



Post-Graduation in Computer Science

**“Energy Consumption and Execution Time
Estimation of Embedded System Applications”**

by

Gustavo Rau de Almeida Callou

MSc Dissertation



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

Recife, March/2009



FEDERAL UNIVERSITY OF PERNAMBUCO
CENTER FOR INFORMATICS
POST-GRADUATION IN COMPUTER SCIENCE

Gustavo Rau de Almeida Callou

“Energy Consumption and Execution Time Estimation of
Embedded System Applications”

*A dissertation submitted to the Post-Graduation
in Computer Science of the Center for
Informatics of Federal University of Pernambuco
in partial fulfillment of requirements for the
degree in Master of Computer Science.*

Adviser: Prof. Dr. Paulo Romero Martins Maciel

Recife, March/2009

Callou, Gustavo Rau de Almeida

Energy consumption and execution time estimation of embedded system applications / Gustavo Rau de Almeida Callou - Recife : O Autor, 2009.

xv, 111 folhas : il., fig., tab.

Dissertação (mestrado) – Universidade Federal de Pernambuco. CIn. Ciência da Computação, 2009.

Inclui bibliografia, anexo e apêndice.

1. Avaliação de desempenho. 2. Redes de Petri – Modelagem de sistemas. 3. Consumo de energia. 4. Simulação. I. Título.

004.029

CDD (22.ed.)

MEI2009-031

This dissertation is dedicated to my parents.

ACKNOWLEDGEMENTS

First of all, I would like to thank God for all conquests that I have been achieving. Also, I am very grateful to my parents, people that have always been supporting and encouraging me to do what I dreamed. Many thanks to my sisters that always contributed me to keep going at all difficult moments. A special thanks for Karina, who has never stopped to give her support and encouragement whether in good or difficult moments.

Thanks to my friend and professor Sérgio Galdino who gave me the opportunity to start my academic life as a researcher and who introduce me to professor Paulo Maciel. Many thanks to my advisor and friend professor Paulo Maciel. He gave me the chance to be part of his research group, and has helped me at all moments. A thanks to professor Meuse Oliveira Jr. who helped me at the initial stages of this research and who provided the hardware implementation for validating the experiments; to Eduardo Tavares, who helped with valuable article reviews; to Bruno Nogueira for implementing the CPN Simulator and for all supporting and help; and to all members of our research group (MODCS - Modeling of Distributed and Concurrent Systems); and to professors Nelson Rosa and Ricardo Salgueiro for their valuable contributions as committee members.

Thanks to several colleagues I made at CIn/UFPE. Among them: Adriana Ribeiro, Antônio Ricardo, Danuza Neiva, Erica Sousa, Ermeson Andrade, Giselia Magalhães, Julian Menezes, Kalil Bispo, Rafael Antonello and Patricia Takako. My special thanks to Ermeson, Erica and Julian for the motivating moments and for all support and help.

Thanks to all professors and staff in the center for informatics of UFPE.

Thanks to Clinic Hospital of Pernambuco (HCPE), specially to all the informatics team members.

Thanks to my English teacher, Cremilda Matos, for all my English improvements.

All models are wrong, but some are useful.

—GEORGE BOX

RESUMO

Nos últimos anos, a redução do consumo de energia das aplicações dos sistemas embarcados tem recebido uma grande atenção da comunidade científica, visto que, como o tempo de resposta e o baixo consumo de energia são requisitos conflitantes, esses estudos tornam-se altamente necessários. Nesse contexto, é proposta uma metodologia aplicada nas fases iniciais de projeto para dar suporte às decisões relativas ao consumo de energia e ao desempenho das aplicações desses dispositivos embarcados.

Além disso, esse trabalho propõe modelos temporizados de eventos discretos que são avaliados através de uma metodologia de simulação estocástica com o objetivo de representar diferentes cenários dos sistemas com facilidade. Dessa forma, para cada cenário é preciso decidir o número máximo de simulações e o tamanho de cada rodada da simulação, onde ambos os fatores podem impactar no desempenho para se obter tais estimativas. Essa metodologia considera também, um modelo intermediário que representa a descrição do comportamento do sistema e, é através desse modelo que cenários são analisados. Esse modelo intermediário é baseado em redes de Petri coloridas temporizadas que permitem não somente a análise do software, mas também fornece suporte a um conjunto de métodos bem estabelecidos para verificações de propriedades.

É nesse contexto que o software, ALUPAS, responsável por estimar o consumo de energia e o tempo de execução dos sistemas embarcados é apresentado. Por fim, um caso de estudo real, assim como também, exemplos customizados são apresentados com a finalidade de mostrar a aplicabilidade desse trabalho, onde usuários não especializados não precisam interagir diretamente com o formalismo de redes de Petri.

Palavras-chave: Redes de Petri Coloridas, Simulação Estocástica, Sistemas Embarcados, Consumo de Energia e Desempenho.

ABSTRACT

Over the last years, the issue of reducing energy consumption in embedded system applications has received considerable attention from the scientific community, since responsiveness and low energy consumption are often conflicting requirements. In this context, this dissertation proposes a methodology applied in early design phases for supporting design decisions on energy consumption and performance of embedded applications.

In addition, this work proposes temporized discrete event models that have been evaluated through a stochastic simulation approach to represent different system scenarios in an easier way. For each scenario, it is important to decide the maximum number of simulations and the duration of each simulation, where both may impact the performance estimates. Such approach also considers an intermediate model which represents the system behavioral description and, through these models, the scenarios are analyzed. The intermediate model is based on timed Colored Petri Net, a formal behavioral model that not only allows the software execution analysis, but it is also supported by a set of well established methods for property verifications.

In this context, a software, named ALUPAS, for estimating energy consumption and execution time of embedded systems is presented. Lastly, a real-world case study as well as customized examples are presented, showing the applicability of this work in which non-specialized users do not need to interact directly with the Petri net formalism.

Keywords: Coloured Petri Net, Stochastic Simulation, Embedded System, Energy Consumption and Performance.

CONTENTS

List of Figures	xii
List of Tables	xv
Chapter 1—Introduction	1
1.1 Objectives and Contributions	4
1.2 Outline	5
Chapter 2—Background	6
2.1 Energy Consumption and Performance Evaluation	6
2.1.1 Evaluation models	7
2.1.2 Measurement Strategies	9
2.2 Simulation Process	11
2.3 System Classifications	12
2.4 Petri Nets	14
2.4.1 Place-Transition Nets	15
2.4.2 Marked Petri Nets	17
2.4.3 Transition Enabling and Firing	17
2.4.4 Petri Net Analysis Methods	18
2.5 Time Extensions	22
2.6 Coloured Petri Nets	23
2.6.1 CPN ML Language	29
2.6.2 Timed Coloured Petri net	33
2.6.3 Hierarchical CPN	35
2.7 Embedded Systems	38

2.8	Summary	40
Chapter 3—Related Works		41
3.1	General Overview of Energy Consumption and Performance Evaluation .	41
3.2	Hardware simulation-related-works	42
3.3	Software simulation-related-works	44
3.4	Hybrid Approach	45
3.5	Summary	46
Chapter 4—Methodology		47
4.1	Methodology	47
4.2	Energy Consumption and Performance Evaluation Framework	49
4.2.1	Characterization Process	50
4.3	Summary	53
Chapter 5—Formal Model		54
5.1	Embedded Software Modeling	54
5.1.1	CPN model for ordinary instructions	55
5.1.2	CPN model for conditional instructions	57
5.1.3	Procedure calls model	57
5.1.4	CPN model for conditional branch instructions	58
5.1.5	CPN model for branching exchange instructions	60
5.1.6	CPN model for Store multiple instruction	62
5.1.7	CPN model for Load multiple instruction	62
5.1.8	CPN model for the stop criteria process	62
5.2	Reduction Rules	64
5.3	Summary	65
Chapter 6—The Simulation Environment		66
6.1	ALUPAS	66
6.2	Assembler and C Compiler	66

6.3	Binary CPN Compiler	67
6.4	CPN Simulator	68
6.4.1	Sim file	69
6.4.2	Simulation Process	71
6.4.3	Simulation Algorithm Variables	72
6.4.4	Stop Criteria Evaluation	74
6.4.5	Enabling and Firing rules	75
6.5	Graphical User Interface	78
6.5.1	Component Integration	79
6.6	Summary	80
Chapter 7—Case Studies		81
7.1	Example One	81
7.1.1	Simulation Results.	82
7.2	Example Two	83
7.3	Binary Search Algorithm	84
7.3.1	Results	87
7.4	BCNT Algorithm	88
7.5	Pulse-oximeter	89
7.6	Summary	94
Chapter 8—Conclusions		96
8.1	Contributions	97
8.2	Future Works	98
Bibliography		99
Appendix A—A Validation Process		106
A.1	Noncorresponding measurements	106
Appendix B—The Simulation Algorithm		108

Appendix C—Binary-CPN Compiler Class Diagram

Appendix D—CPN Simulator Class Diagram

LIST OF FIGURES

1.1	Levels of Abstraction	3
1.2	Cost of correcting a requirements defect according to the stage at which it is discovered.	4
2.1	Performance Evaluation	8
2.2	Histogram showing accuracy and precision.	11
2.3	Simulation process diagram.	12
2.4	System classifications.	14
2.5	Petri net basic elements.	15
2.6	Compact representation of a PN	16
2.7	(a) Mathematical formalism; (b) Graphical representation before firing of t_0 ; (c) Graphical representation after firing of t_0	18
2.8	Simple reduction rule.	20
2.9	Reduction of a PN with a redundant place	20
2.10	Six transformations preserving properties.	21
2.11	Reduction Rule.	29
2.12	Place inscriptions.	32
2.13	Arc inscription.	32
2.14	Transition inscriptions.	34
2.15	Moore's law.	39
3.1	The System with its Environment, Requirements and Constraints.	41
3.2	Performance Evaluation Methodology.	42
4.1	Methodology activity diagram.	48
4.2	The proposed Framework.	50
4.3	A code example for measuring.	51

4.4	The measurement performed on oscilloscope.	52
4.5	Hardware Platform with LPC2106 microprocessor.	52
4.6	The measurement process.	53
4.7	The AMALGHMA engine.	53
5.1	Methodology task-flow.	55
5.2	Ordinary model.	56
5.3	An example of ordinary model.	56
5.4	Conditional model.	57
5.5	Branch and link model	58
5.6	Conditional branch model.	58
5.7	Conditional branch model example	59
5.8	Branching exchange instruction model.	61
5.9	Bx instruction example	61
5.10	Store multiple model.	62
5.11	Load multiple model.	63
5.12	CPN model for the stop criteria process	63
5.13	(a)CPN model, (b)CPN model after reduction process.	64
6.1	Assembler structure.	67
6.2	Basic Binary CPN Compiler structure.	68
6.3	Detailed Binary-CPN Compiler structure.	68
6.4	Basic CPN Simulator structure.	69
6.5	CPN Simulator structure.	69
6.6	Simulation diagram.	71
6.7	Stop criteria diagram.	75
6.8	Concurrent example.	77
6.9	ALUPAS' input interface.	78
6.10	ALUPAS' output interface.	79
6.11	Component Integration.	80
7.1	Annotated Assembly Code	82
7.2	CPN Tools versus CPN Simulator runtime.	84

7.3	Binary Search Code in Assembly.	85
7.4	Binary Search Code in C.	86
7.5	Binary search results of execution time.	87
7.6	Binary search results of energy consumption.	87
7.7	Binary search results of execution time.	88
7.8	Binary search results of energy consumption.	88
7.9	The CPN model for BCNT Algorithm.	89
7.10	Pulse-Oximeter Architecture.	90
7.11	Pulse-oximeter.	91
7.12	Runtime Comparison.	92
B.1	Simulation Algorithm	108
C.1	Binary-CPN Compiler class diagram.	110
D.1	CPN Simulator class diagram.	111

LIST OF TABLES

2.1	Some Pioneers in Performance Evaluation.	7
2.2	Interpretation for places and transitions.	16
6.1	The simulation results.	76
6.2	Comparison: 10 replications versus 653 replications	76
6.3	Firing rule results of Figure 6.8.	78
7.1	Simulation Results of the code on Figure 7.1.	82
7.2	Comparison between simulation results.	83
7.3	Comparison of the runtime simulation.	83
7.4	Binary Search results summary	88
7.5	BCNT results summary.	90
7.6	Pulse-oximeter result summary	91
7.7	CPN Tools runtime x CPN Simulator runtime.	92
7.8	Excitation - comparison of execution time.	93
7.9	Excitation - comparison of energy consumption.	93
7.10	Acquisition - comparison of execution time.	94
7.11	Acquisition - comparison of energy consumption.	94
7.12	Control - comparison of execution time.	95
7.13	Control - comparison of energy consumption.	95

CHAPTER 1

INTRODUCTION

Embedded system is the one whose principal function is not computational, but it is controlled by a computer embedded (e.g.: microprocessor or microcontroller) within it [Wil01]. The word embedded means that the computer lies inside the overall system, hidden from view, forming an integral part of a greater whole and, as a result, the user may be unaware of the computers existence [Tav06].

Nowadays, embedded systems are present in practically all areas of human lives. Mobile phones, clocks, refrigerators, microwaves, oscilloscopes and routers are a few examples of those devices that have a digital processor responsible for performing specific tasks. Within such devices embedded applications that have always been running the same tasks are present and, thus, the software updates after being in production are unusual. Besides, embedded systems do not terminate, unless it fails [Lee02].

Depending on the purpose of the application, the design of embedded systems may have to take into account several constraints, for instance, time, size, weight, cost, reliability and energy consumption. Furthermore, advances in microelectronics have allowed for the development of embedded systems with several complex features, thereby upholding the development of powerful mobile mechanism such as military gadgets (e.g.: spy satellites and guide missiles) and medical devices (e.g.: thermometers and pulse-oximeters). These devices generally rely on constrained energy sources (e.g.: battery), in such a way that if the energy source is depleted, the system stops functioning. The power consumption control is also becoming an important design goal in designs that are not battery-operated, because the excessive heat generated from high power consumption can seriously degrade chip performance and cause physical damage to the chip [HZDS95]. Hence estimating energy consumption in early design phases can provide important insights to the designer about the battery lifetime as well as parts of the application that need optimization.

Embedded applications that deal with time constraints are classified as Embedded Real-Time Systems (ERTS). In these systems, not only the logical results of computations are important, but also the time instant in which they are obtained. Some constraints are considered “hard”, while others are “soft”, meaning the timing deadlines may or may not be violated. In other words, soft ERTS accepts a soft delay to obtain the results (e.g.: web servers, mobile phones, Voice over Internet Protocol (VoIP) Calls, digital TV, web video conferences, and others). On the other hand, in the hard ERTS if the time constraints are not satisfied, a catastrophe may occur (e.g.: car races, health care devices, military

applications, aircraft and nuclear control centers)[TMSO08]. Hence, time predictability is an essential issue on the development life cycle of those systems [BL04, TMS⁺07].

The context of this work is related to embedded systems with timing and energy constraints. More specifically, this work is concerned about the adoption of formal models for modeling hard real-time systems with energy constraints as well as the utilization of techniques for estimating their energy consumption and execution time. A formal approach, based on Coloured Petri Nets (CPN), for estimating execution time as well as energy consumption of embedded system applications through a stochastic simulation method is presented. The formal mechanism, such as CPN, has been adopted in order to trade off and system's representation based on abstraction levels that might focus on processor instructions or high-level programming languages, in which applications may be modeled instruction-by-instruction or by blocks of instructions.

Originally, most part of embedded systems was hardware-based, using for instance ASICs (Application Specific Integrated Circuit). However, with the constant micro-electronic advances, the technology evolved in a such way that the computational capability of processors has been increased and, correspondingly, their cost and size have been decreased. Consequently, the software has been responsible for 80% of an embedded system development so far [SVM01]. This process has been moving functionalities from hardware to software, and some advantages such as flexibility, lower cost, and accessibility are improved. On the other hand, functionalities implemented in hardware still have better performance and consume less energy.

It is important to state that many works deal with energy consumption of embedded systems at different abstraction levels. The differences of each abstraction level [Oel00] [Bea01] resulted in the following classification:

- Application/System level: the energy consumption related to the execution of a particular program can be considered in such level.
- Behavioral/Algorithm level: different algorithms for the same purpose gives different amount of power consumption, and at this level such behavior is analyzed.
- Architectural level: in this level, the power consumption analysis of caches, core and processor buses are performed.
- Logic (gate) level: at this level both the function and the style of any circuit are decided. There are various design styles and each one has its power-performance trade-offs.
- Transistor (circuit) level: this is the lowest level of abstraction; studies of changing the input voltage and reordering of transistors, for example, are analyzed in such level.

Figure 1.1 summarizes the relation of those levels considering capacity, accuracy, speed, resources and energy saving analysis on each level. The reader should have in

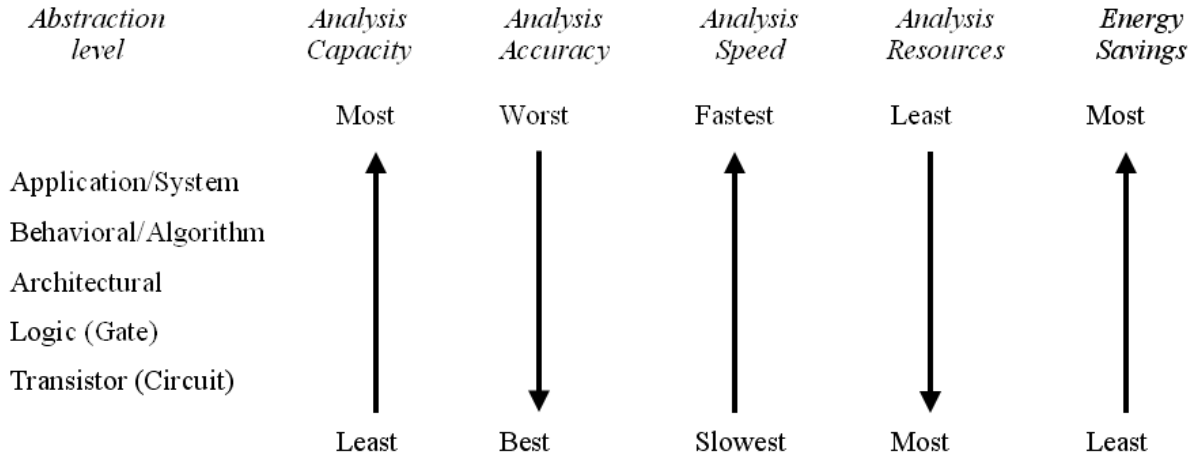


Figure 1.1: Levels of Abstraction

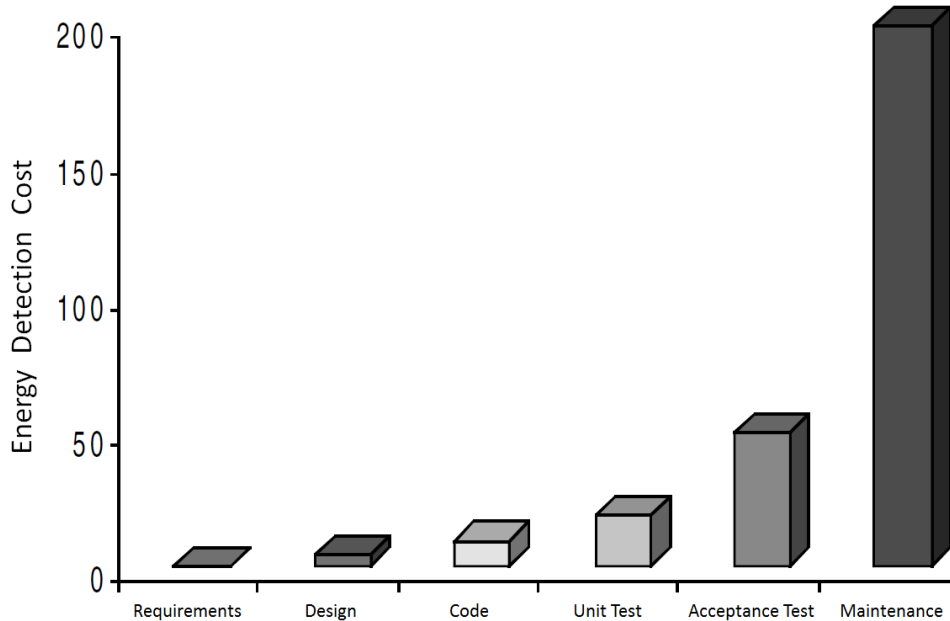
mind that it does not have a level better than the others. On the other hand, a level may be more applicable in a specific case, but all of them are important in order to have a low power energy consumption in embedded systems. Furthermore, this work focuses the application and behavior levels.

Without loss of generality, there are two basic approaches based on simulation for estimating embedded software energy consumption: (i) instruction based simulation and (ii) hardware based simulation [NN02]. In hardware simulation, despite the very high computation effort, more accurate results might be obtained in comparison with instruction simulation due to the laborious system specification. However, instruction simulation has been adopted by many works in order to provide energy consumption estimation in a satisfactory period of time. Although there are some works about these methods, to the best of our knowledge, only a small number may represent the embedded applications at a different abstraction level with good accuracy for estimating energy consumption and execution time in a short period of runtime.

In addition, performance has been a central issue in the design, development and configuration of systems [Wel02]. The performance as well as power will get more importance if we consider embedded systems with energy and time constraints. In this context, it is not enough to know that systems work properly, they must also work effectively in order to respect their constraints. Studies about performance analysis of systems have been conducted to evaluate existing and/or planned softwares, to compare alternative configurations and to find an optimal system configuration. Thus, being able to estimate the performance and power consumption of a system is important because if such requirements are not satisfied, the system designers can make changes in a very early stage of the design, thereby saving both time and money.

The redesign of both software and hardware is costly and may cause late system delivery. Figure 1.2 shows the error detection costs in a different stages of the development life cycle. Thus, in such illustration is demonstrated that earlier detected errors cost less

money for the companies. Moreover, as the system are getting more and more complex, the adoption of formal evaluation models can provide a significant help in order to reduce the global development cost of embedded systems.



Font: [Hal07, p. 2]

Figure 1.2: Cost of correcting a requirements defect according to the stage at which it is discovered.

1.1 OBJECTIVES AND CONTRIBUTIONS

In order to assure that the embedded system constraints (e.g.: energy consumption and execution time) are preserved, this dissertation focuses on providing a methodology that aims at evaluating energy consumption as well as execution time of embedded real-time systems in early design phases. From an Assembly code or C program, models have been built in order to represent the system behavior and compute both energy consumption and execution time of each code instruction.

More specifically, the objectives are:

- to propose a temporized discrete event model supported by a precise semantic in order to be able to represent Assembly language and a representative subset of the C ANSI language;
- to propose a simulator for evaluating the proposed model in order to estimate the energy consumption and execution time of embedded systems applications;
- to propose a characterization mechanism of the microcontroller instruction set considering its energy consumption and performance.

This dissertation presents a methodology for estimating energy consumption and execution time of embedded systems. This work extends the approach proposed by Oliveira [OJ06] by simplifying such methodology considering other microcontroller (ARM7-based instead of 8051) and dealing with C programs. Furthermore, a simulation tool is proposed in order to improve the runtime evaluation. Specific contributions are depicted as follows:

- **Framework.** A mechanism for supporting design decisions on energy consumption and performance of embedded applications in early design phases is proposed;
- **Modeling.** The proposed methodology automatically translates the embedded code into a Coloured Petri net, a formal behavioral model that allows the software execution analysis. The modeling phase is based on composition of basic blocks that represents each relevant behavior of the ARM7-based instruction set microcontroller.
- **Simulating.** A stochastic evaluation approach through discrete event simulation is proposed for output data analysis. A new simulating tool is proposed to simulate the specific CPN models in a much faster simulation runtime than the other generic engines available for CPN simulation.

1.2 OUTLINE

This work is organized as follows:

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

CHAPTER 2

BACKGROUND

This chapter shows a summary of the background information needed for a better understanding about this work. First of all, it is performed an overview of energy consumption and performance evaluation, including measurement techniques and evaluation models. After that, it is presented the system classification. Next, it is shown an overview about Petri nets and Coloured Petri nets (CPN). Afterwards, some definitions such as binding, marking, enable transitions, fire rules and reduction process are introduced.

2.1 ENERGY CONSUMPTION AND PERFORMANCE EVALUATION

Energy is one of the most important non-functional requirements for embedded system design. It is important to stress that the energy consumption of embedded system depends on the hardware platform and software. The energy design problems can be classified into two groups: (i) analysis and (ii) optimization [Yea98]. Analysis problems are concerned with the accurate estimation of the energy consumption in order to assure that the energy consumption constraints are not violated. The analysis techniques differ in their accuracy and efficiency, in which the accuracy depends on the available design information. In early design phases, the focus should be to obtain energy consumption estimates quickly through little design information. Thus, in such phases, less accuracy results are expected. As the design proceeds, more details are available and more accurate results can be obtained through longer analysis time.

Optimization has been considered as the process that improves the design without violating any design specification. An automatic design optimization requires a fast analysis engine to evaluate different design scenarios. On the other hand, manual optimization demands a tool in order to provide energy consumption estimation of different design choices. It is important to highlight that a design decision involves trade-offs from different sources such as the impact to the circuit delay, which affects the performance and throughput of the chip, and the chip area, which may increase the manufacturing costs. Furthermore, the design decisions to achieve a low energy consumption may affect other factors such as cycle time, quality and reliability.

Nowadays, it is not always enough to know that systems work properly, they must also work effectively. Thus, Performance Evaluation (PE) is often a central issue in the design, development, and configuration of systems. The goals of the PE may be to maximize the throughput of the system, process a given workload for a minimum cost (e.g.: to reduce the energy consumption), or any number of other objective functions [Luc71]. These goals provide the overall environment for evaluation and determine what level of effort

can be devoted to the models or measurement techniques that should be applied in order to obtain the performance metric of a system.

In order to have a history overview of the firsts approaches that deal with PE, Table 2.1 shows the pioneers in PE. However, such list showed by [Her02] is incomplete and should contain a hundred or more names. In such table, it is possible to observe that in the sixties, researchers have been adopting PE techniques in the world of computers and computer communication systems. Furthermore, since 80s performance simulation approaches have been taken over in the Internet.

Table 2.1: Some Pioneers in Performance Evaluation.

Pioneers	Year	in	approaches
Erlang	1908-18	Telephone traffic	fundamental delay and loss formulas
Palm	1943	Telephone traffic	long-term variations
Jacobaeus	1950	switching networks	congestion in link systems
Clos	1953	switching networks	nonblocking systems
Wilkinson	1955	toll traffic engineering	alternate routing systems
Cobham	1954	operation research	priority assignment
Jackson	1957	operation research	queuing in networks
Conway	1958-67	operation research	scheduling
Scherr	1965	time-sharing systems	measurement and modeling
Kleinrock	1964-74	ARPA	performance and reliability
Buzen	1971	computers	central server model
Bux	1981	token ring network	performance simulation
Bellcore	80th	Internet traffic	long-range dependency

In addition, performance analysis studies are conducted to evaluate existing or planned systems, to compare alternative configurations, or to find an optimal configuration of a system [Wel02]. The following sections presents an overview of the evaluation models and measurements techniques that have been conducted in order to measure and estimate the energy consumption and execution time of embedded system applications.

2.1.1 Evaluation models

The performance evaluation can be classified into performance modeling and performance measurement [Joh06]. There are advantages and drawbacks to each of these techniques.

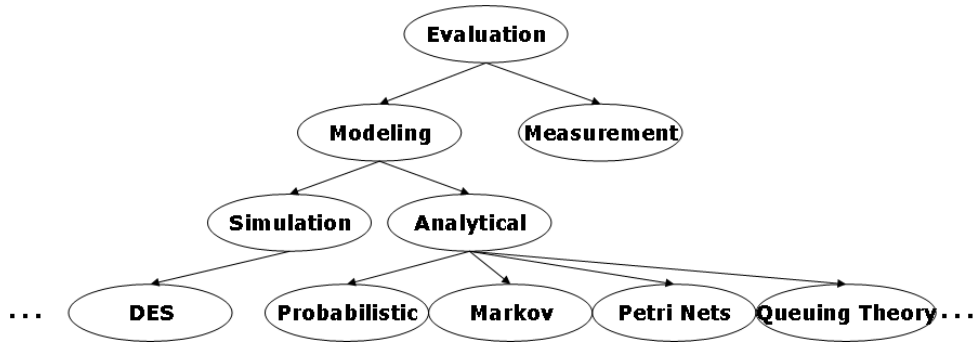


Figure 2.1: Performance Evaluation

The most direct method for performance as well as power evaluation are based on actual measurement of the system under study. Although measurement techniques can provide exact answers regarding the performance and power, during the design phase, the system (hardware prototype) is not always available for such experiments, and yet performance of a given design needs to be predicted to verify that it meets design requirements and to carry out necessary trade-offs [Bol06]. Another drawback of the measurement approach is that performance (energy consumption also) of only the existing configuration can be measured or, in the best cases, it might allow limited reconfiguration through code changing. Furthermore, the measurement results may or may not be accurate depending on the current states of the system, in which such technique has been performed. It is also important to state that a possible solution for such issue could be the adoption of statistical approaches that may guarantee the measurement results. Instead, the computational effort (human also) may turn this solution inadequate.

Modeling methods are typically adopted in early stages of the design process, when entire systems or prototypes are not yet available for measurements. Performance modeling may further be divided into simulation-based modeling and analytical modeling. Figure 2.1 shows the classification of performance evaluation, in which the analytical models deal with probabilistic methods, queuing theory, Markov models, or Petri nets [Joh06]. The basic principle of the analytic approaches is to represent the formal system description either as a single equation from which the interesting measures can be obtained as closed-form solutions, or as a set of system equations from which exact or approximate metrics can be calculated through numerical methods [Bol06]. However, in order to be able to have tractable solutions, simplified assumptions are often made regarding the structure of the model and, hence, a compromise between tractability and accuracy is often a challenge. In fact, Jain [Jai91] has observed that “analytical modeling requires so many simplifications and assumptions that if the results turn out to be accurate, even the analysts are surprised”.

An alternative to analytical models is the adoption of simulation-based models, where the most popular of them are based on discrete-event simulation (DES) [Bol06]. The results obtained through simulation approaches have not been so accurate as the ones

provided by measurements techniques, but it is possible to calculate the estimates precision. The principal drawback of simulation models, however, is the time taken to run such models for large, realistic systems, particularly when results with high accuracy (i.e.: narrow confidence intervals) are desired. Simulation approaches deal with a statistical investigation of output data of both performance and energy analysis, and the verification and validation of simulation experiments.

It is important to state that each technique can be adopted in different situation. Thus, the decision of which approach should be adopted depends on each situation. Another characteristic that the reader should be in mind is to create appropriate models containing only needed details to simplify the models. The next section describes some measurement strategies and their issues.

2.1.2 Measurement Strategies

It is possible to adopt many different types of performance metrics in order to perform a measurement. Different strategies are adopted for measuring the values of these metrics considering the system state changes (events). These events are classified as:

- *Event-count metrics.* Metrics that just counts the number of times a specific event occurs. An Example of event-count metrics is the number of disk input/output requests performed by a software.
- *Secondary-event metrics.* These types of metrics record the values of secondary parameters after an event happens. For instance, in the performance evaluation of a software, its energy consumption value may be computed.
- *Profiles.* A profile is an aggregate metric used to characterize the overall behavior of an application program or of an entire system. Typically, it is used to identify where the program or system is spending its execution time.

The above event-type classification is useful for helping the performance analyst to decide which measurement technique will be adopted, since different types of measurement tools are appropriate for measuring different types of events. There are basically three measurement strategies:

- (i) *Event-driven.* This strategy records only the information necessary to compute the performance metric. The simplest type of an event-driven measurement tool adopts the Event-count metrics in order to produce the results. This measurement technique is usually considered for low-frequency event systems.
- (ii) *Sampling.* Samples of the executing program are taken at fixed points in time. As a consequence, statistical approach has to be adopted to obtain precise results.

- (iii) *Indirect*. This strategy is considered when the desired performance metric is difficult (or impossible) to measure directly. In this case, another metric is measured from which the results are obtained.

There are advantages and drawbacks to each of these techniques. An event-driven measurement tool provides only a higher-level summary of the system behavior, such as overall counts or average durations. In contrast, sampling strategy adopts statistical approaches in order to provide the information. Thus, its results vary slightly each time the experiment is performed. The indirect strategy is just performed when the direct metric is not available. Several of the fundamental techniques that have been used for implementing the various measurement strategies are described in the following sections.

Interval timers

An interval timer is adopted to measure the execution time of a software or any code blocks within an application. It can also provide the time basis for a sampling measurement tool. The interval timer basic idea is to count the number of clock pulses that happens among the events. For that, calls to a routine that record the current timer count values are inserted before and after the predefined events. There are two common implementations of interval timers, one considering software interrupt and one adopting hardware counter.

Hardware timers compute the number of pulses they receive at their clock input from a clock source. The counter starts from “0” when the system is powered up and, so, the value read from the counter corresponds to the number of clock ticks that have occurred. On the other hand, the software interrupt adopts the hardware clock to generate a processor interruption. The interrupt-service routine is responsible to increment the counter variable that is read by an application.

Measurement perturbations

The measurement techniques implementation can perturb the computer systems performance measurement results. In order to obtain higher resolution measurements, for instance, more instrumentation points in a program are adopted. However, this causes more perturbations in the program than in its usual execution behavior. As a result, only the important data to infer the behavior of the system should be considered.

Measurement noise

Time is a fundamental quantity that needs to be measured to determine almost any aspect of a computer system’s performance [Mea00]. There are three important characteristics that determine the quality of the measurement results: (i) *accuracy*, the absolute difference between a measured value and the corresponding reference value; (ii) *precision*, the repeatability of the measurements performed; and (iii) *resolution*, the smallest incremental change that can be detected and displayed by a measuring tool.

Figure 2.2 depicts a histogram that shows the number of times each specific measurement occurred. Moreover, the histogram distribution indicates the measurement precision by the spread of the measurements around the mean value. On the other hand, accuracy is the difference between the mean of the measured values and the true value (see Figure 2.2).

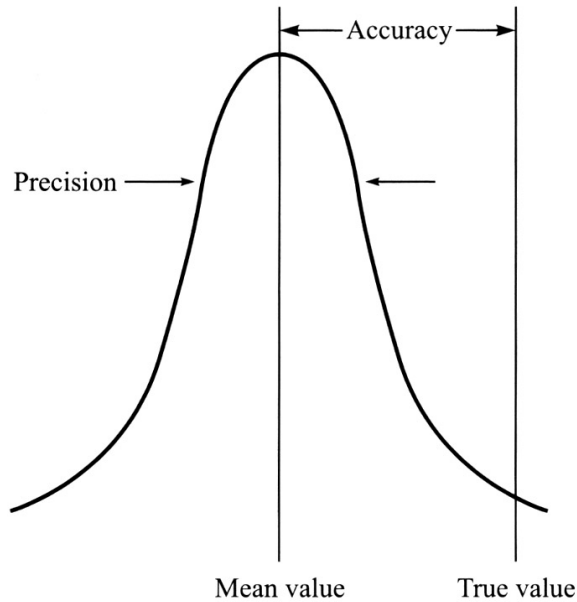


Figure 2.2: Histogram showing accuracy and precision.

Measuring accuracy, precision and resolution individual errors are difficult to be quantified by the results. Instead, a confidence interval for the mean value is adopted to quantify the precision of measurement results. On the other hand, quantifying the accuracy of measurements is more difficult because it involves, for example, the calibration of the clock source with a standard measurement of time.

2.2 SIMULATION PROCESS

Simulation is the execution of a model that reproduces the system behavior that it represents. In this context, there are two types of systems: terminal and non terminal. The terminal systems, also called transient systems, are those ones in which there are initial and final states well determined. The non terminal, also named stationary systems, consists of systems that the simulation is finished through a statistical stop criteria evaluation instead of an event that could happen. A stationary simulation approach has been adopted in this work.

Figure 2.3 depicts a general simulation process [LK99]. The simulation starts on the main program which invokes the initialization routine. The initialization routine sets the

simulation clock to “0” (variable indicating the current value of simulated time), initializes counters (variables used for storing statistical information about system performance and energy consumption), and starts the event list (list that contains the transition times for each transition able to fire). Afterwards, the main program invokes the timing routine which determines the next event type (the transition that is fired) and advances the simulation clock. Next, the main program invokes the event routine, in which the system state and statistical counters are updated, future events are generated and added to the event list. Then, it is determined whether the simulation should be finished or not, according to the stop criteria evaluation. After finishing the simulation, the estimates results are showed.

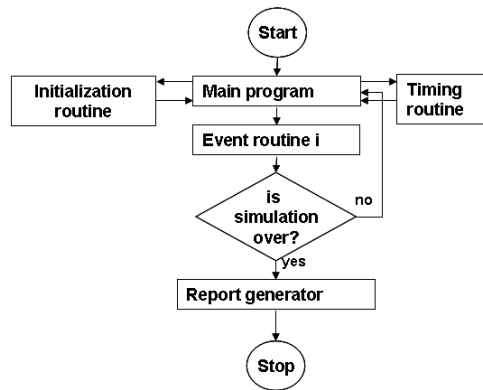


Figure 2.3: Simulation process diagram.

2.3 SYSTEM CLASSIFICATIONS

This dissertation adopts a temporized discrete event system through stochastic simulation. Before getting into the details of this particular class of systems, it is reasonable to start out by simply describing what the word “system” means, and by presenting the system classifications. Systems does not have an exact definition. Three representative system definitions are presented as follows:

- (i) An aggregation or assemblage of things so combined by nature or man as to form an integral or complex whole (Encyclopedia Americana).
- (ii) A regularly interacting or interdependent group of items forming a unified whole (Webster’s Dictionary).
- (iii) A combination of components that act together to perform a function not possible with any of the individual parts (IEEE Standard Dictionary of Electrical and Electronic Terms).

The system classifications have been adopted to describe the scope of different aspects of system and control theory. Such classification is really important in order to understand

the adopted models and simulation mechanism (temporized discrete event models through stochastic simulation) by this work. Figure 2.4 depicts these classifications in which systems are divided into two main groups, *Static*, system that does not depend on the past, and *Dynamic*, which are systems whose output depends on the input. The *Dynamic* systems can be divided into *time-varying* and *time-invariant*. The *time-invariant* systems, also called stationary systems, are divided into *linear* and *non-linear* systems. The *non-linear*, a system whose performance cannot be described by equations, is divided into *discrete-state* and *continuous-state* systems. The *discrete-state* system is divided into event-drive and time-driven systems. The *event-drive* system, in which the state is changed by the occurrence of an event, is divided into *stochastic* and *deterministic* systems. The *stochastic* system is divided into discrete-time and continuous-time. The following items describe each system in more details.

- *Static and Dynamic Systems.* Systems whose output is always independent of past values of the input are classified as static. On the other hand, dynamic systems are those systems whose output depend on past values of the input. In order to describe the behavior of dynamic systems, differential equations are generally required.
- *Time-varying and Time-invariant Systems.* The behavior of time-invariant systems does not change with time. This property, also called stationarity, implies that such systems always respond in the same way.
- *Linear and Nonlinear Systems.* A linear system satisfies the condition $g(a_1u_1 + a_2u_2) = a_1g(u_1) + a_2g(u_2)$, where u_1, u_2 are two input vectors, a_1, a_2 are two real numbers, and $g(\cdot)$ is the resulting output. Thus, linear systems correspond to system where all the interrelationships among the quantities involved cannot be expressed by linear equations (e.g.: algebraic, differential or integral)
- *Continuous-State and Discrete-State Systems.* The state variables can generally take on any real (or complex) value in continuous-state systems. In discrete-state systems, the state variables are elements of a discrete set (e.g.: the non-negative integers).
- *Time-driven and Event-driven Systems.* The state continuously changes as time changes in time-driven systems. In event-driven systems, it is only the occurrence of asynchronously generated discrete events that forces instantaneous state transitions.
- *Deterministic and Stochastic Systems.* A system becomes stochastic whenever one or more of its output variables is a random variable. In this case, the state of the system is described by a stochastic process, and a probabilistic framework is required to characterize the system behavior.
- *Discrete-time and Continuous-time Systems.* In continuous-time systems, all input, state, and output variables are defined for all possible values of time. On the other side, discrete-time systems have one or more of these variables defined at discrete points in time only, usually as the result of some sampling process.

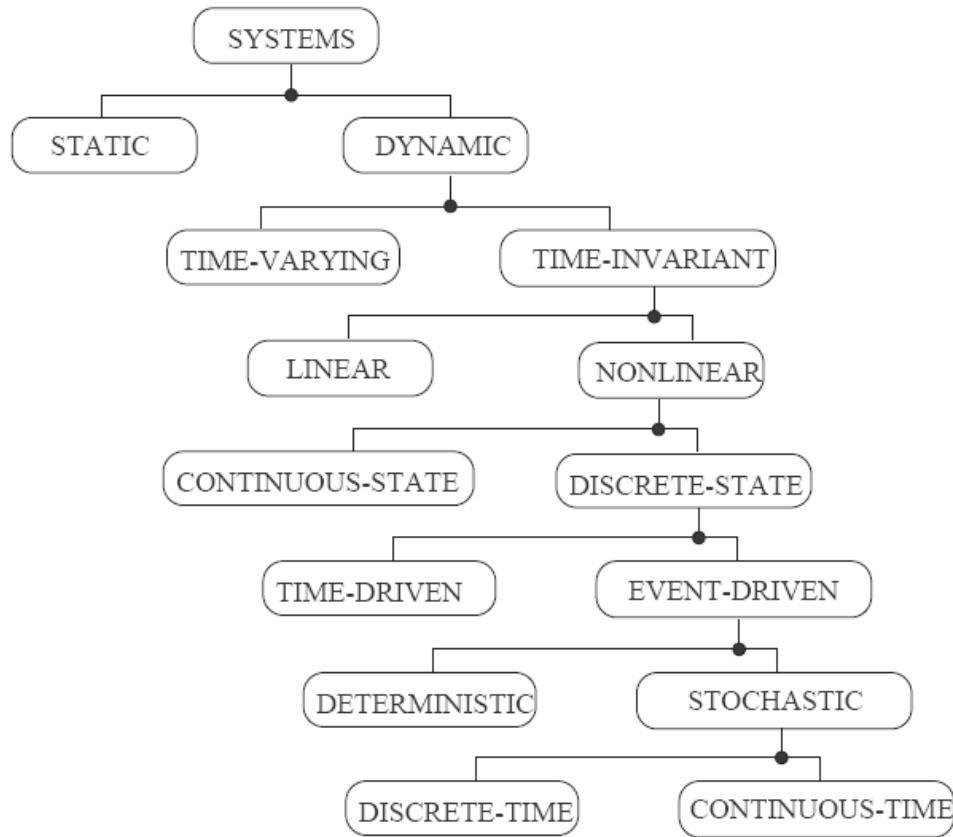


Figure 2.4: System classifications.

2.4 PETRI NETS

Formal methods consist of writing formal descriptions, analyzing those descriptions and, in some cases, producing new descriptions from them in order to obtain refinements [Hal07]. Among the formal methods, Petri nets have been adopted in this work.

Petri nets (PN) were introduced in 1962 by the PhD dissertation of Carl Adams Petri [Pet62], at Technical University of Darmstadt, Germany. The original theory was developed as an approach to model and analyze communication systems. Petri Nets (PNs)[Mur89] are a graphic and mathematical modeling tool that can be applied in several types of systems and allow the modeling of parallel, concurrent, asynchronous and non-deterministic systems. Since its seminal work, many representations and extensions have been proposed for allowing more concise descriptions and for representing systems feature not observed on the early models. Thus, the simple Petri net has subsequently been adapted and extended in several directions, in which timed, stochastic, high-level, object-oriented and coloured nets are a few examples of the proposed extensions.

2.4.1 Place-Transition Nets

Place/Transition Petri nets are one of the most prominent and best studied class of Petri nets, and it is sometimes called just by Petri net (PN). A marked Place/Transition Petri net is a bipartite directed graph, usually defined as follows:

Definition 2.4.1. (Petri Net) A Petri net [Mur89] is a 5-tuple:

$$PN = (P, T, F, W, M_0)$$

where:

- (i) $P = \{p_1, p_2, \dots, p_m\}$ is a finite set of places;
- (ii) $T = \{t_1, t_2, \dots, t_n\}$ is a finite set of transitions;
- (iii) $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (flow relation);
- (iv) $W : F \rightarrow \{1, 2, 3, \dots\}$ is a weight function;
- (v) $M_0 : P \rightarrow \{0, 1, 2, 3, \dots\}$ is the initial marking;

This class of Petri net has two kinds of nodes, called places (P) represented by circles and transitions (T) represented by bars, such that $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$. Figure 2.5 depicts the basic elements of a simple PN. 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. Figure 2.6 shows multiple arcs connecting places and transitions in a compact way by a single arc labeling it with its weight or multiplicity k .

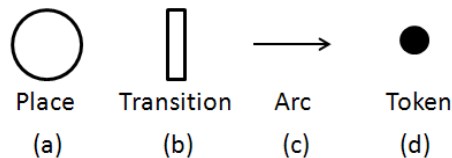


Figure 2.5: Petri net basic elements.

Places and transitions may have several interpretations. Using the concept of conditions and events, places represent conditions, and transitions represent events, such that, an event may have several pre-conditions and post-conditions. For more interpretations, Table 2.2 shows other meanings for places and transitions [Mur89].

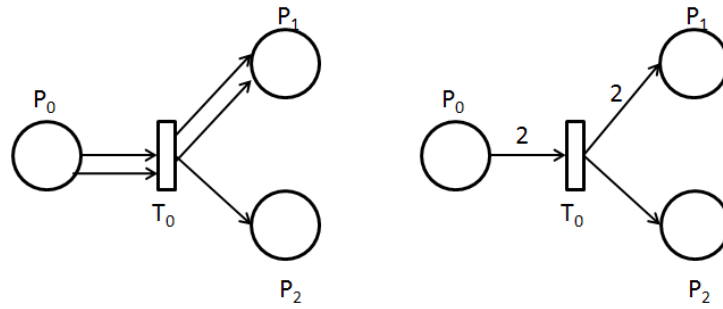


Figure 2.6: Compact representation of a PN

Table 2.2: 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 needed	tasks	resource releasing
conditions	logical clauses	conclusions
buffers	processor	buffers

It is important to show that there are another way to represent PN's elements. As an example, the set of input and output places of transitions is shown in Definition 2.4.2. Similarly, the set of input and output transitions of determinate place is shown in Definition 2.4.3.

Definition 2.4.2. (Input and Output Transitions of a place) The set of input transitions (also called pre-set) of a place $p_i \in P$ is:

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

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

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

Definition 2.4.3. (Input and Output Places of a transition) The set of input places of a transition $t_j \in T$ is:

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

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

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

2.4.2 Marked Petri Nets

A marking (also named token) has a primitive concept in PNs such as place and transitions. Markings are information attributed to places; the number and mark distributions consist of the net state in determined moment. The formal definitions are presented as follows.

Definition 2.4.4. (Marking) Considering the set of places P in a net N , the formal definition of marking is represented by a function that maps the set of places P into non negative integers $M : P \rightarrow \mathbb{N}$.

Definition 2.4.5. (Marking vector) Considering the set of places P in a net N , the marking can be defined as a vector $M = (M(p_1), \dots, M(p_n))$, where $n = \#(P)$, $\forall p_i \in P / M(p_i) \in \mathbb{N}$. Thus, such vector gives the number of tokens in each place for the marking M_i .

Definition 2.4.6. (Marked net) A marked Petri net is defined by a tupla $NM = (N; M_0)$, where N is the net structure and M_0 is the initial marking.

A marked Petri net contains tokens, which reside in places, travel along arcs, and their flow through the net is regulated by transitions. A peculiar distribution (M) of the tokens in the places, represents a specific state of the system. These tokens are denoted by black dots inside the places as shown in Figure 2.5 (d).

2.4.3 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:

- (i) 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 . Adopting a mathematical notation, an enabled transition t for given marking m_i is denoted by $m_i[t >]$, if $m_i(p_j) \geq W(p_j, t), \forall p_j \in P$.
- (ii) An enabled transition may or may not fire (depending on whether or not the respective event takes place).
- (iii) 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' . Using a mathematical notation, the firing of a transition is represented by the equation $m_j(p) = m_i(p) - W(p, t) + W(t, p), \forall p \in P$. If a marking m_j is reachable from m_i by firing a transition t , it is denoted by $m_i[t > m_j]$.

Figure 2.7 (a) shows a Petri net model example with three places and one transition. Figure 2.7 (b) outlines its respective graphical representation, and Figure 2.7 (c) provides the same graphical representation after the firing of t_0 . For this example, the set of reachable markings is $m = \{m_0 = (3, 1, 0), m_1 = (1, 0, 2)\}$. The marking m_1 was obtained by firing t_0 , such that, $m_1(p_0) = 3 - 2 + 0$, $m_1(p_1) = 1 - 1 + 0$, and $m_1(p_2) = 0 - 0 + 2$.

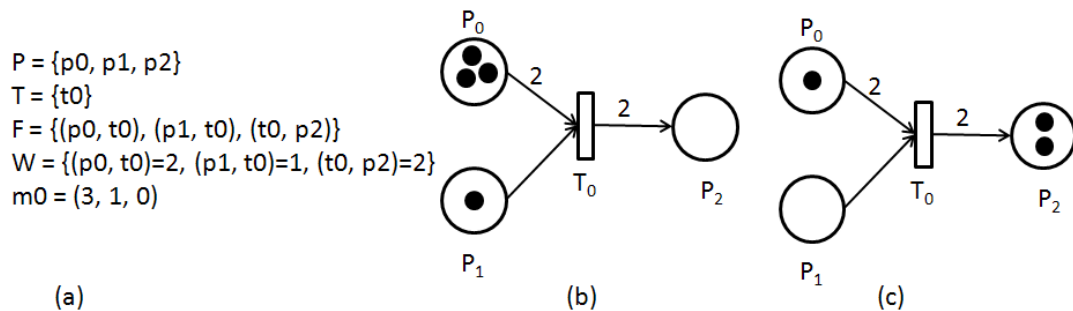


Figure 2.7: (a) Mathematical formalism; (b) Graphical representation before firing of t_0 ; (c) Graphical representation after firing of t_0 .

2.4.4 Petri Net Analysis Methods

In order to verify that a given PN satisfies certain properties (e.g.: deadlock freedom, liveness [Mur89]), it is necessary to adopt some analysis methods. The Petri net analysis methods may be divided into three groups: the reachability tree method, analysis based on the matrix-equations and reduction techniques. In this work, the analysis based on reachability tree and reduction techniques are presented.

Reachability Based Methods

The analysis method namely Reachability Tree is based on the building of a tree that makes possible to represent all reachable markings of a net [MLC96].

From the initial marking of a PN, it is possible to obtain some markings through the fireable transitions. Such possibilities can be represented as a tree, where the nodes correspond the markings and the arcs represent the fired transitions.

The reachability tree has been generated through initial marking of the net and adding directly reachable markings as leaves. Next, the process proceeds by these new markings in order to determine their directly reachable markings. These markings now become the new leaves of the already generated part of the reachability tree. If the desired marking is reached, it is not necessary to continue building the tree any further at that node. Such Reachability trees can be transformed directly into graphs by removing multiple nodes and connecting the nodes appropriately. This graph is called a reachability graph.

Definition 2.4.7. (Reachability Tree) Considering a Marked Petri net $MN = (N; M_0)$, a reachability tree is defined by $RT = (S, A)$, where S represents the markings and A the labeled arcs by $t_j \in T$.

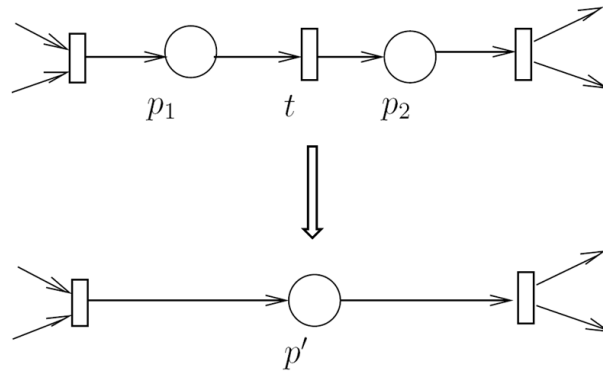
Some PN's properties such as boundedness, safeness, deadlock freedom and reachability can be analyzed through these reachability tree T by adopting the following rules [Mur89]:

- (i) A Marked Petri net $(N; M_0)$ is bounded and thus $R(M_0)$ is finite if and only if (iff) W (from the weight function - Definition 2.4.1) does not appear in any node labels in T ;
- (ii) A Marked Petri net $(N; M_0)$ is safe iff only 0's and 1's appear in code labels in T ;
- (iii) A transition t is dead iff it does not appear as an arc label in T ;
- (iv) If M is reachable from M_0 , then there will be a node labeled M' such that $M \leq M'$.

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.

Reduction techniques

Reduction analysis deals with the reduction of the Place-Transition net by replacing subnets of the net by less complex subnets such that several properties remain invariant. A reduction technique that reduces sequential states (places) has been considered in the

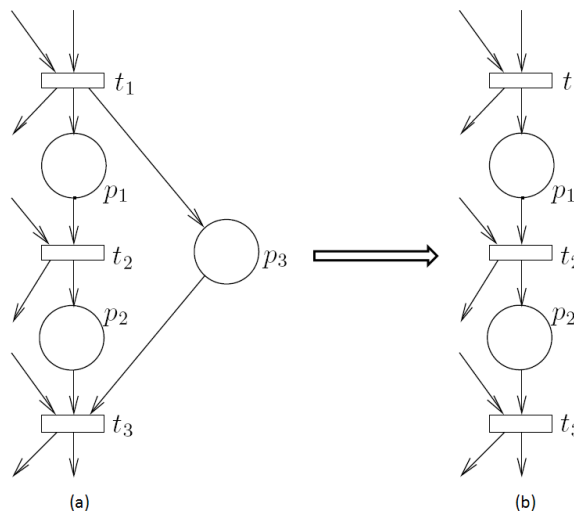


Font: [BK02, p. 113]

Figure 2.8: Simple reduction rule.

Petri net model showed in Figure 2.8. This figure shows that the places p_1 and p_2 were replaced by the place p' .

Figure 2.9 (a) depicts a petri net model in which it is possible to observe that the place p_3 represents a redundant path. In order to reduce the redundant path, a transformation starts by removing arcs from transitions to the place. If all arcs are removed, then the place can be removed from the PN. A place is redundant “when its marking is always sufficient for allowing firings of transitions connected to it” [Ber86]. Figure 2.9 (b) shows the petri net model after performing the reduction technique for removing the redundant path.

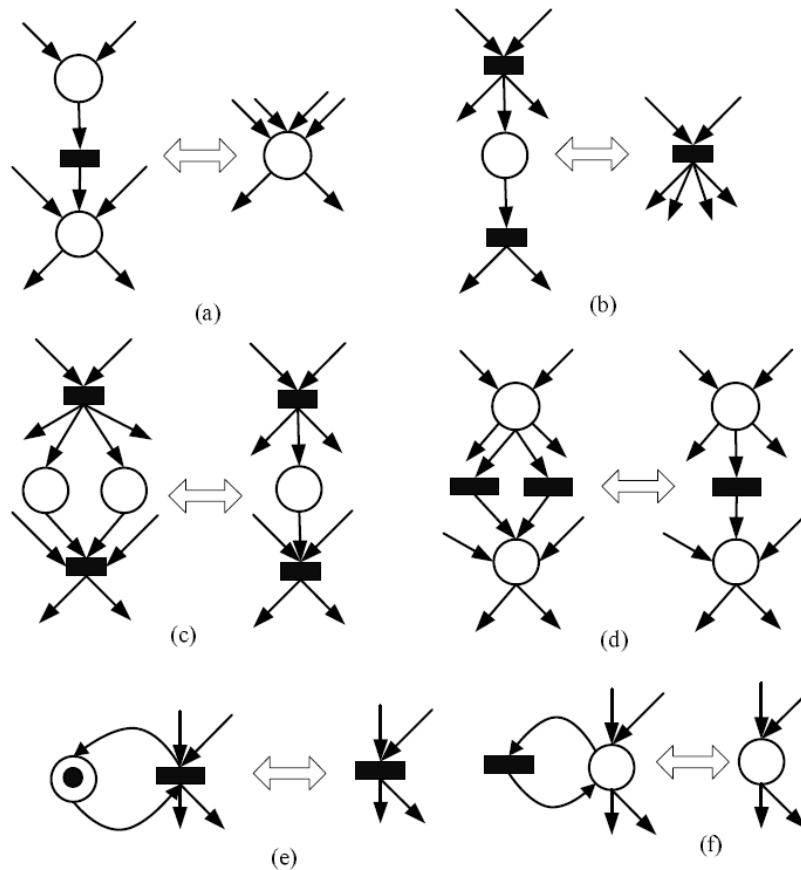


Font: [BK02, p. 114]

Figure 2.9: Reduction of a PN with a redundant place

Murata [Mur89] presented other simple reduction rules, in which the nets have been reduced by applying fusion of places and transitions, and by the elimination of loops. Figure 2.10 depicts six reduction operations.

- (i) Fusion of Series Places, Figure 2.10 (a).
- (ii) Fusion of Series Transitions, Figure 2.10 (b).
- (iii) Fusion of Parallel Places, Figure 2.10 (c).
- (iv) Fusion of Parallel Transitions, Figure 2.10 (d).
- (v) Elimination of Self-Loop Places, Figure 2.10 (e).
- (vi) Elimination of Self-Loop Transition, Figure 2.10 (f).



Font: [Mur89, p. 553]

Figure 2.10: Six transformations preserving properties.

2.5 TIME EXTENSIONS

The original definition of Petri nets does not include any notion of time and their aims are to model the logical behavior of systems by describing the causal relations between their events. Many researches have been proposing different ways for incorporating timing in Petri Nets, and the first ones related to them were presented by P.M Merlin et al. [MF76] and J.D Noe et al. [NN73]. In Timed Petri nets (TPN), time may be associated to places, transitions or tokens [vdAvHR00] such that:

- *Places* : The time can be associated to places, in which the markings in the output places only will be available to fire after a determinate amount of time.
- *Token* : The time can be added to the token, in which it have an information indicating when the token will be available to fire a transition.
- *Transitions* : The time can be associated to transitions. In this case, the fire of a transitions only happens after some delay correspondent to the time associated to the transition.

In TPN, the time can be deterministic, stochastic or between intervals.

- *Deterministic* : In this case, a deterministic time is adopted to represent the events.
- *Interval Computations*: In this case, intervals are adopted in order to describe the higher and shorter limits related to the time of each activity.
- *Stochastic* : This model adopts a probabilistic approach.

Since transitions represent activities that change the state (marking) of the net, it seems natural to associate time to transitions. For this, there are two different firing policies in TPN:

- *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 such tokens are consumed from input places and generated in output places when the transition fires. The firing itself does not consume any time.

In atomic firing, when a transition is able to fire, a timer associated to the transition is started. Such timer decreases in a constant way, and the transition is fired when the timer value goes to zero. There is an issue related to the other transitions timers and, in order to solve such issue, the following approaches have been adopted to represent the memory policies whenever a transition fires [vdAvHR00]:

- *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 reset. 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.

2.6 COLOURED PETRI NETS

Among the Petri net extensions that have been proposed, it is important to stress Jensen's high-level model, the so called Coloured Petri net (CPN)[Jen95] [JKW07]. In this model, a token may have complex data type as in programming languages; each place has the correspondent data type, hence restricting the kind of tokens that it may receive; the transitions process the token values and create new ones with different values; hierarchy structure can be modeled with different abstraction levels, where each transition may describe another net (called subnet), and so on. Indeed, CPN is a high-level model that considers abstract data-types and hierarchy.

Likewise in PN, places are graphically represented by ellipses, transition by rectangles, and arcs by direct arrows. Moreover, a Timed CPN has been adopted in this research. The main difference between timed and non timed CPN models is that the tokens in a timed CPN model, in addition to the token colour, can carry a second value called a time stamp [JKW07].

CPN has some properties which turns such formalism a valuable language for the design, specification and analysis of many different types of systems. Among these properties, it is possible to stress:

- (i) *well-defined semantics*: CPNs have a well-defined semantics that turns them able to represent complex systems;
- (ii) *hierarchical descriptions*: it is possible to build a large and complex CPN by relating smaller CPNs to each other, in a well-defined way. This hierarchical property is similar to subroutines, procedures and modules of programming languages;
- (iii) *time concept*: CPNs can be extended to cover the time concept;
- (iv) *interactive simulations*. In CPN simulations, it is possible to fire a determinate number of transitions and to see their result values instantaneously;
- (v) *inscriptions*: CPNs allow one to associate inscriptions to CPN components such as places, arcs, and transitions in order to improve their functionalities;

- (vi) *reduction process*: due to their precise semantics, it is possible to apply a reduction process where a new simplified CPN model can be obtained preserving determinate properties.

The formal definition of Coloured Petri nets is based on the following entity definitions.

Definition 2.6.1. (Multi-set) Multi-set is a function that describes the set of element collections with identical colour (data type). Let \mathbb{N} be the set of all non-negative integers. The multi-set MS , defined over a non-empty set S , is a function $m : S \rightarrow \mathbb{N}$, where:

$$MS = \sum_{s \in S} m(s)'s$$

S_{MS} denotes the set of all multi-sets over S . The non-negative integers $\{m(s) | s \in S\}$ are the coefficients of the multi-set.

Considering a collection of tokens that represents the pets in a house, this token type is “PET”. A state of the house related to the number of pets can be defined by: 2 tokens representing “dog” value and 4 tokens representing “cat” value. This multi-set is:

$$MS = \sum_{s \in S} m(s)'s = 2'dog + 4'cat.$$

Multi-sets consider a number of standard operations.

Definition 2.6.2. (Multi-set Operations) Let a set of multi-set $\{m, m_1, m_2\} \subseteq SMS$ and n a non-negative integer. The following basic operations are defined among multi-sets:

- (i) $m_1 + m_2 = \sum_{s \in S} (m_1(s) + m_2(s))'s$ (addition)
- (ii) $n * m = \sum_{s \in S} (n * m(s))'s$ (scalar multiplication)
- (iii) $m_1 \neq m_2 \Rightarrow \exists s \in S | m_1(s) \neq m_2(s)$ (comparison \neq)
- (iv) $m_1 \leq m_2 \Rightarrow \exists s \in S | m_1(s) \leq m_2(s)$ (comparison \leq)
- (v) $m_1 \geq m_2 \Rightarrow \exists s \in S | m_1(s) \geq m_2(s)$ (comparison \geq)

$$(vi) |m| = \sum_{s \in S} m(s) \text{ (size)}$$

$$(vii) m_2 - m_1 = \sum_{s \in S} (m_2(s) - m_1(s))'s, \text{ iff(if and only if) } m_2 \geq m_1 \text{ (subtraction)}$$

For a better comprehension of the formal concepts related to CPN, it is also important to define some primitive operators.

Definition 2.6.3. (Type operator) Let V , the set of variables; EXP , expressions and T , model types. The function $Type : V \cup EXP \rightarrow T$ denotes type operator, where $Type(exp)$ maps the variable or expression exp into a valid type.

Definition 2.6.4. (Operator of set of variables) Let V be the set of variables and EXP expressions. The function $Var : EXP \rightarrow V$ denotes the operator of set of variables, where $Var(exp)$ represents the set of variables in an exp expression.

The Formal definition of Coloured Petri nets is presented as follows.

Definition 2.6.5. (Coloured Petri Net) The non-hierarchical definition of Coloured Petri Net [Jen94] is a nine-tuple:

$$CPN = (\sum, P, T, A, N, C, G, E, I)$$

satisfying the following requirements:

- (i) \sum is a finite set of non-empty types, called colour sets;
- (ii) P is a finite set of elements (Places) that represents local states;
- (iii) T is a finite set of elements (Transitions) that depicts events and actions;
- (iv) A is a finite set of arcs such that $P \cap T = P \cap A = T \cap A = \emptyset$;
- (v) N is a node function defined from A into $P \times T \cup T \times P$;

- (vi) C is a colour function defined from P into Σ ;
- (vii) G is a guard function defined from T into expressions such that: $\forall t \in T : [Type(G(t)) = Bool \wedge Type(Var(G(t))) \subseteq \Sigma]$, where $Bool \in \{true, false\}$;
- (viii) E is an arc function defined from A into expressions such that: $\forall a \in A : [Type(E(a)) = C(p(s))_{MS} \wedge Type(Var(E(a))) \subseteq \Sigma]$, where $p(a)$ is the place of $N(a)$ and C_{MS} denotes the set of all multi-sets over C ;
- (ix) I is an initialization function defined from P into closed expressions such that $\forall p \in P : [Type(I(p)) = C(p(s))_{MS}]$.

where:

$Type(expr)$ denotes the type of an expression;

$Var(expr)$ denotes the set of variables in an expression;

$C(p)_{MS}$ denotes a multi-set over $C(p)$.

After analyzing the formal definition of CPN, the reader should conclude that:

The set of types - item(i) - determines the data values, operations and functions that can be adopted in the net expressions (i.e.: arc expressions, guards and initialization expressions).

Places, transitions and arcs - item (ii), (iii), (iv) - are described by three sets P , T and A which are finite and pairwise disjoint ($P \cap T = P \cap A = T \cap A = \emptyset$).

The node function - item (v) - maps each arc into a pair where the first element is the source node and the second the destination node. It is important to stress that both elements must be of different kind (i.e.: place and transition).

The colour function C - item (vi) - maps each place, p , to a type $C(p) \in \Sigma$. This means that each token on p must have a data value that belongs to $C(p)$.

The guard function G - item (vii) - maps each transition, t , into a boolean expression, where all variables have types that belong to S .

The arc expression function E - item (viii) - maps each arc, a , into an expression of type $C(p)_{MS}$. This means that each arc expression must evaluate to multi-sets over the type of the adjacent place, p .

The initialization function I - item (ix) - maps each place, p , into a closed expression which must be of type $C(p)_{MS}$. A *Closed expression* is an expression without variables

which can be evaluated in all bindings, and all evaluations give the same value.

The behavior of CPN models

For a better comprehension of concepts related to the behavior of CPN models, it is important to introduce the following notation for all $t \in T$ and for all pairs of nodes $(x_1, x_2) \in (P \times T \cup T \times P)$:

$$A(t) = \{a \in A \mid N(a) \in (P \times T \cup T \times P)\}$$

$$Var(t) = \{v \mid v \in Var(G(t)) \vee \exists a \mid a \in A(t): v \in Var(E(a))\}$$

$$A(x_1, x_2) = \{a \in A \mid N(a) = (x_1, x_2)\}$$

$$E(x_1, x_2) = \sum_{(a \in A(x_1, x_2))} E(a)$$

In CPNs, sometimes token values refer to token colours, in the same way data types are referred as colour sets. The formal definition of token is depicted as follows.

Definition 2.6.6. (Token) Token represents a value (literal) that can be either primitive or composite data types. A token element is a pair (p, c) where $p \in P$ and $c \in C(p)$. TE denotes the set of all token elements.

Data values are assigned (bound) to variables present in the arc expressions on the surrounding arcs of transitions in order to evaluate whether a transition may or may not occur. The formal definition of binding element is presented as follows.

Definition 2.6.7. Binding. A binding element is a pair (t, b) where $t \in T$ and $b \in B(t)$, where $B(t)$ denotes the set of all bindings for t . A binding of a transition t is a function b defined on $Var(t)$, such that:

$$(i) \forall v \in Var(t): b(v) \in Type(v).$$

$$(ii) G(t) < b > \text{ denotes the evaluation of the guard expression } G(t) \text{ in the binding } b.$$

A state of a CPN is called a marking [KCJ98], and its definition is given as follows.

Definition 2.6.8. Marking. It consists of a number of tokens positioned (distributed) on the individual places. Each token carries a value (colour), which belongs to the type of the place on which the token resides. The tokens present on a particular place are called the marking of that place.

A marking is a multi-set over TE (the set of all token elements) while a *step* is a non-empty and finite multi-set over BE (the set of all binding elements). The initial marking M_0 is the marking which is obtained by evaluating the initialization expressions.

It is possible to attach a boolean expression (with variables), called guard, to each transition. A transition is enabled if each of its input places contains the multi-set specified by the input arc inscription and the guard evaluates to true. The formal definition of enabling a transition is defined as follows.

Definition 2.6.9. Enabled Transitions. A step Y is enabled in a marking M if and only if the following property is satisfied:

$$(i) \quad \forall p \in P : \sum_{(t,b) \in Y} E(p,t) \langle b \rangle \leq M(p).$$

When a transition is enabled it may occur (fire). An occurrence of a transition removes tokens from places connected to incoming arcs (input places), and adds tokens to places connected to outgoing arcs (output places), thereby changing the marking (state) of the CPN [KCJ98]. The number and colour of the tokens are determined by the arc expressions, evaluated for the occurring bindings. The formal definition is presented as follows.

Definition 2.6.10. Firing of an Enabled Transition. When a step Y is enabled in a marking M_1 it may occur, changing the marking M_1 to another marking M_2 , defined by:

$$(i) \quad \forall p \in P : M_2(p) = (M_1(p) - \sum_{(t,b) \in Y} E(p,t) \langle b \rangle) + \sum_{(t,b) \in Y} E(t,p) \langle b \rangle.$$

M_2 is directly reachable from M_1 ($M_1[Y]M_2$).

where:

The expression evaluation $E(p,t) \langle b \rangle$ computes the tokens which are removed from p when t occurs with the binding b .

The expression evaluation $E(t, p) < b >$ computes the tokens which are added to places connected to outgoing arcs with the binding b .

Reduction Rule

CPN can also be analyzed by means of reduction, where the main idea is to define the desired properties to investigate, and then it is applied to a set of rules by which the CPN can be simplified [Jen95] [Mur89]. A typical rule is depicted in Figure 2.11, where two transitions T1 and T2 were replaced by one transition T3.

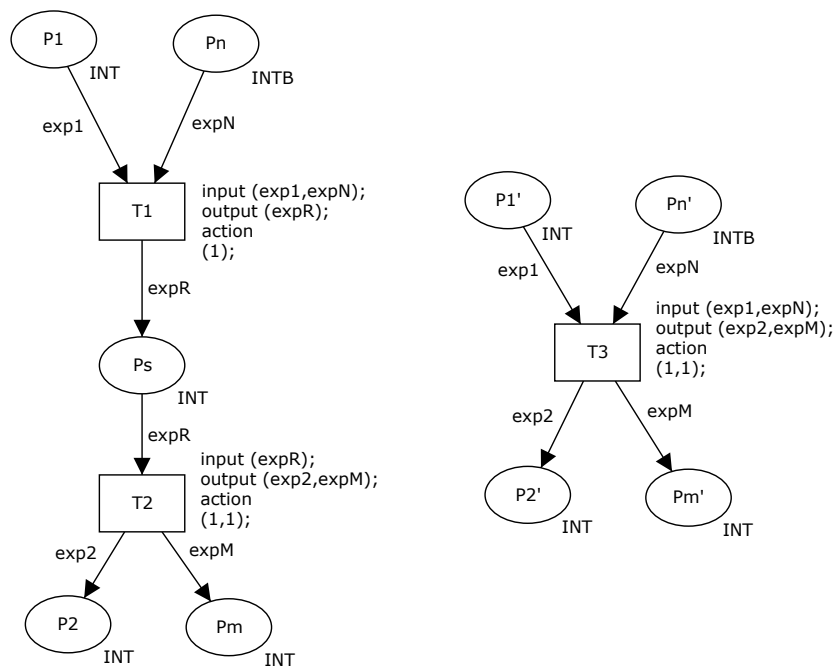


Figure 2.11: Reduction Rule.

2.6.1 CPN ML Language

In order to have a better understating about the Embedded Software Modeling (Chapter 5), it is important to introduce some concepts of the CPN ML Language. First of all, this section describes some history overview of CPN ML Language. Afterwards, it presents some reasons that justify the adoption of a Standard ML Language as the basis of CPN ML. Subsequently, some examples of declarations and net inscriptions are given.

The CPN ML Language is an extension of a well-known functional programming language, Standard ML (SML) [Har08], developed at Edinburgh University. Standard ML is a type-safe programming language that provides a richly expressive and flexible

module system for structuring large programs, including mechanisms for enforcing abstraction, imposing hierarchical structure, and building generic modules. Furthermore, such language is portable across platforms and implementations because it has a precise definition. Moreover, CPN Tools [cpn07], a free environment for CPN models, adopts the CPN ML language for declarations and net inscriptions.

The Standard ML was chosen as the basis of CPN Tools' language because its development team found advantages in adopting an existing language:

- (i) A general and more tested language can be adopted by this way. In addition, the creation of a new programming language is a very slow and expensive process.
- (ii) Instead of developing a new compiler from scratch, it was necessary only to port the compiler to the relevant kind of operating systems and integrate it with CPN Tools.
- (iii) The considerable amount of documentation and tutorial material, which already exists for Standard ML and for functional languages in general, can be reused.
- (iv) Standard ML has types, functions, operations, variables and expressions in a similar way as a typed functional language. Hence it is convenient to build upon such a language.
- (v) Standard ML has also a flexible and extendible syntax for allowing one to write the declarations and net inscriptions in a way which is close to standard mathematics.

Standard ML has compilers for commercial use available, and with such language it is possible to define mathematical functions as long as they are computable. Moreover, with that language the user can declare arbitrarily complex functions and operations. Standard ML also turns possible to perform a smooth integration between code segments and the net inscriptions. A code segment is a sequential piece of code attached to a transition that is executed each time such transition occurs, and it may update files or do other forms of reporting.

The CPNs models encompass three groups: structural, declarations and inscriptions [MLC96]. The declarations and inscriptions of CPNs models are performed through extensions of the Standard ML (as CPN ML). On the other hand, the structure of a CPN model consists basically on a marked graph with places and transitions. In the following lines, some declarations and inscriptions adopting CPN ML Language are presented.

Declarations

In CPNs, declarations are used to define color sets, functions, variables and constants.

Color set declarations: Two examples of color set declarations are depicted as follows:

```
colset INTEGER = int;

colset Context = record jump:INTEGER*test:BOOL;
```

The first example shows the declaration of the color set (colset), *INTEGER*, that considers integer values. The second color set definition shows a way to represent different types that are identified by a unique label, *Context*, in which *jump* represents an *INTEGER* type and *test* represents a boolean type. In order to define such representation, the word *record* has been considered.

Variable declarations: A variable is an identifier whose value can be changed during the execution of the model. The variable types must be one of the color sets previous declared, and such variables are used to guard and arc expressions. The term binding is commonly adopted in order to associate a value with a variable. There is another kind of variable named reference that has as its scope the entire CPN. These reference variables may be read and updated by code segments. The word *var* has been adopted to declare variables and the word *globref* is the one adopted to declare reference variables. Variable declarations and a reference variable are exemplified as follows, in which *i, Ki, Ko* are variables of the type *Context* and *nReplication* is a reference variable.

```
var i, Ki, Ko: Context; (variable)

globref nReplication=10; (reference variable)
```

Constant declarations: In constant declarations, a value is bound to an identifier which works as a constant variable. An example is listed as follows, in which the *CI* is the identifier and *val* the word adopted to declare constant identifiers.

```
(* Conf Interval *)
val CI =0.95;
```

Function declarations: Function declarations are declared in CPN ML as follows:

```
(* Standard Deviation of a List *)
fun standardDeviation(l) = ( let val v = variance(l); in Math.pow(v,0.5) end );
```

Inscriptions

Inscriptions are associated with CPN net components such as places, arcs, and transitions. Some inscriptions may or may not affect the behavior of a net. Depending on the type of inscriptions, different syntactic requirements are needed.

Place Inscriptions: There are three inscriptions that may be associated to a place, in which two are optional and one is required:

- *Colour set inscription* (required): determine the type (color) of all the tokens that can be put in a place.
- *Initial marking inscription* (optional): specify the initial tokens for a place.
- *Place name inscription* (optional): identify the place, and it may contain any sequence of characters.

Figure 2.12 shows inscriptions associated to a place. In this figure, the *name* corresponds to the place name inscription; the *value* is a constant associated to specify the initial marking; and *Context* is the color set of the tokens that such place may receive.

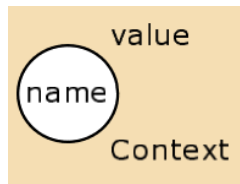


Figure 2.12: Place inscriptions.

Arc Inscriptions: An arc inscription is a CPN ML expression that evaluates a multi-set or a single element. It is important to state that the colour set of the arc expression must match the colour set of the place attached to the arc. Figure 2.13 depicts an arc with the inscription i .

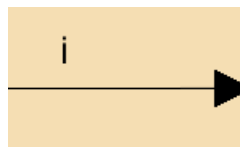


Figure 2.13: Arc inscription.

Transition Inscriptions: There are four inscriptions that may be associated to transitions and all of them are optional:

- *Transition name inscription:* a label that identifies the transition, and it may contain any sequence of characters.
- *Guard inscription:* a guard is a CPN ML boolean expression that may be evaluated to true or false.
- *Time inscription:* a transition delay may be associated such that if current time is 10 and the time delay is “@ + 2”, then the time stamp of tokens sent to the output places will be “12”. A missing time inscription is equivalent to a zero delay.
- *Code segment inscription:* Each transition may have an attached code segment which contains ML code. Code segments are executed when their parent transition occurs.

A function that describes the relation between transitions and code segment inscriptions is not present in the formal definition of Coloured Petri net. However, such function is important for a better understanding of the Chapter 5, in which the embedded software modeling are depicted.

Definition 2.6.11. (Code segment inscription Function) Let T be a set of Transitions and $CodSeg$ a set of code segments, in a Coloured Petri net, a function of code association is defined by $FCod: T \rightarrow CodSeg$. It is important to state that may exist T_i without $FCod$ defined.

The code segments can be used to different purposes such as: (i) to apply algorithms descriptions, (ii) to perform a stochastic simulation approach, (iii) to create protocol interfaces among different engines. Although there are different applicabilities of code segment inscriptions, their adoption may result in some limitations of the net analysis. This may happen because their use can change the model variables. As a consequence, the analysis by reachability tree is limited, since the CPN model behavior is not restrict to the rules present in arcs and guard expressions anymore [HJS91]. However, the adoption of code segment inscriptions are very helpful for approaches focused on the simulation by providing practicability and flexibility on the descriptions.

Figure 2.14 depicts a net that has some inscriptions associated to the transition $Trans$, in which it is possible to observe a guard expression that only enables the transition if $jump = 0$. Moreover, the code segment inscription associated to it changes the $jump$ value.

2.6.2 Timed Coloured Petri net

This work adopted the Timed Coloured Petri net (TCPN) [Jen95], a formal computational model, in order to assure timing constraints. The following subsections present concepts

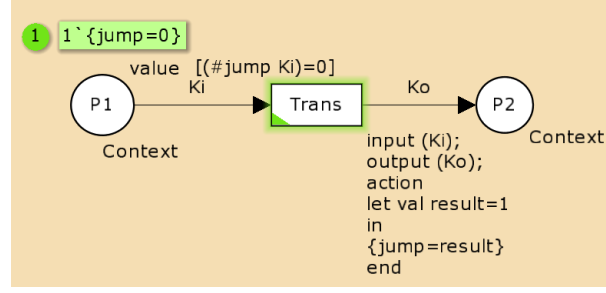


Figure 2.14: Transition inscriptions.

related to the formal model.

TCPN considers a global clock which represents the system model's time. More precisely, each token has a time stamp on it. The time stamp describes the earliest model time at which the token can be removed by a binding element. The current state of a net can be changed when an enabled transition fire. Then, the next enabled transition should be generated when the global clock is greater than or equal to the token's time stamp.

In order to represent token's time stamp, the operator “@” has been adopted in this dissertation to add time stamps. Places with timed colour sets contain timed multi-sets of values, and thus it is important to define a timed multi-set.

Definition 2.6.12. (Timed multi-set) A timed multi-set TM , over a non-empty set of states S , is a function $tm : S \times R \rightarrow \mathbb{N}$, where R is the set of time values (time stamps) and \mathbb{N} the non-negative integers. The time multi-set is defined by:

$$TM = \sum_{s \in S} tm(s)'s@tmv[s]$$

where:

$tm(s)$ is the number of appearances of the element $s \in S$ in the timed multi-set.

The list $tmv[s] = [r_1, r_2, \dots, r_{tm(s)}]$ is defined to contain the time values $r \in R$, in which $tm(s, r) \neq 0$.

S_{TMS} denotes the set of all timed multi-sets over S .

The non-negative integers $\{tm(s) | s \in S\}$ are called the coefficients of the timed multi-set.

Definition 2.6.13. (Timed Coloured Petri nets) A timed non-hierarchical Coloured Petri net is defined by a tuple $TCPN = (CPN, R, r_o)$, where CPN is a Coloured Petri net, R is the set of time values (time stamps) and r_o is the start time element of R .

A state is a pair (M, r) where M is a marking and r a time value. The initial state is the pair (M_o, r_o) . The sets of all markings and states are denoted by M and S , respectively.

It is possible to attach a boolean expression (with variables), called guard, to each transition. Considering the time constraints, a transition at a time r_2 is enabled in a state (M_1, r_1) if the time r_2 is greater than the time r_1 . Furthermore, the transition's input places must contain the multi-set specified by the input arc inscription and the guard is evaluated to true. The formal definition of enabling a transition is defined as follows.

Definition 2.6.14. (Enabled Transition Set) A step Y is enabled in a state (M_1, r_1) at time r_2 iff (if and only if) the following properties are satisfied:

$$(i) \sum_{(t,b) \in Y} E(p, t) < b > r_2 \leq M_1(p), \forall p \in P,$$

$$(ii) r_1 \leq r_2,$$

(iii) r_2 is the smallest element of R for which there exists a step satisfying (i) and (ii).

An occurrence of a transition removes tokens from places connected to incoming arcs (input places), and adds tokens to places connected to outgoing arcs (output places), thereby changing the marking (state) of the TCPN. The firing of an enabled transition is similar to the CPN Firing rule (Definition 2.6.10), in which the number and colour of the tokens are determined by the arc expressions, evaluated for the occurring bindings.

2.6.3 Hierarchical CPN

The basic idea of Hierarchical CPN (HCPN) [JKW07] [KCJ98] is to allow the modeler to construct hierarchical structures, represented by high-level transitions, called substitution transitions. This means that one can model a large CPN by relating smaller CPNs to each other, in a well-defined way. At one level, it is possible to give a simple description of the modeled activity without having to consider internal details about how it is carried out. At another level, it is possible to specify the more detailed behavior. The model that

is represented by the substitution transitions is named subpage, and the higher model, which has substitution transitions, is the page. These pages are connected to each other by input places and some output places called input and output socket places, respectively. The following lines present the formal definitions of HCPN.

Definition 2.6.15. (Hierarchical Coloured Petri Net) The Hierarchical formal definition of Coloured Petri Nets [Jen95] [NLHC03] is a tuple:

$$HCPN = (S, SN, SA, PN, PT, PA, FS, FT, PP)$$

where:

(i) S is a finite set of pages (subnets) such that:

- Each page $s \in S$ is a non-hierarchical CPN:

$$(\sum_s, P_s, T_s, A_s, N_s, C_s, G_s, E_s, I_s).$$

- The sets of net elements are pairwise disjoint:

$$\forall s_1, s_2 \in S : [s_1 \neq s_2 \Rightarrow (P_{s_1} \cup T_{s_1} \cup A_{s_1}) \cap (P_{s_2} \cup T_{s_2} \cup A_{s_2}) = \emptyset].$$

(ii) $SN \subseteq T$ is a set of substitution nodes;

(iii) SA is a page assignment function. It is defined from SN into S such that:

- No page is a subpage of itself:

$$s_0 s_1 \dots s_n \in S^* | n \in \mathbb{N}_+ \wedge s_0 = s_n \wedge \forall k \in 1 \dots n : s_k \in SA(SN_{s_{k-1}}) = \emptyset;$$

(iv) $PN \subseteq P$ is a set of port nodes;

(v) PT is a port type function. It is defined from PN into $\{\text{in, out, i/o}\}$;

(vi) PA is a port assignment function. It is defined from SN into binary relation such that:

- Socket nodes are related to port nodes:

$$\forall x \in SN : PA(x) \subseteq X(x) \times PN_{SA(x)};$$

- Related nodes have identical colour sets and equivalent initialization expressions:

$$\forall x \in SN : \forall (x_1, x_2) \in PA(x) : [C(x_1) = C(x_2) \wedge I(x_1) = I(x_2)];$$

(vii) $FS \subseteq P_s$ is a finite set of fusion sets such that:

- Members of fusion set have identical colour sets and equivalent initialization expressions:

$$\forall f_s \in FS : \forall p_1, p_2 \in f_s : [C(p_1) = C(p_2) \wedge I(p_1) = I(p_2)];$$

(viii) FT is a fusion type function. It is defined from fusion sets such that:

- Each fusion set is of type: global, page or instance.
- Page and instance fusion sets belong to a single page:

$$\forall f_s \in FS : [FT(f_s) \neq global \Rightarrow \exists s \in S : f_s \subseteq P_s]$$

(ix) $PP \in S_{MS}$ is a multi-set of prime pages, where prime pages is a multi-set over the set of all pages.

An HCPN consists of a non-hierarchical set of CPNs, named *Pages* - item (i) of Definition 2.6.15, that are connected by *SA* and *PA* assignment functions - item (iii) and (vi). The *SA* function assigns transitions to pages considering that no one transition assigns to itself page - item (iii). This transition represents the behavior of the connected net, named subpage. The transitions that belongs to *SA* domain are named *substitution nodes* (*SN*) or *substitution transitions* - item (ii). A substitution node is a net entity (at superpage) that substitutes or replaces, a lower hierarchical net and, so, there can also be *substitution places*. There is a well-defined interface between subpage and the corresponding superpage. This interface relates socket nodes (at superpage) to port nodes (at subpage). The *PA* function assigns places of a substitution transition (at superpage) to places (at subpage) that it represents, where the places connected to the substitution transitions are *sockets* and the places within the subpage are *port* nodes - item (vi). This function is related to nodes that have identical colour sets and equivalent initialization expressions. It is important to highlight that the *PA* function can assign a socket to many ports, and a port can be assigned to many sockets.

A set of places (or a set of transitions) can be unified (folded) into a single conceptual node. These nodes may reside on the same page or different pages. A fusion is obtained by defining a fusion set (FS) containing an arbitrary number of places or an arbitrary number of transitions that can be present in different pages. These places have identical colour and equivalent initialization expressions such that they belong to just one fusion set - item (vii). There are three different kind of fusion sets: *global* fusion sets are allowed to have members from many different pages, *page* fusion sets unifies all the places of a page instance, and *instance* fusion sets only have members from a specific page instance - item (viii). The prime pages - item (ix) - is a multi-set over the set of all pages and they determine, together with the page assignment, how many instances the individual pages have.

In addition, it is important to highlight that each non-hierarchical CPN is a hierarchical CPN with a single page. Thus, in non-hierarchical CPN there are no substitution transition, port and fusion nodes. Furthermore, the single page belongs to the multi-set of prime pages with the coefficient one.

2.7 EMBEDDED SYSTEMS

The advances of embedded systems have been providing more and more human-computer interaction, the so called ubiquitous computing. Embedded devices have been so integrated into everyday objects and human activities such as in cars and in telecommunication equipments that it is difficult to note the embedded device within them. A system is said to be embedded when it performs one or a few dedicated tasks, and its environment interactions is continuous through sensors and actuators [Mar03]. The sensors are responsible for collecting information about the embedded system environment and the actuators controls such environment. Such kind of systems only stops working if it is powered down. An embedded system is a special-purpose computer system that has some hard project restrictions such as size, performance, cost, power and others.

Following the success of ubiquitous computing for office and control flow applications, embedded systems are considered to be the most important application area of information technology during the coming years. Due to this fact, the term named post-PC era was created, in which the standard-PCs will not be anymore the dominant kind of hardware.

Due to the computer hardware evolutions, embedded systems have been more common day after day. Moore's law [Moo00] describes a long-term trend in the history of computing hardware. Since the invention of the integrated circuit in 1958, the number of transistors that can be placed inexpensively on an integrated circuit has increased exponentially, doubling approximately every two years. Figure 2.15 shows the doubling of the transistors counts in every two years. Furthermore, the processing power, measured in millions of instructions per second (MIPS), has steadily risen because of such increased transistor counts. Moore's Law previews also the decreasing costs of hardware equipments. As silicon-based technology gains in performance, it becomes less expensive to

produce, more plentiful and powerful, and more seamlessly integrated embedded systems.

CPU Transistor Counts 1971-2008 & Moore's Law

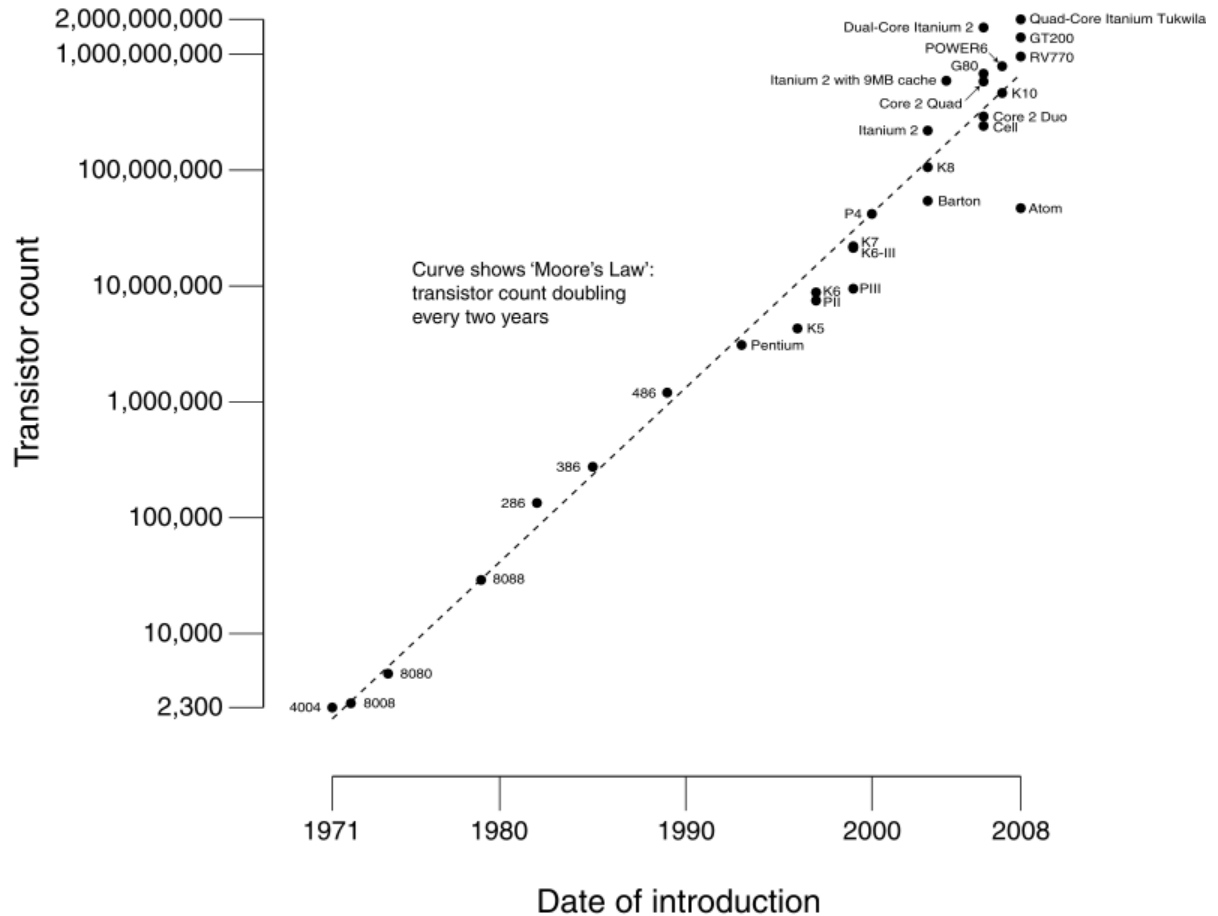


Figure 2.15: Moore's law.

Embedded systems have to be dependable in the sense that some devices are responsible for controlling safety-critical systems such as nuclear power plants, cars, trains and airplanes.

In order to evaluate embedded system efficiency, the following metrics have been adopted:

- *Energy and Runtime efficiency:* Embedded devices deal with restrict amount of resources and in order to increase their battery life time, the energy consumption should be reduced by the adoption of the smallest clock frequencies and supply voltages that respect their time constraints.

- *Code – size* : The code-size should be as small as possible for especially systems on a chip (SoCs), in which the integrated memory should be used very efficiently.
- *Weight* : The low weight is an essential argument for buying mobile devices.
- *Cost* : Low cost is an extremely crucial issue on the market.

Another important characteristic related to embedded system is that their application should be completely dedicated to the device. For an efficient system, an unused memory should not be present. Moreover, the cost to fix a code is very high after the embedded system been in a production phase (see the maintenance cost shown in Figure 1.2). Furthermore, there are two kinds of time constraint. A time constraint is called hard if not meeting that constraint should result in a catastrophe, and all others are soft.

Embedded systems are called hybrid if they include analog and digital parts within it. The analog system adopts continuous signal values in continuous time, and digital ones use discrete signal values in discrete time. This system are said to be reactive systems in sense that it is always waiting for some input.

2.8 SUMMARY

This chapter described the main concepts needed for the understanding of this dissertation. First of all, it was performed an overview of performance evaluation and energy consumption, including models and measurement techniques. After that, the system classification was presented. Next, an overview of Petri nets, a graphic and mathematical modeling tool for modeling and analyzing several kinds of systems (e.g.: parallel and non-deterministic systems), were presented. Additionally, a Petri net extension was presented, namely, Coloured Petri net.

RELATED WORKS

This chapter shows a summary of the relevant related works. First of all, it is performed a general overview of performance evaluation. After, the related works have been divided into three main sections: (i) Hardware simulation-related-works, in which the hardware behavior has been reproduced; (ii) Software simulation-related-works, in which their focus is to simulate the software control flow and its influence to the power and performance; and (iii) Hybrid approach which combine some points related to both hardware and software simulation techniques.

3.1 GENERAL OVERVIEW OF ENERGY CONSUMPTION AND PERFORMANCE EVALUATION

Herzog [Her02] shows the importance of using Formal Methods (FM) to the Performance Evaluation (PE) process. The main goal of such work is to reduce the mutual reservations between both areas, formal specification techniques and performance evaluation. For them, FMs may find their way into a new and very attractive area of applications and some fundamental problems of PE may be overcome. Thus, methodological steps were proposed. Figure 3.1 shows a typical scenario in which the environment generates requests, the so called workload, to the system, where:

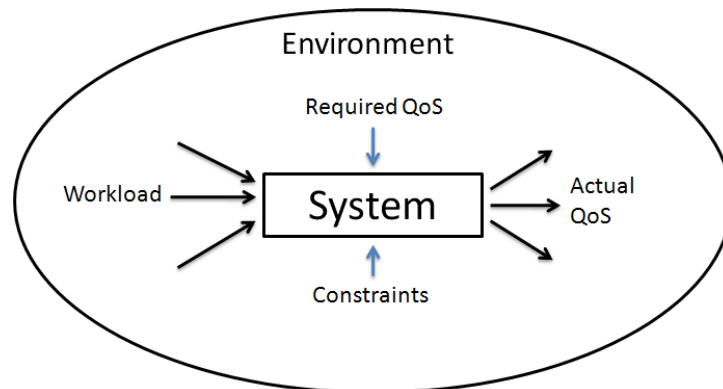


Figure 3.1: The System with its Environment, Requirements and Constraints.

- (i) The workload represents the sum of all needed and desired activities and services.

- (ii) The system consists of one or more components trying to satisfy these requests.
- (iii) The system is considered optimized if the system fulfills all requirements concerning Quality of Service (QoS) as well as all technical and economic constraints.

In addition, such approach presented an overview of performance evaluation methodology and Figure 3.2 depicts its steps. It is important to stress that such methodology can also be applied to consider energy consumption evaluation. The first step shown by the methodology present in Figure 3.2 is to identify the problem and to perform the analysis requirements. In order to identify any problem and/or to perform requirement analysis, some workload characterization and system parameters are needed. After that, two totally different approaches are started, experiments monitoring a real system (measurements) and modeling technique for workload/system behavior studies. Both are followed by analysis steps adopting statistic, stochastic and simulation methods. Thus, the validation is started and, finally, system structures and operating modes are synthesized.

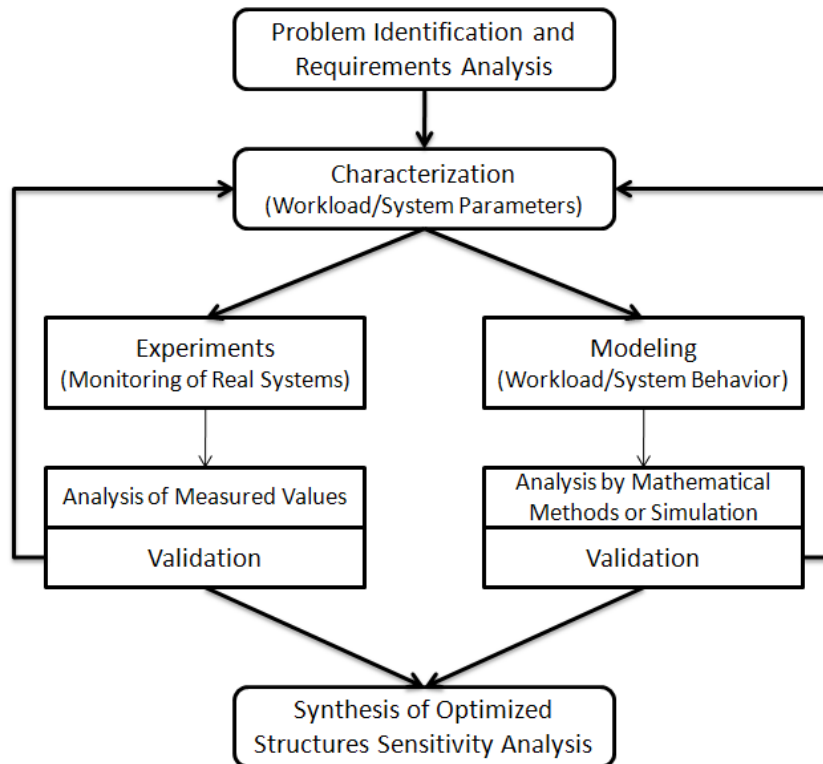


Figure 3.2: Performance Evaluation Methodology.

3.2 HARDWARE SIMULATION-RELATED-WORKS

Hardware components such as CPUs or memories provide structural resources, software components provide pure functionality, and some components, such as I/O controllers,

provide functionality bundled with resources. Thus, in this dissertation, it has been considered as hardware all components related to it (e.g.: busses, I/O controllers).

In order to model the hardware operations through microcontroller descriptions, simulation tools have been developed by some approaches. In these simulation techniques, energy consumption models have been built considering either mathematic models of circuit (a lower level) or higher description level such as Register Transfer Level (RTL). PowerMill [HZDS95] is an example of such transistor level tool that reproduces the current and power behavior in VLSI circuits. In such tool, it is possible to simulate deep-submicron CMOS (Complementary Metal-Oxide-Semiconductor) circuits, including sophisticated circuitries such as exclusive-or gates and sense-amplifiers. Another example is the QuickPower [men08], a simulator that considers the logic abstraction level through circuit simulations.

In addition, another tool named SimplePower [YVKI00][IKV01] was developed to provide the energy consumed in the memory system and on-chip buses using analytical energy models. The PowerTimer toolset [BDB⁺03] is another simulator developed to be used in early-stage microarchitecture-level power-performance analysis of microprocessors. In such approach, energy functions in conjunction with any given cycle-accurate microarchitectural simulator were used. The energy functions model the power consumption of primitive and hierarchically composed building blocks of structures such as pipeline stage latches, queues, buffers and component read/write multiplexers, local clock buffers, register files, and cache array macros. That methodology adopted analytical equations obtained from empirical circuit-level simulation experiments in order to perform the energy consumption estimates. A framework named Wattch [BTM00] was proposed for analyzing and optimizing microprocessor power dissipation at the architecture-level. This approach is considered as a complement to existing lower-level tools, because it allows architects to explore and cull the design space in early design phases.

SimplePower, Wattch and PowerTimer are engines that consider the abstraction architectural level and adopt a RTL model of the desired architectural in order to estimate the power consumption. Even good results have been obtained by the adoption of such techniques, the low abstraction level adopted demanded an enormous computational effort restricting the applicability for real world applications. Thus, such methodologies have been performed in small code applications. Another drawback of the low level approaches is the need of detailed hardware descriptions.

Another work [HH08] adopted a modified version of the Sim-outorder simulator from the SimpleScalar suite [ALE02] [Sim08] in order to investigate some techniques for improving the performance of memory hierarchies for embedded systems. Sim-outorder is an execution driven, cycle-accurate, out-of-order simulator. The adopted methodology considers precise models for the memory hierarchy and for the memory bus, and four different processors with different levels of instruction level parallelism and complexity.

A static performance evaluation methodology was proposed in [RJ03] to support early, architecture-level design space exploration for component-based embedded systems. This approach evaluates the system performance based on a scenario. For this, it focuses on

an interactive definition of evaluation scenarios through incremental refinement of a functional specification to identify control flow paths corresponding to typical case behaviors. Thus, they considered that the energy consumption through instructions does not have a considerable variation. Instead, to them, the energy consumption has a stronger relationship with the code control flow (e.g.: loops) than the specific characteristics of the set of instructions.

Another work [BM08] adopted the SimpleScalar [BA97] architecture simulator in order to extend the cache at the circuit level to allow power and performance trade-offs to be managed. This research divided the power consumption into two components, active power and leakage power. They defined active power as the power consumed by switching parts of the digital circuits, and the leakage power as the power consumed by the transistors when they are off. The variation effects of power supply voltage, threshold voltage and of channel length in the leakage power were performed. Their conclusions were that the channel length impacts more in the leakage power consumption.

3.3 SOFTWARE SIMULATION-RELATED-WORKS

The energy consumption of a microprocessor is directly correspondent to the software in execution. Thus, a lower energy consumption has become an essential challenge for optimizing the embedded system applications. In general, the energy consumption of a software is described considering the instruction set of the processor under study. The instruction can consume energy basically by two mechanisms: (i) during the instruction execution, a sequence of internal processor states is generated and the state transitions results in the hardware energy consumption pattern, named Instruction Base Cost [LFTM97]; (ii) due to the instruction operands, the instruction can perform register changes and memory access that implies in a dynamic energy consumption. Furthermore, some factors can increase its base cost energy consumption through register transitions and memory access. The register numbers, register values, immediate values of the instructions, operand address and the operand values are examples of those factors [NKN02]. Although these factors has some influence, a mean value in the energy consumption can be obtained to each instruction.

Over the last years, many approaches have been developed to deal with the estimation of software execution time and energy consumption in embedded systems. Tiware et al. [TMW94] developed an instruction level simulation mechanism that quantifies the energy cost of individual instruction. This approach divides the code in basic blocks, which define a contiguous section with exactly one entry and exit points. Thus, it is possible to get the energy cost of a program after multiplying each base cost block by the number of times it was executed. The main limitation of this approach is that it will not work for programs with larger execution times since the ammeter may not show a stable reading.

An approach for power-aware code exploration, through an analysis mechanism based on Coloured Petri Net (CPN), is presented in [JNM⁺06]. In that approach, a methodology for stochastic modeling of 8051 based microcontroller instruction set is demonstrated.

The presented method allows setting probabilities on conditional instructions for representing complex application scenarios. The main drawback of that method is the model complexity, and as a direct consequence, a higher runtime evaluation is required. Another drawback was the adoption of a generic engine for the CPN models evaluation. These restrictions does not allow to evaluate a real life complex application or even reasonable size programs and, also, only Assembly codes were considered.

Another approach related to energy consumption estimation is based on functional decomposition [LSJM01]. In that method, the power consumption of each functional block is computed from a set of consumption rules. These rules are represented as mathematical functions obtained from several measurements of different codes and configuration parameter values, which are extracted from the code. Thus, the energy consumption is obtained by adding the consumption of all blocks. This work has been extended [SLJM04], so that a tool to estimate the power and energy consumption related to C programs and assembly code was proposed. This work does not provide means for structural and behavioral property analysis and verification.

Another paper [KCNL08] presents an energy consumption modeling technique for embedded systems based on a microcontroller, in which the number of cycles instead of the number of executed instructions is considered and it computes the energy by a polynomial expression. In order to obtain such expressions, the software tasks that run on the embedded system were profiled, and their characteristics were analyzed. The type of executed assembly instructions, as well as the number of accesses to the memory and the analog-to-digital converter, is the required information for the derivation of the proposed model. An appropriate instrumentation setup was developed for measuring and modeling the energy consumption in the corresponding digital circuits. This work adopted analytical models that may require so many simplifications and assumptions that may turn the results not so accurate.

Muttreja et al. [MRRJ07] presented a methodology to speed up simulation-based software-performance / energy estimation adopting macromodeling. However, this methodology is only applicable to data that follows the same distribution as the data used to train the model. Thus, this restriction reduces the applicability of that methodology.

An adaptation of the instruction-level power estimation model to soft-core processors implemented in FPGAs is presented in [dHAW⁺07]. In order to validate their methodology, the Nios II softcore processor was adopted. In such approach the inter-instruction costs (cost correspondent to the transition from one kind of instruction to another) and pipeline stalls were not modeled directly, instead of that, a correction factor was adopted.

3.4 HYBRID APPROACH

A framework that combines hybrid simulation, cache simulation and online trace-driven replay techniques to accurately predict performance of programmable elements in embedded environments was proposed in [GKK⁺08]. A simulator, called HySim, combines

a target architecture specific ISS (Instruction Set Simulator) execution with native code execution on the simulation host for achieving high simulation speed. For this, a similar methodology is adopted where an entire application is compiled through the target compiler to produce a target specific binary (the input of their framework).

Another work [SBVR08] presents an hybrid method that solves performance issues by combining the advantages of simulation-based and analytical approaches with the objective of gaining simulation runtime speed without remarkable loss of accuracy. The methodology is based on the generation of SystemC code out of the original C code and back-annotation of statically determined cycle information into the generated code. One drawback of that methodology is the difficulty to find corresponding parts of the binary code in the C source code if the compiler optimizes or changes the structure of the binary code too much. Thus, such approach does not work well for some processors and/or compilers.

3.5 SUMMARY

This chapter summarized the main works related with estimating performance and energy consumption of embedded system applications. In several works, tools have been developed in order to model the hardware operations through microcontroller descriptions. Although these techniques can provide accurate results, their computational costs are too high which reduce their applicability in many situations. In contrast to these hardware simulations, approaches that make use of software simulation techniques provide their results in a shorter runtime. Nevertheless, the majority of such approaches adopts some methods (e.g.: analytical models) in which their results are less accurate than the hardware ones. Moreover, the hybrid approach are quite new, and so, their results need more refinements.

METHODOLOGY

This chapter depicts the proposed methodology for building embedded critical software. Next, the proposed framework in order to estimate the energy consumption and performance of embedded system applications is presented. Afterwards, the characterization process that has been adopted to obtain the energy consumption and execution time values of an ARM7-based microcontroller instruction set is demonstrated.

4.1 METHODOLOGY

This section briefly introduces the proposed **Methodology for Embedded Critical Software Construction (MEMBROS)**. Figure 4.1 depicts the core activities of MEMBROS methodology, which organizes the activities in three groups: (i) Requirements Validation; (ii) Energy Consumption and Performance Evaluation; and (iii) Software Synthesis. Although this dissertation focuses is related to the energy consumption and execution time estimates, an overview of the whole methodology is important in order to show that this dissertation subject is engaged with other works.

Initially, the activities regarding requirement validation are performed. After carrying out the requirement analysis, the system requirements are modeled using a set of SysML (Systems Modeling Language) diagrams (SDs)[Sys07] that represents the functionalities of the embedded software to be developed. The SDs provide the designer an intuitive language for modeling the requirements without knowing the details of the internal formalism, which is utilized in further activities for reasoning about quantitative/qualitative properties. Since timing and energy constraints are of utmost importance in the systems of interest, the SDs are annotated with timing and energy consumption information (e.g.: initial estimates) using MARTE (Modeling and Analysis of Real-Time and Embedded Systems) [MAR07]. Next, the annotated SDs are automatically mapped into time Petri net (TPN) models in order to lay down a mathematical basis for analysis and verification of properties (e.g.: absence of deadlock conditions between requirements). This activity also concerns to obtain best and worst-case execution times and the respective energy consumptions, in such a way that the requirements are also evaluated whether timing and energy constraints can be met. More details in [CMC⁺08].

Afterwards, the embedded software development process is started taking into account the results obtained in previous activities. Once a first source code release is available, the designer analyzes the code in order to assign probability values to conditional and iterative

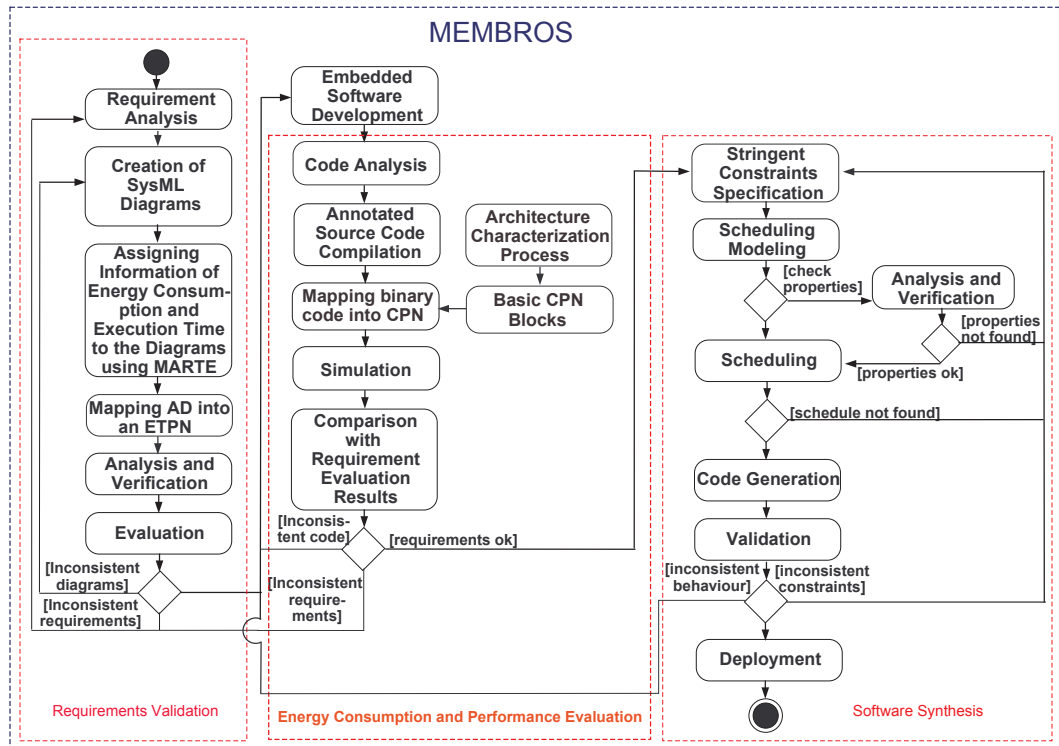


Figure 4.1: Methodology activity diagram.

structures. It is important to state that with such values it is possible to simulate different software scenarios just changing the probability instruction annotations. Furthermore, the compiled code with probability annotations allows designers to evaluate in the context of time and energy consumption, in such a way that these costs may be estimated before the whole system (hardware prototype) is available. For that, the compiled code is automatically translated into a Coloured Petri net (CPN) model [JKW07] in order to provide a basis for the stochastic simulation of the embedded software. An architecture characterization activity is also considered to permit the construction of a library of basic CPN blocks [CMA⁺08b], which provides the foundation for the automatic generation of CPN stochastic models (see Section 4.2). From the CPN model (generated by the composition of basic blocks), a stochastic simulation of the compiled code is carried out considering the characteristics of the target platform. If the simulation results are in agreement with the requirements, the software synthesis is performed.

Software synthesis activities are concerned with the stringent constraints (e.g.: time and energy), and, in the general sense, it is composed of two subgroups of activities: (i) tasks' handling; and (ii) code generation. Tasks' handling is responsible for tasks' scheduling, resource management, and inter-task communication, whereas code generation deals with the static generation of the final source code, which includes a customized runtime support, namely, dispatcher. It is important to state that the concept of task is similar to *process*, in the sense that it is a concurrent unit activated during system

runtime. For the following activities, it is assumed that the embedded software has been implemented as a set of concurrent hard real-time tasks.

Initially, the task timing information as well as the information regarding the hardware energy consumption are computed through the energy consumption and performance evaluation activities. Next, the designer defines the specification of the system stringent constraints, which consists of a set of concurrent tasks with their respective constraints, behavioral descriptions, information related to the hardware platform (e.g.: voltage/frequency and energy consumption) as well as the energy constraint. Afterwards, the specification is translated into an internal model able to represent concurrent activities, timing information, inter-task relations, such as precedence and mutual exclusion, as well as energy constraints. The adopted internal model is a time Petri net extension (TPNE), labeled with energy consumption values and code annotations. After generating the internal model (TPNE), the designer may firstly choose to perform property analysis/verification or carry out the scheduling activity. This work adopts a pre-runtime scheduling approach in order to find out a feasible schedule that satisfies timing, inter-task and energy constraints. Next, the feasible schedule is adopted as an input to the automatic code generation mechanism, such that a tailored code is obtained with the respective runtime control, namely, dispatcher. Finally, the application is validated on a Dynamic Voltage Scaling (DVS) platform in order to check the system behavior as well as the respective constraints. Once the system is validated, it can be deployed to the real environment. The basic goal of DVS is to adjust the processor' operating voltage at run-time to the minimum level in order to reduce the energy consumption considering the application time constraints.

4.2 ENERGY CONSUMPTION AND PERFORMANCE EVALUATION FRAMEWORK

The model evaluation concerns to execution time and energy consumption estimates. This process aims to help designers to identify the application blocks that need to be optimized and, moreover, it helps them to decide which code parts should be transformed into hardware components.

The proposed framework takes into account Assembly and C codes (see Figure 4.2) labeled with probabilities assigned to conditional instructions in order to specify the system scenarios as well as parameters for the stop criteria. The assembly or C codes are provided as an input to the assembler or compiler that generates two outputs: the Binary Code (machine code) and the Listing file (file in which the probabilities and the stop criteria parameters are captured). The listing file is an output file of compilers and it is adopted to help designers in the debug process (e.g.: to identify the compilation issues). After that, the Binary-CPN Compiler reads these two files (generated by the Assembler or by the Compiler) and, also, the Basic CPN Models, and generates two CPN Models to

be analyzed. The CPN-Optimized is the model used to estimate the energy consumption and execution time and, the other model is adopted to validate the optimized one. These CPN models are represented by the basic models and can be read by CPN Tools and/or by the CPN Simulator in order to generate the estimate results.

The CPN Simulator is a tool that evaluates the proposed CPN models in order to compute the energy consumption and execution time. This tool has been conceived as an alternative to CPN Tools, since CPN Tools' simulation mechanism is quite time consuming when analyzing large models because it is a general purpose evaluation environment. Moreover, an automatic CPN Generator receives the processor characterization tables in order to create the Instruction-CPN Models for the ARM7 (and others) processors. The execution times and the energy consumption values could be obtained from datasheets, characterization processes, measuring and so on.

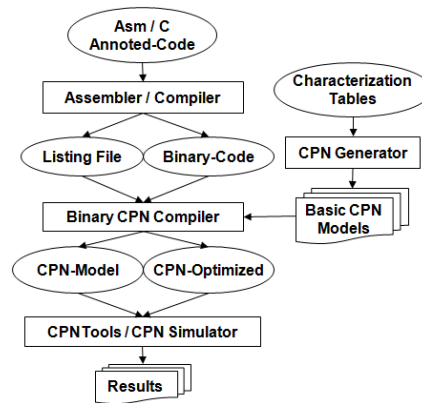


Figure 4.2: The proposed Framework.

4.2.1 Characterization Process

In order to obtain the energy consumption and execution time values of a microcontroller instruction set, it may be necessary to adopt some measurement techniques in case such values cannot be obtained from manuals and datasheets. It is essential to present processor particularities to perform a characterization process. Hence, this section initiates by introducing some points related to the adopted Philips LPC2106 microprocessor. Afterwards, the characterization scheme is detailed.

An ARM7-based microcontroller, Philips LPC2106 [man03], has been adopted due to its widespread use in embedded systems as the hardware platform for conducting the case study validation. LPC2106 is a 32-bit microprocessor based on Reduced Instruction Set Computer (RISC) principles. Furthermore, its instruction set and its related decode mechanism are much simpler than those of microprogrammed based on Complex Instruction Set Computers (CISC). As a consequence, high instruction throughput and

impressive real-time interrupt response from a small and cost-effective processor core have been performed.

Another important characteristic that should be mentioned about LPC2106 microprocessor is the absence of internal cache memory [Fur00]. Instead, a Memory Accelerator Module (MAM) is available. The MAM corresponds to a technique, adopted by LPC2100 family microprocessors, that attempts to have the next ARM instruction in its local memory in time for the CPU to execute. Although this mechanism improves the microprocessor performance, it has not been considered throughout the scope of this work. The proposed methodology has considered a general approach to evaluate the ARM7-based microprocessors. Thus, the characterization process was performed with the MAM mechanism turned off. However, it is possible to cover this particularity by extending the proposed methodology in order to perform an evaluation before each instruction be executed to reproduce the MAM technique approach.

Additionally, it is important to state that the energy consumption depends on the instruction parameters (register values). In other words, if the same instruction is executed with different parameters, the energy consumption and execution time may have small differences. Tiwari et al [TMW94] showed that good estimates can be obtained without considering such issue, in which their experiment results demonstrated that the range of those variation corresponds to less than 5%.

Figure 4.3 depicts a code example adopted to characterize the LPC2106 instruction set, in which an oscilloscope synchronization marker has been adopted to the code analysis. Ten thousands of instruction replications have been performed, as Figure 4.3 depicts on lines 8 to 12, in order to have a precise characterization of an instruction. Furthermore, Figure 4.4 depicts such code on the oscilloscope (Agilent DSO03202A), in which the signal voltages can be viewed as a two-dimensional graph. The wave shape of the electrical signal shown in Figure 4.4 represents the execution of the code in analysis.

```

1
2 while(1){
3   int i;
4   IOSET = IOPIN | 0X00000080; /* oscilloscope synchronization marker */
5
6   /* code */
7   __asm{
8     mov r1, #0
9     mov r1, #0
10    ... (representing 9.996 mov r1, #0)
11    mov r1, #0
12    mov r1, #0
13  }
14  /* end of code */
15
16  IOCLR = (~IOPIN) | 0X00000080 /* oscilloscope synchronization marker */
17  for (i=0; i<5300; i++); /* delay */
18 }
```

Figure 4.3: A code example for measuring.

Characterization Scheme

A Characterization Scheme has been adopted in order to characterize such proces-

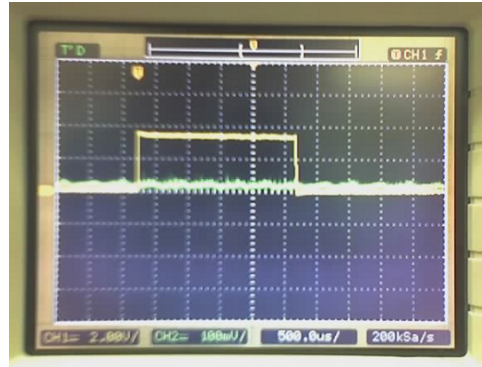


Figure 4.4: The measurement performed on oscilloscope.



Figure 4.5: Hardware Platform with LPC2106 microprocessor.

processor, in which a hardware platform with the LPC2106 microcontroller (Figure 4.5), an oscilloscope and a desktop computer (PC) are connected as shown in Figure 4.6. Thus, the PC runs AMALGHMA (Advanced Measurement Algorithms for Hardware Architectures) [CTM08], a tool (Figure 4.7) developed to automate the measuring activities that captures the information displayed by the oscilloscope. This characterization process is similar to the ones adopted by [TMW94] and [RJ98]. Hence, the oscilloscope captures the CPU drained current by measuring the voltage drop across a sense resistor, and the AMALGHMA acquires the energy consumption and execution time through the oscilloscope. For this, the oscilloscope is connected to an I/O port pin of the target CPU, and through it the code start and end times are indicated.

AMALGHMA adopts a set of statistical methods such as bootstrap [ET93] and parametric methods [Chu04] in order to cope with oscilloscope resolution and resistor error. The results obtained through AMALGHMA to the LPC2106 microcontroller have been validated considering LPC2106 datasheet as well as ARM7TDMI reference manual.

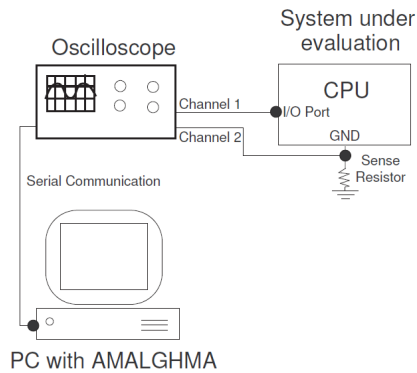


Figure 4.6: The measurement process.

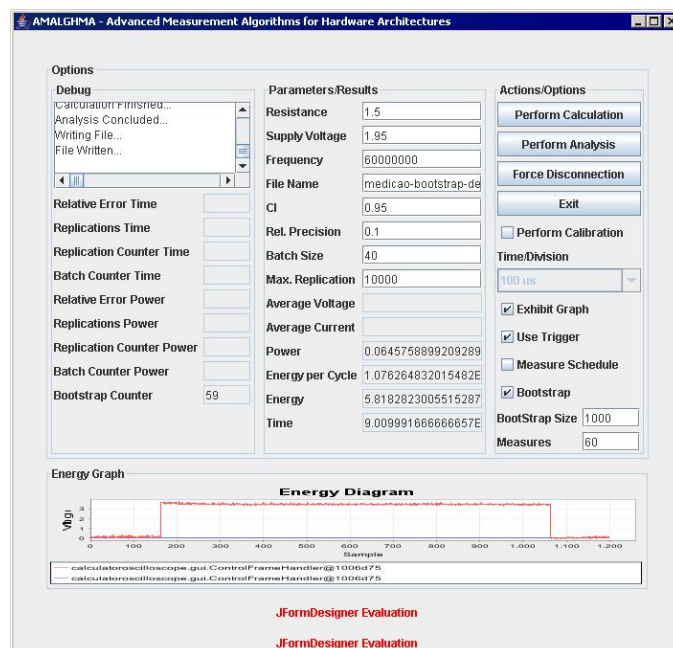


Figure 4.7: The AMALGHMA engine.

4.3 SUMMARY

This chapter described that the proposed methodology of this dissertation for estimating energy consumption and execution time of embedded systems corresponds to one part of the MEMBROS methodology. The MEMBROS methodology was divided into three groups: (i) Requirements Validation; (ii) Energy Consumption and Performance Evaluation; and (iii) Software Synthesis. The first group focuses on the activities regarding requirement validation. The second one consists of this dissertation focuses and its activities are related to the energy consumption and performance evaluation of embedded softwares in early phases of the design. The Software Synthesis optimize the embedded software by improving the schedule routine and by adopting a technique called DVS in order to create the embedded code that can be deployed to the real environment.

This chapter presents the proposed Coloured Petri net models, discrete event models, that represent the ARM instruction set for estimating energy consumption and execution time of embedded software. Afterwards, it depicts the adopted reduction rules in order to simplify the proposed CPN models.

5.1 EMBEDDED SOFTWARE MODELING

The embedded software models should be as simple as possible in order to consider only the important characteristics that affect the energy consumption and performance of such applications. Thus, this work proposes a model representation that does not consider, for example, register values. As a directly consequence, the model has been considering just the code control flow instead of all hardware operations (it does not reproduce all the compiler operations) in order to improve its runtime simulation.

The proposed approach considers each processor instruction like a transition, in which values related to energy consumption and performance are labeled. Furthermore, the adopted building process considers the code control flow, and so, a software structure is obtained as a result, in which loops and procedures are also represented. Places represent the control flow states, a token in a CPN place represents the current state, and a transition firing means an instruction (or a block of instructions) execution.

CPN models allow modeling embedded applications at different abstraction levels. The proposed model is represented by a two-level CPN model. At the lower level, the basic CPN models are present; the higher level concerns the composition of such basic models.

The proposed methodology [CMA⁺08c], by this dissertation, is based on the instruction set of the microcontroller instead of considering the processor architecture as the proposed CPN processor models presented in [BKY98] and [BKY00]. The main drawback of these works is the detailed hardware description that is not usually available for designers. The proposed method considers only the energy consumption and execution time related to the processor instruction set. Therefore, if this information is not available in datasheets, the designers can perform some measurement techniques in order to obtain such values.

Figure 5.1 overviews the proposed methodology. The machine code is transformed into a CPN model through the adopted compiler. Afterwards, the generated model is provided as input to the simulator. Besides the model, the scenario of interest has to be specified in order to start the evaluation process. As a result of the evaluation process, the energy consumption and performance estimates are provided.

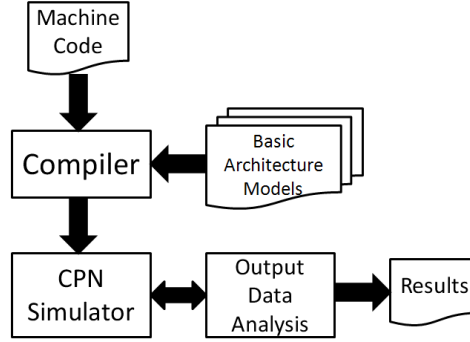


Figure 5.1: Methodology task-flow.

Seven basic CPN instruction models [CMA⁺08a] [CMA⁺08b] have been conceived to characterize the ARM7 processor instruction set. Each basic model considers the values obtained through the measurement process described in the Section 4.2.1, and its evaluation computes the energy consumption and execution time values. These individual nets are represented in a XML format compatible with CPN Tools [cpn07].

In addition, it is important to stress that the evaluation of the proposed basic CPN models consider the global variable *globalTime* that computes the executed time. Thus, instead of changing a token's time stamp, the value of a global variable has been incremented. Although this concept is slightly different from the formal definition of TCPN (see Definition 2.6.2), it has been adopted in order to improve the CPN Tools simulation runtime.

Definition 5.1.1. (Inst) *Inst* is the whole set of instructions of the target microcontroller.

Definition 5.1.2. (InstName) *InstName* is the instruction name of each microcontroller instruction.

5.1.1 CPN model for ordinary instructions

Ordinary instructions are those that do not change their control flow execution. Figure 5.2 depicts the CPN model for ordinary instructions. This model is composed by one input place (*S₋Input*), one output place (*S₋Output*) and one transition (*InstName_i*) named by the instruction name that it represents. Moreover, the *Context* is the colour (type) of the places and *Value* represents their initial markings. The formal definition of the CPN model for these instructions is presented as follows.

Definition 5.1.3. (CPN model for ordinary instructions) Let $OrdInst \in Inst$. The Coloured Petri net described by a tuple $CPNMOrd = (\sum, P, T, A, N, C, G, E, I)$, defines the ordinary instruction model of *Inst*, where:

- (i) $\sum = \{Context\}$,
- (ii) $P = \{S_Input, S_Output\}$,

- (iii) $T = \{InsName_i \mid FCod(InsName_i) = CodInst_{name}\},$
- (iv) $A = \{a_0, a_1\},$
- (v) $N(a_0) = (S_Input, InsName)$ and $N(a_1) = (InsName, S_Output),$
- (vi) $\forall P_k \in P, C(P_k) = Context,$
- (vii) $G(InsName_i) = true,$
- (viii) $E(a_0)=K_i, E(a_1)=K_o \mid Type(K_i) = Type(K_o) = Context,$
- (ix) $I(S_Input) = I(S_Output) = \text{not defined}.$

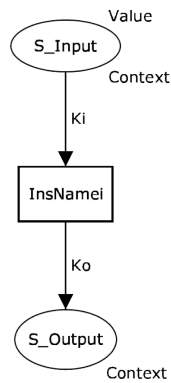


Figure 5.2: Ordinary model.

Figure 5.3 depicts an example of ordinary instruction model. This model assigns the energy consumption (*val energy*) and execution time (*val cy*) through the function *addData(energy,cy)* as shown on Figure 5.3 (see the code segment inscriptions). This model has two ports, one input type and one output type port.

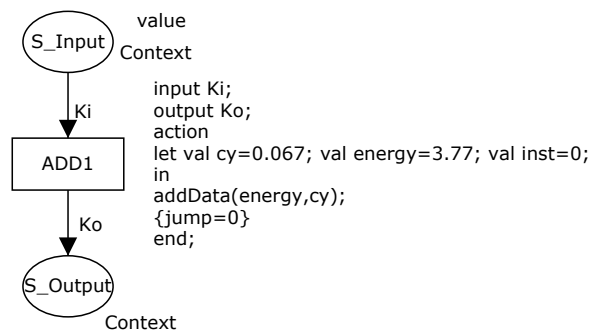


Figure 5.3: An example of ordinary model.

5.1.2 CPN model for conditional instructions

ARM7 instructions have a specific characteristic in the sense that all instructions may or may not be conditional. Conditional instructions are those that evaluate the micro-controller's flags and whether the instruction is executed or not. If the instruction is not executed, the instruction operation is not considered and the control flow continues on to the next instruction. In this work, random variates are generated according to uniform distribution within the interval $[0,1]$ for representing conditional evaluation.

Figure 5.4 presents the conditional model. Definition 5.1.3 also represents this model because of their differences are in the code segment inscriptions, where variables such as *prob* and *aval* are present. *Prob* represents the probability of conditional instructions to be executed and *aval* has been conceived to evaluate this probability. In order to obtain the *aval* value, a conceived function ($uniform(0.0, 1.0) \leq prob$) is evaluated as true if the respective condition is satisfied, and as false otherwise. It is important to state that this model computes different energy consumption and execution time values depending on the *Aval* results.

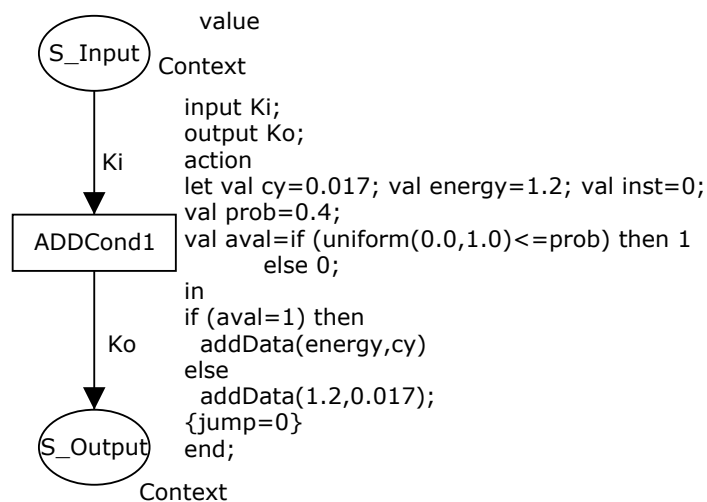


Figure 5.4: Conditional model.

5.1.3 Procedure calls model

The model defined for representing Procedure Calls is also represented by Definition 5.1.3. Figure 5.5 shows an example of this model by representing the branch and link operation. This model, besides using the function *addData* (already depicted in Section 5.1.1), considers the function *push*. The function *push* stores (pushes) the current address position into a stack, which is adopted by return from procedure calls model.

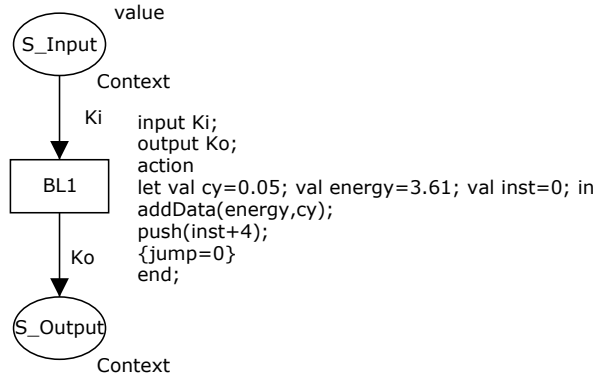


Figure 5.5: Branch and link model

5.1.4 CPN model for conditional branch instructions

The conditional model is shown in Figure 5.6, in which a branch is executed if the respective condition is satisfied. This model is composed by one input place (S_Input), one intermediate state ($Inter_State$) and two output places ($S_Output1$ and $S_Output2$). Each output place corresponds to a different code control flow execution.

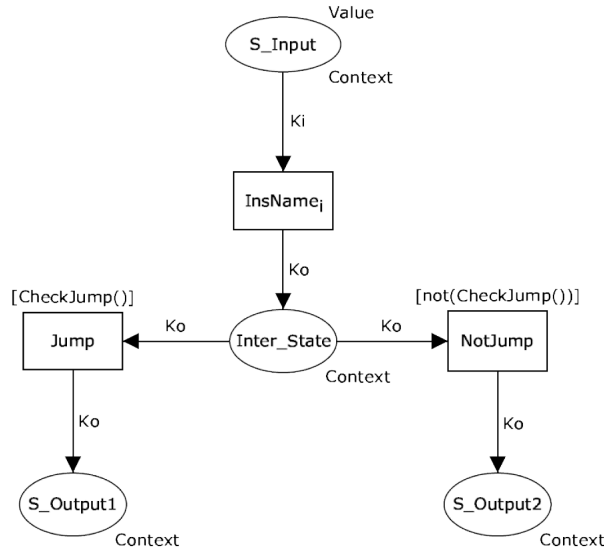


Figure 5.6: Conditional branch model.

Definition 5.1.4. (CPN model for conditional branch) Let $CondInst \in Inst$. The Coloured Petri net $CPNMCond = (\Sigma, P, T, A, N, C, G, E, I)$ defines the conditional instruction model of $Inst$, where:

- (i) $\Sigma = \{Context\}$,
- (ii) $P = \{S_Input, Inter_State, S_Output1, S_Output2\}$.

- (iii) $T = \{Jump, NotJump, InsName_i\}$,
- (iv) $A = \{a_k \mid 0 \leq k \leq 5\}$,
- (v) $N(a_0) = (S_Input, InsName_i)$, $N(a_1) = (InsName_i, Inter_State)$,
 $N(a_2) = (Inter_State, Jump)$, $N(a_3) = (Inter_State, NotJump)$,
 $N(a_4) = (Jump, S_Output1)$, $N(a_5) = (NotJump, S_Output2)$,
- (vi) $\forall P_k \in P, C(P_k) = Context$,
- (vii) $G(InsName) = true$, $G(Jump) = CheckJump()$, $G(NotJump) = not(CheckJump)$,
 where $CheckJump()$ is an operation that results on $true$ or $false$.
- (viii) $E(a_0) = K_i$; $\forall a_j, a_j \neq a_0 \in A, E(a_j) = K_o$, where $Type(K_i) = Type(K_o) = Context$,
- (ix) $I(S_Input) = I(Inter_State) = I(S_Output1) = I(S_Output2) = \text{not defined}$.

The conditional statement is stochastically represented through the transitions *Jump* and *NotJump* and their respective guard expressions that define which transition should be fired (see Figure 5.7). *Jump* transition is fireable if the token value is equal to “1” (one). *Notjump* transition can be fired if the token carries the value “0” (zero). The reader should observe that these transitions are connected to two different output ports that represent distinct program execution flows. It is also important to remark that *prob* represents the probability of making a branch, and *aval* is considered for evaluating the possibility of branching where its value determines which transition (*Jump* or *NotJump*) should occur.

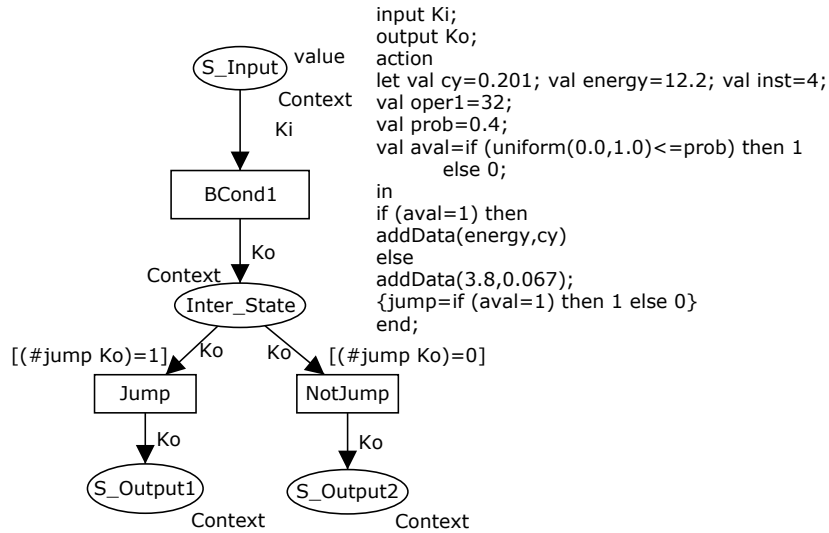


Figure 5.7: Conditional branch model example

5.1.5 CPN model for branching exchange instructions

Another basic model has been considered to represent the BX (branch exchange) instruction. In this model, the number of transitions, places, arcs and output ports depend on the application code that is analyzed. This model can be adopted, for example, to perform procedure return (when it just performs the branch). Thus, if a procedure was called more than once, then its model is connected to different places in order to reproduce their different control flow. The definition of this CPN model is presented as follows.

Definition 5.1.5. (CPN model for branching exchange instructions) Let $RetInst \in Inst$. The Coloured Petri net that represents branching exchange instruction is defined by the tuple $CPNMRet = (\Sigma, P, T, A, N, C, G, E, I)$, where:

- (i) $\Sigma = \{Context\}$,
- (ii) $P = \{S_Input, Inter_State\} \cup P_o$, $P_o = \{S_output_1, S_output_2, \dots, S_output_n\}$, $n \in \mathbb{N}$, where n represents the different control flow executions.
- (iii) $T = \{InsName_i\} \cup T_i$, $T_i = \{jump_1, jump_2, \dots, jump_n\}$, $n \in \mathbb{N}$, where n represents the different control flow executions,
- (iv) $A = \{a_k \mid 0 \leq k \leq m, m \in \mathbb{N}, \text{ where } m = (n \times 2) + 2\}$,
- (v) $N(a_0) = (S_Input, InsName_i)$, $N(a_1) = (InsName_i, Inter_State)$,
 $N(a_2) = (Inter_State, jump_1)$, ..., $N(a_2) = (Inter_State, jump_n)$,
 $N(a_3) = (jump_1, S_output_1)$, ..., $N(a_4) = (jump_n, S_output_n)$,
- (vi) $\forall P_k \in P, C(P_k) = Context$,
- (vii) $G(InsName_i) = true$,
 $G(jump_1) = [(\#jumpKo)=w_1]$, ...,
 $G(jump_n) = [(\#jumpKo)=w_2]$, $w \in \mathbb{N}$, where w represents different memory address.
- (viii) $E(a_0)=K_i$; $\forall a_j, a_j \neq a_0, a_j \in A, E(a_j) = K_o$, where $Type(K_i) = Type(K_o) = Context$,
- (ix) $I(S_Input) = I(Inter_State) = I(S_output_1) = \dots = I(S_output_n) = \text{not defined}$.

Figure 5.8 depicts the CPN model for branching exchange instruction. This model is composed by one input place (S_Input), one intermediate state ($Inter_State$) and n output places, where n represents the different control flow executions. This model is also composed by $InsName_i$ transition, and n $Jump$ transitions. Moreover, the $Context$ is the type of the places and $Value$ represents their initial markings.

Figure 5.9 presents an example of such model, in which there are three ports where one is an input type port and all the others are output type ports. This structure has been adopted, since this model represents a subnet of a hierarchical net that is linked to different code control flow (places). It is important to state that only one of these

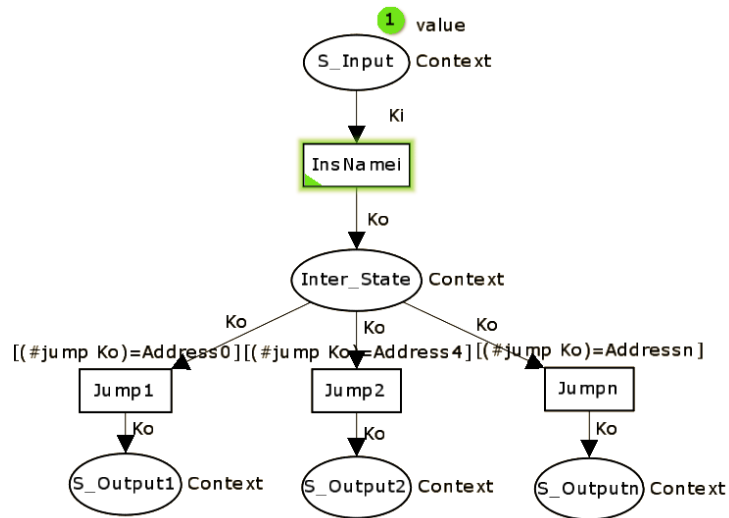


Figure 5.8: Branching exchange instruction model.

transitions (*Jump1* or *Jump2*) is able to “fire” at a time (the input port stores at most one token and then the firing of one these transitions disables the other). Guard expressions are assigned to those transitions, so that *Jump1* transition firing represents a branch to the address “32” and *Jump2* to “0”. The desired return address is provided by the variable *pos*.

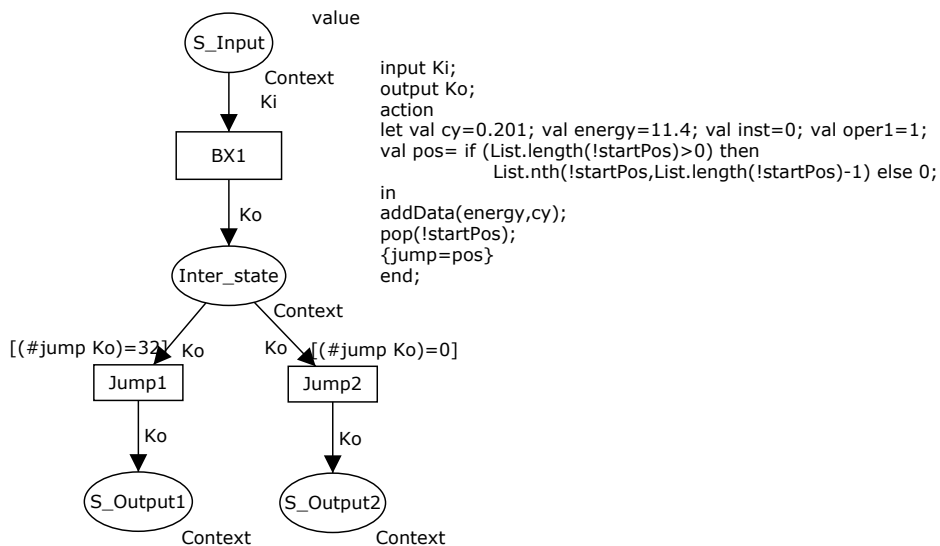


Figure 5.9: Bx instruction example

That last basic model has also been adopted to represent the final instruction of the code in analysis. In this context, during the net evaluation, “0” is assigned to the variable *pos* meaning that this model is connected to the one that represents the stop criteria evaluation process (see Section 5.1.8).

5.1.6 CPN model for Store multiple instruction

In a program when a procedure call is performed, it should be necessary to store the current state of the microprocessor, and as soon as the routine is ended that state will be loaded. For this, there are ARM7 instructions that have a specific characteristic in the sense that just one instruction may load or store different amount of values. This can be done by the instructions LDM (Load Multiple) or STM (Store Multiple). Such store instruction is able to store a non-empty subset of the general-purpose registers to sequential memory locations. Definition 5.1.3 also represents this model.

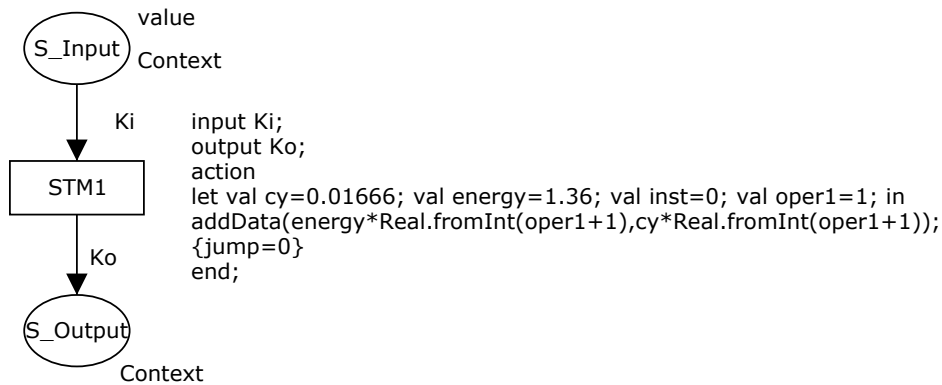


Figure 5.10: Store multiple model.

Figure 5.10 shows the CPN model for Store multiple instruction. Such model has to consider the number of registers that has been adopted to record the data information. The variable *oper1* informs the number of register considered. Thus, Figure 5.10 shows that the energy consumption and execution time values depend on the number of registers that has been handled.

5.1.7 CPN model for Load multiple instruction

The load multiple instruction, already introduced (see Section 5.1.6), is useful for block loads, stack operations and procedure exit sequences. This model computes the energy consumption and execution time in a similar way of the CPN model for STM instruction, but in case *R15 (PC)* register value is changed, hence the energy consumption and execution time values are different. In this case, the CPN model depicted by Figure 5.11 represents branches and it is similar to the CPN model for branching exchange instruction (Section 5.1.5). The differences concerns only in the code segment inscriptions.

5.1.8 CPN model for the stop criteria process

The steady-state simulation stop criteria takes into account the specified error and a confidence degree. Thus, the designer should define the confidence degree and maximal error related to each metrics (energy and time). More details in Section 6.4.4.

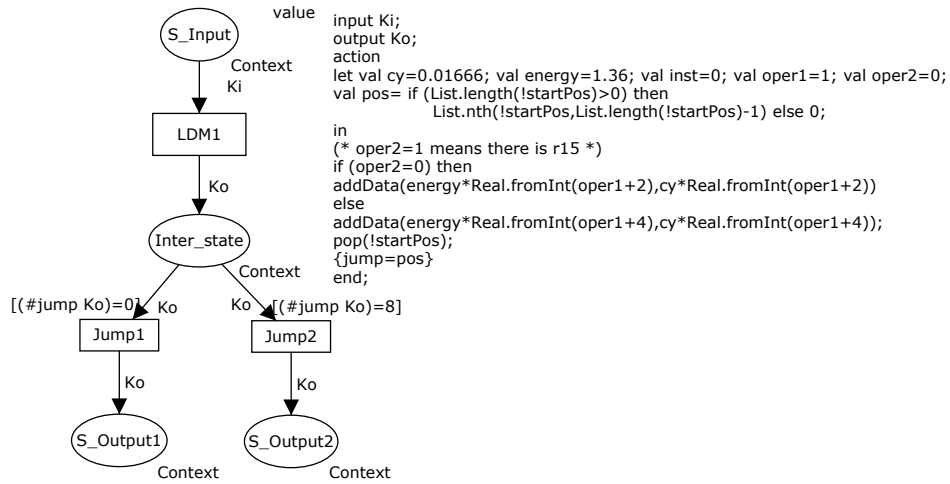


Figure 5.11: Load multiple model.

Figure 5.12 depicts the adopted CPN model for the stop criteria process, in which the energy consumption and execution time of a complete run of the CPN model are computed by the function *addIterData()*. Furthermore, the function *checkEnd()* evaluates whether the simulation may or may not continue. In case the simulation is continued, another function *newIter()* is performed to begin a new turn with the adopted statistic variables reconfigured. Otherwise, the simulation is finished and the mean time, time standard deviation, time error, mean energy, energy standard deviation and energy error are recorded into a file by the function *writeFile()*.

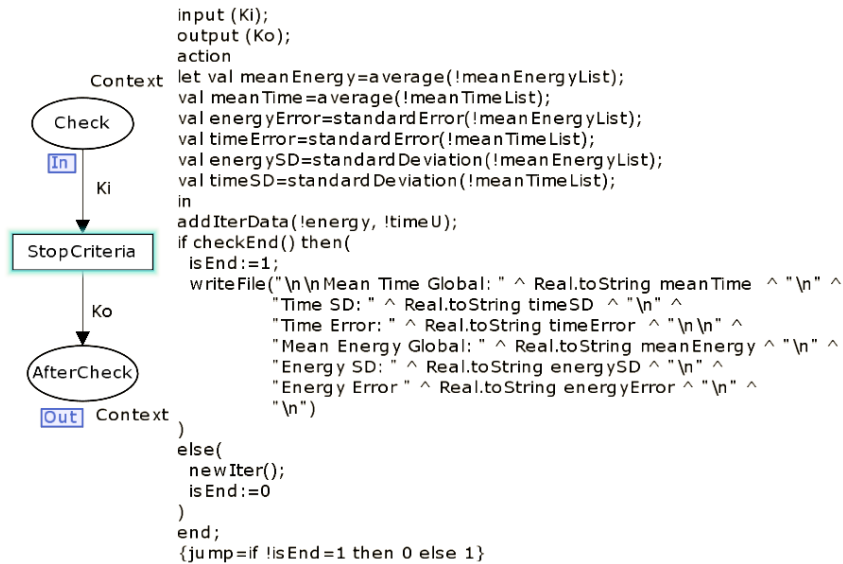


Figure 5.12: CPN model for the stop criteria process

5.2 REDUCTION RULES

In this dissertation, the CPN reduction rules have been adopted in order to transform a CPN model into an equivalent simplified model, in which all important characteristics for estimating the energy consumption and execution time are preserved. The adopted process has three basic rules: (i) any place to be reduced cannot have more than one input arc; (ii) candidate transitions for merging cannot have more than one output arc; (iii) candidate transitions to be reduced cannot have functions assigned to them (such as *push()* and *pop()*), in other words, candidate transitions cannot represent a procedure call model or return of procedure calls. Thus, if all these rules are satisfied, these elements will be clustered.

Figure 5.13 (a) depicts a CPN model, and Figure 5.13 (b) represents a snapshot of the CPN model after being provided the reduction process. It is important to state that the energy consumption and execution time values associated to those three transitions (*Inst_0_MOV1R1, #1*, *Inst_1_MOV1R1, #1*, *Inst_2_MOV1R1, #1*) were summarized (clustered) into just one transition (*BLOCK ID7209*). Thus, when the transition *BLOCK ID7209* occurs, the energy consumption and execution time related to those three transitions are computed.

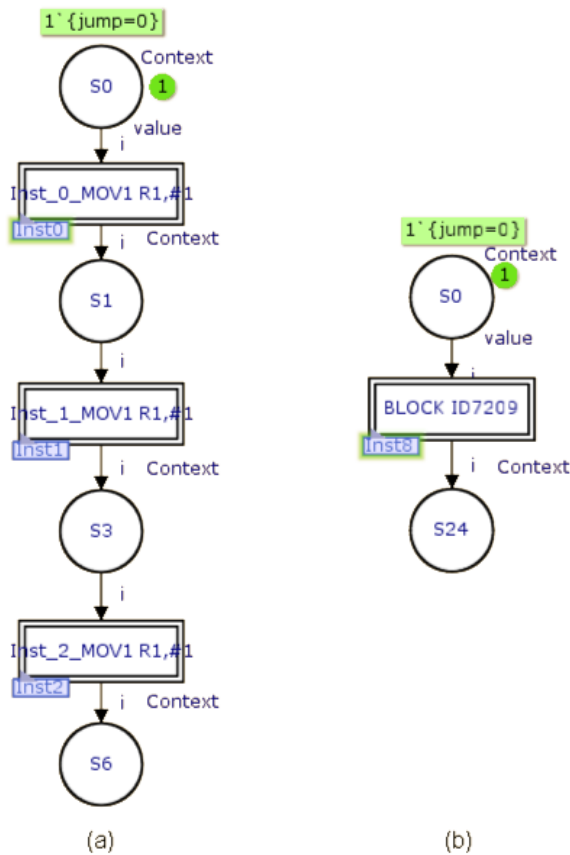


Figure 5.13: (a)CPN model, (b)CPN model after reduction process.

5.3 SUMMARY

This chapter detailed each basic Coloured Petri net model that has been conceived to characterize the ARM7 processor instruction set. The basic models have been divided into seven groups: (i) CPN model for ordinary instructions that consist of those instructions that do not change their control flow execution; (ii) CPN model for conditional instructions, in which a stochastic method has been adopted to evaluate whether the instruction is executed or not; (iii) Procedure calls model, adopted to represent procedure calls; (iv) CPN model for conditional branch instructions, in which a stochastic method evaluates whether a branch is executed or not; (v) CPN model for branching exchange instruction, adopted to consider the BX (branch exchange) instruction; (vi) CPN model for store multiple instruction, conceived to represent STM (Store Multiple) instruction; (vii) CPN model for load multiple instruction, conceived to represent LDM (Load Multiple) instruction. Furthermore, this chapter also provided the CPN model for evaluating the stop criteria process. Lastly, this chapter showed the adopted reduction rules conceived to transform a CPN model into an equivalent simplified model, in which all important characteristics for estimating the energy consumption and execution time are preserved.

CHAPTER 6

THE SIMULATION ENVIRONMENT

This chapter presents the proposed simulation environment, named, ALUPAS, for providing, in early design phases, a mechanism that assists design decisions on energy consumption and performance concerning embedded applications. First of all, an overview of ALUPAS is presented, and, the adopted simulation process is detailed.

6.1 ALUPAS

A Unified environment for Low Power Application Simulation (ALUPAS) [CM08] has been developed to combine the proposed framework functionalities (Section 4.2) into an integrated environment in order to cope with the complexities related to non-functional requirements (e.g.: energy consumption and execution time estimates). This tool provides a graphic interface, in which the designer does not need to interact directly with the internal formalism (CPN).

The ALUPAS is composed of an Assembler, C Compiler, a Binary CPN Compiler, a CPN Simulator and a Graphical User Interface (GUI). For a better understanding, the following sections describe each component.

6.2 ASSEMBLER AND C COMPILER

An assembly language is a low-level language for programming computers defined by the hardware manufacturer. Thus, this language implements a symbolic representation of the numeric machine codes and other constants needed to program a particular CPU architecture. Assembler is a software component that translates programs implemented in assembly (ASM) language into the machine language (binary code) of the target platform. In order to perform this task, the assembler creates an object code by translating assembly instruction mnemonics (instruction abbreviations) into opcodes (operation codes) as well as resolving symbolic names for memory locations. In contrast to Assembler, a compiler translates a high-level language, such as C, into assembly language first and then into machine language.

This work adopts the Keil's ARM Assembler and Keil's C Compiler [kei08] in order to generate the binary code. The Assembler and C compiler options were set up to also generate the listing (LST) file (see Figure 6.1) from which probabilities and the stop criteria parameters are captured.

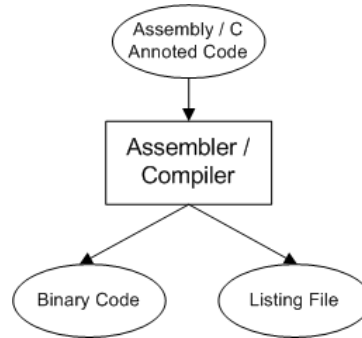


Figure 6.1: Assembler structure.

Annotations

The code annotations have been conceived in order to set up stop criteria parameters and probability values of the conditional statements. The stop criteria annotations have to be set on the first code line following the specification below.

; <confidence interval| desired energy error| desired time error| batch size| max number of replication>

The reader should note that stop criteria parameters deal with confidence interval, desired energy consumption and time errors, batch size and maximum number of replications on the simulation. See Section 6.4.4 for more information about the adopted stop criteria process. The designer should put annotations to conditional instructions as the following specification depicts.

conditional statements ; <@probabilistic value@>

The designer may use the Equation 6.1 in order to obtain the probabilistic value of conditional statements such as a loop.

$$p = 1 - (1/N) \quad (6.1)$$

where: $N \in \mathbb{N}$, and represents the sample number.

6.3 BINARY CPN COMPILER

This dissertation proposes a Binary-CPN compiler which automatically translates an embedded software, implemented in C language or ASM, into a CPN model. This compiler reads the two files (machine code and listing file) and, also, the Basic CPN Models, in order to generate a CPN Model to be analyzed. Figure 6.2 depicts the binary-CPN compiler basic structure.

Figure 6.3 shows the binary-CPN compiler structure in more details. The parser processes the compiled code in order to generate an internal representation of each code instruction. Next, a XML parser interprets the listing file in order to create a XML file

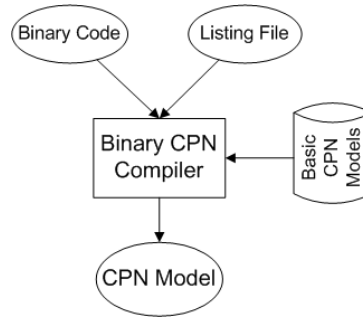


Figure 6.2: Basic Binary CPN Compiler structure.

(prob-XML) containing the probabilistic values of each conditional instruction. Afterwards, a Model Factory creates the final formal model (CPN) and the Sim file through the basic-CPN models and prob-XML file. This Sim file consists of a XML representing the formal model, and it is adopted as an input to the CPN Simulator. Finally, a reduction process is performed through some rules to transform the evaluation models (CPN model and Sim File) into equivalent ones. It is important to state that all characteristics regarding to energy consumption and execution time are preserved. An overview of the proposed implementation is present in Appendix C, in which the binary-CPN compiler class diagram is detailed.

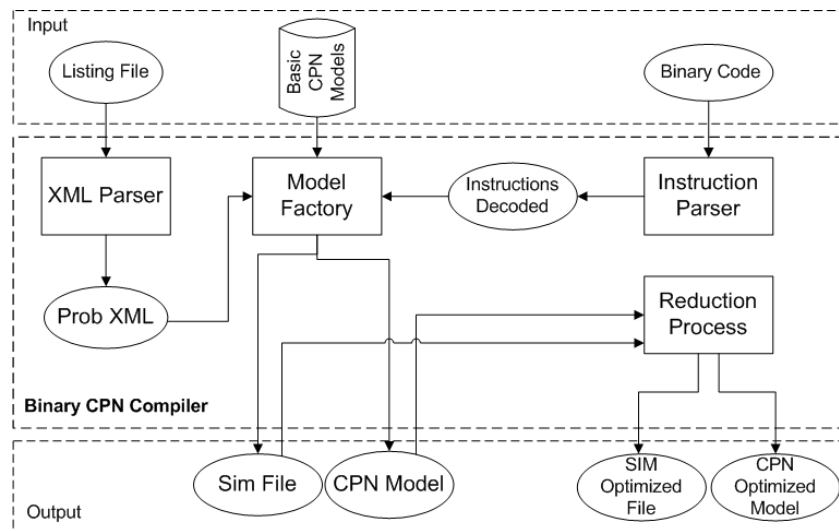


Figure 6.3: Detailed Binary-CPN Compiler structure.

6.4 CPN SIMULATOR

CPN Simulator is a tool developed to evaluate the proposed CPN models. This tool has been conceived as an alternative to CPN Tools [VLM⁺03], since the respective simulation mechanism consumes a significant amount of time when analyzing large models. Figure 6.4 depicts the basic CPN Simulator structure which considers a XML model speci-

cation (*Sim* file) to estimate the energy consumption and execution time of embedded applications.

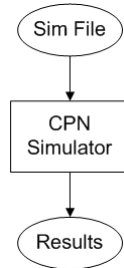


Figure 6.4: Basic CPN Simulator structure.

Figure 6.5 shows the CPN Simulator structure in more details. The parser processes the *Sim* file in order to create an internal element representation to transitions, places, arcs and tokens. This parser also identifies the stop criteria parameters such as confidence interval, desired energy and time errors. Afterwards, the proposed Net Factory has been considered to connect the created internal elements in order to generate the internal model to be evaluated. This model has been simulated through the simulator driver, in which statistical methods related to the execution time and energy consumption estimates are considered. The simulation continues according to the stop criteria evaluation (see Section 6.4.4). An overview of the proposed implementation is present in Appendix D, in which the CPN Simulator class diagram is detailed.

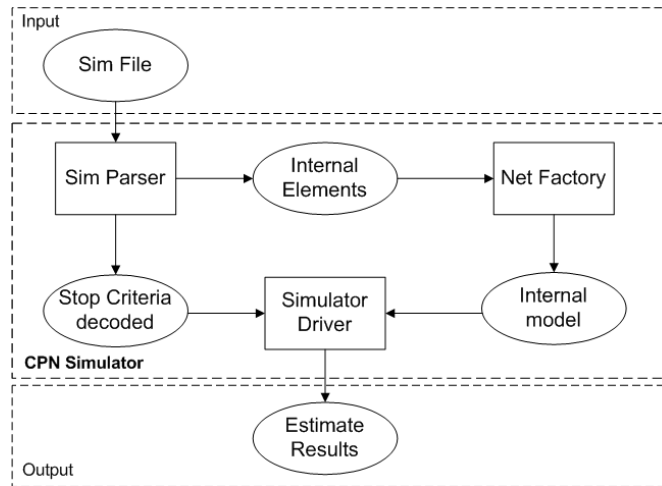


Figure 6.5: CPN Simulator structure.

6.4.1 Sim file

The *Sim* file considers each Coloured Petri net structures (e.g.: places) and the stop criteria parameters (e.g.: confidence interval) through XML specification. For a better visualization, the following lines present the XML model specification adopted by CPN Simulator.

Place

Place specifications have an internal identification (*id*) and the place name.

```
< Place id="identification" name="label" >
```

Token

Token specifications have a token color tag (*TokenColor*) and *placeID* that identifies the token's place.

```
<tokens><TokenColor placeID="identification" /></tokens>
```

Arcs

Place-transition (input) arcs have four tags: *fromID*, identifies the place; *toID*, identifies the transition; *id*, arc identification; and *name*, arc name.

```
< InputArc fromID="PlaceIdentification" id="identification" name="arcName" toID="TransitionIdentification" >
```

Transition-place (output) arcs have four tags: *fromID*, identifies the transition; *toID*, identifies the place; *id*, arc identification; and *name*, arc name.

```
< OutputArc fromID="TransitionIdentification" id="identification" name="arcName" toID="PlaceIdentification" >
```

Transitions

Ordinary transitions are adopted in order to represent the ordinary models (see Section 5.1.1). The execution time and energy consumption values are associated to these transitions.

```
< OrdinaryTransition cy="TimeValue" energyCost="EnergyValue" id="identification" name="label" >
```

Conditional transitions are adopted to represent the conditional model (see Section 5.1.2). In addition to the time and energy values, a *probability* value is associated to these transitions.

```
< ConditionalTransition cy="TimeValue" energyCost="EnergyValue" id="identification" name="label" probability="value" >
```

Stop Criteria parameters

The desired execution time (*ErrorMaxTime*) and energy consumption (*ErrorMaxEner*) errors and the confidence interval (*IC*) are set up by the following XML.

```
< configuration ErrorMaxEner="value" ErrorMaxTime="value" IC="confidence interval" >
```

6.4.2 Simulation Process

The simulation starts by reading the *Sim* file, initializing statistical counters (variables used for storing statistical information about system performance and energy consumption) and setting the stop criteria parameters (confidence interval, desired energy consumption and execution time errors). Next, the simulation clock is set to “0” (variable indicating the current value of simulated time) and the event list is created (list that contains the enabled transitions).

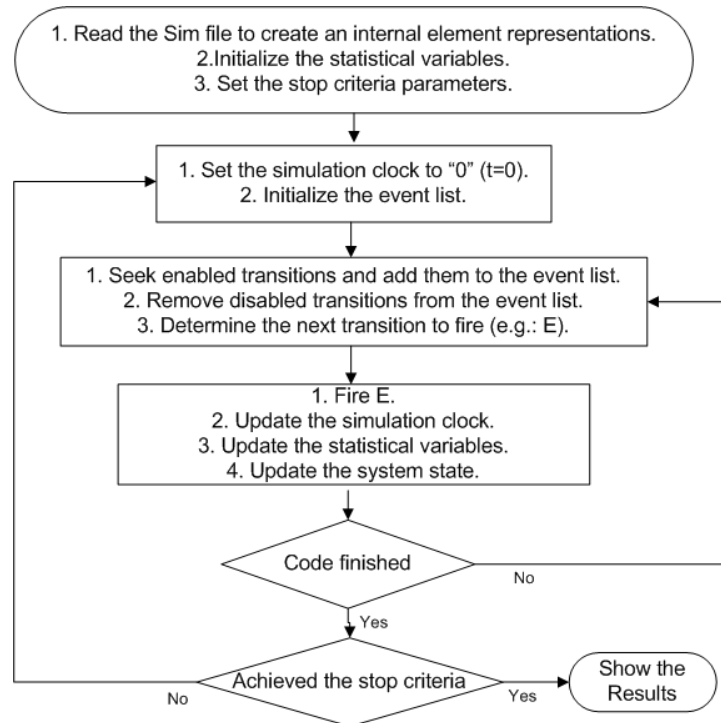


Figure 6.6: Simulation diagram.

Afterwards, the simulation process get into the loop that is present on the proposed algorithm (see Appendix B). The simulation loop corresponds to the evaluation of the proposed CPN model until the estimate results take into account the specified confidence degree. The loop is started and the event list is updated by a method that adds the enabled transitions and removes disabled ones. A transition with the smallest time associated is chosen from the event list to be fired. It is important to recover that a transition fire means that an event (instruction or block of instructions) was executed.

Always after performing an event activity (a transition fires), the simulation clock, statistical counters and the system state are updated in order to represent the new state of the CPN model evaluation. Moreover, after all transition fires, it should be evaluate

if it is the end of the code or not. In case it is the code finish, a method is conceived in order to determine whether the simulation should be finished or not according to the stop criteria evaluation (Section 6.4.4). After finishing the simulation, the estimate results (e.g.: energy consumption and execution time values) are showed. Figure 6.6 shows the simulation diagram of the adopted simulation process.

6.4.3 Simulation Algorithm Variables

In order to perform the adopted simulation algorithm (see Appendix B) considering the CPN Simulator implementation (see Appendix D), some variables have been used to provide the estimate metrics. These variables can be classified into: (i) input variables, adopted by the stop criteria evaluation; (ii) auxiliary variables, compute the number of iterations and replications simulated; (iii) counter variables for the metrics, compute the desired metrics (energy consumption and execution time values); (iv) auxiliary lists for the metrics, adopted to contain the simulation results in order to perform the statistical calculation; (v) event lists, lists that contain the events that may or may not happen. The following lines depicts this classification.

Input Variables

- (i) *nIterations* : determines the number of iterations of each replication;
- (ii) *iConf*: adopted confidence degree;
- (iii) *desiredTime* : desired execution time precision;
- (iv) *desiredEnergy* : desired energy consumption precision;
- (v) *nMinReplications* : determines the minimum number of replications;
- (vi) *nMaxReplications* : determines the maximum number of replications.

Auxiliary Variables

- (i) *iteration* : counter that computes the number of iteration;
- (ii) *currentReplication* : counter that computes the number of replication.

Counter Variables for the Metrics

- (i) *currentTime*: counter for computing the executed time of the current iteration;

- (ii) *currentCost* : counter for computing the energy consumption value of the current iteration.

Auxiliary Lists for the Metrics

- (i) *iTime* : List that contains the execution time values of each iteration;
- (ii) *iCost* : List that contains the mean energy consumption values of each iteration;
- (iii) *costAnalysis*: List that contains the mean energy consumption results of each replication (the mean values after performed *nIterations*);
- (iv) *timeAnalysis*: List that contains the mean execution time results of each replication.

Event Lists

- (i) *eventList* : List that contains the enabled transitions;
- (ii) *readyTrasitions*: List that contains the transitions ready to fire.

It is important to highlight that a simulation can be stopped by the stop criteria evaluation (see Section 6.4.4) or by overcoming the maximum number of replications. Considering the stop criteria evaluation, the simulation runtime depends directly on the desired accuracy - input variables (iii) and (iv) - and the confidence degree - input variable (ii) - that are configured by the designer. On the other hand, the simulation can be finished when the *currentReplication* - auxiliary variable (ii) - overcomes the set up value for the maximum number of replications - input variable (vi).

CurrentTime and *currentCost* - counter variables (i) and (ii) - represent counters for computing the executed time and energy consumption of the current iterations. The iteration results are stored into lists - auxiliary lists (i) and (ii) - that contain the execution time values and energy consumption of each iteration. After performing *nIterations* (a replication) - input variable (i), the mean value of *iTime* and *iCost* lists are computed and stored into the *timeAnalysis* and *costAnalysis* - auxiliary lists (iii) and (iv). When the minimum number of replications is executed - input variable (v), it is calculated the *absolute precision* for both energy and time samples. These values are compared with the desired precisions (variables *desiredTime* and *desiredEnergy*). In case the computed results are smaller than the desired ones, the simulation is finished. Otherwise, the simulation continues by calculating the number of replication needed.

In addition, a list - event list (i) - that contains the next events (enabled transitions) is also adopted. This list should be updated each time that an event happens. Considering the possibility of two events be enabled at the same time, another list is used - event list

(ii). This list takes into account the timing constraints of the events, in which the event (transition) with the smallest time is chosen. Considering the case that two transitions have the same timing constraints, one transition is selected randomly.

6.4.4 Stop Criteria Evaluation

A stop criteria evaluation is adopted in order to provide simulation results taking into account specified confidence degree. As a narrow confidence interval has been considered, the simulation process has to be executed several times (runs) to provide the estimates results [Chu04]. The number of runs depends on (among other factors) the specified confidence degree. The initial number of replication runs adopted is specified by the analyzer.

Stop Criteria considers: *Absolute Precisions* for energy consumption and execution time (the designer informs the desired precisions), means and standard deviations. The *Absolute Precision*, calculated by Equation 6.2, in which the t critical value is calculated for $1 - \alpha/2$ confidence degree and $n - 1$ degrees of freedom; s is the standard deviation of replication and n is the number of replications in the example.

$$AbsolutePrecision = t_{1-\alpha/2, n-1} \times \frac{s}{\sqrt{n}} \quad (6.2)$$

Afterwards, the desired precisions related to both energy consumption and execution time are compared with the current results. The simulation is finished if these calculated values are smaller than the desired precisions specified. Otherwise, the simulation proceeds by calculating the required number of new simulation runs (replications) through Equation 6.3 considering the desired precision specified. There are two replication values, one for energy consumption and other for execution time. Figure 6.7 depicts the adopted stop criteria evaluation process.

$$i = \left[\frac{t_{1-\alpha/2, n-1} \times s}{DesiredPrecision} \right]^2 \quad (6.3)$$

Exemplifying

For a better understanding, an example has been considered to demonstrate the stop criteria evaluation process. First of all, it is determined both desired precisions values for energy consumption and execution time which were $0,5\mu J$ and $1.0\mu s$, respectively. The considered confidence interval was 95%. Table 6.1 shows the simulation results. It is important to stress that each replication number considered 40 runs and, so, each value is a mean value.

Afterwards, the desired precisions related to both energy consumption and execution time are compared with the current simulation results (see Table 6.2). The reader should

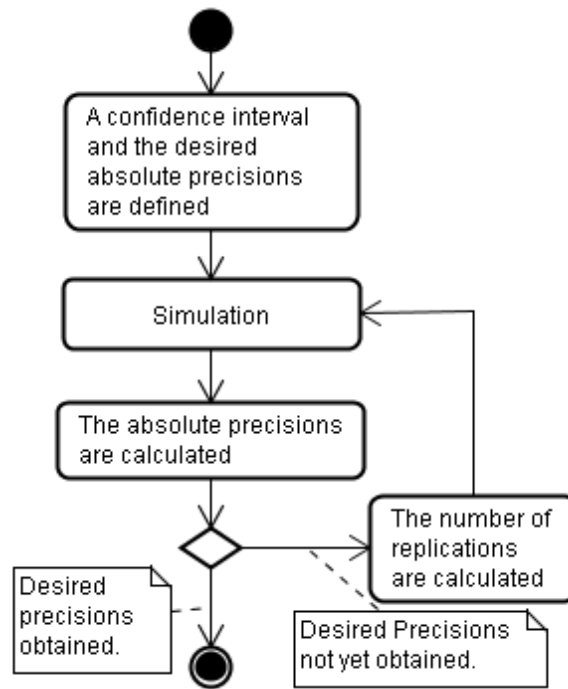


Figure 6.7: Stop criteria diagram.

observe that both calculated absolute precisions for energy and time were higher than the desired values, where the energy result was $4,011\mu J$ (desired $0,5\mu J$) and the time computed was $2,147\mu s$ (desired $1,0\mu s$). Thus, the simulation proceeded by calculating the required number of new replication runs (i) through Equation 6.3. The number of more replication needed for the time metric was 46 and for energy was 643. Table 6.2 also shows the simulation summary results considering 653 simulations, in which it is possible to remark that the computed absolute precisions were smaller than the desired. As a result, the simulation is finished, where the mean time was $24,049\mu s$ and the mean energy consumption value was $46,580\mu J$.

6.4.5 Enabling and Firing rules

Although the proposed CPN models have not considered parallelism, the CPN Simulator can evaluate concurrent systems. CPN simulator adopts the atomic firing rule and considers enabling memory method (see Section 2.5). In order to perform this task, a timer measuring the execution time is adopted, and all transitions have a time associated. The firing rule conceived that transition with the smallest time is fired first. Considering the case that two transitions have the same timing constraints, one transition is selected randomly.

A transition t is said to be enabled, if each input places p of t contains the multi-set

Table 6.1: The simulation results.

Replication Number	Mean Time (μs)	Mean Energy (μJ)
1	21,125	41,6455
2	21,55	41,9305
3	25,7	49,92575
4	29,9	58,1395
5	22,875	44,36325
6	22,425	42,99325
7	26,45	51,33025
8	25,775	49,8
9	23,325	46,5175
10	28,675	54,68025

Table 6.2: Comparison: 10 replications versus 653 replications

Case Study	10 replications		653 replications	
	Time(μs)	Energy(μJ)	Time(μs)	Energy(μJ)
Mean Value	24,78	48,133	24,049	46,580
Standard Deviation	3,002	5,607	1,999	3,846
<i>AbsolutePrecision</i>	2,147	4,011	0,153	0,295

specified by the input arc inscription, the guard expression is evaluated to true and the timing constraints are considered. A step Y is enabled in a state (M_1, r_1) at time r_2 iff (if and only if) the following properties are satisfied (see Section 2.6.2 for more details):

$$(i) \sum_{(t,b) \in Y} E(p, t) < b > r_2 \leq M_1(p), \forall p \in P$$

$$(ii) r_1 \leq r_2$$

(iii) r_2 is the smallest element of \mathbb{R} for which there exists a step satisfying (i) and (ii).

An occurrence of a transition removes tokens from places connected to incoming arcs (input places), and adds tokens to places connected to outgoing arcs (output places), thereby changing the marking (state) of the TCPN. The number and colour of the tokens are determined by the arc expressions, evaluated for the occurring bindings. The formal definition is presented as follows.

When a step Y is enabled in a marking M_1 it may occur, changing the marking M_1 to another marking M_2 , defined by:

$$(i) \quad M_2(p) = (M_1(p) - \sum_{(t,b) \in Y} E(p,t) \langle b \rangle) + \sum_{(t,b) \in Y} E(t,p) \langle b \rangle, \forall p \in P.$$

where:

The expression evaluation $E(p,t) \langle b \rangle$ computes the tokens which are removed from p when t occurs with the binding b .

The expression evaluation $E(t,p) \langle b \rangle$ computes the tokens which are added to places connected to outgoing arcs with the binding b .

For a better comprehension of concepts related to the adopted enabling and firing rules, a Petri net $PN = (P = \{p_0, p_1, p_2, p_3, p_4, p_5\}, T = \{t_0, t_1, t_2, t_3, t_4\}, F = \{(p_0, t_0), (t_0, p_1), (t_0, p_2), (p_1, t_1), (p_2, t_2), (t_1, p_3), (t_2, p_4), (p_3, t_3), (t_3, p_5), (t_2, p_4), (p_4, t_5)\}, W = \{(p_0, t_0, 1), (t_0, p_1, 1), (t_0, p_2, 1), (p_1, t_1, 1), (p_2, t_2, 1), (t_1, p_3, 1), (t_2, p_4, 1), (p_3, t_3, 1), (t_3, p_5, 1), (t_2, p_4, 1), (p_4, t_5, 1)\}, m_0 = [1, 0, 0, 0, 0, 0])$ is depicted in Figure 6.8. Conceiving the time associated to the transition $t_0 = 0, t_1 = 2, t_2 = 1, t_3 = 3$ and $t_4 = 2$. Since the initial marking only enables the firing of t_0, t_0 is fired. Thus, the marking changed to $m_1 = [0, 1, 0, 0, 0, 0]$ and the transitions t_1 and t_2 are enabled. At this moment, the transitions t_2 is fired due to its execution time be smaller. Afterwards, transitions t_1 and t_4 are enabled to fire, and t_1 occurred. The reader should bear in mind that the execution time value increased just one (“1”) time unit instead of two (“2” - time associated to t_1). This happened due to the adopted enabling memory method. Table 6.3 shows the firing results of the concurrent example depicted in Figure 6.8.

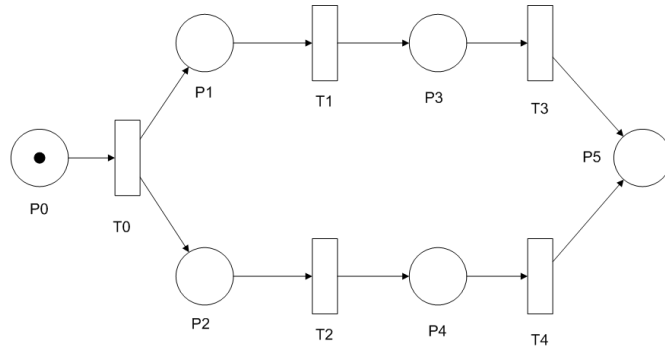


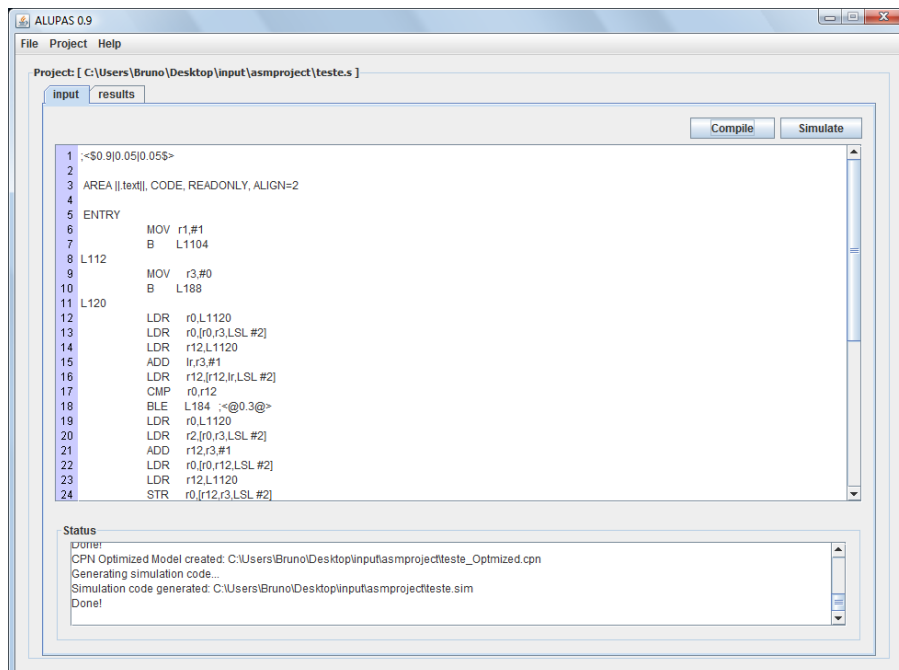
Figure 6.8: Concurrent example.

Table 6.3: Firing rule results of Figure 6.8.

enabled	fired	execution time
t0	t0	0
t1, t2	t2	1
t1, t4	t1	2
t3, t4	t4	3
t3	t3	5

6.5 GRAPHICAL USER INTERFACE

The ALUPAS' Graphical User Interface (GUI) provides a mechanism in which the designer does not interact directly with the internal formalism (CPN). Figure 6.9 depicts the ALUPAS' input interface. A simulation process starts when the designer creates a new project and puts the code to be analyzed. Afterwards, the designer inserts the probabilistic values on conditional instructions and sets up the stop criteria evaluation parameters (e.g.: confidence intervals, desired energy consumption and execution time errors). After a successful compilation process, a CPN model is created to be evaluated (simulated) in order to obtain the energy consumption and execution time estimates.

**Figure 6.9:** ALUPAS' input interface.

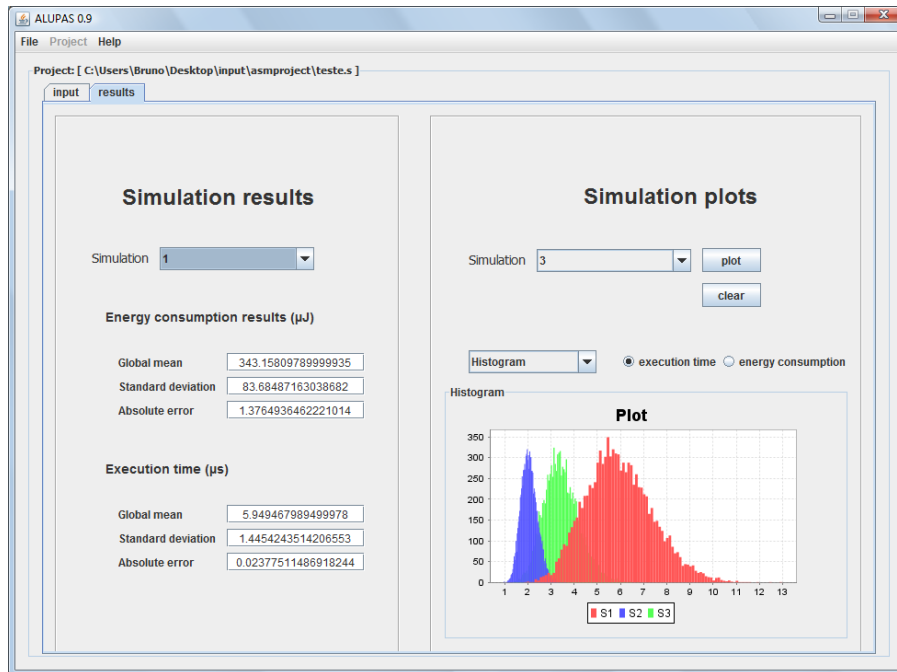


Figure 6.10: ALUPAS' output interface.

ALUPAS can evaluate different control flow scenarios, where the designer just changes the probabilistic annotation values of the conditional instructions. The estimate results are stored and can be compared to other simulation results. Figure 6.10 shows the ALUPAS' output interface. The *Simulation Results* frame depicts the estimated metric results (mean energy consumption and execution time values, and their respective standard deviations and errors). After performing the simulation, it is important to highlight that graphic representations (histogram and box plot) can be plotted. The reader should observe that different simulation results can be plotted into the same plot area (see Figure 6.10). This fact helps the designer to compare different simulation scenarios.

6.5.1 Component Integration

ALUPAS has been developed to combine the proposed framework functionalities (Section 4.2) into a unified environment with a GUI (see Figure 6.11). The reader should have in mind that the designer just interacts with the GUI. However, trained designers can view and edit the CPN model that represents the code behavior. Furthermore, the file with all simulation results is available from which the designer can consider to perform other statistical calculations. It is important to highlight that the component integration is quite similar to the proposed framework and, so, it has not been not described in details.

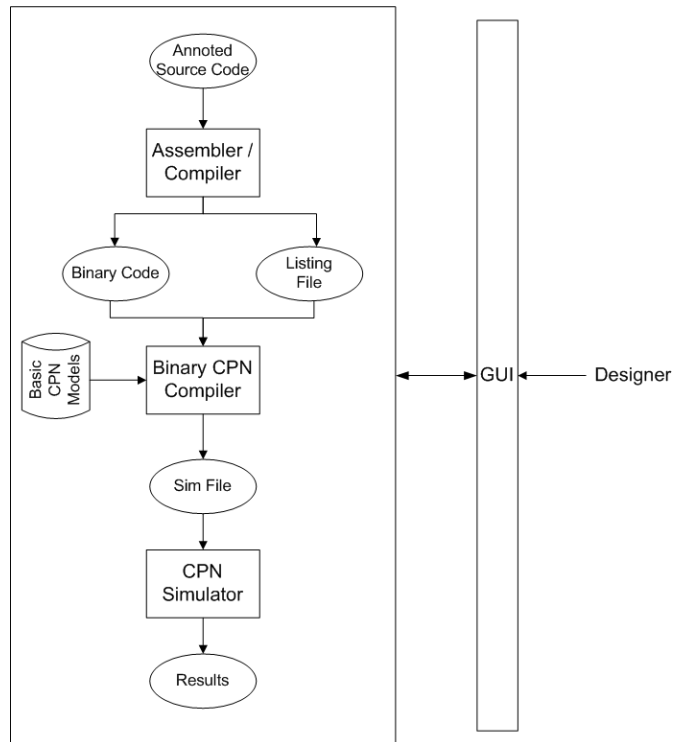


Figure 6.11: Component Integration.

6.6 SUMMARY

This chapter presented the simulation environment, named, ALUPAS, that provides, in early design phases, a mechanism that assists design decisions on energy consumption and performance concerning embedded applications. Thus, this chapter detailed the ALUPAS' components. The Binary CPN Compiler and CPN Simulator components are the most important. The Binary CPN Compiler automatically translates an embedded software, implemented in C language or Assembly, into a CPN model, whereas the CPN Simulator is a tool developed to evaluate the proposed CPN models.

CHAPTER 7

CASE STUDIES

In order to illustrate the practical usability of the proposed methodology, this chapter presents five experimental results in details. All experiments were performed on an AMD Turion 64X2 1.6GHz, 2Gb RAM, and OS WinXP. The first one exemplifies the proposed methodology by a small code application which was adopted for explaining the process to estimate both energy consumption and execution time of embedded softwares. The second experiment is performed in order to illustrate a runtime comparison between the CPN Simulator, specific engine to evaluate CPN models, and CPN Tools. Experiments adopting a binary search algorithm are considered, in the third study case, to depict that different scenarios can be created by changing the probabilistic annotations for the conditional code instructions. The fourth experiment, the BCNT algorithm, is considered to evaluate the proposed methodology and to illustrate in details the proposed optimized CPN model (model after reduction process). The last experiment, the pulse-oximeter case study, is conducted in order to apply the proposed methodology in a real-world case.

7.1 EXAMPLE ONE

This example has been conducted in order to exemplify the proposed methodology. Figure 7.1 depicts a small application code, where the values for the stop criteria evaluation are depicted on the first code line. Note that the confidence degree is set to 95%, the specified precision for energy consumption is $200\eta\text{J}$, and $20\mu\text{s}$ is the specified precision for the execution time. The number of runs of each replications is set to 40 times, and the maximum number of replications is 10.000 states (if the simulation is finished by this condition, there is no guarantee that the confidence degree is gotten). It is important to highlight that these values are chosen by the designer from a previous acknowledgement about the code under analysis.

The registers' values have not been considered in the model, and instead of comparing them, a probabilistic approach is adopted. In this example, there is a loop (lines 16-20) that is executed 10 times, so, the probability is performed using the equation $p = 1 - (1/N)$. In this case, $p = 1 - 1/10 = 0,9$ (probability of the conditional instruction *blt loop*). The probability has been adopted to set *prob* variable (see Figure 5.7) in the conditional CPN model.

```

1  ;<$0.95|200.0|20.0|40|10000$>
2  AREA  ARMex, CODE, READONLY
3  ENTRY
4  main bl proc
5      bx r14
6
7  proc stmdb r13!,{r14}
8      bl for
9      bl for
10     mov r1,#0
11     mov r2,#0
12     add r1,r1,#1
13     add r2,r2,#2
14     ldmia r13!,{r15}
15
16     for mov r4,#1
17     b test
18     loop add r4,r4,#1
19     test cmp r4,#0xa
20     blt loop ;<@0.9@>
21     bx r14
22     END

```

Figure 7.1: Annotated Assembly Code

7.1.1 Simulation Results.

The code depicted in Figure 7.1 was simulated in three different ways: (i) using CPN Tools with the CPN model, (ii) using CPN Tools with the optimized CPN Model, (iii) and adopting the proposed CPN Simulator with the optimized CPN Model. The simulation results using CPN Tools considering both CPN model and optimized model are identical. Thus, the proposed CPN Simulator has been considering only the optimized model.

Table 7.1 presents both CPN Tools and CPN Simulator simulation results. The reader should observe that we are considering standard deviation and errors for both metrics (energy consumption and execution time). The time standard deviation obtained when CPN Tools was considered was $0,19\mu s$. The confidence degree adopted was 95% (see header annotation on Figure 7.1), so that the execution time value ($2,23\mu s$) should be within $[2,10\mu s; 2,37\mu s]$.

Table 7.1: Simulation Results of the code on Figure 7.1.

CPN Tools	CPN Simulator
Mean Time: $2,2320 \mu s$	Mean Time: $2,2279 \mu s$
Time SD: $0,1905 \mu s$	Time SD: $0,1444 \mu s$
Time Error: $0,1363 \mu s$	Time Error: $0,1033 \mu s$
Mean Energy: $167,4008 \eta J$	Mean Energy: $167,2632 \eta J$
Energy SD: $14,1533 \eta J$	Energy SD: $10,7243 \eta J$
Energy Error $10,1240 \eta J$	Energy Error: $7,6717 \eta J$

Table 7.2 compares the simulation results obtained through CPN tools (considering the non-optimized model) and the proposed CPN simulator in order to show that these results are very close. The simulation results provided by the CPN Simulator are similar

to those obtained by the CPN Tools, since the differences are smaller than 2%. Table 7.2 also compares the simulation runtime of both environments CPN Tools and CPN Simulator. The CPN Tools spent 22s, meanwhile the CPN Simulator spent less than 1s. Thus, this simple example showed that the simulation runtime of CPN Simulator were much faster then the CPN Tools.

Table 7.2: Comparison between simulation results.

	CPN Tools	CPN Simulator
	CPN Model	CPN Optimized
Execution Time	2,2320 μs	2,2279 μs
Energy Consumption	167,4008 ηJ	167,2632 ηJ
Runtime	22 s	172 ms

7.2 EXAMPLE TWO

This experiment has been adopted aiming to show the importance of both CPN reduction process and CPN Simulator. It consists of codes with instructions that only use the ordinary model (see Figure 5.3). Thus, these codes do not perform branches in the code control flow and the examples were performed with 10, 20, 30, 40, 50, 100, 200, 400 instructions in order to perform the runtime comparison between CPN Tools and CPN Simulator.

Table 7.3 shows a runtime comparison of those simulation models in which different numbers of instructions were taken into account. It is worth stressing that the runtime simulation on CPN tools is quite time consuming when analyzing large models.

Table 7.3: Comparison of the runtime simulation.

N Inst.	CPN Model		CPN Tools /
	CPN Tools	CPN Simulator	CPN Simulator
10	17s	187ms	91
20	21s	219ms	96
30	30s	234ms	128
40	41s	281ms	146
50	56s	297ms	189
100	1min 51s	515ms	216
200	5min 6s	1s 219ms	251
400	17min 25s	3s 438ms	304

Table 7.3 also presents the CPN Simulator runtime which is at least 91 times shorter than the respective time on CPN Tools environment. The column named *CPN Tools / CPN Simulator* shows the ratio between CPN Simulator runtime and CPN Tools runtime. The CPN Simulator has performed much faster simulations than CPN tools. However, it is also important to mention that CPN Simulator performs even better for larger models. For a 20-instruction application, the CPN Simulator was 96 times faster than CPN Tools; and for a 200-instruction application, the CPN Simulator was 251 times faster.

Figure 7.2 depicts the simulation time comparison between CPN Tools and CPN Simulator. The CPN Simulator was much faster than CPN Tools. A possible explanation for the runtime in CPN Tools has not been good enough to evaluate the conceived models (huge models) is the fact that CPN Tools is a generic environment to create, edit and simulate CPN models. Thus, such environment needs to perform a syntax analysis process before being able to simulate. Furthermore, as CPN Simulator is a specific tailored environment for simulating the conceived CPN models, it does not spend time checking models' syntax. Nevertheless, it is important to stress that the obtained models are syntactically correct and their semantics represent the programs' control flow.

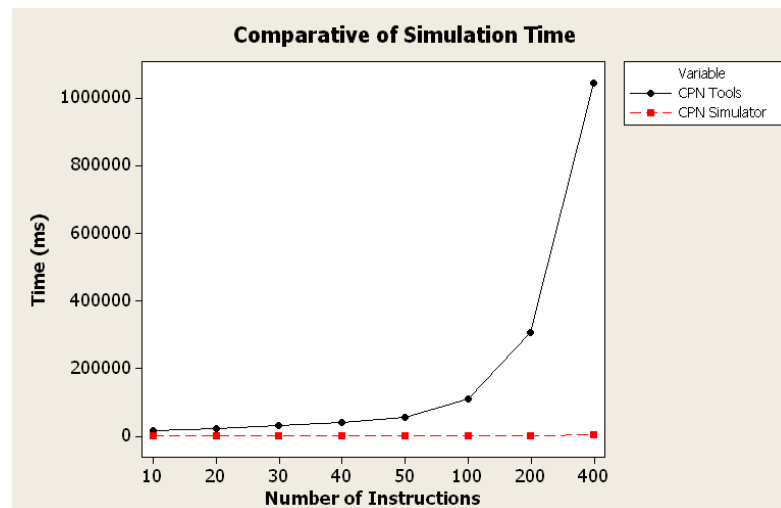


Figure 7.2: CPN Tools versus CPN Simulator runtime.

In this example, the adopted reduction process clustered all the instructions when it is considered the optimized CPN model. The CPN tools runtime and CPN Simulator runtime were 6s and 187ms, respectively for all of these experiment examples. Thus, the simulation runtime with optimized CPN model (6s) for the 400-instruction application was 174 times faster than the simulation with non optimized CPN model (17min 25s).

7.3 BINARY SEARCH ALGORITHM

A binary search algorithm is a technique for finding a particular value in a sorted list (array) of values. This method starts by selecting the middle element of an array, comparing

its value to the target value, and determining if the selected value is greater than, less than, or equal to the target value. The selected element (by guessing its index) whose value turns out to be higher becomes the new upper bound of the array, and if its value is lower, it becomes the new lower bound. This technique continues iteratively and such algorithm reduces the search by a factor of two each time.

```

1          ;<$0.95|0.1|0.1>
2          AREA    ARMex, CODE, READONLY
3          ENTRY
4          main MOV   r0,#0
5            MOV   r1,#0xfe
6            MVN   r3,#0
7            B     loop
8
9          begin ADD   r12,r0,r1
10           ASR   r2,r12,#1
11           LDR   r12,[L1.4084]
12           LDR   r12,[r12,r2,LSL #3]
13           CMP   r12,#0x12c
14           BNE   notfound           ;<@1.0@>
15           LDR   r12,[L1.4084]
16           ADD   r12,r12,r2,LSL #3
17           LDR   r13,[r12,#4]
18           B     end
19
20 notfound LDR   r12,[L1.4084]
21           LDR   r12,[r12,r2,LSL #3]
22           CMP   r12,#0x12c
23           BGE   inf                 ;<@0.0>
24           ADD   r0,r2,#1
25           B     loop
26
27           inf SUB   r1,r2,#1
28
29           loop CMP   r0,r1
30            BLE   begin               ;<@0.88@>
31
32           end MOV   r0,r0
33            BX   lr

```

Figure 7.3: Binary Search Code in Assembly.

In this example, an array with 255 elements was considered. Figure 7.3 depicts this algorithm in assembly language and Figure 7.4 shows the same algorithm in C. It is possible to observe that the code behavior depends on the searched element (*key*) and the elements present on the array. In order to represent such behavior, the conditional branch instructions present in Figure 7.3 in the lines 14, 23 and 30 received code annotations representing their probability for branching. During the code execution, the instructions on line 14 and 30 are the ones that determine the number of iterations on the code. The instruction present on line 23 defines the next array bound (lower or higher).

The code was evaluated in three different scenarios: Best Case Execution Time (BCET), Typical Case Execution Time (TCET) and Worst Case Execution Time (WCET). For each scenario, different probabilistic values were associated to the conditional instructions in order to reproduce their respective behavior.

The worst case occurs when the binary search algorithm looks for a non-present element on the array. In this case, the iteration number is 8 ($\log_2(255)$). On the other hand, in the best case scenario, the searched element is found in the first iteration (in the middle of the array). Nevertheless, the probability of finding an element on an i iteration is $\frac{2^{i-1}}{255}$, considering a typical case, then the mean iteration number in order to find an element is $\sum_{i=1}^8 i \cdot \frac{2^{i-1}}{255} \cong 7$.

```

1  #DEFINE TARGET 300
2
3  struct DATA {
4      int key;
5      int value;
6  };
7
8  struct DATA data[255];
9
10 int binary_search()
11 {
12     int fvalue, mid, up, low;
13     low = 0;
14     up = 254;
15     fvalue = -1 /* all data are positive */ ;
16     while (low <= up) { ;<@0.88@>
17         mid = (low + up) >> 1;
18         if (data[mid].key == TARGET) { //<@1.00@> /* found */
19             fvalue = data[mid].value;
20             break;
21         } else
22         if (data[mid].key < TARGET) { //<@0.00@> /* not found */
23             low = mid + 1;
24         } else {
25             up = mid - 1;
26         }
27     }
28     return fvalue;
29 }

```

Figure 7.4: Binary Search Code in C.

The instruction in line 30 of Figure 7.3 is responsible for the conditional behavior (*while*) related in the Figure 7.4. As a consequence, in the worst case, such conditional instruction should be evaluated as true 8 times, so its probability is $p = \frac{8}{9}$. Likewise, the probability of the instruction present in line 14 is “1” in order to represent a non-present element searched on-the-array. The reader should note that this conditional statement is represented in line 18 of the C code as Figure 7.4 depicts. Such value may seem odd, but the reader should bear in mind that it reproduces the probability for not finding an element on the array, hence its value should be *true* (“1”) considering the instruction that makes a branch instead of getting inside the code. In addition, the probability set on the other conditional instruction (line 14 of Figure 7.3) was set up to perform the worst case scenario. Thus, both figures (Figure 7.3 and Figure 7.4) depict the annotated code in order to represent the worst case scenario of the binary search algorithm.

In order to evaluate the best case, the instructions present in lines 14 and 30 should have their probabilities set to “0” and “1”, respectively. It is important to state that the probability related to the other conditional instruction (present in line 23) is not relevant because the code execution, in this case, does not reach this point.

In order to evaluate the typical case scenario, the mean iteration number (7) previously calculated was considered. Hence, the probabilities related to instructions present in lines 14, 23 and 30 were $\frac{6}{7}$ (meaning that in the 7th try, the element is found), “1” and “0,5”, respectively.

7.3.1 Results

Table 7.4 shows the results for the typical, the best and the worst scenarios of a binary search algorithm. The lowest energy consumption and execution time values occurred in the best case. On the other hand, the highest consumption and execution time were performed by the worst scenario. The typical scenario results were between the best and worst case. Moreover, the estimated results are quite close to the measured ones performed on the hardware platform. For example, the estimated execution time value for the worst case was $15,3\mu s$ and the measured one was $15,2\mu s$. Figures 7.5 and 7.6 depict the respective results.

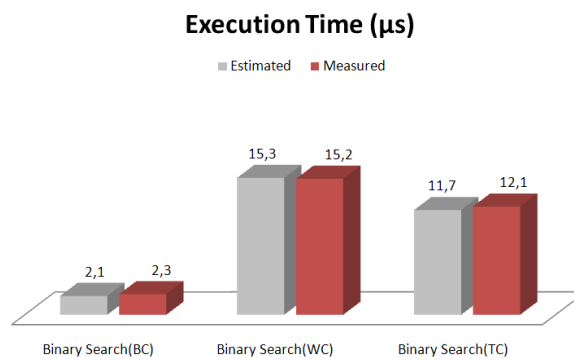


Figure 7.5: Binary search results of execution time.

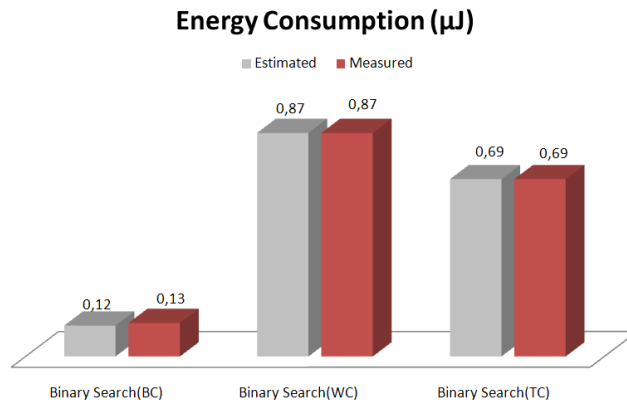


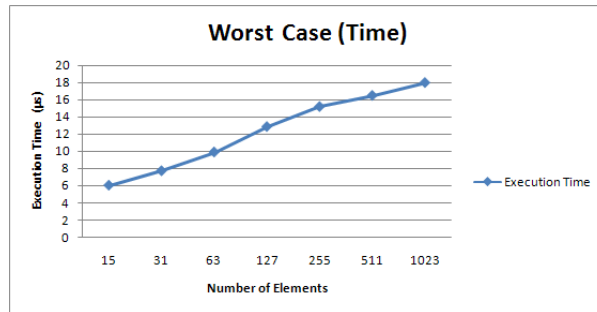
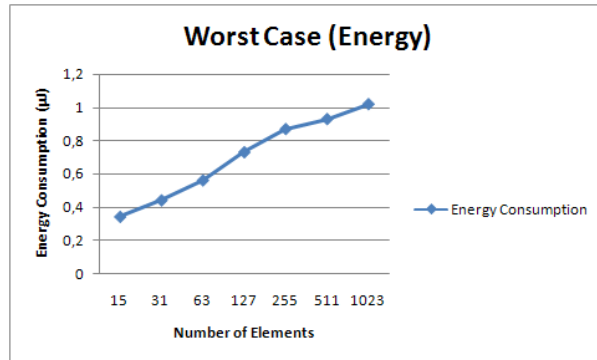
Figure 7.6: Binary search results of energy consumption.

After the results had been compared (validated) with the hardware platform for an array with 255 elements, another experiments were performed taking into account other arrays with different lengths for estimating the worst and best cases. As it was already explained, the conditional instruction present on line 30 is the one responsible to determine the list length, so that its probability were $p = \frac{4}{5}$ (for 15 elements), $p = \frac{5}{6}$ (for 31 elements), $p = \frac{6}{7}$ (for 63 elements) and so on until $p = \frac{10}{11}$ (for 1023 elements). Figure 7.7 shows the results considering execution time worst case and Figure 7.8 depicts the results

Table 7.4: Binary Search results summary

Case Study	Estimated		Hardware	
	Time(μs)	Energy(μJ)	Time(μs)	Energy(μJ)
1. Binary Search(BCET)	2,1	0,12	2,3	0,13
2. Binary Search(WCET)	15,3	0,87	15,2	0,87
3. Binary Search(TCET)	11,7	0,69	12,1	0,69

related to energy consumption worst case. As the reader may observe, the results of both energy consumption and execution time worst case increase when the binary search algorithm is performed in higher arrays.

**Figure 7.7:** Binary search results of execution time.**Figure 7.8:** Binary search results of energy consumption.

7.4 BCNT ALGORITHM

The BCNT Algorithm was proposed by Motorola as an integrated part of Power Stone Benchmark. The BCNT adopts a series of operations between two arrays, explores the memory space manipulation, and it also adopts bitwise operations. Figure 7.9 depicts the optimized CPN model for the BCNT algorithm, in which it is possible to represent a set

of instructions by only one transition (reductions). The *BLOCKID* transition represents these clustered instructions.

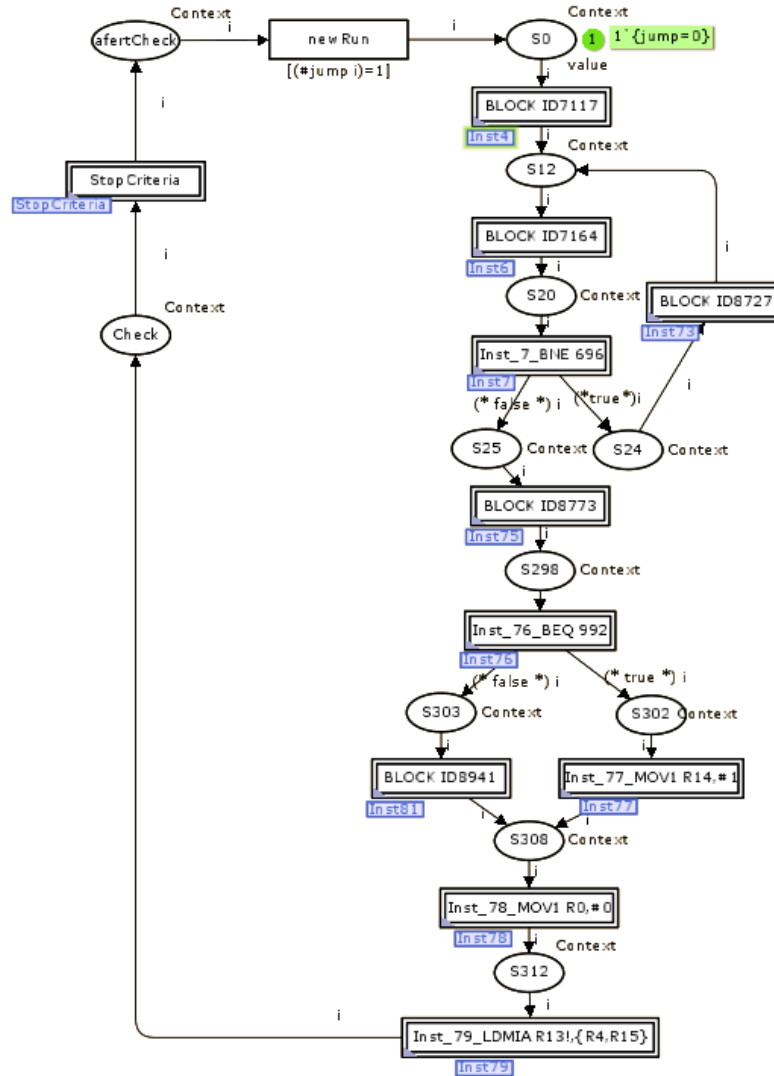


Figure 7.9: The CPN model for BCNT Algorithm.

Table 7.5 depicts a comparative study between the estimated values and measurements conducted on hardware platform according to the methodology described in [TM08]. The execution time measured on hardware was $96,39 \mu s$ and the energy consumption was $5,73 \mu J$. The estimated time error was 2,27% and the energy error 4,23%.

7.5 PULSE-OXIMETER

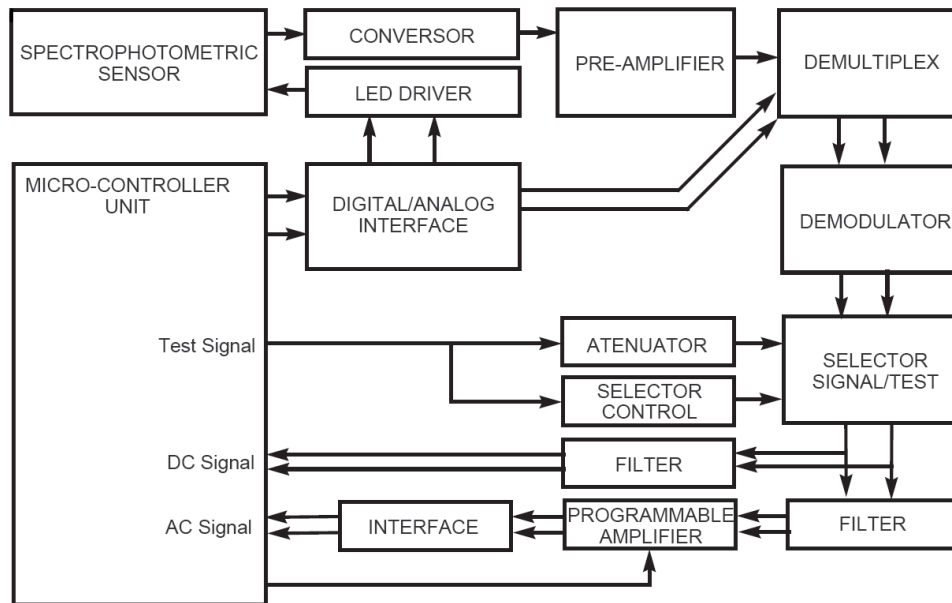
This case study is considered in order to apply the proposed methodology for estimating the energy consumption and execution time of a real embedded system application. Pulse-

Table 7.5: BCNT results summary.

	Time (μs)	Energy (μJ)
Estimated	94,25	5,50
Measured	96,39	5,73
Error	2,27%	4,23%

oximeter is a widely-used biomedical device and a portable one is battery operated, therefore its battery-life time is of great importance. This electronic device is responsible for non-invasively measurements of the blood oxygen saturation and has been widely used in Center Care Units (CCU).

The architecture of the adopted pulse-oximeter device can be seen in Figure 7.10. Such 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.

**Figure 7.10:** Pulse-Oximeter Architecture.

The micro-controller controls the synchronization and amplitude of the led driver, which dispatches non-simultaneous stream pulses to the infrared and red leds (see Figure 7.11). Both leds generate, respectively, infrared and red radiation pulses through the finger of a patient. A photo-diode detects the radiations level. The micro-controller

calculates the related oxygen saturation level based on data received, and shows the result on a display.

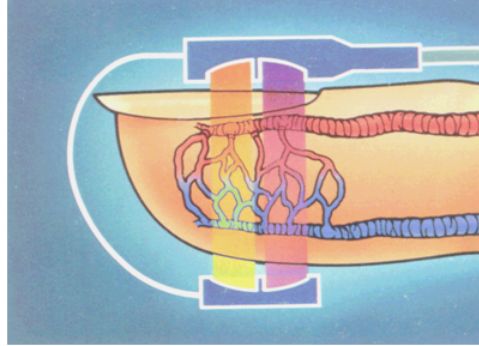


Figure 7.11: Pulse-oximeter.

The pulse-oximeter code was divided into three processes: (i) *excitation* which is responsible for dispatching stream pulses to the leds in order to generate radiation pulses; (ii) *acquisition*, which deals with the data captured from the radiations on the patient's finger; and (iii) *control* which is responsible to perform the calculation of oxygen saturation level. For each process, a CPN model was built in order to estimate the respective energy consumption and execution time. Table 7.6 presents a comparative study between the estimated values and measurements conducted on the hardware platform according to the methodology described in [CM08].

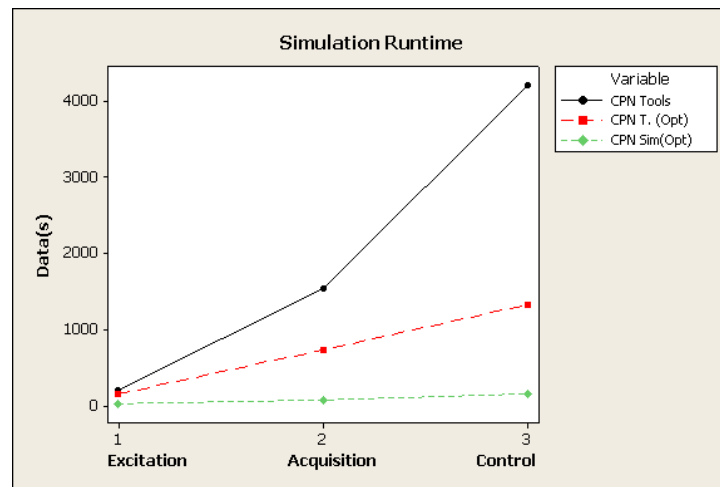
Table 7.6: Pulse-oximeter result summary

Case Study	Estimated		Hardware		Error	
	Time(μs)	Energy(μJ)	Time(μs)	Energy(μJ)	Time(%)	Energy(%)
1. Excitation	38,48	2,20	38,88	2,25	1,04	2,20
2. Acquisition	86,61	5,16	91,18	5,55	5,28	7,66
3. Control	12410,78	722,54	12745,99	779,46	2,70	7,88

Furthermore, Table 7.7 presents a runtime comparison of CPN Tools versus CPN Simulator, and Figure 7.12 also depicts this comparison through a graphic representation. In this graphic, there are three lines representing the simulations: (i) CPN Tools adopting CPN Model as input; (ii) CPN Tools with the CPN Optimized model as input; and (iii) CPN Simulator evaluating the CPN Optimized model. The reader should observe that the first line increases much faster than the others, hence, one may observe that the CPN Simulator has provided results at good accuracy in a much faster runtime than CPN Tools does.

Table 7.7: CPN Tools runtime x CPN Simulator runtime.

	CPN Tools		CPN Simulator
	CPN Model	CPN Opt	CPN Opt
Excitation	203s	150s	22s
Acquisition	1540s	729s	71s
Control	4203s	1325s	151s

**Figure 7.12:** Runtime Comparison.

Nevertheless, in order to validate the estimated values taking into account a statistical method, the Two-Sample T-Test was conducted. Such test compares the effectiveness of these estimated values with the measurement conducted on the hardware platform by determining whether or not there is evidence that highlights difference between the results. Thus, this approach has been adopted to determine whether the estimate values compared with the hardware measurements have or have not a statistically significant differences. For more details about how to perform such test, see Appendix A.

Excitation:

Table 7.8 depicts the execution time data for the difference T-test of excitation process. After performing this test, with 95% of confidence interval, the difference between the execution times measured and simulated is (-0,293; 0,313). As the reader may observe, zero (0) belongs to the interval, thus no difference was detected to highlight distinction between the two sets of data (obtained by measurement and through simulation). Besides, the p-value (0,948) is far greater than commonly chosen α -levels, hence this value might be observed as the “power” of the assertion presented.

Table 7.8: Excitation - comparison of execution time.

	Sample Size	Mean	Standard Deviation
Simulator	416	38,89	3,14
Hardware	1000	38,88	0,007

Table 7.9: Excitation - comparison of energy consumption.

	Sample Size	Mean	Standard Deviation
Simulator	416	2,29	0,18
Hardware	1000	2,25	0,06E-5

Table 7.9 depicts the energy consumption data for performing the difference T-test considering 95% of confidence interval degree. The confidence interval calculated is (0,02265; 0,05735). As the interval does not include zero, it does not suggest that they are equal. Instead, there are evidences to assert that the difference between those data sets is 0,05735 with 95% of confidence. In other words, the difference is 2,5% with a confidence interval of 95%.

Acquisition:

Table 7.10 depicts the execution time data for the difference T-test of acquisition process. After performing this test, with 95% of confidence interval, between the execution times measured and simulated is (-4,804; -4,336). As the reader may observe, zero (0) does not belong to the interval, it does not suggest that they are equal.

Similarly, Table 7.11 depicts the energy consumption data for the difference T-test of acquisition process. After performing this test, with 95% of confidence interval, the difference between energy consumption values estimated and measured is (-0,40389; -0,37611). As the interval does not include zero, it does not suggest that they are equal. Nevertheless, there are evidences to assert with 95% of confidence that the difference between those execution time data sets is 5,5%, and 7,8% for the energy consumption data sets.

Control:

Table 7.12 depicts the execution time data for the difference T-test of control process. After performing this test, with 95% of confidence interval, the difference between execution times simulated and measured is (-689; 19). As the reader may observe, zero (0) belongs to the interval, thus no difference was detected to highlight distinction between the two sets of data (obtained by measurement and through simulation). Besides, the p-value (0,063) is greater than commonly chosen α -levels, hence there is no difference

Table 7.10: Acquisition - comparison of execution time.

	Sample Size	Mean	Standard Deviation
Simulator	2225	86,61	5,62
Hardware	1000	91,18	3E-7

Table 7.11: Acquisition - comparison of energy consumption.

	Sample Size	Mean	Standard Deviation
Simulator	2255	5,16	0,334
Hardware	1000	5,55	0,7E-5

between estimated and measured values.

Table 7.13 depicts the energy consumption data for performing difference T-test considering 95% of confidence interval degree. The confidence interval calculated is (-91,0; -50,6). As the interval does not include zero, it does not suggest that they are equal. Instead, there are evidences to assert that the difference between those data sets is 11% with 95% of confidence.

7.6 SUMMARY

This chapter presented five experiments adopted to evaluate the proposed methodology, and also, to demonstrate the importance of the CPN reduction process as well as to justify the development of the specific simulation tool, named, CPN Simulator. The first one exemplified the proposed methodology by a small code application which was adopted for explaining the process to estimate both energy consumption and execution time of embedded softwares.

The second experiment was performed in order to illustrate a runtime comparison between the CPN Simulator, specific engine to evaluate CPN models, and CPN Tools. It demonstrated that the CPN Simulator runtime was 304 times faster, in some cases, than CPN Tools environment. Moreover, this case study also depicted the runtime evaluation adopting the reduction process that was, in some cases, 174 times faster than simulations adopting the non optimized CPN model.

Experiments adopting a binary search algorithm were considered in the third study case. These experiments were performed in order to depict that different scenarios can be created by changing the probabilistic annotations for the conditional code instructions.

The fourth experiment, the BCNT algorithm, is an integrated part of Motorola's Power Stone Benchmark. It was considered to evaluate the proposed framework and to

Table 7.12: Control - comparison of execution time.

	Sample Size	Mean	Standard Deviation
Simulator	92	12411	1708
Hardware	1000	12746	1,1E-6

Table 7.13: Control - comparison of energy consumption.

	Sample Size	Mean	Standard Deviation
Simulator	92	708,6	97,5
Hardware	1000	779,4	0,5E-5

illustrate in details the proposed optimized CPN model (model after reduction process).

The last experiment, the pulse-oximeter case study, was conducted in order to apply the proposed methodology in a real-world case. This equipment is an electro-medical device responsible for measuring the oxygen saturation in the blood system using a non-invasive method.

It is also important to highlight that the energy consumption and execution time estimates obtained from the proposed model evaluation are 93% close to the respective measures obtained from the real hardware platform. Moreover, the CPN Simulator runtime provided accurate results with much smaller computation effort. Finally, the reduction process has provided meaningful results.

CONCLUSIONS

Over the last years, the time-to-market has always been demanding high complexity embedded systems in even shorter times. Furthermore, since software implementations have some advantages than hardware, due to flexibility and lower cost, nowadays, 80% of a embedded system development is related to software. Due to this fact, the design of embedded software has significantly increased its complexity. Moreover, little attention has been given to execution time and energy consumption constraints, which are issues that must be concerned, since several applications demand safety properties.

In this context, this dissertation presented an approach based on Coloured Petri nets for estimating both embedded software execution time and energy consumption. The presented work proposes a methodology which reproduces the code control flow through the composing of the proposed set of basic CPN models that represents the behavior of the instruction set microcontroller. The desired estimate metrics are obtained by simulating the CPN model.

CPN models can also be analyzed by means of reduction, where the main idea is to define the desired properties to investigate and, then, to apply a set of reduction rules by which the model is simplified. Thus, this work proposes a set of CPN reduction rules in order to transform a CPN model into an equivalent simplified model, in which all important characteristics for estimating energy consumption and execution time are preserved. The runtime evaluation adopting this reduction process was, in some cases, 174 times faster than simulations adopting the non-optimized CPN model.

In addition, this work aims to provide, in the design phase, a methodology that helps the designer by informing the energy consumption and the performance of either Assembly codes or C programs. In order to accomplish this, a set CPN models have been proposed to reproduce each instruction behavior of an ARM7-based microcontroller. It is important to state that the concept of the proposed methodology can be applied to other families of processors.

This work also provides a simulation infrastructure of integrated tools that allows the automatic translation of a compiled code into a CPN Model, such that non-specialized users do not need to interact directly with the Petri net formalism. Hence, ALUPAS, a unified environment for estimating energy consumption and execution time, has been developed in order to provide such functionalities in which system design complexity is considerably reduced and inconsistencies related to non-functional requirements are detected earlier without great difficulty.

ALUPAS adopts a stochastic discrete event simulation, in which complex systems and different control flow scenarios can be easily evaluated. In order to represent these scenarios, the designer just changes the probabilistic annotation values associated to the conditional instructions. The estimate results are stored and can be compared with other simulation results. Furthermore, being able to estimate the performance and energy consumption of a system is important because if such requirements are not satisfied, the system designers can make changes in a very early stage of the design, thereby saving both time and money. Hence, ALUPAS can provide important insights to the designer about the battery lifetime as well as parts of the application that needs optimization.

ALUPAS is composed of some components such as Binary CPN Compiler, that automatically generates a CPN model through a compiled code and the proposed basic CPN models, and the CPN Simulator, a tool developed to evaluate the proposed CPN models since the runtime simulation on CPN tools is quite time consuming when analyzing large models. Although the proposed methodology does not consider parallel tasks, the CPN Simulator was developed in such way that turns it able to evaluate this kind of system. Thus, in future works, this tool can also be adopted to simulate parallel systems.

It is worth mentioning that the estimated values obtained via simulation on CPN Tools and the simulation through CPN Simulator are quite close. However, the simulation on CPN Simulator was 300 times faster, in some cases, than simulations on CPN Tools when considering the optimized CPN Model. For sake of fairness, the reader should also bear in mind that CPN Tools is a general purpose environment, and it provides many other functionalities than the CPN Simulator does.

The presented case studies clearly show that the proposed methodology and the framework have provided meaningful results with small errors using a real-world device of center care units, called pulse-oximeter, and other customized examples. The estimates obtained from the model are 93% close to the respective measures obtained from the real hardware platform. It is also important to highlight that pieces of codes that are either energy or timing consuming were also identified. Moreover, the simulations provide accurate results with much smaller computational effort than measurements on the hardware platform.

8.1 CONTRIBUTIONS

The main contribution of this dissertation is the proposed methodology for estimating energy consumption and execution time of embedded systems in early design phases. This work extends the approach proposed by Oliveira [OJ06] by simplifying such methodology considering other microcontroller (ARM7-based instead of 8051) and dealing with C programs. Furthermore, a simulation tool was developed in order to improve the runtime evaluation of the proposed CPN models. Specific contributions are depicted as follows:

- **Framework.** A framework that considers the proposed methodology is proposed for supporting design decisions on energy consumption and performance of embedded applications in early design phases;

- **Modeling.** The proposed methodology automatically translates the embedded code by the proposed binary-CPN Compiler into a Coloured Petri net, a formal behavioral model that allows the software execution analysis. The modeling phase is based on composition of basic blocks that represents each relevant behavior of the ARM7-based instruction set microcontroller.
- **Simulating.** A stochastic evaluation approach through discrete event simulation is proposed for output data analysis. A new simulating tool was developed to simulate the specific CPN models in a much faster simulation runtime than the other generic engines available for CPN simulation.

8.2 FUTURE WORKS

It is important to stress that the proposed methodology is not restricted to ARM7 based microprocessors. Thus, as future directions, we plan to extend the proposed methodology to study other processor families. The proposed CPN Generator is a tool developed in order to help the designer to extend the proposed methodology to other processor families by the automatic creation of the basic CPN models. Moreover, the proposed methodology can be extended to cover pipeline, which is a technique adopted by processors to allow overlapping execution of multiple instructions at the same time.

Similarly, another extension can be to consider not only simple task operations, but also to estimate the energy consumption and execution time of multi-processors, in which the Coloured Petri Net has a precise formal semantic that can easily represent parallel systems. Moreover, the CPN Simulator tool was developed considering this future work and it is able to evaluate parallel and concurrent systems.

Another possible future work is related to estimate energy consumption and execution time to more complex and large systems such as data centers. Again, the formal semantic provided by the adoption of Petri net can support this more complex kind of system.

BIBLIOGRAPHY

- [ALE02] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.
- [BA97] Doug Burger and Todd M. Austin. The simpleScalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.
- [BDB⁺03] Brooks, David, Bose, Pradip, Srinivasan, Vijayalakshmi, Gschwind, and Michael K. New methodology for early-stage, microarchitecture-level power-performance analysis of microprocessors. *IBM Journal of Research and Development*, 2003.
- [Bea01] Pradip Bose and et al. Power-efficient design: Modeling and optimizations. *Tutorial of IEEE International Symposium on Circuits and Systems*, 2001.
- [Ber86] Gérard Berthelot. Checking properties of nets using transformation. In *Advances in Petri Nets 1985, covers the 6th European Workshop on Applications and Theory in Petri Nets-selected papers*, pages 19–40, London, UK, 1986. Springer-Verlag.
- [BK02] Falko Bause and Pieter S. Kritzinger. *Stochastic Petri Nets: An Introduction to the Theory*. 2002.
- [BKY98] Frank Burns, Albert Koelmans, and Alexandre Yakovlev. Analysing superscalar processor architectures with coloured petri nets. *International Journal on Software Tools for Technology Transfer*, 2:182–191, 1998.
- [BKY00] Frank Burns, Albert Koelmans, and Alexandre Yakovlev. Wcet analysis of superscalar processors using simulation with coloured petri nets. *Real-Time Syst.*, 18(2-3):275–288, 2000.
- [BL04] R. Barreto and R. Lima. A novel approach for off-line multiprocessor scheduling in embedded hard real-time systems. *Design Methods And Applications For Distributed Embedded Systems*, 2004.
- [BM08] Mahmoud Bannaser and Csaba Andras Moritz. Power and performance tradeoffs with process variation resilient adaptive cache architectures. In *SBCCI '08: Proceedings of the 21st annual symposium on Integrated circuits and system design*, pages 123–128, New York, NY, USA, 2008. ACM.

- [Bol06] G. Bolch. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons, Inc, 2nd edition, 2006.
- [BTM00] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattach: a framework for architectural-level power analysis and optimizations. *SIGARCH Comput. Archit. News*, 28(2):83–94, 2000.
- [Chu04] C.A. Chung. *Simulation Modeling Handbook: A Practical Approach*. CRC Press, 2004.
- [CM08] G. Callou and P. Maciel. Alupas software. <http://www.cin.ufpe.br/~grac/alupas>, 2008.
- [CMA⁺08a] G. Callou, P. Maciel, E. Andrade, B. Nogueira, and E. Tavares. A coloured petri net based approach for estimating execution time and energy consumption in embedded systems. In *SBCCI '08: Proceedings of the 21st annual symposium on Integrated circuits and system design*, pages 134–139, New York, NY, USA, 2008. ACM.
- [CMA⁺08b] G. Callou, P. Maciel, E. Andrade, B. Nogueira, and E. Tavares. Estimation of energy consumption and execution time in early phases of design lifecycle: an application to biomedical systems. *Electronics Letters*, 44(23):1343–1344, November 2008.
- [CMA⁺08c] G. Callou, P. Maciel, E. Andrade, B. Nogueira, E. Tavares, and M. Oliveira. A formal approach for estimating embedded system execution time and energy consumption. In *PATMOS '08: Proceedings of the 18th International Workshop on Power and Timing Modeling, Optimization and Simulation*, 2008.
- [CMC⁺08] E. Carneiro, P. Maciel, G. Callou, T. Tavares, and B. Nogueira. Mapping sysml state machine diagram to time petri net for analysis and verification of embedded real-time systems with energy constraints. In *ENICS '08: Proceedings of the 2008 International Conference on Advances in Electronics and Micro-electronics*, pages 1–6, Washington, DC, USA, 2008. IEEE Computer Society.
- [cpn07] Cpn tools version 2.2.0. <http://wiki.daimi.au.dk/cpntools/cpntools.wiki>, 2007.
- [CTM08] G. Callou, E. Tavares, and P. Maciel. Modcs - tools. http://www.modcs.org/?page_id=15, 2008.
- [dHAW⁺07] J.A. de Holanda, J. Assumpcao, D.F. Wolf, E. Marques, and J.M.P. Cardoso. On adapting power estimation models for embedded soft-core processors. *Industrial Embedded Systems, 2007. SIES '07. International Symposium on*, pages 345–348, July 2007.

- [ET93] B. Efron and R. Tibshirani. *An Introduction to the Bootstrap*. Chapman and Hall, 1993.
- [Fur00] S. Furber. *Arm System-On-Chip Architecture*. Addison-Wesley, 2000.
- [GKK⁺08] Lei Gao, Kingshuk Karuri, Stefan Kraemer, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. Multiprocessor performance estimation using hybrid simulation. In *DAC '08: Proceedings of the 45th annual conference on Design automation*, pages 325–330, New York, NY, USA, 2008. ACM.
- [Hal07] A. Hall. Realising the Benefits of Formal Methods. *Journal of Universal Computer Science*, 13(5):669–678, 2007.
- [Har08] Robert Harper. *Programming in Standard ML*. Carnegie Mellon University, 2008.
- [Her02] Ulrich Herzog. Formal methods for performance evaluation. pages 1–37, 2002.
- [HH08] Giancarlo C. Heck and Roberto A. Hexsel. The performance of pollution control victim cache for embedded systems. In *SBCCI '08: Proceedings of the 21st annual symposium on Integrated circuits and system design*, pages 46–51, New York, NY, USA, 2008. ACM.
- [HJS91] Peter Huber, Kurt Jensen, and Robert M. Shapiro. Hierarchies in coloured petri nets. In *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets*, pages 313–341, London, UK, 1991. Springer-Verlag.
- [HZDS95] Charlie X. Huang, Bill Zhang, An-Chang Deng, and Burkhard Swirski. The design and implementation of powermill. In *ISLPED '95: Proceedings of the 1995 international symposium on Low power design*, pages 105–110, New York, NY, USA, 1995. ACM.
- [IKV01] M. Irwin, M. Kandemir, and N. Vijaykrishnan. Simplepower: A cycle-accurate energy simulator. *IEEE CS Technical Committee on Computer Architecture Newsletter*, 2001.
- [Jai91] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley - Interscience, 1991.
- [Jen94] Kurt Jensen. An introduction to the theoretical aspects of coloured petri nets. In *A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium*, pages 230–272, London, UK, 1994. Springer-Verlag.
- [Jen95] Kurt Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use, vol. 2*. Springer-Verlag, London, UK, 1995.

- [JKW07] Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *Int. J. Softw. Tools Technol. Transf.*, 9(3):213–254, 2007.
- [JNM⁺06] Meuse N. O. Junior, Silvino Neto, Paulo Maciel, Ricardo Lima, Angelo Ribeiro, Raimundo Barreto, Eduardo Tavares, and Frederico Braga. Analyzing software performance and energy consumption of embedded systems by probabilistic modeling: An approach based on coloured petri nets. *Petri Nets and Other Models of Concurrency - ICATPN 2006*, 4024/2006:261–281, 2006.
- [Joh06] L. John. *Performance Evaluation and Benchmarking*. CRC Press, 2006.
- [KCJ98] Lars M. Kristensen, Soren Christensen, and Kurt Jensen. The practitioner’s guide to coloured petri nets. *International Journal on Software Tools for Technology Transfer*, 2:98–132, 1998.
- [KCNL08] V. Konstantakos, A. Chatzigeorgiou, S. Nikolaidis, and T. Laopoulos. Energy consumption estimation in embedded systems. *IEEE Transactions on Instrumentation and Measurement*, 57(3):797–804, 2008.
- [kei08] Keil software version 3.33. <https://www.keil.com>, 2008.
- [Lee02] E. A. Lee. *Embedded software*, volume 56. 2002.
- [LFTM97] Mike Tien-Chien Lee, Masahiro Fujita, Vivek Tiwari, and Sharad Malik. Power analysis and minimization techniques for embedded dsp software. *IEEE Trans. Very Large Scale Integr. Syst.*, 5(1):123–135, 1997.
- [LK99] Averill M. Law and David M. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill Higher Education, 1999.
- [LSJM01] J. Laurent, E. Senn, N. Julien, and E. Martin. High Level Energy Estimation for DSP Systems. *Proc. Int. Workshop on Power And Timing Modeling, Optimization and Simulation PATMOS*, pages 311–316, 2001.
- [Luc71] Henry Lucas, Jr. Performance evaluation and monitoring. *ACM Comput. Surv.*, 3(3):79–91, 1971.
- [man03] Lpc2106/2105/2104 user manual, philips electronics. http://www.nxp.com/acrobat_download/usermanuals/UM_LPC2106_2105_2104_1.pdf, 2003.
- [Mar03] Peter Marwedel. *Embedded System Design*. 2003.
- [MAR07] OMG MARTE. Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE), Beta1. 2007.

- [Mea00] *Measuring computer performance: a practitioner's guide*. Cambridge University Press, New York, NY, USA, 2000.
- [men08] Mentor. <http://www.mentor.com/>, 2008.
- [MF76] P. Merlin and D. J. Faber. Recoverability of communication protocols: Implications of a theoretical study. *IEEE Transactions on Communications*, 24(9):1036–1043, Sept. 1976.
- [MLC96] P.R.M. Maciel, R.D. Lins, and P.R.F. Cunha. Introdução às Redes de Petri e Aplicações. *X Escola de Computação, Campinas, SP*, 1996.
- [Moo00] Gordon E. Moore. Cramming more components onto integrated circuits. pages 56–59, 2000.
- [MRRJ07] A. Muttreja, A. Raghunathan, S. Ravi, and N. Jha. Automated energy/performance macromodeling of embedded software. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26:2229–2256, 2007.
- [Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
- [NKN02] S. Nikolaidis, N. Kavvadias, and P. Neofotistos. Instruction level power models for embedded processors. Technical report, IST-2000-30093/EASY Project, Deliv. 21, Dec. 2002.
- [NLHC03] E. Németh, R. Lakner, K.M. Hangos, and I.T. Cameron. Hierarchical cpn model-based diagnosis using hazop knowledge. Research report scl-009/2003, Process Control Research Group - Research Report SCL-009/2003, 2003.
- [NN73] JD Noe and GJ Nutt. Macro E-Nets for Representation of Parallel Systems. *IEEE Transactions on Computers*, 31(9):718–727, 1973.
- [NN02] N.K.S. Nikolaidis and P. Neofotistos. Instruction-level Power Measurement Methodology. *Electronics Lab, Physics Dept., Aristotle University of Thessaloniki, Greece, March*, 2002.
- [Oel00] Bengt Oelmann. Asynchronous and mixed synchronous/asynchronous design techniques for low power. *Proceedings of KTH*, 2000.
- [OJ06] M. Oliveira Júnior. *Estimativa do Consumo de Energia Devido ao Software: Uma abordagem em Redes de Petri Coloridas*. PhD thesis, Centro de Informática, Universidade Federal de Pernambuco, 2006.
- [Pet62] C. A. Petri. *Kommunikation mit Automaten*. PhD Dissertation, Darmstadt University, Germany, 1962.

- [RJ98] J.T. Russell and M.F. Jacome. Software power estimation and optimization for high performance, 32-bit embedded processors. In *ICCD '98: Proceedings of the International Conference on Computer Design*, page 328, Washington, DC, USA, 1998. IEEE Computer Society.
- [RJ03] Jeffrey T. Russell and Margarida F. Jacome. Architecture-level performance evaluation of component-based embedded systems. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 396–401, New York, NY, USA, 2003. ACM.
- [SBVR08] Jürgen Schnerr, Oliver Bringmann, Alexander Viehl, and Wolfgang Rosenstiel. High-performance timing simulation of embedded software. In *DAC '08: Proceedings of the 45th annual conference on Design automation*, pages 290–295, New York, NY, USA, 2008. ACM.
- [Sim08] SimpleScalar llc. <http://www.simplescalar.com/>, 2008.
- [SLJM04] E. Senn, J. Laurent, N. Julien, and E. Martin. SoftExplorer: estimation, characterization and optimization of the power and energy consumption at the algorithmic level. *Proc of PATMOS Conference*, pages 342–351, 2004.
- [SVM01] Alberto Sangiovanni-Vincentelli and Grant Martin. Platform-based design and software design methodology for embedded systems. *IEEE Des. Test*, 18(6):23–33, 2001.
- [Sys07] OMG SysML. Systems Modeling Language (SysML) Specification final report. *Object Management Group*, 2007.
- [Tav06] Eduardo Tavares. A time petri net based approach for software synthesis in hard real-time embedded systems with multiple processors. Master's thesis, Centro de Informática, Universidade Federal de Pernambuco, 2006.
- [TM08] E. Tavares and P. Maciel. Amalghma tool. <http://www.cin.ufpe.br/~eagt/tools/>, 2008.
- [TMS⁺07] E. Tavares, P. Maciel, B. Silva, M. Oliveira, and R. Rodrigues. Modelling and scheduling hard real-time biomedical systems with timing and energy constraints. *Electronics Letters*, 43(19):1015–1017, 2007.
- [TMSO08] E. Tavares, P. Maciel, B. Silva, and M.N. Oliveira. Hard real-time tasks' scheduling considering voltage scaling, precedence and exclusion relations. *Information Processing Letters*, 2008.
- [TMW94] V. Tiwari, S. Malik, and A. Wolfe. Power Analysis of Embedded Software: A First Step Towards Software Power Minimization. *Readings in Hardware/Software Co-Design*, 1994.

- [vdAvHR00] WMP van der Aalst, KM van Hee, and HA Reijers. Analysis of discrete-time stochastic petri nets. *Statistica Neerlandica*, 54(2):237–255, 2000.
- [VLM⁺03] R.A. Vinter, W. Lisa, L.H. Michael, et al. CPN Tools for Editing, Simulating, and Analysing Coloured Petri Net. *Proceedings of Applications and Theory of Petri Nets*, pages 23–27, 2003.
- [Wel02] Lisa Wells. *Performance Analysis using Coloured Petri Nets*. PhD thesis, Department of Computer Science, University of Aarhus, 2002.
- [Wil01] T. Wilmshurtz. *An Introduction to the Design of Small-scale Embedded Systems*. 2001.
- [Yea98] Gary K. Yeap. *Practical low power digital VLSI design*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.
- [YVKI00] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The design and use of simplepower: A cycleaccurate energy estimation tool. *Proceedings of Design Automation Conference*, 2000.

APPENDIX A

A VALIDATION PROCESS

Metrics estimations are subject to accuracy and precision errors among others. Confidence intervals have been widely adopted to quantify the precision of the simulation estimates. Since there is noise in any measurement, it is necessary to adopt a technique for determining whether random fluctuations in the measurements are actually significant or not.

In order to validate the simulation results, an approach has been adopted to determine whether the estimates values compared with the hardware measurements have or have not a statistically significant differences. One of the simplest approach for comparing alternative evaluation is to determine whether the confidence intervals for the two data sets of measurements being compared overlap [Mea00]. If they do, then there is no evidence to state that there is a significant difference between them at the specified confidence degree. If they do not overlap, however, it is possible to conclude that those data sets are significantly different.

It is important to state that when the confidence intervals do not overlap, it is not possible to say with complete assurance that there actually is a real difference between the alternatives. It is only possible to say that there is no reason to believe that there is not a difference. There is still the probability, however, that the differences observed were simply due to the random fluctuations of the alternatives.

A.1 NONCORRESPONDING MEASUREMENTS

The measurements are noncorresponding or unpaired when there is not a corresponding number of the measurements made to compare two different systems. In this context, as there is no correspondence or pairing between the measurements, it is necessary to adopt an approach that computes the means \bar{x}_1 and \bar{x}_2 , and the standard deviations, s_1 and s_2 , for each set of measurements separately. Then it is computed the difference of the means, $\bar{x} = \bar{x}_1 - \bar{x}_2$.

Next, it is computed the standard deviation of that difference of mean values through Equation A.1. This equation is the sum of the standard deviations of each set of measurements, fittingly weighted by the total number of measurements in each set.

$$S_x = \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}} \quad (\text{A.1})$$

In order to find the confidence interval (c_1, c_2) for the difference of the means, Equation A.2 and A.3 are performed. At the confidence level chosen, there is no significant difference between the two sets of measurements if the resulting confidence interval includes “0”. On the other side, if the confidence interval (c_1, c_2) does not includes “0”, there will still the probability that the differences observed were due simply to random fluctuations in the both measurements. Although this type of ambiguous conclusion is often not very satisfying, it is unfortunately the best that can be done given the statistical nature of such measurements.

$$c_1 = \bar{x} - Z_{1-\alpha/2, n_{df}} S_x \quad (\text{A.2})$$

$$c_2 = \bar{x} + Z_{1-\alpha/2, n_{df}} S_x \quad (\text{A.3})$$

When at least 30 measurements have been made for each system (both $n_1 \geq 30$ and $n_2 \geq 30$), the number of degrees of freedom in the t distribution is performed through the Equation A.4. On the other hand, either $n_1 \leq 30$ or $n_2 \leq 30$ the number of degrees of freedom in this case is computed by Equation A.5.

$$n_{df} = n_1 + n_2 - 2 \quad (\text{A.4})$$

$$n_{df} = \frac{\left(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right)^2}{\frac{(s_1^2/n_1)^2}{(n_1-1)} + \frac{(s_2^2/n_2)^2}{(n_2-1)}} \quad (\text{A.5})$$

APPENDIX B

THE SIMULATION ALGORITHM

A simulation algorithm has been adopted in order to perform both energy consumption and execution time estimates. Figure B.1 depicts this algorithm, in which the reader should observe that the simulation can be finished by the stop criteria evaluation (see Section 6.4.4) or by overcoming the maximum number of replications. In case the simulation is finished through overcoming the adopted value for maximum number of replications, there is no guarantee that the desired confidence degree is obtained.

```
Algorithm Simulation  
Input  $nMaxReplication, nMinReplication, iterations$ : non-negative integers;  
         $desiredEnergy, desiredTime, iConf$ : non-negative reals;  
Output  $energyConsumption, executiontime$ : non-negative reals;  
1 initialize statistical counters and lists;  
2 repeat  
3   update eventList;  
4   {fire enabled transition with shorter time}  
5   fire readyTransition;  
6    $currentTime \leftarrow transition.Time$ ;  
7    $currentCost \leftarrow transition.Cost$ ;  
8   if codeFinished then  
9      $iTime \leftarrow currentTime$ ;  
10     $iCost \leftarrow currentCost$ ;  
11    inc iteration;  
12     $currentTime \leftarrow 0.0$ ;  
13     $currentCost \leftarrow 0.0$ ;  
14    {completed one replication}  
15    if iteration = nIterations then  
16       $timeAnalysis.add \leftarrow mean(iTime)$ ;  
17       $costAnalysis.add \leftarrow mean(iCost)$ ;  
18      inc currentReplication;  
19      {finished the simulation of initial replications}  
20      if currentReplication  $\geq nMinReplications$  then  
21        {compute absolutePrecision for iTime and iCost (Equation 6.2)}  
22        if  $absTime \leq desiredTime$  and  $absEnergy \leq desiredEnergy$   
23        then  
24          {stop the simulation}  
25          return ( $mean(timeAnalysis), mean(costAnalysis)$ );  
26        else  
27          {compute i replication needed (Equation 6.3)}  
28           $nMinReplications \leftarrow i$ ;  
29          {restart the statistical counters}  
30          iteration  $\leftarrow 0$ ;  
31           $iCost \leftarrow null$ ;  
32           $iTime \leftarrow null$ ;  
32 until currentReplication  $\leq NMaxReplication$ 
```

Figure B.1: Simulation Algorithm

The simulation algorithm receives as input a confidence degree, the desired precisions for both energy consumption and execution time, the number of iterations, the maxi-

mum and minimum number of replications. Moreover, the output of this algorithm are the energy consumption and execution time estimates which their respective standard deviation and errors.

The simulation algorithm starts by initializing: (i) statistical counters, variables used for storing statistical information about system performance and energy consumption, (ii) auxiliary lists for the metrics, adopted to contain the simulation results in order to perform the statistical calculation, and (iii) event lists, lists that contain the events that may or may not happen.

Afterwards, a loop is started by the event list being updated - line 3. The next event (transition) occurs - line 4. Thus, statistical counters - lines 6 and 7 - are updated. Considering that the transition fired represents the last code instruction, the current iteration results are stored into lists - lines 9 and 10 - that contain the execution time values and energy consumption of each iteration. Next, it is incremented the number of simulated iterations and current statistical counters are reset - lines 11 - 13. In case the number of iterations executed corresponds to the replication number, the mean values of the lists that contain the replication results are stored into the replication lists - lines 16 and 17. Next, it is incremented the number of replications executed - line 18. When the minimum number of replications is executed, it is calculated the *absolute precision* for both energy and time samples - lines 20 and 21. These values are compared with the desired precisions (variables *desiredTime* and *desiredEnergy*) - line 22. In case the computed results are smaller than the desired ones, the simulation is finished - line 24. Otherwise, the simulation continues by calculating the number of replication needed - line 27. Lines 29 - 31 restart the iteration variables. Finally, line 32 is responsible for finishing the simulation in case the number of replications overcomes the maximum number previous configured.

BINARY-CPN COMPILER CLASS DIAGRAM

Figure C.1 overviews the binary-CPN Compiler class diagram implementation. The *XMLParser* class generates the *XMLProb* file, in which the probabilistic values of conditional instructions are present. Afterwards, the *parserARM7* class identifies each instruction code to create an internal representation. Next, the *instructionReaderARM7* class utilizes the *factoryARM7* class to read each internal instruction representation and to create different instructions types. Afterwards, the *instructionReaderARM7* class considers the *modelFactory* class in order to construct the CPN model that represents the code behavior. It is important to highlight that an internal instruction representation is created to cope with the needed information for the *simGenerator* class. The *parserARM7* has also a method that performs reduction rules into the created evaluation models. Finally, *simGenerator* class creates a XML file (*Sim* file), compatible with CPN Simulator, that represents the code control flow.

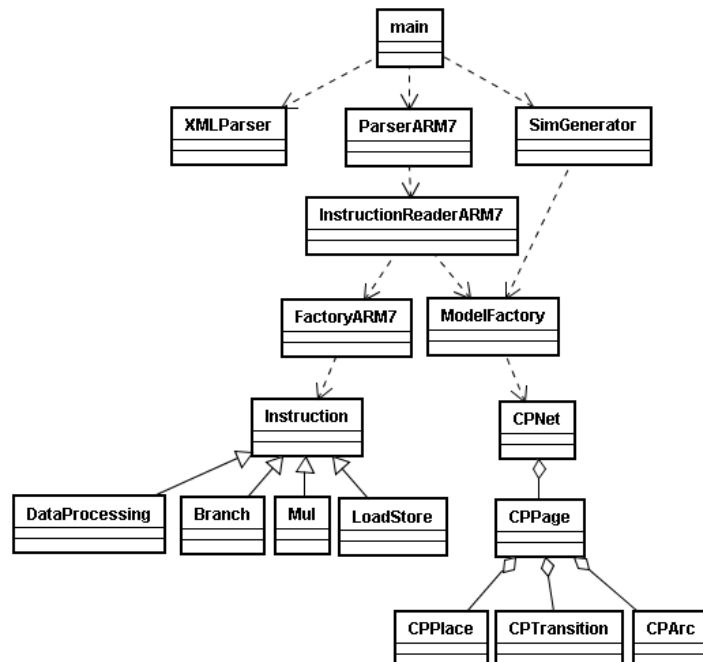


Figure C.1: Binary-CPN Compiler class diagram.

CPN SIMULATOR CLASS DIAGRAM

Considering the implementation of the CPN Simulator, Figure D.1 depicts its class diagram. *SimParser* class creates an internal element representation to transitions, places, arcs and tokens considering the *Sim* file information. This class also sets the stop criteria parameters, such as confidence interval, desired energy and time errors. Afterwards, *Net* class connects the created elements in order to generate the net. *Driver* class evaluates this net. The *analysis* class is adopted to compute statistical methods related to the execution time and energy consumption estimates. The simulation continues according to the stop criteria evaluation (see Section 6.4.4).

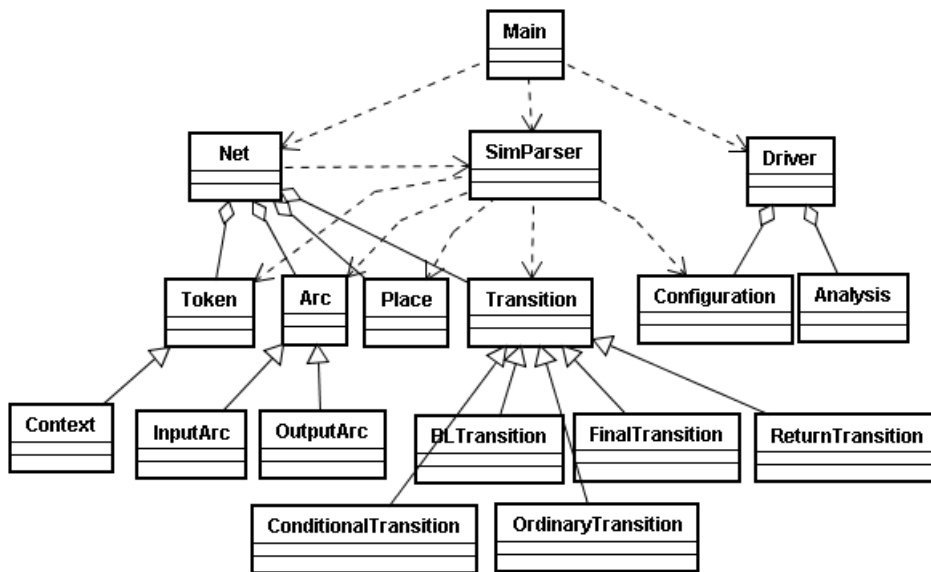
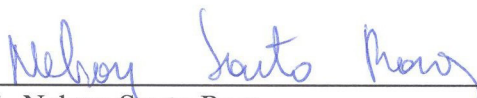


Figure D.1: CPN Simulator class diagram.

This volume has been typeset in L^AT_EX with the UFPET_{hesis} class (www.cin.ufpe.br/~paguso/ufpethesis).

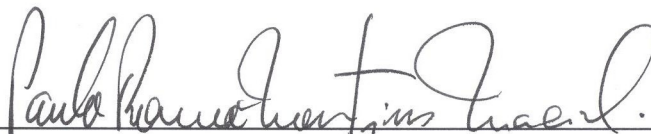
Dissertação de Mestrado apresentada por **Gustavo Rau de Almeida Callou** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título “**Energy Consumption and Execution Times Estimation of Embedded System Applications**”, orientada pelo **Prof. Paulo Romero Martins Maciel** e aprovada pela Banca Examinadora formada pelos professores:



Prof. Nelson Souto Rosa
Centro de Informática / UFPE



Prof. Ricardo José de Paiva Britto Salgueiro
Departamento de Computação e Estatística / UFS



Prof. Paulo Romero Martins Maciel
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 16 de fevereiro de 2009.



Prof. FRANCISCO DE ASSIS TENÓRIO DE CARVALHO
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.