b) shows that g is true at all children of the right child of s_0 (indicated by filled circles) and hence the property is true for the right subtree of s_0. Thus the property is true for all subtrees of s_0 and hence it is also true at s_0.

Figure A.4 summarizes the similar reasoning used to check properties stated in other forms such as $EG\ g$ and $AG\ g$. Of course, in practice, the model-checking algorithms are really far more complex than this; they use sophisticated tricks to prune the state space to avoid checking those parts where the property is guaranteed to be true.

In SMV, a property to be verified is given by the user in the SPEC section. The logical connectives not, or, and, implies (if-then) are represented by !, |, &, and $\rightarrow$ respectively. The CTL temporal connectives are AF, AG, EF, EG, and so on. The property $AF(pump = on)$ states that for every path beginning at the initial state, there is a state in that path at which the pump is on. This property is clearly false in the initial state since there is a path from the initial state in which the pump always remains off (for example, if tank A forever remains empty). If this property is specified in the SPEC section, SMV generates the following counterexample for the property. The loop indicates an infinite sequence of states (in other words, a path) beginning at the initial state such that tank B is full in every state of the path and hence pump is off.

```
-- specification AF pump = on is false
-- as demonstrated by the following execution sequence
-- loop starts here
state 1.1:
level_a = full
level_b = full
pump = off

state 1.2:
```

The dual property $AF(pump = off)$ states that for every path beginning at the initial state, there is a state in that path at which the pump is off. This property is trivially true at the initial state, since in the initial state itself (which is included in all paths) pump = off is true.

You can specify interesting and complex properties by combining temporal and logical connectives. The property AG $((pump = off) \rightarrow AF(pump = on))$ states that it's always the case that if pump is off then it eventually becomes on. This property is clearly false in the initial state. The property $AGAF(pump = off \rightarrow (level\_a = empty \mid level\_b = full))$ states that pump is always off if ground tank is empty or the upper tank is full. The property $AG(EF(level\_b = ok \mid level\_b = full))$ states that it is always possible to reach a state when the upper tank is ok or full.

A.5 Model Checking in Practice

Model checking has proven to be a tremendously successful technology to verify requirements and design for a variety of systems, particularly in hardware systems and real-time embedded and safety-critical systems. For example, the SPIN model-checker was used to verify the multi-threaded plan execution module in NASA's DEEP SPACE mission and discovered five previously unknown concurrency errors [40].

However, there are some major issues to deal with when using model checking in practice. For example:

- Every model-checking tool comes with its own modeling language that provides no way to automatically translate informal requirement descriptions into this language. This translation is necessarily manual and hence it is difficult to check whether the model correctly represents your system. In fact, there may be parts of your requirements that may be difficult or even impossible to model in the given modeling notation.

- Similar problems exist for the tool-specific property specification notation, which is often a variant of CTL, CTL*, or propositional linear temporal logic (PLTL). Some properties to be verified may be difficult or even impossible to express in the notation.

- The number of states in your model may be extremely large. Although model-checking algorithms include ingenious ways to reduce this state space, the model checker may still take too long to verify a given property or "give up" during this task. In such cases the user has to put in more work, such as verifying parts of the model separately or reducing the state space by reducing domains of variables.

Nevertheless, model checking is likely to prove an invaluable way to verify system requirements or design. It often leads to early detection of the shortcomings in the requirements or design, thereby leading to large savings in later rework.

Appendix B

Model Checking Verification Steps

B.1 Mutual Exclusive Marking

```
s1 sat? AG-(P16 &P17 )
..s1 sat? -(P16 &P17 )
...s1 sat? (P16 &P17 )
....s1 sat? P16
....s1 sat P16 : FALSE
...s1 sat (P16 &P17 ): FALSE
..s1 sat -(P16 &P17 ): TRUE
..s1 =16=> s2
..s2 sat? -(P16 &P17 )
...s2 sat? (P16 &P17 )
....s2 sat? P16
....s2 sat P16 : FALSE
...s2 sat (P16 &P17 ): FALSE
..s2 sat -(P16 &P17 ): TRUE
..s2 ==1=> s3
..s3 sat? -(P16 &P17 )
...s3 sat? (P16 &P17 )
....s3 sat? P16
....s3 sat P16 : FALSE
...s3 sat (P16 &P17 ): FALSE
..s3 sat -(P16 &P17 ): TRUE
..s3 ==2=> s4
..s4 sat? -(P16 &P17 )
...s4 sat? (P16 &P17 )
....s4 sat? P16
....s4 sat P16 : FALSE
...s4 sat (P16 &P17 ): FALSE
..s4 sat -(P16 &P17 ): TRUE
..s4 ==7=> s5
..s5 sat? -(P16 &P17 )
...s5 sat? (P16 &P17 )
....s5 sat? P16
....s5 sat P16 : FALSE
...s5 sat (P16 &P17 ): FALSE
..s5 sat -(P16 &P17 ): TRUE
..s5 ==8=> s6
..s6 sat? -(P16 &P17 )
...s6 sat? (P16 &P17 )
....s6 sat? P16
....s6 sat P16 : FALSE
...s6 sat (P16 &P17 ): FALSE
..s6 sat -(P16 &P17 ): TRUE
..s6 =15=> s7
..s7 sat? -(P16 &P17 )
...s7 sat? (P16 &P17 )
....s7 sat? P16
....s7 sat P16 : FALSE
...s7 sat (P16 &P17 ): FALSE
..s7 sat -(P16 &P17 ): TRUE
..s7 ==4=> s8
..s8 sat? -(P16 &P17 )
...s8 sat? (P16 &P17 )
....s8 sat? P16
....s8 sat P16 : FALSE
...s8 sat (P16 &P17 ): FALSE
..s8 sat -(P16 &P17 ): TRUE
..s8 ==6=> s9
..s9 sat? -(P16 &P17 )
...s9 sat? (P16 &P17 )
....s9 sat? P16
....s9 sat P16 : FALSE
...s9 sat (P16 &P17 ): FALSE
..s9 sat -(P16 &P17 ): TRUE
..s9 =12=> s10
..s10 sat? -(P16 &P17 )
...s10 sat? (P16 &P17 )
....s10 sat? P16
....s10 sat P16 : FALSE
...s10 sat (P16 &P17 ): FALSE
..s10 sat -(P16 &P17 ): TRUE
..s10 ==9=> s11
..s11 sat? -(P16 &P17 )
...s11 sat? (P16 &P17 )
....s11 sat? P16
....s11 sat P16 : TRUE
.........s11 sat? P17
.........s11 sat P17 : FALSE
...s11 sat (P16 &P17 ): FALSE
..s11 sat -(P16 &P17 ): TRUE
..s11 ==5=> s12
..s12 sat? -(P16 &P17 )
...s12 sat? (P16 &P17 )
....s12 sat? P16
....s12 sat P16 : TRUE
.........s12 sat? P17
.........s12 sat P17 : FALSE
...s12 sat (P16 &P17 ): FALSE
..s12 sat -(P16 &P17 ): TRUE
..s12 =11=> s13
..s13 sat? -(P16 &P17 )
...s13 sat? (P16 &P17 )
....s13 sat? P16
....s13 sat P16 : TRUE
.........s13 sat? P17
.........s13 sat P17 : FALSE
...s13 sat (P16 &P17 ): FALSE
..s13 sat -(P16 &P17 ): TRUE
..s13 no successor
s13 sat AG-(P16 &P17 ): TRUE
..s8 =12=> s14
..s14 sat? -(P16 &P17 )
...s14 sat? (P16 &P17 )
....s14 sat? P16
....s14 sat P16 : FALSE
...s14 sat (P16 &P17 ): FALSE
..s14 sat -(P16 &P17 ): TRUE
..s14 ==9=> s15
..s15 sat? -(P16 &P17 )
...s15 sat? (P16 &P17 )
....s15 sat? P16
....s15 sat P16 : TRUE
```

```
.........s15 sat? P17
.........s15 sat P17 : FALSE
...s15 sat (P16 &P17 ): FALSE
..s15 sat -(P16 &P17 ): TRUE
..s15 ==5=> s16
..s16 sat? -(P16 &P17 )
...s16 sat? (P16 &P17 )
....s16 sat? P16
....s16 sat P16 : TRUE
.........s16 sat? P17
.........s16 sat P17 : FALSE
...s16 sat (P16 &P17 ): FALSE
..s16 sat -(P16 &P17 ): TRUE
..s16 ==6=> s12 visited
..s15 ==6=> s11 visited
..s5 =12=> s17
..s17 sat? -(P16 &P17 )
...s17 sat? (P16 &P17 )
....s17 sat? P16
....s17 sat P16 : FALSE
...s17 sat (P16 &P17 ): FALSE
..s17 sat -(P16 &P17 ): TRUE
..s17 ==9=> s18
..s18 sat? -(P16 &P17 )
...s18 sat? (P16 &P17 )
....s18 sat? P16
....s18 sat P16 : TRUE
.........s18 sat? P17
.........s18 sat P17 : FALSE
...s18 sat (P16 &P17 ): FALSE
..s18 sat -(P16 &P17 ): TRUE
..s18 ==5=> s19
..s19 sat? -(P16 &P17 )
...s19 sat? (P16 &P17 )
....s19 sat? P16
....s19 sat P16 : TRUE
.........s19 sat? P17
.........s19 sat P17 : FALSE
...s19 sat (P16 &P17 ): FALSE
..s19 sat -(P16 &P17 ): TRUE
..s19 ==8=> s20
..s20 sat? -(P16 &P17 )
...s20 sat? (P16 &P17 )
....s20 sat? P16
....s20 sat P16 : TRUE
.........s20 sat? P17
.........s20 sat P17 : FALSE
...s20 sat (P16 &P17 ): FALSE
..s20 sat -(P16 &P17 ): TRUE
..s20 =13=> s21
..s21 sat? -(P16 &P17 )
...s21 sat? (P16 &P17 )
....s21 sat? P16
....s21 sat P16 : TRUE
.........s21 sat? P17
.........s21 sat P17 : FALSE
...s21 sat (P16 &P17 ): FALSE
..s21 sat -(P16 &P17 ): TRUE
..s21 =10=> s22
..s22 sat? -(P16 &P17 )
...s22 sat? (P16 &P17 )
....s22 sat? P16
....s22 sat P16 : FALSE
...s22 sat (P16 &P17 ): FALSE
..s22 sat -(P16 &P17 ): TRUE
..s22 ==6=> s23
..s23 sat? -(P16 &P17 )
...s23 sat? (P16 &P17 )
....s23 sat? P16
....s23 sat P16 : FALSE
...s23 sat (P16 &P17 ): FALSE
..s23 sat -(P16 &P17 ): TRUE
..s23 =11=> s24
..s24 sat? -(P16 &P17 )
...s24 sat? (P16 &P17 )
....s24 sat? P16
....s24 sat P16 : FALSE
...s24 sat (P16 &P17 ): FALSE
..s24 sat -(P16 &P17 ): TRUE
..s24 no successor
s24 sat AG-(P16 &P17 ): TRUE
..s18 ==8=> s25
..s25 sat? -(P16 &P17 )
...s25 sat? (P16 &P17 )
....s25 sat? P16
....s25 sat P16 : TRUE
.........s25 sat? P17
.........s25 sat P17 : FALSE
...s25 sat (P16 &P17 ): FALSE
..s25 sat -(P16 &P17 ): TRUE
..s25 ==5=> s20 visited
..s25 =13=> s26
..s26 sat? -(P16 &P17 )
...s26 sat? (P16 &P17 )
....s26 sat? P16
....s26 sat P16 : TRUE
.........s26 sat? P17
.........s26 sat P17 : FALSE
...s26 sat (P16 &P17 ): FALSE
..s26 sat -(P16 &P17 ): TRUE
..s26 =10=> s27
..s27 sat? -(P16 &P17 )
...s27 sat? (P16 &P17 )
....s27 sat? P16
....s27 sat P16 : FALSE
...s27 sat (P16 &P17 ): FALSE
..s27 sat -(P16 &P17 ): TRUE
..s27 ==5=> s22 visited
..s27 ==6=> s28
..s28 sat? -(P16 &P17 )
...s28 sat? (P16 &P17 )
....s28 sat? P16
....s28 sat P16 : FALSE
...s28 sat (P16 &P17 ): FALSE
..s28 sat -(P16 &P17 ): TRUE
..s28 ==5=> s23 visited
..s4 ==8=> s29
..s29 sat? -(P16 &P17 )
...s29 sat? (P16 &P17 )
....s29 sat? P16
....s29 sat P16 : FALSE
...s29 sat (P16 &P17 ): FALSE
..s29 sat -(P16 &P17 ): TRUE
..s29 =15=> s30
..s30 sat? -(P16 &P17 )
...s30 sat? (P16 &P17 )
....s30 sat? P16
....s30 sat P16 : FALSE
...s30 sat (P16 &P17 ): FALSE
..s30 sat -(P16 &P17 ): TRUE
..s30 ==4=> s31
..s31 sat? -(P16 &P17 )
...s31 sat? (P16 &P17 )
....s31 sat? P16
....s31 sat P16 : FALSE
...s31 sat (P16 &P17 ): FALSE
..s31 sat -(P16 &P17 ): TRUE
..s31 ==6=> s32
..s32 sat? -(P16 &P17 )
...s32 sat? (P16 &P17 )
....s32 sat? P16
....s32 sat P16 : FALSE
...s32 sat (P16 &P17 ): FALSE
..s32 sat -(P16 &P17 ): TRUE
..s32 ==7=> s9 visited
..s31 ==7=> s8 visited
..s30 ==7=> s7 visited
..s3 ==7=> s33
..s33 sat? -(P16 &P17 )
...s33 sat? (P16 &P17 )
....s33 sat? P16
....s33 sat P16 : FALSE
...s33 sat (P16 &P17 ): FALSE
..s33 sat -(P16 &P17 ): TRUE
..s33 ==2=> s5 visited
..s33 =12=> s34
..s34 sat? -(P16 &P17 )
...s34 sat? (P16 &P17 )
....s34 sat? P16
....s34 sat P16 : FALSE
...s34 sat (P16 &P17 ): FALSE
..s34 sat -(P16 &P17 ): TRUE
..s34 ==9=> s35
..s35 sat? -(P16 &P17 )
...s35 sat? (P16 &P17 )
....s35 sat? P16
....s35 sat P16 : TRUE
.........s35 sat? P17
.........s35 sat P17 : FALSE
...s35 sat (P16 &P17 ): FALSE
..s35 sat -(P16 &P17 ): TRUE
..s35 ==2=> s18 visited
..s35 ==5=> s36
..s36 sat? -(P16 &P17 )
...s36 sat? (P16 &P17 )
....s36 sat? P16
....s36 sat P16 : TRUE
```

```
.........s36 sat? P17
.........s36 sat P17 : FALSE
...s36 sat (P16 &P17 ): FALSE
..s36 sat -(P16 &P17 ): TRUE
..s36 ==2=> s19 visited
..s2 ==2=> s37
..s37 sat? -(P16 &P17 )
...s37 sat? (P16 &P17 )
....s37 sat? P16
....s37 sat P16 : FALSE
...s37 sat (P16 &P17 ): FALSE
..s37 sat -(P16 &P17 ): TRUE
..s37 ==1=> s4 visited
..s37 ==8=> s38
..s38 sat? -(P16 &P17 )
...s38 sat? (P16 &P17 )
....s38 sat? P16
....s38 sat P16 : FALSE
...s38 sat (P16 &P17 ): FALSE
..s38 sat -(P16 &P17 ): TRUE
..s38 =15=> s39
..s39 sat? -(P16 &P17 )
...s39 sat? (P16 &P17 )
....s39 sat? P16
....s39 sat P16 : FALSE
...s39 sat (P16 &P17 ): FALSE
..s39 sat -(P16 &P17 ): TRUE
..s39 ==1=> s30 visited
..s39 ==4=> s40
..s40 sat? -(P16 &P17 )
...s40 sat? (P16 &P17 )
....s40 sat? P16
....s40 sat P16 : FALSE
...s40 sat (P16 &P17 ): FALSE
..s40 sat -(P16 &P17 ): TRUE
..s40 ==1=> s31 visited
..s40 ==6=> s41
..s41 sat? -(P16 &P17 )
...s41 sat? (P16 &P17 )
....s41 sat? P16
....s41 sat P16 : FALSE
...s41 sat (P16 &P17 ): FALSE
..s41 sat -(P16 &P17 ): TRUE
..s41 ==1=> s32
s1 sat AG-(P16 &P17 ): TRUE
```

B.2 Processor Utilization

```
s1 sat? AG-(P6 &P7 )
..s1 sat? -(P6 &P7 )
...s1 sat? (P6 &P7 )
....s1 sat? P6
....s1 sat P6 : FALSE
...s1 sat (P6 &P7 ): FALSE
..s1 sat -(P6 &P7 ): TRUE
..s1 =16=> s2
..s2 sat? -(P6 &P7 )
...s2 sat? (P6 &P7 )
....s2 sat? P6
....s2 sat P6 : FALSE
...s2 sat (P6 &P7 ): FALSE
..s2 sat -(P6 &P7 ): TRUE
..s2 ==1=> s3
..s3 sat? -(P6 &P7 )
...s3 sat? (P6 &P7 )
....s3 sat? P6
....s3 sat P6 : FALSE
...s3 sat (P6 &P7 ): FALSE
..s3 sat -(P6 &P7 ): TRUE
..s3 ==2=> s4
..s4 sat? -(P6 &P7 )
...s4 sat? (P6 &P7 )
....s4 sat? P6
....s4 sat P6 : FALSE
...s4 sat (P6 &P7 ): FALSE
..s4 sat -(P6 &P7 ): TRUE
..s4 ==7=> s5
..s5 sat? -(P6 &P7 )
...s5 sat? (P6 &P7 )
....s5 sat? P6
....s5 sat P6 : FALSE
...s5 sat (P6 &P7 ): FALSE
..s5 sat -(P6 &P7 ): TRUE
..s5 ==8=> s6
..s6 sat? -(P6 &P7 )
...s6 sat? (P6 &P7 )
....s6 sat? P6
....s6 sat P6 : FALSE
...s6 sat (P6 &P7 ): FALSE
..s6 sat -(P6 &P7 ): TRUE
..s6 =15=> s7
..s7 sat? -(P6 &P7 )
...s7 sat? (P6 &P7 )
....s7 sat? P6
....s7 sat P6 : FALSE
...s7 sat (P6 &P7 ): FALSE
..s7 sat -(P6 &P7 ): TRUE
..s7 ==4=> s8
..s8 sat? -(P6 &P7 )
...s8 sat? (P6 &P7 )
....s8 sat? P6
....s8 sat P6 : FALSE
...s8 sat (P6 &P7 ): FALSE
..s8 sat -(P6 &P7 ): TRUE
..s8 ==6=> s9
..s9 sat? -(P6 &P7 )
...s9 sat? (P6 &P7 )
....s9 sat? P6
....s9 sat P6 : FALSE
...s9 sat (P6 &P7 ): FALSE
..s9 sat -(P6 &P7 ): TRUE
..s9 =12=> s10
..s10 sat? -(P6 &P7 )
...s10 sat? (P6 &P7 )
....s10 sat? P6
....s10 sat P6 : TRUE
.........s10 sat? P7
.........s10 sat P7 : FALSE
...s10 sat (P6 &P7 ): FALSE
..s10 sat -(P6 &P7 ): TRUE
..s10 ==9=> s11
..s11 sat? -(P6 &P7 )
...s11 sat? (P6 &P7 )
....s11 sat? P6
....s11 sat P6 : FALSE
...s11 sat (P6 &P7 ): FALSE
..s11 sat -(P6 &P7 ): TRUE
..s11 ==5=> s12
..s12 sat? -(P6 &P7 )
...s12 sat? (P6 &P7 )
....s12 sat? P6
....s12 sat P6 : FALSE
...s12 sat (P6 &P7 ): FALSE
..s12 sat -(P6 &P7 ): TRUE
..s12 =11=> s13
..s13 sat? -(P6 &P7 )
...s13 sat? (P6 &P7 )
....s13 sat? P6
....s13 sat P6 : FALSE
...s13 sat (P6 &P7 ): FALSE
..s13 sat -(P6 &P7 ): TRUE
..s13 no successor
s13 sat AG-(P6 &P7 ): TRUE
..s8 =12=> s14
..s14 sat? -(P6 &P7 )
...s14 sat? (P6 &P7 )
....s14 sat? P6
....s14 sat P6 : TRUE
.........s14 sat? P7
.........s14 sat P7 : FALSE
...s14 sat (P6 &P7 ): FALSE
..s14 sat -(P6 &P7 ): TRUE
..s14 ==9=> s15
..s15 sat? -(P6 &P7 )
...s15 sat? (P6 &P7 )
....s15 sat? P6
....s15 sat P6 : FALSE
...s15 sat (P6 &P7 ): FALSE
..s15 sat -(P6 &P7 ): TRUE
..s15 ==5=> s16
..s16 sat? -(P6 &P7 )
...s16 sat? (P6 &P7 )
```

```
....s16 sat? P6
....s16 sat P6 : FALSE
...s16 sat (P6 &P7 ): FALSE
..s16 sat -(P6 &P7 ): TRUE
..s16 ==6=> s12 visited
..s15 ==6=> s11 visited
..s5 =12=> s17
..s17 sat? -(P6 &P7 )
...s17 sat? (P6 &P7 )
....s17 sat? P6
....s17 sat P6 : TRUE
........s17 sat? P7
........s17 sat P7 : FALSE
...s17 sat (P6 &P7 ): FALSE
..s17 sat -(P6 &P7 ): TRUE
..s17 ==9=> s18
..s18 sat? -(P6 &P7 )
...s18 sat? (P6 &P7 )
....s18 sat? P6
....s18 sat P6 : FALSE
...s18 sat (P6 &P7 ): FALSE
..s18 sat -(P6 &P7 ): TRUE
..s18 ==5=> s19
..s19 sat? -(P6 &P7 )
...s19 sat? (P6 &P7 )
....s19 sat? P6
....s19 sat P6 : FALSE
...s19 sat (P6 &P7 ): FALSE
..s19 sat -(P6 &P7 ): TRUE
..s19 ==8=> s20
..s20 sat? -(P6 &P7 )
...s20 sat? (P6 &P7 )
....s20 sat? P6
....s20 sat P6 : FALSE
...s20 sat (P6 &P7 ): FALSE
..s20 sat -(P6 &P7 ): TRUE
..s20 =13=> s21
..s21 sat? -(P6 &P7 )
...s21 sat? (P6 &P7 )
....s21 sat? P6
....s21 sat P6 : FALSE
...s21 sat (P6 &P7 ): FALSE
..s21 sat -(P6 &P7 ): TRUE
..s21 =10=> s22
..s22 sat? -(P6 &P7 )
...s22 sat? (P6 &P7 )
....s22 sat? P6
....s22 sat P6 : FALSE
...s22 sat (P6 &P7 ): FALSE
..s22 sat -(P6 &P7 ): TRUE
..s22 ==6=> s23
..s23 sat? -(P6 &P7 )
...s23 sat? (P6 &P7 )
....s23 sat? P6
....s23 sat P6 : FALSE
...s23 sat (P6 &P7 ): FALSE
..s23 sat -(P6 &P7 ): TRUE
..s23 =11=> s24
..s24 sat? -(P6 &P7 )
...s24 sat? (P6 &P7 )
....s24 sat? P6
....s24 sat P6 : FALSE
...s24 sat (P6 &P7 ): FALSE
..s24 sat -(P6 &P7 ): TRUE
..s24 no successor
s24 sat AG-(P6 &P7 ): TRUE
..s18 ==8=> s25
..s25 sat? -(P6 &P7 )
...s25 sat? (P6 &P7 )
....s25 sat? P6
....s25 sat P6 : FALSE
...s25 sat (P6 &P7 ): FALSE
..s25 sat -(P6 &P7 ): TRUE
..s25 ==5=> s20 visited
..s25 =13=> s26
..s26 sat? -(P6 &P7 )
...s26 sat? (P6 &P7 )
....s26 sat? P6
....s26 sat P6 : FALSE
...s26 sat (P6 &P7 ): FALSE
..s26 sat -(P6 &P7 ): TRUE
..s26 =10=> s27
..s27 sat? -(P6 &P7 )
...s27 sat? (P6 &P7 )
....s27 sat? P6
....s27 sat P6 : FALSE
...s27 sat (P6 &P7 ): FALSE
..s27 sat -(P6 &P7 ): TRUE
..s27 ==5=> s22 visited
..s27 ==6=> s28
..s28 sat? -(P6 &P7 )
...s28 sat? (P6 &P7 )
....s28 sat? P6
....s28 sat P6 : FALSE
...s28 sat (P6 &P7 ): FALSE
..s28 sat -(P6 &P7 ): TRUE
..s28 ==5=> s23 visited
..s4 ==8=> s29
..s29 sat? -(P6 &P7 )
...s29 sat? (P6 &P7 )
....s29 sat? P6
....s29 sat P6 : FALSE
...s29 sat (P6 &P7 ): FALSE
..s29 sat -(P6 &P7 ): TRUE
..s29 =15=> s30
..s30 sat? -(P6 &P7 )
...s30 sat? (P6 &P7 )
....s30 sat? P6
....s30 sat P6 : FALSE
...s30 sat (P6 &P7 ): FALSE
..s30 sat -(P6 &P7 ): TRUE
..s30 ==4=> s31
..s31 sat? -(P6 &P7 )
...s31 sat? (P6 &P7 )
....s31 sat? P6
....s31 sat P6 : FALSE
...s31 sat (P6 &P7 ): FALSE
..s31 sat -(P6 &P7 ): TRUE
..s31 ==6=> s32
..s32 sat? -(P6 &P7 )
...s32 sat? (P6 &P7 )
....s32 sat? P6
....s32 sat P6 : FALSE
...s32 sat (P6 &P7 ): FALSE
..s32 sat -(P6 &P7 ): TRUE
..s32 ==7=> s9 visited
..s31 ==7=> s8 visited
..s30 ==7=> s7 visited
..s3 ==7=> s33
..s33 sat? -(P6 &P7 )
...s33 sat? (P6 &P7 )
....s33 sat? P6
....s33 sat P6 : FALSE
...s33 sat (P6 &P7 ): FALSE
..s33 sat -(P6 &P7 ): TRUE
..s33 ==2=> s5 visited
..s33 =12=> s34
..s34 sat? -(P6 &P7 )
...s34 sat? (P6 &P7 )
....s34 sat? P6
....s34 sat P6 : TRUE
........s34 sat? P7
........s34 sat P7 : FALSE
...s34 sat (P6 &P7 ): FALSE
..s34 sat -(P6 &P7 ): TRUE
..s34 ==9=> s35
..s35 sat? -(P6 &P7 )
...s35 sat? (P6 &P7 )
....s35 sat? P6
....s35 sat P6 : FALSE
...s35 sat (P6 &P7 ): FALSE
..s35 sat -(P6 &P7 ): TRUE
..s35 ==2=> s18 visited
..s35 ==5=> s36
..s36 sat? -(P6 &P7 )
...s36 sat? (P6 &P7 )
....s36 sat? P6
....s36 sat P6 : FALSE
...s36 sat (P6 &P7 ): FALSE
..s36 sat -(P6 &P7 ): TRUE
..s36 ==2=> s19 visited
..s2 ==2=> s37
..s37 sat? -(P6 &P7 )
...s37 sat? (P6 &P7 )
....s37 sat? P6
....s37 sat P6 : FALSE
...s37 sat (P6 &P7 ): FALSE
..s37 sat -(P6 &P7 ): TRUE
..s37 ==1=> s4 visited
..s37 ==8=> s38
..s38 sat? -(P6 &P7 )
...s38 sat? (P6 &P7 )
....s38 sat? P6
....s38 sat P6 : FALSE
...s38 sat (P6 &P7 ): FALSE
..s38 sat -(P6 &P7 ): TRUE
..s38 =15=> s39
```

```
..s39 sat? -(P6 &P7 )
...s39 sat? (P6 &P7 )
....s39 sat? P6
....s39 sat P6 : FALSE
...s39 sat (P6 &P7 ): FALSE
..s39 sat -(P6 &P7 ): TRUE
..s39 ==1=> s30 visited
..s39 ==4=> s40
```

```
..s40 sat? -(P6 &P7 )
...s40 sat? (P6 &P7 )
....s40 sat? P6
....s40 sat P6 : FALSE
...s40 sat (P6 &P7 ): FALSE
..s40 sat -(P6 &P7 ): TRUE
..s40 ==1=> s31 visited
..s40 ==6=> s41
```

```
..s41 sat? -(P6 &P7 )
...s41 sat? (P6 &P7 )
....s41 sat? P6
....s41 sat P6 : FALSE
...s41 sat (P6 &P7 ): FALSE
..s41 sat -(P6 &P7 ): TRUE
..s41 ==1=> s32
s1 sat AG-(P6 &P7 ): TRUE
```

B.3 Precedence Relation

```
s1 sat? AG-((P10 &P6 )&(P8 &-P2 ))
..s1 sat? -((P10 &P6 )&(P8 &-P2 ))
...s1 sat? ((P10 &P6 )&(P8 &-P2 ))
....s1 sat? (P10 &P6 )
.....s1 sat? P10
.....s1 sat P10 : FALSE
....s1 sat (P10 &P6 ): FALSE
...s1 sat ((P10 &P6 )&(P8 &-P2 )): FALSE
..s1 sat -((P10 &P6 )&(P8 &-P2 )): TRUE
..s1 =12=> s2
..s2 sat? -((P10 &P6 )&(P8 &-P2 ))
...s2 sat? ((P10 &P6 )&(P8 &-P2 ))
....s2 sat? (P10 &P6 )
.....s2 sat? P10
.....s2 sat P10 : FALSE
....s2 sat (P10 &P6 ): FALSE
...s2 sat ((P10 &P6 )&(P8 &-P2 )): FALSE
..s2 sat -((P10 &P6 )&(P8 &-P2 )): TRUE
..s2 =10=> s3
..s3 sat? -((P10 &P6 )&(P8 &-P2 ))
...s3 sat? ((P10 &P6 )&(P8 &-P2 ))
....s3 sat? (P10 &P6 )
.....s3 sat? P10
.....s3 sat P10 : FALSE
....s3 sat (P10 &P6 ): FALSE
...s3 sat ((P10 &P6 )&(P8 &-P2 )): FALSE
..s3 sat -((P10 &P6 )&(P8 &-P2 )): TRUE
..s3 ==7=> s4
..s4 sat? -((P10 &P6 )&(P8 &-P2 ))
...s4 sat? ((P10 &P6 )&(P8 &-P2 ))
....s4 sat? (P10 &P6 )
.....s4 sat? P10
.....s4 sat P10 : FALSE
....s4 sat (P10 &P6 ): FALSE
...s4 sat ((P10 &P6 )&(P8 &-P2 )): FALSE
..s4 sat -((P10 &P6 )&(P8 &-P2 )): TRUE
..s4 =11=> s5
..s5 sat? -((P10 &P6 )&(P8 &-P2 ))
...s5 sat? ((P10 &P6 )&(P8 &-P2 ))
....s5 sat? (P10 &P6 )
.....s5 sat? P10
.....s5 sat P10 : FALSE
....s5 sat (P10 &P6 ): FALSE
...s5 sat ((P10 &P6 )&(P8 &-P2 )): FALSE
..s5 sat -((P10 &P6 )&(P8 &-P2 )): TRUE
..s5 ==8=> s6
..s6 sat? -((P10 &P6 )&(P8 &-P2 ))
...s6 sat? ((P10 &P6 )&(P8 &-P2 ))
....s6 sat? (P10 &P6 )
.....s6 sat? P10
.....s6 sat P10 : FALSE
....s6 sat (P10 &P6 ): FALSE
...s6 sat ((P10 &P6 )&(P8 &-P2 )): FALSE
..s6 sat -((P10 &P6 )&(P8 &-P2 )): TRUE
..s6 ==6=> s7
..s7 sat? -((P10 &P6 )&(P8 &-P2 ))
...s7 sat? ((P10 &P6 )&(P8 &-P2 ))
....s7 sat? (P10 &P6 )
.....s7 sat? P10
.....s7 sat P10 : FALSE
....s7 sat (P10 &P6 ): FALSE
...s7 sat ((P10 &P6 )&(P8 &-P2 )): FALSE
..s7 sat -((P10 &P6 )&(P8 &-P2 )): TRUE
..s7 ==2=> s8
..s8 sat? -((P10 &P6 )&(P8 &-P2 ))
...s8 sat? ((P10 &P6 )&(P8 &-P2 ))
....s8 sat? (P10 &P6 )
.....s8 sat? P10
.....s8 sat P10 : FALSE
....s8 sat (P10 &P6 ): FALSE
...s8 sat ((P10 &P6 )&(P8 &-P2 )): FALSE
..s8 sat -((P10 &P6 )&(P8 &-P2 )): TRUE
..s8 ==4=> s9
..s9 sat? -((P10 &P6 )&(P8 &-P2 ))
...s9 sat? ((P10 &P6 )&(P8 &-P2 ))
....s9 sat? (P10 &P6 )
.....s9 sat? P10
.....s9 sat P10 : FALSE
....s9 sat (P10 &P6 ): FALSE
...s9 sat ((P10 &P6 )&(P8 &-P2 )): FALSE
..s9 sat -((P10 &P6 )&(P8 &-P2 )): TRUE
..s9 =13=> s10
..s10 sat? -((P10 &P6 )&(P8 &-P2 ))
...s10 sat? ((P10 &P6 )&(P8 &-P2 ))
....s10 sat? (P10 &P6 )
.....s10 sat? P10
.....s10 sat P10 : TRUE
..........s10 sat? P6
..........s10 sat P6 : FALSE
....s10 sat (P10 &P6 ): FALSE
```

```
...s10 sat ((P10 &P6 )&(P8 &-P2 )): FALSE
..s10 sat -((P10 &P6 )&(P8 &-P2 )): TRUE
..s10 ==5=> s11
..s11 sat? -((P10 &P6 )&(P8 &-P2 ))
...s11 sat? ((P10 &P6 )&(P8 &-P2 ))
....s11 sat? (P10 &P6 )
.....s11 sat? P10
.....s11 sat P10 : FALSE
....s11 sat (P10 &P6 ): FALSE
...s11 sat ((P10 &P6 )&(P8 &-P2 )): FALSE
..s11 sat -((P10 &P6 )&(P8 &-P2 )): TRUE
..s11 ==1=> s12
..s12 sat? -((P10 &P6 )&(P8 &-P2 ))
...s12 sat? ((P10 &P6 )&(P8 &-P2 ))
....s12 sat? (P10 &P6 )
.....s12 sat? P10
.....s12 sat P10 : FALSE
....s12 sat (P10 &P6 ): FALSE
...s12 sat ((P10 &P6 )&(P8 &-P2 )): FALSE
..s12 sat -((P10 &P6 )&(P8 &-P2 )): TRUE
..s12 ==3=> s13
..s13 sat? -((P10 &P6 )&(P8 &-P2 ))
...s13 sat? ((P10 &P6 )&(P8 &-P2 ))
....s13 sat? (P10 &P6 )
.....s13 sat? P10
.....s13 sat P10 : FALSE
....s13 sat (P10 &P6 ): FALSE
...s13 sat ((P10 &P6 )&(P8 &-P2 )): FALSE
..s13 sat -((P10 &P6 )&(P8 &-P2 )): TRUE
..s13 ==9=> s14
..s14 sat? -((P10 &P6 )&(P8 &-P2 ))
...s14 sat? ((P10 &P6 )&(P8 &-P2 ))
....s14 sat? (P10 &P6 )
.....s14 sat? P10
.....s14 sat P10 : FALSE
....s14 sat (P10 &P6 ): FALSE
...s14 sat ((P10 &P6 )&(P8 &-P2 )): FALSE
..s14 sat -((P10 &P6 )&(P8 &-P2 )): TRUE
..s14 no successor
s14 sat AG-((P10 &P6 )&(P8 &-P2 )): TRUE
..s3 =11=> s15
..s15 sat? -((P10 &P6 )&(P8 &-P2 ))
...s15 sat? ((P10 &P6 )&(P8 &-P2 ))
....s15 sat? (P10 &P6 )
.....s15 sat? P10
.....s15 sat P10 : FALSE
....s15 sat (P10 &P6 ): FALSE
...s15 sat ((P10 &P6 )&(P8 &-P2 )): FALSE
..s15 sat -((P10 &P6 )&(P8 &-P2 )): TRUE
..s15 ==7=> s5 visited
..s15 ==8=> s16
..s16 sat? -((P10 &P6 )&(P8 &-P2 ))
...s16 sat? ((P10 &P6 )&(P8 &-P2 ))
....s16 sat? (P10 &P6 )
.....s16 sat? P10
.....s16 sat P10 : FALSE
....s16 sat (P10 &P6 ): FALSE
...s16 sat ((P10 &P6 )&(P8 &-P2 )): FALSE
..s16 sat -((P10 &P6 )&(P8 &-P2 )): TRUE
..s16 ==6=> s17
..s17 sat? -((P10 &P6 )&(P8 &-P2 ))
...s17 sat? ((P10 &P6 )&(P8 &-P2 ))
....s17 sat? (P10 &P6 )
.....s17 sat? P10
.....s17 sat P10 : FALSE
....s17 sat (P10 &P6 ): FALSE
...s17 sat ((P10 &P6 )&(P8 &-P2 )): FALSE
..s17 sat -((P10 &P6 )&(P8 &-P2 )): TRUE
..s17 ==2=> s18
..s18 sat? -((P10 &P6 )&(P8 &-P2 ))
...s18 sat? ((P10 &P6 )&(P8 &-P2 ))
....s18 sat? (P10 &P6 )
.....s18 sat? P10
.....s18 sat P10 : FALSE
....s18 sat (P10 &P6 ): FALSE
...s18 sat ((P10 &P6 )&(P8 &-P2 )): FALSE
..s18 sat -((P10 &P6 )&(P8 &-P2 )): TRUE
..s18 ==4=> s19
..s19 sat? -((P10 &P6 )&(P8 &-P2 ))
...s19 sat? ((P10 &P6 )&(P8 &-P2 ))
....s19 sat? (P10 &P6 )
.....s19 sat? P10
.....s19 sat P10 : FALSE
....s19 sat (P10 &P6 ): FALSE
...s19 sat ((P10 &P6 )&(P8 &-P2 )): FALSE
..s19 sat -((P10 &P6 )&(P8 &-P2 )): TRUE
..s19 ==7=> s9 visited
..s18 ==7=> s8 visited
..s17 ==7=> s7 visited
..s16 ==7=> s6 visited
..s2 =11=> s20
..s20 sat? -((P10 &P6 )&(P8 &-P2 ))
...s20 sat? ((P10 &P6 )&(P8 &-P2 ))
....s20 sat? (P10 &P6 )
.....s20 sat? P10
.....s20 sat P10 : FALSE
....s20 sat (P10 &P6 ): FALSE
...s20 sat ((P10 &P6 )&(P8 &-P2 )): FALSE
..s20 sat -((P10 &P6 )&(P8 &-P2 )): TRUE
..s20 ==8=> s21
..s21 sat? -((P10 &P6 )&(P8 &-P2 ))
...s21 sat? ((P10 &P6 )&(P8 &-P2 ))
....s21 sat? (P10 &P6 )
.....s21 sat? P10
.....s21 sat P10 : FALSE
....s21 sat (P10 &P6 ): FALSE
...s21 sat ((P10 &P6 )&(P8 &-P2 )): FALSE
..s21 sat -((P10 &P6 )&(P8 &-P2 )): TRUE
..s21 ==6=> s22
..s22 sat? -((P10 &P6 )&(P8 &-P2 ))
...s22 sat? ((P10 &P6 )&(P8 &-P2 ))
....s22 sat? (P10 &P6 )
.....s22 sat? P10
.....s22 sat P10 : FALSE
....s22 sat (P10 &P6 ): FALSE
...s22 sat ((P10 &P6 )&(P8 &-P2 )): FALSE
..s22 sat -((P10 &P6 )&(P8 &-P2 )): TRUE
..s22 ==2=> s23
```

```
..s23 sat? -((P10 &P6 )&(P8 &-P2 ))
...s23 sat? ((P10 &P6 )&(P8 &-P2 ))
....s23 sat? (P10 &P6 )
.....s23 sat? P10
.....s23 sat P10 : FALSE
....s23 sat (P10 &P6 ): FALSE
...s23 sat ((P10 &P6 )&(P8 &-P2 )): FALSE
..s23 sat -((P10 &P6 )&(P8 &-P2 )): TRUE
..s23 ==4=> s24
..s24 sat? -((P10 &P6 )&(P8 &-P2 ))
...s24 sat? ((P10 &P6 )&(P8 &-P2 ))
....s24 sat? (P10 &P6 )
.....s24 sat? P10
.....s24 sat P10 : FALSE
....s24 sat (P10 &P6 ): FALSE
...s24 sat ((P10 &P6 )&(P8 &-P2 )): FALSE
..s24 sat -((P10 &P6 )&(P8 &-P2 )): TRUE
..s24 =10=> s19
s1 sat AG-((P10 &P6 )&(P8 &-P2 )): TRUE
```

B.4 Exclusion Relation

```
s1 sat? AG-(P6 &P7 )
..s1 sat? -(P6 &P7 )
...s1 sat? (P6 &P7 )
....s1 sat? P6
....s1 sat P6 : FALSE
...s1 sat (P6 &P7 ): FALSE
..s1 sat -(P6 &P7 ): TRUE
..s1 =12=> s2
..s2 sat? -(P6 &P7 )
...s2 sat? (P6 &P7 )
....s2 sat? P6
....s2 sat P6 : FALSE
...s2 sat (P6 &P7 ): FALSE
..s2 sat -(P6 &P7 ): TRUE
..s2 =10=> s3
..s3 sat? -(P6 &P7 )
...s3 sat? (P6 &P7 )
....s3 sat? P6
....s3 sat P6 : FALSE
...s3 sat (P6 &P7 ): FALSE
..s3 sat -(P6 &P7 ): TRUE
..s3 ==7=> s4
..s4 sat? -(P6 &P7 )
...s4 sat? (P6 &P7 )
....s4 sat? P6
....s4 sat P6 : FALSE
...s4 sat (P6 &P7 ): FALSE
..s4 sat -(P6 &P7 ): TRUE
..s4 =11=> s5
..s5 sat? -(P6 &P7 )
...s5 sat? (P6 &P7 )
....s5 sat? P6
....s5 sat P6 : FALSE
...s5 sat (P6 &P7 ): FALSE
..s5 sat -(P6 &P7 ): TRUE
..s5 ==8=> s6
..s6 sat? -(P6 &P7 )
...s6 sat? (P6 &P7 )
....s6 sat? P6
....s6 sat P6 : FALSE
...s6 sat (P6 &P7 ): FALSE
..s6 sat -(P6 &P7 ): TRUE
..s6 =13=> s7
..s7 sat? -(P6 &P7 )
...s7 sat? (P6 &P7 )
....s7 sat? P6
....s7 sat P6 : FALSE
...s7 sat (P6 &P7 ): FALSE
..s7 sat -(P6 &P7 ): TRUE
..s7 ==5=> s8
..s8 sat? -(P6 &P7 )
...s8 sat? (P6 &P7 )
....s8 sat? P6
....s8 sat P6 : TRUE
........s8 sat? P7
........s8 sat P7 : FALSE
...s8 sat (P6 &P7 ): FALSE
..s8 sat -(P6 &P7 ): TRUE
..s8 ==1=> s9
..s9 sat? -(P6 &P7 )
...s9 sat? (P6 &P7 )
....s9 sat? P6
....s9 sat P6 : FALSE
...s9 sat (P6 &P7 ): FALSE
..s9 sat -(P6 &P7 ): TRUE
..s9 ==3=> s10
..s10 sat? -(P6 &P7 )
...s10 sat? (P6 &P7 )
....s10 sat? P6
....s10 sat P6 : FALSE
...s10 sat (P6 &P7 ): FALSE
..s10 sat -(P6 &P7 ): TRUE
..s10 =14=> s11
..s11 sat? -(P6 &P7 )
...s11 sat? (P6 &P7 )
....s11 sat? P6
....s11 sat P6 : FALSE
...s11 sat (P6 &P7 ): FALSE
..s11 sat -(P6 &P7 ): TRUE
..s11 ==6=> s12
..s12 sat? -(P6 &P7 )
...s12 sat? (P6 &P7 )
....s12 sat? P6
....s12 sat P6 : FALSE
...s12 sat (P6 &P7 ): FALSE
..s12 sat -(P6 &P7 ): TRUE
..s12 ==2=> s13
..s13 sat? -(P6 &P7 )
...s13 sat? (P6 &P7 )
....s13 sat? P6
....s13 sat P6 : FALSE
...s13 sat (P6 &P7 ): FALSE
..s13 sat -(P6 &P7 ): TRUE
..s13 ==4=> s14
..s14 sat? -(P6 &P7 )
...s14 sat? (P6 &P7 )
....s14 sat? P6
....s14 sat P6 : FALSE
...s14 sat (P6 &P7 ): FALSE
..s14 sat -(P6 &P7 ): TRUE
..s14 ==9=> s15
..s15 sat? -(P6 &P7 )
...s15 sat? (P6 &P7 )
....s15 sat? P6
....s15 sat P6 : FALSE
...s15 sat (P6 &P7 ): FALSE
..s15 sat -(P6 &P7 ): TRUE
..s15 no successor
s15 sat AG-(P6 &P7 ): TRUE
..s6 =14=> s16
..s16 sat? -(P6 &P7 )
...s16 sat? (P6 &P7 )
....s16 sat? P6
....s16 sat P6 : FALSE
...s16 sat (P6 &P7 ): FALSE
..s16 sat -(P6 &P7 ): TRUE
..s16 ==6=> s17
..s17 sat? -(P6 &P7 )
...s17 sat? (P6 &P7 )
....s17 sat? P6
....s17 sat P6 : FALSE
...s17 sat (P6 &P7 ): FALSE
..s17 sat -(P6 &P7 ): TRUE
..s17 ==2=> s18
..s18 sat? -(P6 &P7 )
...s18 sat? (P6 &P7 )
....s18 sat? P6
....s18 sat P6 : FALSE
...s18 sat (P6 &P7 ): FALSE
```

```
..s18 sat -(P6 &P7 ): TRUE
..s18 ==4=> s19
..s19 sat? -(P6 &P7 )
...s19 sat? (P6 &P7 )
....s19 sat? P6
....s19 sat P6 : FALSE
...s19 sat (P6 &P7 ): FALSE
..s19 sat -(P6 &P7 ): TRUE
..s19 =13=> s20
..s20 sat? -(P6 &P7 )
...s20 sat? (P6 &P7 )
....s20 sat? P6
....s20 sat P6 : FALSE
...s20 sat (P6 &P7 ): FALSE
..s20 sat -(P6 &P7 ): TRUE
..s20 ==5=> s21
..s21 sat? -(P6 &P7 )
...s21 sat? (P6 &P7 )
....s21 sat? P6
....s21 sat P6 : TRUE
........s21 sat? P7
........s21 sat P7 : FALSE
...s21 sat (P6 &P7 ): FALSE
..s21 sat -(P6 &P7 ): TRUE
..s21 ==1=> s22
..s22 sat? -(P6 &P7 )
...s22 sat? (P6 &P7 )
....s22 sat? P6
....s22 sat P6 : FALSE
...s22 sat (P6 &P7 ): FALSE
..s22 sat -(P6 &P7 ): TRUE
..s22 ==3=> s14 visited
..s5 =13=> s23
..s23 sat? -(P6 &P7 )
...s23 sat? (P6 &P7 )
....s23 sat? P6
....s23 sat P6 : FALSE
...s23 sat (P6 &P7 ): FALSE
..s23 sat -(P6 &P7 ): TRUE
..s23 ==5=> s24
..s24 sat? -(P6 &P7 )
...s24 sat? (P6 &P7 )
....s24 sat? P6
....s24 sat P6 : TRUE
........s24 sat? P7
........s24 sat P7 : FALSE
...s24 sat (P6 &P7 ): FALSE
..s24 sat -(P6 &P7 ): TRUE
..s24 ==1=> s25
..s25 sat? -(P6 &P7 )
...s25 sat? (P6 &P7 )
....s25 sat? P6
....s25 sat P6 : FALSE
...s25 sat (P6 &P7 ): FALSE
..s25 sat -(P6 &P7 ): TRUE
..s25 ==3=> s26
..s26 sat? -(P6 &P7 )
...s26 sat? (P6 &P7 )
....s26 sat? P6
....s26 sat P6 : FALSE
...s26 sat (P6 &P7 ): FALSE
..s26 sat -(P6 &P7 ): TRUE
..s26 ==8=> s10 visited
..s25 ==8=> s9 visited
..s24 ==8=> s8 visited
..s23 ==8=> s7 visited
..s4 =13=> s27
..s27 sat? -(P6 &P7 )
...s27 sat? (P6 &P7 )
....s27 sat? P6
....s27 sat P6 : FALSE
...s27 sat (P6 &P7 ): FALSE
..s27 sat -(P6 &P7 ): TRUE
..s27 ==5=> s28
..s28 sat? -(P6 &P7 )
...s28 sat? (P6 &P7 )
....s28 sat? P6
....s28 sat P6 : TRUE
........s28 sat? P7
........s28 sat P7 : FALSE
...s28 sat (P6 &P7 ): FALSE
..s28 sat -(P6 &P7 ): TRUE
..s28 ==1=> s29
..s29 sat? -(P6 &P7 )
...s29 sat? (P6 &P7 )
....s29 sat? P6
....s29 sat P6 : FALSE
...s29 sat (P6 &P7 ): FALSE
..s29 sat -(P6 &P7 ): TRUE
..s29 ==3=> s30
..s30 sat? -(P6 &P7 )
...s30 sat? (P6 &P7 )
....s30 sat? P6
....s30 sat P6 : FALSE
...s30 sat (P6 &P7 ): FALSE
..s30 sat -(P6 &P7 ): TRUE
..s30 =11=> s26 visited
..s29 =11=> s25 visited
..s28 =11=> s24 visited
..s27 =11=> s23 visited
..s3 =11=> s31
..s31 sat? -(P6 &P7 )
...s31 sat? (P6 &P7 )
....s31 sat? P6
....s31 sat P6 : FALSE
...s31 sat (P6 &P7 ): FALSE
..s31 sat -(P6 &P7 ): TRUE
..s31 ==7=> s5 visited
..s31 ==8=> s32
..s32 sat? -(P6 &P7 )
...s32 sat? (P6 &P7 )
....s32 sat? P6
....s32 sat P6 : FALSE
...s32 sat (P6 &P7 ): FALSE
..s32 sat -(P6 &P7 ): TRUE
..s32 ==7=> s6 visited
..s32 =14=> s33
..s33 sat? -(P6 &P7 )
...s33 sat? (P6 &P7 )
....s33 sat? P6
....s33 sat P6 : FALSE
...s33 sat (P6 &P7 ): FALSE
..s33 sat -(P6 &P7 ): TRUE
..s33 ==6=> s34
..s34 sat? -(P6 &P7 )
...s34 sat? (P6 &P7 )
....s34 sat? P6
....s34 sat P6 : FALSE
...s34 sat (P6 &P7 ): FALSE
..s34 sat -(P6 &P7 ): TRUE
..s34 ==2=> s35
..s35 sat? -(P6 &P7 )
...s35 sat? (P6 &P7 )
....s35 sat? P6
....s35 sat P6 : FALSE
...s35 sat (P6 &P7 ): FALSE
..s35 sat -(P6 &P7 ): TRUE
..s35 ==4=> s36
..s36 sat? -(P6 &P7 )
...s36 sat? (P6 &P7 )
....s36 sat? P6
....s36 sat P6 : FALSE
...s36 sat (P6 &P7 ): FALSE
..s36 sat -(P6 &P7 ): TRUE
..s36 ==7=> s19 visited
..s35 ==7=> s18 visited
..s34 ==7=> s17 visited
..s33 ==7=> s16 visited
..s2 =11=> s37
..s37 sat? -(P6 &P7 )
...s37 sat? (P6 &P7 )
....s37 sat? P6
....s37 sat P6 : FALSE
...s37 sat (P6 &P7 ): FALSE
..s37 sat -(P6 &P7 ): TRUE
..s37 ==8=> s38
..s38 sat? -(P6 &P7 )
...s38 sat? (P6 &P7 )
....s38 sat? P6
....s38 sat P6 : FALSE
...s38 sat (P6 &P7 ): FALSE
..s38 sat -(P6 &P7 ): TRUE
..s38 =10=> s32 visited
..s38 =14=> s39
..s39 sat? -(P6 &P7 )
...s39 sat? (P6 &P7 )
....s39 sat? P6
....s39 sat P6 : FALSE
...s39 sat (P6 &P7 ): FALSE
..s39 sat -(P6 &P7 ): TRUE
..s39 ==6=> s40
..s40 sat? -(P6 &P7 )
...s40 sat? (P6 &P7 )
....s40 sat? P6
....s40 sat P6 : FALSE
...s40 sat (P6 &P7 ): FALSE
..s40 sat -(P6 &P7 ): TRUE
```

```
..s40 ==2=> s41
..s41 sat? -(P6 &P7 )
...s41 sat? (P6 &P7 )
....s41 sat? P6
....s41 sat P6 : FALSE
...s41 sat (P6 &P7 ): FALSE
..s41 sat -(P6 &P7 ): TRUE
..s41 ==4=> s42
..s42 sat? -(P6 &P7 )
...s42 sat? (P6 &P7 )
....s42 sat? P6
....s42 sat P6 : FALSE
...s42 sat (P6 &P7 ): FALSE
..s42 sat -(P6 &P7 ): TRUE
..s42 =10=> s36
s1 sat AG-(P6 &P7 ): TRUE
```

Tese de Doutorado apresentada por **Raimundo da Silva Barreto** a Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título **"Uma Metodologia Baseada em Rede de Petri Temporizada para a Síntese de Software em Sistemas Embarcados e de Tempo-Real Críticos"**, elaborada sob a orientação do **Prof. Paulo Romero Martins Maciel** e aprovada pela Banca Examinadora formada pelos professores:

Prof. Paulo Roberto Freire Cunha
Departamento de Sistemas de Computação - CIn / UFPE

Prof. Nelson Souto Rosa
Departamento de Sistemas de Computação - CIn / UFPE

Prof. Ricardo Massa Ferreira Lima
Escola Politécnica de Pernambuco / UPE

Prof. Siang Wun Song
Instituto de Matemática e Estatística / USP

Prof. Antonio Otavio Fernandes
Departamento de Ciência da Computação / UFMG

Visto e permitida a impressão.
Recife, 29 de abril de 2005.

Prof. JAELSON FREIRE BRELAZ DE CASTRO
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.