Linköping Studies in Science and Technology
Dissertation No. 920

Verification and Scheduling Techniques for Real-Time Embedded Systems

Luis Alejandro Cortés

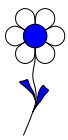
Linköpings universitet
INSTITUTE OF TECHNOLOGY

Department of Computer and Information Science
Linköping University, S-581 83 Linköping, Sweden

Linköping, 2005

ISBN 91-85297-21-6 ISSN 0345-7524
Printed by UniTryck
Linköping, Sweden

To Lina María



Abstract

Embedded computer systems have become ubiquitous. They are used in a wide spectrum of applications, ranging from household appliances and mobile devices to vehicle controllers and medical equipment.

This dissertation deals with design and verification of embedded systems, with a special emphasis on the real-time facet of such systems, where the time at which the results of the computations are produced is as important as the logical values of these results. Within the class of real-time systems two categories, namely hard real-time systems and soft real-time systems, are distinguished and studied in this thesis.

First, we propose modeling and verification techniques targeted towards hard real-time systems, where correctness, both logical and temporal, is of prime importance. A model of computation based on Petri nets is defined. The model can capture explicit timing information, allows tokens to carry data, and supports the concept of hierarchy. Also, an approach to the formal verification of systems represented in our modeling formalism is introduced, in which model checking is used to prove whether the system model satisfies its required properties expressed as temporal logic formulas. Several strategies for improving verification efficiency are presented and evaluated.

Second, we present scheduling approaches for mixed hard/soft real-time systems. We study systems that have both hard and soft real-time tasks and for which the quality of results (in the form of utilities) depends on the completion time of soft tasks. Also, we study systems for which the quality of results (in the form of rewards) depends on the amount of computation allotted to tasks. We introduce quasi-static techniques, which are able to exploit at low cost the dynamic slack caused by variations in actual execution times, for maximizing utilities/rewards and for minimizing energy.

Numerous experiments, based on synthetic benchmarks and realistic case studies, have been conducted in order to evaluate the proposed approaches. The experimental results show the merits and worthiness of the techniques introduced in this thesis and demonstrate that they are applicable on real-life examples.

Acknowledgments

It has been a long path towards the completion of this dissertation and many people have along the way contributed to it. I wish to express my sincere gratitude to them all.

First of all, and not merely because it is the convention, I want to thank my thesis advisors, Zebo Peng and Petru Eles. Not only have they given me invaluable support and guidance throughout my doctoral studies but also, and more importantly, they have always encouraged me the way a good friend does.

Many thanks to former and present members of the Embedded Systems Laboratory (ESLAB), and in general to all my colleagues in the Department of Computer and Information Science at Linköping University, for providing a friendly working environment.

My friends, those with whom I shared the "wonder years" as well as those I met later, have in many cases unknowingly made this journey more pleasant. The simple fact that I can count on them anytime is a source of joy.

I wish to acknowledge the financial support of CUGS—Swedish National Graduate School of Computer Science—and SSF—Swedish Foundation for Strategic Research—via the STRINGENT program. This work would not have been possible without their funding.

I am thankful to my family for their constant support. Especially I owe my deepest gratitude and love to Dad and Mom for being my great teachers. And finally, Lina María, my beloved wife, deserves the most special recognition for her endless patience, encouragement, and love. The least I can do is to dedicate this work to her.

Luis Alejandro Cortés
Linköping, January 2005

Table of Contents

Part I

Preliminaries

Chapter 1

Introduction

The semiconductor industry has evolved at an incredible pace since the conception of the transistor in 1947. Such a high pace of development could hardly be matched by other industries. If the automotive industry, often used as a comparison point, had advanced at the same rate as the the semiconductor industry, an automobile today would cost less than a cent, weigh less than a gram, and consume less than 10^{-5} liters per hundred kilometers [Joh98].

The amazing evolution of the electronic technologies still continues at the present time, progressing rapidly and making it possible to fabricate smaller and cheaper electronic devices that perform more complex functions at higher speeds. And yet, beyond the technological achievements, the so-called electronic revolution has opened up new challenges and frontiers in the human capabilities.

The first electronic digital computer, ENIAC (Electronic Numerical Integrator And Computer), contained 18.000 vacuum tubes and hundreds of thousands of resistors, capacitors, and inductors [McC99]. It weighed over 30 tons, took up 167 m^2, and consumed around 175 kW of power. ENIAC could perform 5.000 addition operations per second. Today, a state-of-the-art microprocessor contains around 50 million transistors and can execute billions of additions per second, in an area smaller than a thumbnail, consuming a couple of tens of watts.

The remarkable development of computer systems is partly due to the advances in semiconductor technology. But also, to a great extent, new paradigms and design methodologies have made it possible to design and deploy devices with such extraordinary computation capabilities. Innovative design frameworks have thus exploited the rapid technological progress in order to create more powerful computer systems at lower cost.

In the mid-1960s Gordon Moore made the observation that the number

of transistors on an integrated circuit would double approximately every 18 months [Moo65]. This exponential growth has held since the invention of the integrated circuit in 1959. While fundamental physical limits will eventually be reached, the trend predicted by Moore's law is expected to continue for at least one more decade.

With semiconductor technology advancing rapidly, it is nowadays feasible to fully integrate complex systems on a single chip. However, the capabilities to design such systems are not growing at the same rate as the capabilities to fabricate them. The semiconductor technology is simply outpacing the design capabilities, which creates consequently a productivity gap: every year, the number of available raw transistors increases by 58% while the designer's capabilities to design them grows only by 21% [Kah01]. This drives the need for innovative design frameworks that help to bridge this gap. And these design paradigms will play an increasing role in sustaining the development of computer-based systems.

Digital computer-based systems have become ubiquitous. These systems have various types of applications including automotive and aircraft controllers, cellular phones, network switches, household appliances, medical devices, and consumer electronics. Typical households in developed countries have, for instance, desktop or laptop computers, scanners, printers, fax machines, TV sets, DVD players, stereo systems, video game consoles, telephones, food processors, microwave ovens, washing machines, vacuum cleaners, refrigerators, video and photo cameras, and personal digital assistants, among many others. Each one of the devices listed above has at its heart at least one microprocessor controlling or implementing the functions of the system. This widespread use of digital systems has been boosted by the enormous computation capabilities that are nowadays available at very low cost.

In the devices mentioned above, except desktops and laptops, the computer is a component within a larger system. In such cases the computer is embedded into a larger system, hence the term *embedded systems*. In the case of the desktop and the laptop, the computer is the system itself. Desktop and laptop computers, as well as workstations and mainframes, belong to the class of *general-purpose systems*. They can be programmed to implement any computable function. Embedded systems, as opposed to general-purpose systems, implement a dedicated set of functions that is particular to the application.

The vast majority of computer systems is today used in embedded applications. Less than 2% of the billions of microprocessors sold annually are actually used in general-purpose systems [Tur02]. The number of embedded systems in use will continue to grow rapidly as they become more pervasive

in our everyday life.

1.1 Motivation

Embedded systems are used in a large number of applications and the spectrum of application fields will continue to expand. Despite the variety and diversity of application areas, embedded systems are characterized by a number of generic features:

- Embedded systems are intended for particular applications, that is, one such system performs a set of dedicated functions that is well defined in advance and, once the system is deployed, the functionality is not modified during normal operation. The digital controller of a home appliance such as a vacuum cleaner, for example, is designed and optimized to perform that particular function and will serve the same function during the operational life of the product.
- Due to the interaction with their environment, embedded systems must fulfill strict temporal requirements, typically in the form of deadlines. Thus the term *real-time system* is frequently used to emphasize this aspect. The correct behavior of these systems depends not only on the logical results of the computations but also on the time at which these results are produced [But97]. For instance, the ABS (Anti-locking Brake System) controller in modern vehicles must acquire data from the sensors, process it, and output the optimal force to be applied to the brake pads, in a time frame of few milliseconds subject to a maximal-reaction time constraint.
- For many embedded applications, especially mobile devices, energy consumption is an essential design consideration. For this type of devices, it is crucial to use as efficiently as possible the energy provided by an exhaustible source such as a battery.
- Embedded systems are generally heterogeneous, that is, they include hardware as well as software elements. The hardware components, such as application-specific integrated circuits and field-programmable gate arrays, provide the speed and low-power dimension needed in many applications. The software components, such as programmable processors, give the flexibility for extending the system with increased functionality and adding more features to new generations of the product.
- Embedded systems have high requirements in terms of reliability and correctness. Errors in safety-critical applications, such as avionics and automotive electronics, may have disastrous consequences. Therefore safety-critical applications demand techniques that ensure the reliable and correct operation of the system.

The design of systems with such characteristics is a difficult task. Embedded systems must not only implement the desired functionality but must also satisfy diverse constraints (power and energy consumption, performance, correctness, size, cost, flexibility, etc.) that typically compete with each other. Moreover, the ever increasing complexity of embedded systems combined with small time-to-market windows poses great challenges for the designers.

A key issue in the design of embedded systems is the simultaneous optimization of competing design metrics [VG02]. The designer must explore several alternatives and trade off among the different design objectives, hence the importance of sound methodologies that allow the systematic exploration of the design space. It is through the application of rigorous and systematic techniques that the design of embedded systems can be carried out in an efficient and productive manner.

Due to the diversity of application areas, design techniques must be tailored to the particular class of embedded systems. The type of system dictates thus the most relevant design goals. The design methods must consequently exploit the information characteristic of the application area. In portable, battery-powered devices, such as mobile phones, for example, energy consumption is one of the most important design considerations, which might not necessarily be the case for a home appliance such as a washing machine.

In this thesis we place special emphasis on the real-time aspects of embedded systems. Depending on the consequences of failing to meet a deadline, real-time systems are usually categorized in two classes, namely *hard real-time systems* and *soft real-time systems* [Kop97], [Lap04]. Basically, a hard real-time system is one in which a deadline miss may lead to a catastrophic failure. A soft real-time system is one in which a deadline miss might degrade the system performance but poses no serious risk to the system or environment integrity.

We propose in this thesis techniques targeted towards these two classes of systems. In Part II (Modeling and Verification) we address hard real-time systems, where correctness, both logical and temporal, is of prime importance. In Part III (Scheduling Techniques) we discuss several approaches for mixed hard/soft real-time systems, which may include parts that are loosely constrained, for example, soft tasks for which deadline misses can be tolerated at the expense of quality of results.

In the next section we elaborate on a generic design flow, indicating the particular steps to which the techniques proposed in Parts II and III can be applied.

1.2 Generic Design Flow

This section presents a generic design flow for embedded systems. We highlight the parts of such a flow that are directly addressed in this thesis in order to demarcate the different contributions of our work.

A generic design flow is shown in Figure 1.1. The process starts with a *system specification* which describes the functionality of the system as well as performance, cost, energy, and other constraints of the intended design. Such a specification states the functionality without giving implementation details, that is, it specifies what the system must do without making assumptions about how it must be implemented. In the design of embedded systems many different specification languages are available [GVNG94]. The system specification can be given using a variety of languages that range from natural language to languages with strong formal semantics, although it is preferable to specify the system using a language with precise semantics as this allows the use of tools that assist the designer from the initial steps of the design flow.

Once the system specification is given, the designer must come up with a *system model* that captures aspects from the functional part of the specification as well as non-functional attributes. Modeling is a fundamental aspect of the design methodology. A model of computation with precise mathematical meaning is essential for carrying out in a systematic way the different steps from specification to implementation: a sound representation allows the designer to capture unambiguously the functionality of the system as well as non-functional constraints, verify the correctness of the system, reason formally about the refinement steps during the synthesis process, and use CAD tools throughout the different stages of the design flow [SLSV00]. As detailed in Section 2.1, a large variety of modeling formalisms have been used for representing embedded systems. These models of computation encompass diverse styles, attributes, and application domains.

Then, once the system model has been obtained, the designer must decide the underlying architecture of the system, that is, select the type and number of components as well as the way to interconnect them. This stage is known as *architecture selection*. The components may include various processor cores, custom modules, communication elements such as buses and buffers, I/O interfaces, and memories. The architecture selection step, as well as subsequent design steps, corresponds to the exploration of the design space in search of solutions that allow the implementation of the desired functionality and the satisfaction of the non-functional constraints.

Based on the selected *system architecture*, in the *partitioning and mapping* phase, the tasks or processes of the system model are grouped and

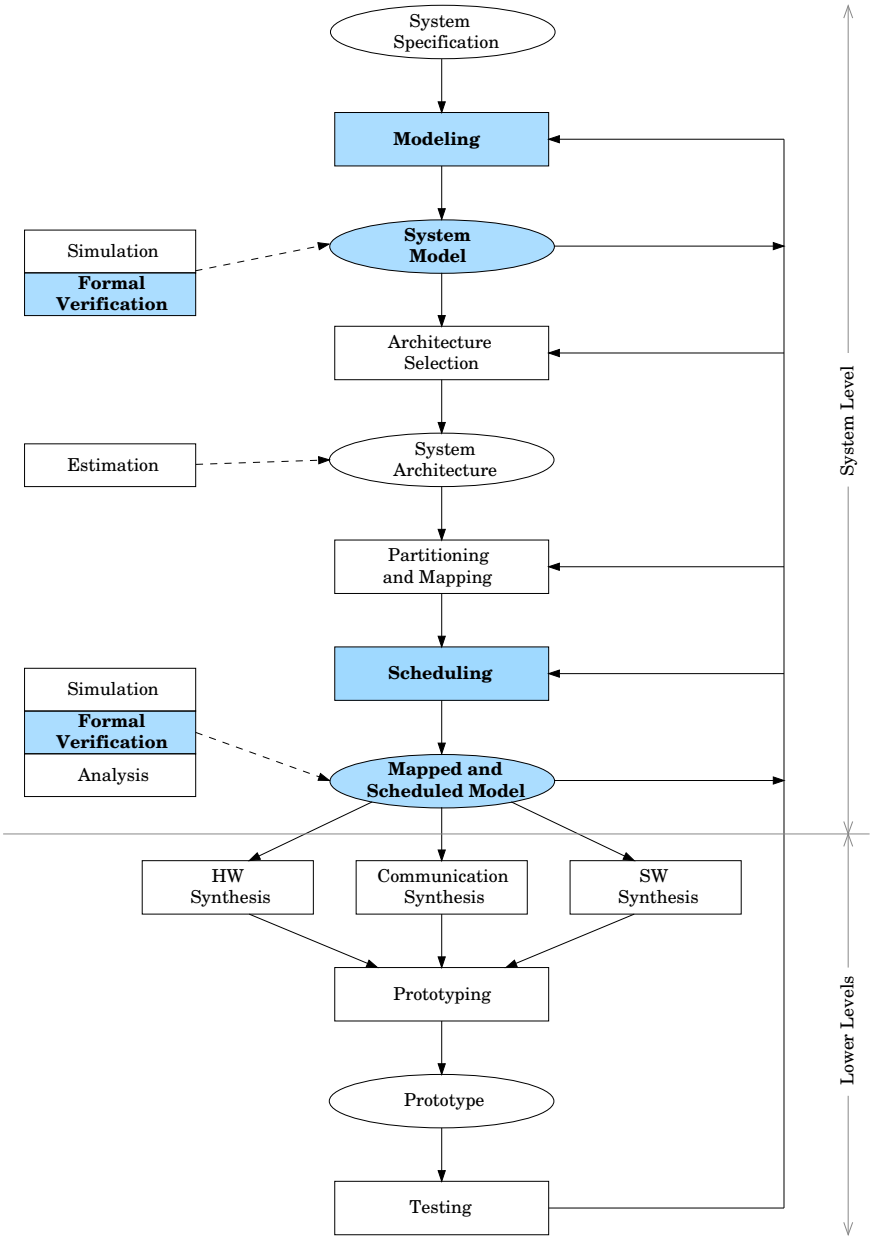


Figure 1.1: A generic design flow for embedded systems

mapped onto the selected components. Hardware/software partitioning in the context of embedded systems refers to the physical partition of system functionality into custom integrated circuits (hardware components) and programmable processors (software components) [DeM97]. The partition of

the system into hardware and software has particularly a great impact on the cost and the performance of the resulting design.

Once it has been determined what parts are to be implemented on which components, certain decisions concerning the execution order of tasks or their priorities have to be taken. This design step is called *scheduling*. Since several computational tasks have typically to share the same processing resource, as dictated by the mapping of tasks onto processing elements, it is necessary to make a temporal distribution of each of the resources among the tasks mapped onto it, in such a way that precedence and timing constraints are fulfilled. This includes selecting the criteria (scheduling policies) for assigning the computational resources to the various tasks as well as the set of rules that determine the order in which tasks are executed [But97]. Moreover, power and energy considerations have become very important in the design of embedded systems and must be taken into account during the system-level design phases, especially during the scheduling phase: modern processors allow the supply voltage to be dynamically varied, which has a direct impact on the energy consumption as well as on the performance (reducing the supply voltage has the benefit of quadratic energy savings at the expense of approximately linear performance loss). Thus the voltage level is a new dimension that has to be taken into consideration in the exploration of solutions that satisfy the timing constraints. Therefore scheduling concerns not only the execution order of tasks but also the selection of voltages at which such tasks run.

At this point, the model must include the information about the design decisions taken in the stages of architecture selection, partitioning, and scheduling (*mapped and scheduled model*).

The design process continues further with the so-called *lower-level* phases, including *SW synthesis*, *HW synthesis*, and *communication synthesis*, and later with *prototyping*. Once the *prototype* has been produced, it must thoroughly be checked during the *testing* phase in order to find out whether it functions correctly.

The design flow includes iterations, where it is sometimes necessary to go back to previous steps because some of the design goals cannot be fulfilled, and therefore it is needed to explore different design alternatives by revising decisions taken earlier in precedent design phases.

Simulation can be used to validate the design at different stages of the process and, therefore, can be carried out at different levels of accuracy [Row94], [FFP04]. Validation of modern embedded systems has become an enormous challenge because of their size and complexity. The nature of simulation (generating functional test vectors, executing the model of system according to these vectors, and observing the behavior of the system

under the test stimuli) means that is not feasible to validate large designs by exhaustive simulation. In spite of the advances in simulation techniques, the fraction of system behavior that can be covered by simulation is declining [Dil98]. *Formal Verification* has emerged as a viable alternative to the problem of verifying complex systems. Formal verification methods embody analytical and mathematical techniques to prove properties about a design. Formal verification can also be performed at different points of the design flow, for example, on the initial system model or on the mapped and scheduled model. Formal verification methods have grown mature and can overcome some of the limitations of traditional validation methods like simulation. Formal verification, however, does not provide a universal solution and there still exist issues to be tackled in this field. Nonetheless, formal verification has proved to be a powerful tool when it comes to the goal of designing correct systems.

Our work contributes to various system-level phases of the flow presented above. The main contributions of this thesis correspond to the parts highlighted in Figure 1.1 as shaded boxes/ovals and are detailed in Section 1.3. Part II deals with modeling and formal verification and and Part III addresses the scheduling phase.

1.3 Contributions

Different classes of real-time embedded systems and different stages of their design cycle are addressed in this thesis. The main contributions of this dissertation are summarized as follows:

Modeling and Verification

- We define a sound model of computation. PRES+, short for Petri Net based Representation for Embedded Systems, is an extension to the classical Petri nets model that captures explicitly timing information, allows systems to be represented at different levels of granularity, and improves expressiveness by allowing tokens to carry information. Furthermore, PRES+ supports the concept of hierarchy [CEP99], [CEP00a], [CEP00c], [CEP01], [CEP03].
- We propose an approach to the formal verification of systems represented in PRES+. Model checking is used to automatically determine whether the system model satisfies its required properties expressed in temporal logics. A systematic procedure to translate PRES+ models into timed automata is presented so that it is possible to make use of existing model checking tools [CEP00b], [CEP00c], [CEP01], [CEP03].

- Strategies for improving verification efficiency are introduced. First, correctness-preserving transformations are applied to the system model in order to obtain a simpler, yet semantically equivalent, one. Thus the verification effort can be reduced. Second, by exploiting the structure of the system model and, in particular, information concerning the degree of concurrency of the system, the translation of PRES+ into timed automata can be improved and, therefore, verification complexity can considerably be reduced [CEP01], [CEP02b], [CEP02a], [CEP03].

Scheduling Techniques

- We present scheduling algorithms for real-time systems that include both hard and soft tasks, considering that there exist utility functions that capture the relative importance of soft tasks as well as how the quality of results is affected when a soft deadline is missed. Static scheduling techniques are proposed and evaluated. Also, a quasi-static scheduling approach, aimed at exploiting the dynamic time slack caused by tasks finishing ahead of their worst-case execution time, is introduced [CEP04c], [CEP04b], [CEP04a], [CEP05b].
- We propose quasi-static techniques for assigning voltages and allotting amount of computation in real-time systems with energy considerations, for which it is possible to trade off performance for energy consumption and also to trade off precision for timeliness. First, methods for maximizing rewards (value obtained as a function of the amount of computation allotted to tasks in the system) subject to energy constraints are presented. Second, techniques for minimizing energy consumption subject to constraints in the total reward are introduced [CEP05a].

It must be observed that in this thesis modeling and verification, on the one hand, and scheduling techniques, on the other hand, are treated separately in Parts II and III respectively. However, modeling and verification as well as scheduling are constituent parts of an integral design flow. Thus in a practical design flow, as the one presented in Figure 1.1, verification and scheduling are not completely independent parts. As it was mentioned previously, verification can be performed at different stages of the design flow, among which also after the scheduling phase; that is, once the decisions related to scheduling are taken, it is important to verify the correctness of the system. Also, scheduling information affects significantly the complexity of the verification process: on the one hand, the system model grows larger because information related to the task execution order must be included in the representation; on the other hand, the temporal distribution of computational resources among tasks makes the state space much smaller because

the degree of parallelism and non-determinism is reduced.

Nonetheless, although our verification and scheduling techniques are addressed separately, they all are targeted towards real-time embedded systems, which are the type of systems we focus on in this dissertation. A distinguishing feature, that is common to the techniques presented in Parts II and III—and also differentiates our work from approaches discussed previously in the literature, is the consideration of varying execution times for tasks in the form of time intervals.

1.4 List of Papers

Parts of the contents of this dissertation have been presented in the following papers:

[CEP99] L. A. Cortés, P. Eles, and Z. Peng. **A Petri Net based Model for Heterogeneous Embedded Systems**. In *Proc. NORCHIP Conference*, Oslo, Norway, pages 248–255, 1999.

[CEP00a] L. A. Cortés, P. Eles, and Z. Peng. **Definitions of Equivalence for Transformational Synthesis of Embedded Systems**. In *Proc. Intl. Conference on Engineering of Complex Computer Systems*, Tokyo, Japan, pages 134–142, 2000.

[CEP00b] L. A. Cortés, P. Eles, and Z. Peng. **Formal Coverification of Embedded Systems using Model Checking**. In *Proc. Euromicro Conference (Digital Systems Design)*, Maastricht, The Netherlands, volume 1, pages 106–113, 2000.

[CEP00c] L. A. Cortés, P. Eles, and Z. Peng. **Verification of Embedded Systems using a Petri Net based Representation**. In *Proc. Intl. Symposium on System Synthesis*, Madrid, Spain, pages 149–155, 2000.

[CEP01] L. A. Cortés, P. Eles, and Z. Peng. **Hierarchical Modeling and Verification of Embedded Systems**. In *Proc. Euromicro Symposium on Digital System Design*, Warsaw, Poland, pages 63–70, 2001.

[CEP02a] L. A. Cortés, P. Eles, and Z. Peng. **An Approach to Reducing Verification Complexity of Real-Time Embedded Systems**. In *Proc. Euromicro Conference on Real-Time Systems (Work-in-progress Session)*, Vienna, Austria, pages 45–48, 2002.

[CEP02b] L. A. Cortés, P. Eles, and Z. Peng. **Verification of Real-Time Embedded Systems using Petri Net Models and Timed Automata**. In *Proc. Intl. Conference on Real-Time Computing Systems and Applications*, Tokyo, Japan, pages 191–199, 2002.

[CEP03] L. A. Cortés, P. Eles, and Z. Peng. **Modeling and Formal Verification of Embedded Systems based on a Petri Net Representation**. *Journal of Systems Architecture*, 49(12-15):571–598, December 2003.

[CEP04a] L. A. Cortés, P. Eles, and Z. Peng. **Combining Static and Dynamic Scheduling for Real-Time Systems**. In *Proc. Intl. Workshop on*

Software Analysis and Development for Pervasive Systems, Verona, Italy, pages 32–40, 2004. Invited paper.

[CEP04b] L. A. Cortés, P. Eles, and Z. Peng. **Quasi-Static Scheduling for Real-Time Systems with Hard and Soft Tasks**. In *Proc. Design, Automation and Test in Europe Conference*, Paris, France, pages 1176–1181, 2004.

[CEP04c] L. A. Cortés, P. Eles, and Z. Peng. **Static Scheduling of Monoprocessor Real-Time Systems composed of Hard and Soft Tasks**. In *Proc. Intl. Workshop on Electronic Design, Test and Applications*, Perth, Australia, pages 115–120, 2004.

[CEP05a] L. A. Cortés, P. Eles, and Z. Peng. **Quasi-Static Assignment of Voltages and Optional Cycles for Maximizing Rewards in Real-Time Systems with Energy Constraints**. 2005. Submitted for publication.

[CEP05b] L. A. Cortés, P. Eles, and Z. Peng. **Quasi-Static Scheduling for Multiprocessor Real-Time Systems with Hard and Soft Tasks**. 2005. Submitted for publication.

1.5 Thesis Overview

This thesis is divided into four parts and consists of ten chapters. The first part presents the introductory discussion and general overview of the thesis. The second part presents our modeling and verification techniques for hard real-time systems, where correctness plays a primordial role. The third part addresses approaches that target hard/soft real-time systems, where the soft part provides the flexibility for trading off quality of results with other design metrics. The fourth and last part concludes the dissertation by summarizing its main points and discussing ideas for future work. The structure of the rest of this thesis, together with a brief description of each chapter, is as follows:

Part II: Modeling and Verification

- Chapter 2 (Related Approaches) addresses related work in the areas of modeling and formal verification.
- Chapter 3 (Design Representation) defines a model of computation based on Petri nets, PRES+, that is used as design representation in Part II. Several notions of equivalence are introduced and, based on them, a concept of hierarchy for PRES+ models is presented.
- Chapter 4 (Formal Verification of PRES+ Models) introduces an approach to the formal verification of systems modeled using PRES+. A translation procedure from PRES+ into the input formalism of available verification tools is proposed.

- Chapter 5 (Improving Verification Efficiency) discusses two techniques for ameliorating the verification process: first, a transformation-based approach that seeks to simplify the system model is presented; second, a technique that exploits information on the degree of concurrency of the system is introduced.

Part III: Scheduling Techniques

- Chapter 6 (Introduction and Related Approaches) gives a brief introduction to Part III and presents related approaches in the areas of scheduling for systems composed of hard and soft real-time tasks as well as scheduling under the framework of the Imprecise Computation model.
- Chapter 7 (Systems with Hard and Soft Real-Time Tasks) addresses the problem of scheduling for real-time systems with hard and soft tasks. Static scheduling solutions for both monoprocessor and multiprocessor systems are discussed. The problem is also addressed under the framework of a quasi-static approach with the goal of improving the quality of results by exploiting the dynamic time slack.
- Chapter 8 (Imprecise-Computation Systems with Energy Considerations) studies real-time systems (under the Imprecise Computation model) for which it is possible to trade off precision for timeliness as well as performance for energy. Two different approaches, in which deadlines, energy, and reward are considered under a unified framework, are addressed in this chapter.

Part IV: Conclusions and Future Work

- Chapter 9 (Conclusions) summarizes the main points of the proposed techniques and presents the thesis conclusions.
- Chapter 10 (Future Work) discusses possible directions in our future research based on the results presented in this dissertation.

Part II

Modeling and Verification

Chapter 2

Related Approaches

Modeling is an essential part of any design methodology. Many models of computation have been proposed in the literature to represent computer systems. These models encompass a broad range of styles, characteristics, and application domains. Particularly in embedded systems design, a variety of models have been developed and used as system representation.

In the field of formal verification, many approaches have also been proposed. There is a significant amount of theoretical results and many of them have been applied in realistic settings. However, approaches targeted especially to embedded systems and considering systematically real-time issues have until now not been very common.

This chapter presents related work in the areas of modeling and formal verification for embedded systems.

2.1 Modeling

Many different models of computation have been proposed to represent embedded systems [ELLSV97], [LSVS99], [Jan03], including extensions to finite state machines, data flow graphs, communicating processes, and Petri nets, among others. This section presents various models of computation for embedded systems reported in the literature.

Finite State Machines

The classical Finite State Machine (FSM) representation [Koz97] is probably the most well-known model used for describing control systems. One of the disadvantages of FSMs, though, is the exponential growth of the number of states that have to be explicitly captured in the model as the system

complexity rises. A number of extensions to the classical FSM model have been suggested in different contexts.

Codesign Finite State Machines (CFSMs) are the underlying model of computation of the POLIS design environment [BCG+97]. A CFSM is an extended FSM including a control part and a data computation part [CGH+93]. Each CFSM behaves synchronously from its own perspective. A system is composed of a number of CFSMs that communicate among themselves asynchronously through signals, which carry information in the form of events. Such a semantics provides a GALS model: Globally Asynchronous (at the system level) and Locally Synchronous (at the CFSM level). CFSMs are mainly intended for control-oriented systems.

In order to make it more suitable for data-oriented systems, the FSM model has been extended by introducing a set of internal variables, thus leading to the concept of *Finite State Machine with Datapath (FSMD)* [GR94]. The transition relation depends not only on the present state and input signals but also on a set of internal variables. Although the introduction of variables in the FSMD model helps to reduce the number of represented states, the lack of explicit support for concurrency and hierarchy is a drawback because the state explosion problem is still present.

The *FunState* model [STG+01] consists of a network and a finite state machine. The so-called network corresponds to the data intensive part of the system. The network is composed of storage units, functions, and arcs that relate storage units and functions. Data is represented by valued tokens in the storage units. The activation of functions in the network is controlled by the state machine. In the FunState model, an arbitrary number of components (network and FSM) can be arranged in a hierarchical structure.

Statecharts extend FSMs by allowing hierarchical composition and concurrency [Har87]. A particular state can be composed of substates which means that being in the higher-level state is interpreted as being in one of the substates. In this way, Statecharts avoids the potential for state explosion by permitting condensed representations. Furthermore, timing is specified by using linear inequalities in the form of time-outs. The problem with Statecharts is that the model falls short when representing data-oriented systems.

Dataflow Graphs

Dataflow graphs [DFL72] are very popular for modeling data-dominated systems. Computationally intensive systems might conveniently be represented by a directed graph where the nodes describe computations and the arcs capture data dependencies between tasks. The computations are executed only when the required operands are available and the operations behave as

atomic operations without side effects. However, the conventional dataflow graph model is inadequate for representing the control unit of systems.

Dataflow Process Networks are mainly used for representing signal processing systems [LP95]. Programs are specified by directed graphs where nodes (actors) represent computations and arcs (streams) represent sequences of data. Processing is done in series of iterated firings in which an actor transforms input data into output ones. Dataflow actors have firing rules to determine when they must be enabled and then execute a specific operation. A special case of dataflow process networks is *Synchronous Data Flow (SDF)* where the actors consume and produce a fixed number of data tokens in each firing because of their static rules [LM87].

Conditional Process Graph (CPG) is an abstract graph representation introduced to capture both the data- and the control-flow of a system [EKP+98]. A CPG is a directed, acyclic, and polar graph, consisting of nodes as well as simple and conditional edges. Each node represents a process which can be assigned to one of the processing elements. The graph has two special nodes (source and sink) used to represent the first and the last task. The model allows each process to be characterized by an execution time and a guard which is the condition necessary to activate that process. In this way, it is possible to capture control information in a dataflow graph.

Communicating Processes

Several models have been derived from Hoare's Communicating Sequential Processes (CSP) [Hoa85]. In CSP, systems are composed of processes that communicate with each other through unidirectional channels using a synchronizing protocol.

SOLAR is a model of computation based on CSP, where each process corresponds to an extended FSM, similar to Statecharts, and communication is performed by dedicated units [JO95]. Thus communication is separated from the rest of the design so that it can be optimized and reused. By focusing on efficient implementation and refinement of the communication units, SOLAR is best suited for communication-driven design processes. SOLAR is the underlying model of the COSMOS design environment [IAJ94].

Interacting Processes are also derived from CSP and consist of independent interacting sequential processes [TAS93]. The communication is performed through channels but, unlike CSP, there exist additional primitives that permit unbuffered transfer and synchronization without data.

The *Formal System Design (ForSyDe)* methodology [SJ04] uses a design representation where the system is modeled as network of concurrent processes which communicate with each other through signals (ordered sequences of events). The ForSyDe methodology provides a framework for the

stepwise design process of embedded systems where a high-level functional model is transformed through a number of refinements into an synthesizable implementation model.

Petri Nets

Modeling of systems using Petri Nets (PN) has been applied widely in many fields of science [Pet81], [Mur89]. The mathematical formalism developed over the years, which defines its structure and firing rules, has made Petri nets a well-understood and powerful model. A large body of theoretical results and practical tools have been developed around Petri nets. Several drawbacks, however, have been pointed out, especially when it comes to modeling embedded systems:

- Petri nets tend to become large, even for relatively small systems. The lack of hierarchical composition makes it difficult to specify and understand complex systems using the conventional model
- The classical PN model lacks the notion of time. However, as pointed out in Section 1.1, time is an essential factor in embedded applications.
- Regular Petri nets lack expressiveness for formulating computations as long as tokens are considered as "black dots".

Several formalisms have independently been proposed in different contexts in order to overcome the problems cited above, such as introducing the concepts of hierarchy [Dit95], time [MF76], and valued tokens [JR91]. Some of the PN formalisms previously proposed to be used in embedded systems design are described in the following.

Petri net based Unified REpresentation (PURE) is a model with data and control notation [Sto95]. It consists of two different, but closely related, parts: a control unit and a computational/data part. Timed Petri nets with restricted transition rules are used to represent the control flow. Hardware and software operations are represented by datapaths and instruction dependence graphs respectively. Hierarchy is however not supported by this model.

In *Colored Petri Nets (CPN)*, tokens may have "colors", that is, data attached to them [Jen92]. The arcs between transitions/places have expressions that describe the behavior associated to transitions. Thus transitions describe actions and tokens carry values. The CPN model permits hierarchical constructs and a strong mathematical theory has been built up around it. The problem of CPN is that timing is not explicitly defined in the model. It is possible to treat time as any other value attached to tokens but, since there is no semantics given for the order of firing along the time horizon, timing inconsistencies can happen. Approaches using CPN that target particularly

embedded systems include [Ben99] and [HG02].

Dual Transition Petri Nets (DTPN) are another model of computation where control and data flow are tightly linked [VAH01]. There are two types of transitions (control and data transitions) as well as two types of arcs (control and data flow arcs). Tokens may have values which are affected by the firing of data transitions. Control transitions may have guards that depend on token values so that guards constitute the link between the control and data domains. The main drawback of DTPN is that it lacks an explicit notion of time. Nor does it support hierarchical constructs.

Several other models extending Petri nets have been used in embedded systems design [MBR99], [SLWSV99], [ETT98]. A more detailed discussion about Petri nets and their fundamentals is presented in Section 3.1.

Our design representation, defined in Chapter 3, differs from other modeling formalisms in the area of Petri nets in several aspects: our model includes an explicit notion of time; it supports hierarchical composition; it can capture both data and control aspects of the system. Some models of computation introduced previously in the literature address separately the points mentioned above. The key difference is that our modeling formalism combines these aspects in a single design representation.

2.2 Formal Verification

Though formal methods are not yet very common in embedded systems design, several verification approaches have been proposed recently. Some of them are presented in this section. We focus on the more automatic approaches like model checking since these are closely related to our work. However, it is worthwhile to mention that theorem proving [Fit96], [Gal87] is a well-established approach in the field of formal verification. This section presents related work in the area of verification of embedded systems.

The verification of Codesign Finite State Machines (CFSMs) has been addressed in [BHJ$^+$96]. In this approach, CFSMs are translated into traditional state automata in order to make use of automata theory techniques. The verification task is to check whether all possible sequences of inputs and outputs of the system satisfy the desired properties (specification). The sequences that meet the requirements constitute the language of another automaton. The problem is then reduced to checking language containment between two automata. Verification requires showing that the language of the system automaton is contained in the language of the specification automaton. The drawback of the approach is that it is not possible to check explicit timing properties, only order of events.

Most of the research on continuous-time model checking is based on the timed automata model [Alu99]. Efficient algorithms have been proposed to verify systems represented as timed automata [ACD90], [LPY95]. Also tools, such as UPPAAL [UPP] and KRONOS [KRO], have successfully been developed and tested on realistic examples. However, timed automata are a fairly low-level representation, especially during the system-level phases of the design flow.

Based on the hybrid automata model [ACHH93], model checking techniques have also been developed [AHH96], [Hsi99]. Arguing that the hardware and software parts of the system have different time scales, Hsiung's approach uses different clock rates to keep track of the time in the hardware and software parts [Hsi99]. It must be mentioned that while the linear hybrid automata model is more expressive than timed automata, the problem of model checking of hybrid automata is harder than the one based on timed automata.

The FunState model can formally be verified by using model checking, as discussed in [STG+01]. The proposed verification strategy is based on an auxiliary representation, very much alike a FSM, into which the FunState model is translated. The set of required properties are expressed as Computation Tree Logic (CTL) formulas. However, no quantitative timing behavior can be reasoned based on CTL.

Model checking based of the Dual Transition Petri Nets (DTPN) model has been addressed in [VAHC+02]. The DTPN model is translated into a Kripke structure and then BDD-based symbolic model checking is used to determine the truth of Linear Temporal Logic (LTL) and CTL formulas. Since there is no explicit notion of time in DTPN, however, timing requirements cannot be verified.

The approaches cited above show that model checking is gaining popularity in the system design community and different related areas are being explored. There has been, for example, a recent interest in defining coverage metrics in terms of the incompleteness of a set of formal properties, that is, in finding out up to which extent a system is considered correct if all the defined properties are satisfied [FPF+03]. This and other works show that special attention is being paid to using formal methods in the system-level phases of the design flow.

In this thesis (Chapters 4 and 5) we propose an approach to the formal verification of embedded systems, which differs from the related work presented in this section in several regards: we deal with quantitative timing properties in our verification approach; and the underlying model of computation allows representations at different levels of granularity so that formal verification is possible at several abstraction levels.

Chapter 3

Design Representation

From the initial conception of a computer system to its final implementation, several design activities must be accomplished. These activities require an abstraction, that is a model, of the system under design. The model captures the characteristics and properties of the system that are relevant for a particular design activity [Jan03].

Along the design flow, different design decisions are taken and these are progressively incorporated into the model of the system. Therefore, an essential issue of any systematic methodology aiming at designing computer systems is the underlying model of computation.

We introduce in this chapter a model of computation called PRES+ (Petri Net based Representation for Embedded Systems). PRES+ captures relevant features of embedded systems and can consequently be used as design representation when devising such systems. PRES+ is an extension to the classical Petri nets model that captures explicitly timing information, allows systems to be represented at different levels of granularity, and improves the expressiveness by allowing tokens to carry information.

It can be mentioned at this point that system specifications given in the functional programming language Haskell [Has] can automatically be translated into the design representation introduced in this chapter by using a systematic translation procedure defined in [CPE01] and assisted by a software tool developed by our research group. This illustrates that our modeling formalism can indeed be used as part of a realistic design flow for embedded systems.

First we present a number of basic concepts related to the theory of Petri nets that will facilitate the presentation of our ideas in subsequent sections and chapters. Then we formally define our model of computation PRES+ and present several modeling examples.

3.1 Fundamentals of Petri Nets

Petri nets are a model applicable to many types of systems. They have been used as a graphical and mathematical modeling tool in a wide variety of application areas [Mur89]. A Petri net can be thought of as a directed bipartite graph (consisting of two types of nodes, namely places and transitions) together with an initial state called the initial marking. The classical definition of a Petri net is as follows.

Definition 3.1 A *Petri net* is a five-tuple $N = (\mathbf{P}, \mathbf{T}, \mathbf{I}, \mathbf{O}, M_0)$ where: $\mathbf{P} = \{P_1, P_2, \ldots, P_m\}$ is a finite non-empty set of *places*; $\mathbf{T} = \{T_1, T_2, \ldots, T_n\}$ is a finite non-empty set of *transitions*; $\mathbf{I} \subseteq \mathbf{P} \times \mathbf{T}$ is a finite non-empty set of *input arcs* which define the flow relation between places and transitions; $\mathbf{O} \subseteq \mathbf{T} \times \mathbf{P}$ is a finite non-empty set of *output arcs* which define the flow relation between transitions and places; and $M_0 : \mathbf{P} \to \mathbb{N}_0$ is the initial *marking*. ❑

Figure 3.1 shows an example of a Petri net where $\mathbf{P} = \{P_a, P_b, P_c, P_d\}$, $\mathbf{T} = \{T_1, T_2\}$, $\mathbf{I} = \{(P_a, T_1), (P_b, T_1), (P_b, T_2)\}$, and $\mathbf{O} = \{(T_1, P_c), (T_2, P_d)\}$. Places are graphically represented by circles, transitions by boxes, and arcs by arrows.

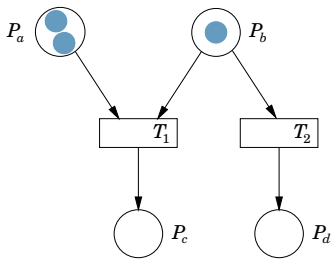


Figure 3.1: A Petri net

In the classical Petri nets model a marking $M : \mathbf{P} \to \mathbb{N}_0$ assigns to a place P a non-negative integer $M(P)$, representing the number of tokens in P. For the example shown in Figure 3.1, $M_0(P_a) = 2$, $M_0(P_b) = 1$, $M_0(P_c) = M_0(P_d) = 0$.

Definition 3.2 The *pre-set* ${}^\circ T = \{P \in \mathbf{P} \mid (P, T) \in \mathbf{I}\}$ of a transition $T \in \mathbf{T}$ is the set of *input places* of T. Similarly, the *post-set* $T^\circ = \{P \in \mathbf{P} \mid (T, P) \in \mathbf{O}\}$ of a transition $T \in \mathbf{T}$ is the set of *output places* of T. The *pre-set* ${}^\circ P$ and the *post-set* P° of a place $P \in \mathbf{P}$ are given by ${}^\circ P = \{T \in \mathbf{T} \mid (T, P) \in \mathbf{O}\}$ and $P^\circ = \{T \in \mathbf{T} \mid (P, T) \in \mathbf{I}\}$ respectively. ❑

The dynamic behavior of a Petri net is given by the change of marking which obeys the firing rule stated by the following definition.

Definition 3.3 A transition T is *enabled* if $M(P) > 0$ for all $P \in {}^\circ T$. The *firing* of an enabled transition (which changes the marking M into a new marking M') removes one token from each input place of T ($M'(P) = M(P) - 1$ for all $P \in {}^\circ T$) and adds one token to each output place of T ($M'(P) = M(P) + 1$ for all $P \in T^\circ$). ❑

The rest of the definitions presented in this section are related to the classical Petri nets model but they are also valid for our design representation PRES+. We include here those notions that are needed for the later discussion.

Definition 3.4 A marking M' is *immediately reachable* from M if there exists a transition $T \in \mathbf{T}$ whose firing changes M into M'. ❑

Definition 3.5 The *reachability set* $\mathscr{R}(N)$ of a net N is the set of all markings reachable from M_0 and is defined by:

(i) $M_0 \in \mathscr{R}(N)$;

(ii) If $M \in \mathscr{R}(N)$ and M' is immediately reachable from M, then $M' \in \mathscr{R}(N)$. ❑

Definition 3.6 Two transitions T and T' are in *conflict* if ${}^\circ T \cap {}^\circ T' \neq \emptyset$. ❑

Definition 3.7 A net N is *conflict-free* if, for all $T, T' \in \mathbf{T}$ such that $T \neq T'$, ${}^\circ T \cap {}^\circ T' = \emptyset$. ❑

Definition 3.8 A net N is *free-choice* if, for any two transitions T and T' in conflict, $|{}^\circ T| = |{}^\circ T'| = 1$. ❑

Definition 3.9 A net N is *extended free-choice* if, for any two transitions T and T' in conflict, ${}^\circ T = {}^\circ T'$. ❑

Definition 3.10 A net N is *safe* if the number of tokens in each place, for any reachable marking, is at most one. ❑

Definition 3.11 A net N is *live* if, for every reachable marking $M \in \mathscr{R}(N)$ and every transition $T \in \mathbf{T}$, there exists a marking M' reachable from M that enables T. ❑

3.2 Basic Definitions

In the following we present the formal definition of the design representation introduced in this chapter.

Definition 3.12 A *PRES+* model is a five-tuple $N = (\mathbf{P}, \mathbf{T}, \mathbf{I}, \mathbf{O}, M_0)$ where: $\mathbf{P} = \{P_1, P_2, \ldots, P_m\}$ is a finite non-empty set of *places*; $\mathbf{T} = \{T_1, T_2, \ldots, T_n\}$ is a finite non-empty set of *labeled transitions*; $\mathbf{I} \subseteq \mathbf{P} \times \mathbf{T}$ is a finite non-empty set of *input arcs*; $\mathbf{O} \subseteq \mathbf{T} \times \mathbf{P}$ is a finite non-empty set of *output arcs*; and M_0 is the initial *marking* (see Definition 3.15) with tokens carrying values and time stamps. ❑

We make use of the example shown in Figure 3.2 in order to illustrate the different definitions corresponding to our model. For this example, the set of places is $\mathbf{P} = \{P_a, P_b, P_c, P_d, P_e\}$, the set of transitions is $\mathbf{T} = \{T_1, T_2, T_3, T_4, T_5\}$, the set of input arcs is $\mathbf{I} = \{(P_a, T_1), (P_b, T_1), (P_c, T_2), (P_d, T_3), (P_d, T_4), (P_e, T_5)\}$, and the set of output arcs is $\mathbf{O} = \{(T_1, P_c), (T_1, P_d), (T_2, P_a), (T_3, P_b), (T_4, P_e), (T_5, P_b)\}$.

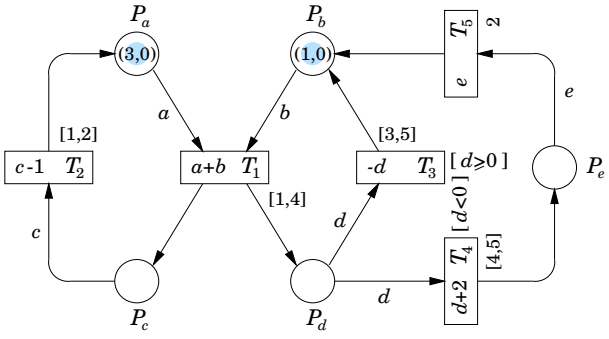


Figure 3.2: A PRES+ model

The definition of a PRES+ model (Definition 3.12) seems at first sight almost identical to that of a classical Petri net (Definition 3.1). Note, however, that PRES+ extends Petri nets in a number of ways that make such a representation suitable for the modeling of embedded systems. The coming definitions introduce those extensions.

Definition 3.13 A *complex token* in a PRES+ model is a pair $K = (v, t)$ where v is the *token value* and t is the *token time*. The type of the token value is referred to as *token type*. The token time is a non-negative real number representing the time stamp of the complex token. ❑

In the sequel, whenever it is clear that the net under consideration corresponds to a PRES+ model, *complex tokens* will simply be referred to as *tokens* and *labeled transitions* will just be referred to as *transitions*.

For the initial marking in the example net presented in Figure 3.2, for instance, in place P_a there is a token K_a with token value $v_a = 3$ and token time $r_a = 0$.

A token value may be of any type, for example boolean, integer, string, etc., or user-defined type of any complexity such as a list, a set, or any data structure. A token type is defined by the set of possible values that the token may take. We use $\boldsymbol{\zeta}$ in order to denote the set of all possible token types for a given system. For example, for a system in which token values may only be integer numbers, as it is the case of the PRES+ model shown in Figure 3.2, $\boldsymbol{\zeta} = \{\mathbb{Z}\}$.

Definition 3.14 The *type function* $\zeta : \mathbf{P} \rightarrow \boldsymbol{\zeta}$ associates every place $P \in \mathbf{P}$ with a token type. $\zeta(P)$ denotes the set of possible token values that tokens may bear in P. ❑

The set of possible tokens in place P is given by $\mathbf{K}_P \subseteq \{(v, r) \mid v \in \zeta(P) \text{ and } r \in \mathbb{R}_0^+\}$. We use $\mathbf{K} = \bigcup_{P \in \mathbf{P}} \mathbf{K}_P$ to denote the set of all tokens.

It is worth pointing out that the token type related to a certain place is fixed, that is, it is an intrinsic property of that place and will not change during the dynamic behavior of the net. For the example given in Figure 3.2, $\zeta(P) = \mathbb{Z}$ for all $P \in \mathbf{P}$, that is all places have token type *integer*. Thus the set of all possible tokens in the system is $\mathbf{K} \subseteq \{(v, r) \mid v \in \mathbb{Z} \text{ and } r \in \mathbb{R}_0^+\}$.

Definition 3.15 A *marking* M is an assignment of tokens to places of the net. The marking of a place $P \in \mathbf{P}$, denoted $M(P)$, can be represented as a multi-set[1] over $\mathbf{K}_P$. For a particular marking M, a place P is said to be *marked* iff $M(P) \neq \emptyset$. ❑

The initial marking M_0 in the net of Figure 3.2 shows P_a and P_b as the only places initially marked: $M_0(P_a) = \{(3, 0)\}$ and $M_0(P_b) = \{(1, 0)\}$, whereas $M_0(P_c) = M_0(P_d) = M_0(P_e) = \emptyset$.

Definition 3.16 All output places of a given transition have the same token type, that is, $P, Q \in T^\circ \Rightarrow \zeta(P) = \zeta(Q)$. ❑

The previous definition is motivated by the fact that there is one transition function associated to a given transition (as formally stated in Definition 3.17), so that when it fires all its output places get tokens with the same value and therefore such places must have the very same token type.

3.3 Description of Functionality

Definition 3.17 For every transition $T \in \mathbf{T}$ in a PRES+ model there exists a *transition function* $f : \zeta(P_1) \times \zeta(P_2) \times \ldots \times \zeta(P_a) \rightarrow \zeta(Q)$ associated to T, where ${}^\circ T = \{P_1, P_2, \ldots, P_a\}$ and $Q \in T^\circ$. ❑

[1] A *multi-set* or *bag* is a collection of elements over some domain in which, unlike a set, multiple occurrences of the same element are allowed. For example, $\{a, b, b, b\}$ is a multi-set over $\{a, b, c\}$.

Transition functions are used to capture the functionality associated with the transitions. They allow systems to be modeled at different levels of granularity with transitions representing simple arithmetic operations or complex algorithms. In Figure 3.2 we inscribe transition functions inside transition boxes: the transition function associated to T_1, for example, is given by $f_1(a, b) = a + b$. We use inscriptions on the input arcs of a transition in order to denote the arguments of its transition function.

Definition 3.18 For every transition $T \in \mathbf{T}$, there exist a *best-case transition delay* τ^{bc} and a *worst-case transition delay* τ^{wc}, which are non-negative real numbers such that $\tau^{\mathrm{bc}} \leq \tau^{\mathrm{wc}}$, and represent, respectively, the lower and upper limits for the execution time of the function associated to the transition. ❑

Referring again to Figure 3.2, for instance, the best-case transition delay of T_2 is $\tau_2^{\mathrm{bc}} = 1$ and its worst-case transition delay is $\tau_2^{\mathrm{wc}} = 2$ time units. Note that when $\tau^{\mathrm{bc}} = \tau^{\mathrm{wc}} = \tau$ we just inscribe the value τ close to the transition, like in the case of the transition delay $\tau_5 = 2$.

Definition 3.19 A transition $T \in \mathbf{T}$ may have a *guard* g associated to it. The guard of a transition T is a predicate $g : \zeta(P_1) \times \zeta(P_2) \times \ldots \times \zeta(P_a) \rightarrow \{0, 1\}$ where ${}^\circ T = \{P_1, P_2, \ldots, P_a\}$. ❑

The concept of guard plays an important role in the enabling rule for transitions in PRES+ models (see Definition 3.21). Note that the guard of a transition T is a function of the token values in places of its pre-set ${}^\circ T$. For instance, in Figure 3.2, $d < 0$ represents the guard

$$g_4(d) = \begin{cases} 1 & \text{if } d < 0, \\ 0 & \text{otherwise.} \end{cases}$$

3.4 Dynamic Behavior

Definition 3.20 A transition $T \in \mathbf{T}$ is *bound*, for a given marking M, iff all its input places are marked. A *binding* B of a bound transition T with pre-set ${}^\circ T = \{P_1, P_2, \ldots, P_a\}$, is an ordered tuple of tokens $B = (K_1, K_2, \ldots, K_a)$ where $K_i \in M(P_i)$ for all $P_i \in {}^\circ T$. ❑

Observe that, for a particular marking M, a transition may have different bindings. This is the case when there are several tokens in at least one of the input places of the transition. Let us consider the net shown in Figure 3.3. In this case $M(P_a) = \{(2, 0)\}$, $M(P_b) = \{(6, 0), (4, 1)\}$, and $M(P_c) = \emptyset$. For this marking, T has two different bindings $B^{\mathrm{i}} = ((2, 0), (6, 0))$ and $B^{\mathrm{ii}} = ((2, 0), (4, 1))$.

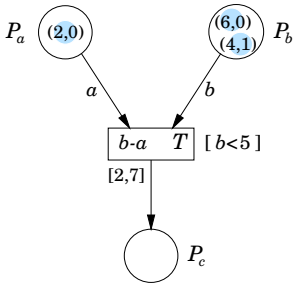


Figure 3.3: Net used to illustrate the concept of *binding*

The existence of a binding is a necessary condition for the enabling of a transition. For the initial marking of the net shown in Figure 3.2, for example, transition T_1 is bound: it has a binding $B_1 = ((3,0),(1,0))$. Since T_1 has no guard, it is enabled for the initial marking (as formally stated in Definition 3.21).

We introduce the following notation which will be useful in the coming definitions. Given a binding $B = (K_1, K_2, \ldots, K_a)$, the token value of the token K_i is denoted v_i and the token time of K_i is denoted t_i.

Definition 3.21 A bound transition $T \in \mathbf{T}$ with guard g is *enabled*, for a binding $B = (K_1, K_2, \ldots, K_a)$, if $g(v_1, v_2, \ldots, v_a) = 1$. A transition $T \in \mathbf{T}$ with no guard is *enabled* if T is bound. ❑

The transition T in the example given in Figure 3.3 has two bindings but it is enabled only for the binding $B^{\text{ii}} = ((2,0),(4,1))$, because of its guard $b < 5$. Thus, upon firing T, the tokens $(2,0)$ and $(4,1)$ will be removed from P_a and P_b respectively, and a new token will be added to P_c (see Definition 3.24).

Definition 3.22 The *enabling time* et of an enabled transition $T \in \mathbf{T}$ for a binding $B = (K_1, K_2, \ldots, K_a)$ is the time instant at which T becomes enabled. The value of et is given by the maximum token time of the tokens in the binding B, that is, $et = \max(t_1, t_2, \ldots, t_a)$. ❑

The enabling time of transition T in the net of Figure 3.3 is $et = \max(0,1) = 1$.

Definition 3.23 The *earliest trigger time* $tt^{\text{bc}} = et + \tau^{\text{bc}}$ and the *latest trigger time* $tt^{\text{wc}} = et + \tau^{\text{wc}}$ of an enabled transition $T \in \mathbf{T}$, for a binding $B = (K_1, K_2, \ldots, K_a)$, are the lower and upper time limits for the firing of T. An enabled transition $T \in \mathbf{T}$ may not fire before its earliest trigger time tt^{bc} and must fire before or at its latest trigger time tt^{wc}, unless T becomes disabled by the firing of another transition. ❑

Definition 3.24 The *firing* of an enabled transition $T \in \mathbf{T}$, for a binding $B = (K_1, K_2, \ldots, K_a)$, changes a marking M into a new marking M'. As a result of firing the transition T, the following occurs:

(i) The tokens $K_1, K_2, \ldots, K_a$ corresponding to the binding B are removed from the pre-set ${}^\circ T$, that is, $M'(P_i) = M(P_i) - \{K_i\}$ for all $P_i \in {}^\circ T$;

(ii) One new token $K = (v, t)$ is added to each place of the post-set T°, that is, $M'(P) = M(P) + \{K\}$ [2] for all $P \in T^\circ$. The token value of K is calculated by evaluating the transition function f with token values of tokens in the binding B as arguments, that is, $v = f(v_1, v_2, \ldots, v_a)$. The token time of K is the instant at which the transition T fires, that is, $t = tt$ where $tt \in [tt^{\mathrm{bc}}, tt^{\mathrm{wc}}]$;

(iii) The marking of places different from input and output places of T remains unchanged, that is, $M'(P) = M(P)$ for all $P \in \mathbf{P} \setminus {}^\circ T \setminus T^\circ$. ❑

The execution time of the function of a transition is considered in the time stamp of the new tokens. Note that, when a transition fires, all the tokens in its output places get the same token value and token time. The token time of a token represents the instant at which it was "created". The timing semantics of PRES+ makes the firing of transitions be consistent with an implicit global system time, that is, the firing of transitions occurs in an order that is in accordance to the time horizon.

In Figure 3.2, transition T_1 is the only one initially enabled (binding $((3,0),(1,0))$) so that its enabling time is $et_1 = 0$. Therefore, T_1 may not fire before 1 time units and must fire before or at 4 time units. Let us assume that T_1 fires at 2 time units: tokens $(3,0)$ and $(1,0)$ are removed from places P_a and P_b respectively, and a new token $(4,2)$ is added to both P_c and P_d. At this moment, only T_2 and T_3 are enabled (T_4 is bound but not enabled because $4 \not< 0$, hence its guard is not satisfied for the binding $((4,2))$). Note also that transition T_2 has to fire strictly before transition T_3: according to the firing rules for PRES+ nets, T_2 must fire no earlier than 3 and no later than 4 time units, while T_3 is restricted to fire in the interval $[5,7]$. Figure 3.4 illustrates a possible behavior of the PRES+ model presented in Figure 3.2.

3.5 Notions of Equivalence and Hierarchy

Several notions of equivalence for systems modeled in PRES+ are defined in this section. Such notions constitute the foundations of a framework for

[2] Observe that the multi-set sum $+$ is different from the multi-set union $\cup$. For instance, given $A = \{a, c, c\}$ and $B = \{c\}$, $A + B = \{a, c, c, c\}$ while $A \cup B = \{a, c, c\}$. An example of multi-set difference $-$ is $A - B = \{a, c\}$.

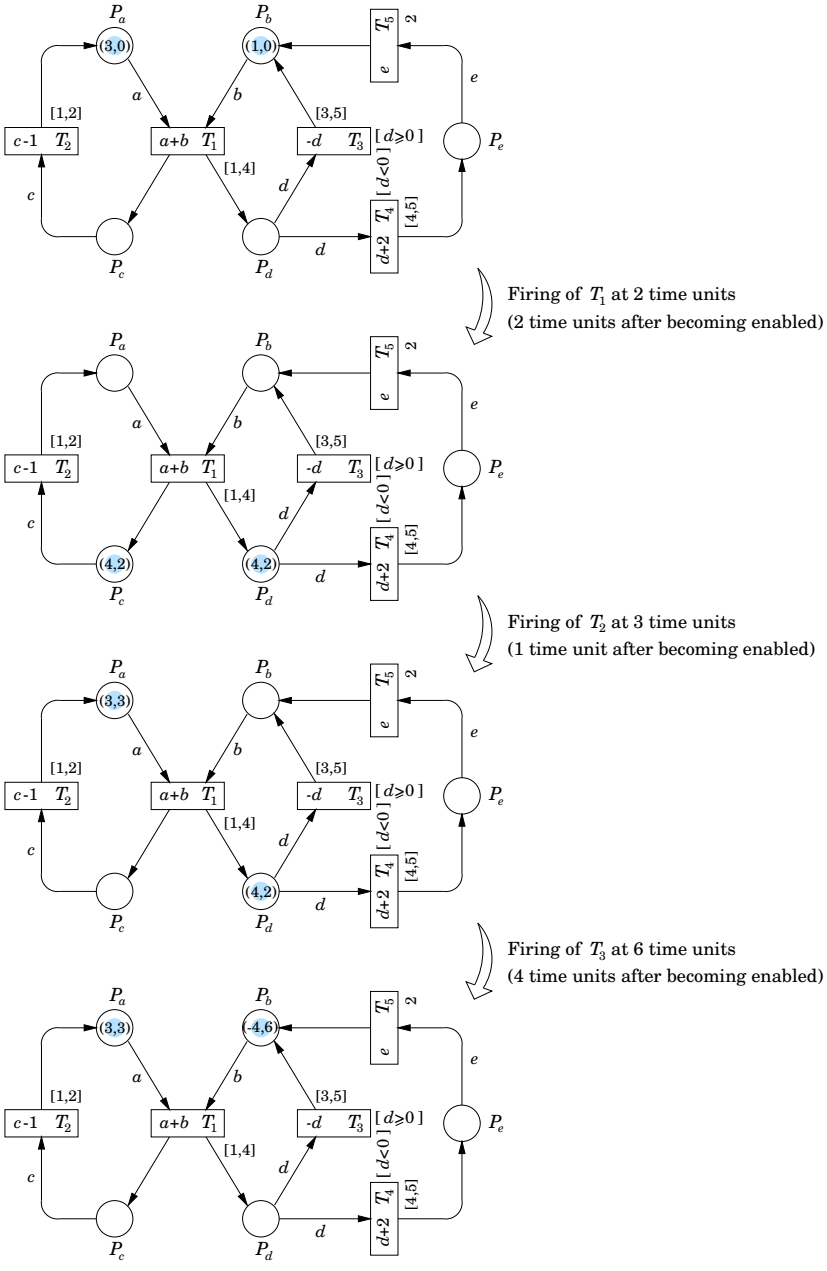


Figure 3.4: Illustration of the dynamic behavior of a PRES+ model

comparing PRES+ models. A concept of hierarchy for this design representation is also introduced. Hierarchy is a convenient way to structure the system so that modeling can be done in a comprehensible form. Hierarchi-

cal composition eases the design process when it comes to specifying and understanding large systems.

3.5.1 Notions of Equivalence

The synthesis process requires a number of refinement steps starting from the initial system model until a more detailed representation is achieved. Such steps correspond to transformations in the system model in such a way that design decisions are included in the representation.

The validity of a transformation depends on the concept of equivalence in which it is contrived. When we claim that two systems are equivalent, it is very important to understand the meaning of equivalence. Two equivalent systems are not necessarily the same but have properties that are common to both of them. A clear notion of equivalence allows us to compare systems and point out the properties in terms of which the systems are equivalent.

The following definition introduces a couple of concepts to be used when defining the notions of equivalence for systems modeled in PRES+.

Definition 3.25 A place $P \in \mathbf{P}$ is said to be an *in-port* if $(T, P) \notin \mathbf{O}$ for all $T \in \mathbf{T}$, that is, there is no transition T for which P is output place. Similarly, a place $P \in \mathbf{P}$ is said to be an *out-port* if $(P, T) \notin \mathbf{I}$ for all $T \in \mathbf{T}$, that is, there is no transition T for which P is input place. The set of in-ports is denoted **inP** while the set of out-ports is denoted **outP**. ❑

Before formally presenting the notions of equivalence, we first give an intuitive idea about them. These notions rely on the concepts of in-ports and out-ports: the initial condition to establish an equivalence relation between two nets N_1 and N_2 is that both have the same number of in-ports as well as out-ports. In this way, it is possible to define a one-to-one correspondence between in-ports and out-ports of the nets. Thus we can assume the same initial marking in corresponding in-ports and then check the tokens obtained in the out-ports after some transition firings in the nets. It is like an external observer putting in the same data in both nets and obtaining output information. If such an external observer cannot distinguish between N_1 and N_2, based on the output data he gets, then N_1 and N_2 are assumed to be "equivalent". As defined later, such a concept is called *total-equivalence*. We also define weaker concepts of equivalence in which the external observer may actually distinguish between N_1 and N_2, but still there is some commonality in the data obtained in corresponding out-ports, such as number of tokens, token values, or token times.

We introduce the following notation to be used in the coming definitions: for a given marking M_i, $m_i(P)$ denotes the number of tokens in place P, that is, $m_i(P) = |M_i(P)|$.

Definition 3.26 Two nets N_1 and N_2 are *cardinality-equivalent* iff:

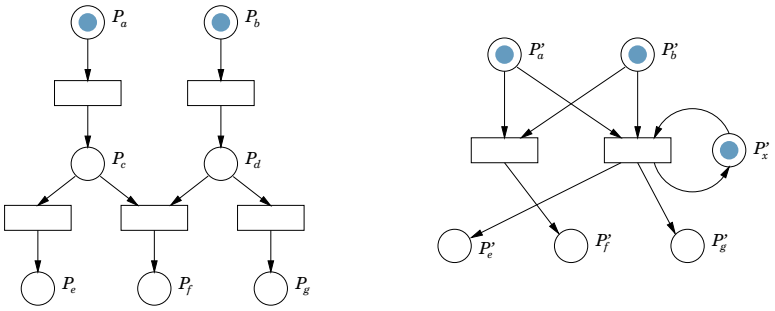
(i) There exist bijections $h_{in} : \mathbf{inP}_1 \rightarrow \mathbf{inP}_2$ and $h_{out} : \mathbf{outP}_1 \rightarrow \mathbf{outP}_2$ that define one-to-one correspondences between in(out)-ports of N_1 and N_2;

(ii) The initial markings $M_{1,0}$ and $M_{2,0}$ satisfy
$M_{1,0}(P) = M_{2,0}(h_{in}(P)) \neq \emptyset$ for all $P \in \mathbf{inP}_1$,
$M_{1,0}(Q) = M_{2,0}(h_{out}(P)) = \emptyset$ for all $Q \in \mathbf{outP}_1$;

(iii) For every $M_1 \in \mathscr{R}(N_1)$ such that
$m_1(P) = 0$ for all $P \in \mathbf{inP}_1$,
$m_1(R) = m_{1,0}(R)$ for all $R \in \mathbf{P}_1 \setminus \mathbf{inP}_1 \setminus \mathbf{outP}_1$
there exists $M_2 \in \mathscr{R}(N_2)$ such that
$m_2(P) = 0$ for all $P \in \mathbf{inP}_2$,
$m_2(R) = m_{2,0}(R)$ for all $R \in \mathbf{P}_2 \setminus \mathbf{inP}_2 \setminus \mathbf{outP}_2$,
$m_2(h_{out}(Q)) = m_1(Q)$ for all $Q \in \mathbf{outP}_1$
and vice versa. ❑

The above definition expresses that if the same tokens are put in corresponding in-ports of two cardinality-equivalent nets, then the same number of tokens will be obtained in corresponding out-ports. Let us consider the nets N_1 and N_2 shown in Figures 3.5(a) and 3.5(b) respectively, in which we have abstracted away information not relevant for the current discussion, like transition delays and token values. For these nets we have that $\mathbf{inP}_1 = \{P_a, P_b\}$, $\mathbf{outP}_1 = \{P_e, P_f, P_g\}$, $\mathbf{inP}_2 = \{P'_a, P'_b\}$, $\mathbf{outP}_2 = \{P'_e, P'_f, P'_g\}$, and h_{in} and h_{out} are defined by $h_{in}(P_a) = P'_a$, $h_{in}(P_b) = P'_b$, $h_{out}(P_e) = P'_e$, $h_{out}(P_f) = P'_f$, and $h_{out}(P_g) = P'_g$. Let us assume that $M_{1,0}$ and $M_{2,0}$ satisfy condition (ii) in Definition 3.26. A simple reachability analysis shows that there exist two cases m_1^{i} and m_1^{ii} in which the first part of condition (iii) in Definition 3.26 is satisfied: a) $m_1^{\mathrm{i}}(P) = 1$ if $P \in \{P_f\}$, and $m_1^{\mathrm{i}}(P) = 0$ for all other places; b) $m_1^{\mathrm{ii}}(P) = 1$ if $P \in \{P_e, P_g\}$, and $m_1^{\mathrm{ii}}(P) = 0$ for all other places. For each of these cases there exists a marking satisfying the second part of condition (iii) in Definition 3.26, respectively: a) $m_2^{\mathrm{i}}(P) = 1$ if $P \in \{P'_f, P'_x\}$, and $m_2^{\mathrm{i}}(P) = 0$ for all other places; b) $m_2^{\mathrm{ii}}(P) = 1$ if $P \in \{P'_e, P'_g, P'_x\}$, and $m_2^{\mathrm{ii}}(P) = 0$ for all other places. Hence N_1 and N_2 are cardinality-equivalent.

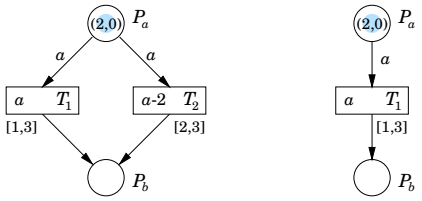
Before defining the concepts of function-equivalence and time-equivalence, let us study the simple nets N_1 and N_2 shown in Figures 3.6(a) and 3.6(b) respectively. It is straightforward to see that N_1 and N_2 fulfill the conditions established in Definition 3.26 and therefore are cardinality-equivalent. However, note that N_1 may produce tokens with different values in its output: when T_1 fires, the token in P_b will be $K_b = (2, t_b^{\mathrm{i}})$ with $t_b^{\mathrm{i}} \in [1, 3]$, but when T_2 fires the token in P_b will be $K_b = (0, t_b^{\mathrm{ii}})$ with $t_b^{\mathrm{ii}} \in [2, 3]$. The reason for



(a) (b)

Figure 3.5: Cardinality-equivalent nets

this behavior is the non-determinism of N_1. On the other hand, when the only out-port of N_2 is marked, the corresponding token value is $v_b = 2$.



(a) (b)

Figure 3.6: Cardinality-equivalent nets with different behavior

As shown in the example of Figure 3.6, even if two nets are cardinality-equivalent the tokens in their outputs may be different, although their initial marking is identical. For instance, there is no marking $M_2 \in \mathscr{R}(N_2)$ in which the out-port has a token with value $v_b = 0$, whereas it does exist a marking $M_1 \in \mathscr{R}(N_1)$ in which the out-port is marked and $v_b = 0$. Thus the external observer could distinguish between N_1 and N_2 because of different token values—moreover different token times—in their out-ports when marked.

Definition 3.27 Two nets N_1 and N_2 are *function-equivalent* iff:

(i) N_1 and N_2 are cardinality-equivalent;

(ii) Let M_1 and M_2 be markings satisfying condition (iii) in Definition 3.26. For every $(v_1, t_1) \in M_1(Q)$, where $Q \in \mathbf{outP}_1$, there exists $(v_2, t_2) \in M_2(h_{out}(Q))$ such that $v_1 = v_2$, and vice versa. ❑

Definition 3.28 Two nets N_1 and N_2 are *time-equivalent* iff:

(i) N_1 and N_2 are cardinality-equivalent;

(ii) Let M_1 and M_2 be markings satisfying condition (iii) in Definition 3.26. For every $(v_1, t_1) \in M_1(Q)$, where $Q \in \mathbf{outP}_1$, there exists $(v_2, t_2) \in M_2(h_{out}(Q))$ such that $t_1 = t_2$, and vice versa. ❑

Two nets are function-equivalent if, besides being cardinality-equivalent, the tokens obtained in corresponding out-ports have the same token value. Similarly, if tokens obtained in corresponding out-ports have the same token time, the nets are time-equivalent.

Definition 3.29 Two nets N_1 and N_2 are *total-equivalent* iff:
(i) N_1 and N_2 are function-equivalent;
(ii) N_1 and N_2 are time-equivalent. ❑

Figure 3.7 shows the relation between the notions of equivalence introduced above. Cardinality-equivalence is necessary for time-equivalence and also for function-equivalence. Similarly, total-equivalence implies all other equivalences. Total-equivalence is the strongest notion of equivalence defined in this section. Note however that two total-equivalent nets need not be identical (see Figure 3.8).

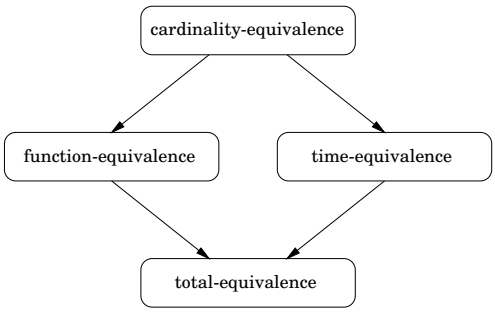


Figure 3.7: Relation between the notions of equivalence

3.5.2 Hierarchical PRES+ Model

PRES+ supports systems modeled at different levels of granularity with transitions representing simple arithmetic operations or complex algorithms. However, in order to efficiently handle the modeling of large systems, a mechanism of hierarchical composition is needed so that the model may be constructed in a structured manner, composing simple units fully understandable by the designer. Hierarchy can conveniently be used as a form to handle complexity and also to analyze systems at different abstraction levels.

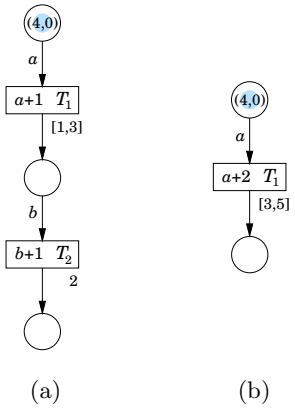


Figure 3.8: Total-equivalent nets

Hierarchical modeling can favorably be applied along the design process. In a top-down approach, for instance, a designer may define the interface to each component and then gradually refine those components. On the other hand, a system may also be constructed reusing existing elements such as Intellectual Property (IP) blocks in a bottom-up approach.

A flat representation of a real-life system can be too big and complex to handle and understand. The concept of hierarchy allows systems to be modeled in a structured way. Thus the system may be broken down into a set of comprehensible nets structured in a hierarchy. Each one of these nets may represent a sub-block of the current design. Such a sub-block can be a pre-designed IP component as well as a design alternative corresponding to a subsystem of the system under design.

In the following we formalize a concept of hierarchy for PRES+ models. A new element called *super-transition* is introduced. Super-transitions can be thought of as "interfaces" in a hierarchical model. Some simple examples are used in order to illustrate the definitions.

Definition 3.30 A transition $T \in \mathbf{T}$ is an *in-transition* of $N = (\mathbf{P}, \mathbf{T}, \mathbf{I}, \mathbf{O}, M_0)$ if $\bigcup_{P \in \mathbf{inP}} P^\circ = \{T\}$. In a similar manner, a transition $T \in \mathbf{T}$ is an *out-transition* of N if $\bigcup_{P \in \mathbf{outP}} {}^\circ P = \{T\}$. ❑

Note that the existence of non-empty sets **inP** and **outP** (in- and out-ports) is a necessary condition for the existence of in- and out-transitions. Also, according to Definition 3.30, if there exists an in-transition T_{in} in a given net N, it is unique (T_{in} is the only in-transition in N). Similarly, an out-transition T_{out} is unique. For the net N_1 shown in Figure 3.9, $\mathbf{inP}_1 = \{P_a, P_b\}$, $\mathbf{outP}_1 = \{P_d\}$, and T_x and T_y are the in-transition and out-transition respectively.

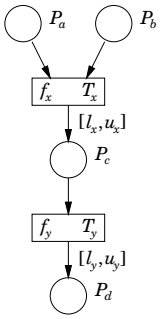


Figure 3.9: A simple subnet N_1

Definition 3.31 An *abstract PRES+* model is a six-tuple $H = (\mathbf{P}, \mathbf{T}, \mathbf{ST}, \mathbf{I}, \mathbf{O}, M_0)$ where $\mathbf{P} = \{P_1, P_2, \ldots, P_m\}$ is a finite non-empty set of places; $\mathbf{T} = \{T_1, T_2, \ldots, T_n\}$ is a finite set of transitions; $\mathbf{ST} = \{ST_1, ST_2, \ldots, ST_l\}$ is a finite set of *super-transitions* ($\mathbf{T} \cup \mathbf{ST} \neq \emptyset$); $\mathbf{I} \subseteq \mathbf{P} \times (\mathbf{T} \cup \mathbf{ST})$ is a finite set of input arcs; $\mathbf{O} \subseteq (\mathbf{T} \cup \mathbf{ST}) \times \mathbf{P}$ is a finite set of output arcs; M_0 is the initial marking. ❑

Observe that a (non-abstract) PRES+ net is a particular case of an abstract PRES+ net with $\mathbf{ST} = \emptyset$. Figure 3.10 illustrates an abstract PRES+ net. Super-transitions are represented by thick-line boxes.

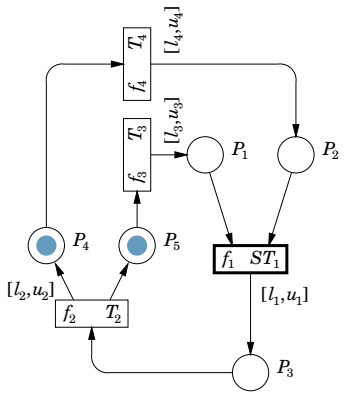


Figure 3.10: An abstract PRES+ model

Definition 3.32 The *pre-set* ${}^\circ ST$ and *post-set* ST° of a super-transition $ST \in \mathbf{ST}$ are given by ${}^\circ ST = \{P \in \mathbf{P} \mid (P, ST) \in \mathbf{I}\}$ and $ST^\circ = \{P \in \mathbf{P} \mid (ST, P) \in \mathbf{O}\}$ respectively. ❑

Similar to transitions, the pre(post)-set of a super-transition $ST \in \mathbf{ST}$ is the set of input(output) places of ST.

Definition 3.33 For every super-transition $ST \in \mathbf{ST}$ there exists a *high-level function* $f : \zeta(P_1) \times \zeta(P_2) \times \ldots \times \zeta(P_a) \rightarrow \zeta(Q)$ associated to ST, where ${}^\circ ST = \{P_1, P_2, \ldots, P_a\}$ and $Q \in ST^\circ$. ❑

Recall that $\zeta(P)$ denotes the token type associated with the place $P \in \mathbf{P}$, that is the type of value that a token may bear in that place. High-level functions associated to super-transitions may be rather useful in, for instance, a top-down approach: for a certain component of the system, the designer may define its interface and a high-level description of its functionality through a super-transition, and in a later design phase refine the component. In current design methodologies it is also very common to reuse predefined elements such as IP blocks. In such cases, the internal structure of the component is unknown to the designer and therefore the block is best modeled by a super-transition and its high-level function.

Definition 3.34 For every super-transition $ST \in \mathbf{ST}$ there exist a *best-case delay* τ^{bc} and a *worst-case delay* τ^{wc}, where $\tau^{\mathrm{bc}} \leq \tau^{\mathrm{wc}}$ are non-negative real numbers that represent the lower and upper limits for the execution time of the high-level function associated to ST. ❑

Definition 3.35 A super-transition may not be in *conflict* with other transitions or super-transitions, that is:

(i) ${}^\circ ST_1 \cap {}^\circ ST_2 = \emptyset$ and $ST_1^\circ \cap ST_2^\circ = \emptyset$ for all $ST_1, ST_2 \in \mathbf{ST}$ such that $ST_1 \neq ST_2$;

(ii) ${}^\circ ST \cap {}^\circ T = \emptyset$ and $ST^\circ \cap T^\circ = \emptyset$ for all $T \in \mathbf{T}$, $ST \in \mathbf{ST}$. ❑

In other words, a super-transition may not share input or output places with other transitions/super-transitions. The restriction imposed by Definition 3.35 avoids time inconsistencies when refining a super-transition with a lower-level subnet. In what follows, the input and output places of a super-transition $ST \in \mathbf{ST}$ will be called the *surrounding* places of ST.

Definition 3.36 A super-transition $ST_i \in \mathbf{ST}$ together with its surrounding places in the net $H = (\mathbf{P}, \mathbf{T}, \mathbf{ST}, \mathbf{I}, \mathbf{O}, M_0)$ is a *semi-abstraction* of the subnet $N_i = (\mathbf{P}_i, \mathbf{T}_i, \mathbf{ST}_i, \mathbf{I}_i, \mathbf{O}_i, M_{i,0})$ (or conversely, N_i is a *semi-refinement* of ST_i and its surrounding places) if:

(i) There exists an in-transition $T_{in} \in \mathbf{T}_i$;

(ii) There exists an out-transition $T_{out} \in \mathbf{T}_i$;

(iii) There exists a bijection $h_{in} : {}^\circ ST_i \rightarrow \mathbf{inP}_i$ that maps the input places of ST_i onto the in-ports of N_i;

(iv) There exists a bijection $h_{out} : ST_i^\circ \rightarrow \mathbf{outP}_i$ that maps the output places of ST_i onto the out-ports of N_i;

(v) $M_0(P) = M_{i,0}(h_{in}(P))$ and $\zeta(P) = \zeta(h_{in}(P))$ for all $P \in {}^\circ ST_i$;

(vi) $M_0(P) = M_{i,0}(h_{out}(P))$ and $\zeta(P) = \zeta(h_{out}(P))$ for all $P \in ST_i^\circ$;
(vii) For the initial marking $M_{i,0}$, T is disabled for all $T \in \mathbf{T}_i \setminus \{T_{in}\}$. ❑

Note that a subnet may, in turn, contain super-transitions. It is simple to prove that the subnet N_1 shown in Figure 3.9 is indeed a semi-refinement of ST_1 in the net shown in Figure 3.10.

If a net N_i is the semi-refinement of some super-transition ST_i, it is possible to *characterize* N_i in terms of both function and time, by putting tokens in its in-ports and then observing the value and time stamp of tokens in its out-ports after a certain firing sequence. If the time stamp of all tokens deposited in the in-ports of N_i is zero, the token time of tokens obtained in the out-ports is called the *execution time* of N_i. For example, the net N_1 shown in Figure 3.9 can be characterized by putting tokens $K_a = (v_a, 0)$ and $K_b = (v_b, 0)$ in its in-ports P_a and P_b, respectively, and observing the token $K_d = (v_d, t_d)$ after firing T_x and T_y. Thus the execution time of N_1 is equal to the token time t_d, in this case bounded by $l_x + l_y \leq t_d \leq u_x + u_y$. The token value v_d is given by $v_d = f_y(f_x(v_a, v_b))$, where f_x and f_y are the transition functions of T_x and T_y respectively.

The above definition of semi-abstraction/refinement allows a complex design to be constructed in a structured way by composing simpler entities. However, it does not give a semantic relation between the functionality of super-transitions and their refinements. Below we define the concepts of strong and weak refinement of a super-transition.

Definition 3.37 A subnet $N_i = (\mathbf{P}_i, \mathbf{T}_i, \mathbf{ST}_i, \mathbf{I}_i, \mathbf{O}_i, M_{i,0})$ is a *strong refinement* of the super-transition $ST_i \in \mathbf{ST}$ together with its surrounding places in the net $H = (\mathbf{P}, \mathbf{T}, \mathbf{ST}, \mathbf{I}, \mathbf{O}, M_0)$ (or ST_i and its surrounding places is a *strong abstraction* of N_i) if:

(i) N_i is a semi-refinement of ST_i;
(ii) N_i *implements* ST_i, that is, N_i is function-equivalent to ST_i and its surrounding places;
(iii) The best-case delay τ_i^{bc} of ST_i is equal to the lower bound of the execution time of N_i;
(iv) The worst-case delay τ_i^{wc} of ST_i is equal to the lower bound of the execution time of N_i. ❑

The subnet N_1 shown in Figure 3.9 is a semi-refinement of ST_1 in the net shown in Figure 3.10. N_1 is a strong refinement of the super-transition ST_1 if, in addition: (a) $f_1 = f_y \circ f_x$; (b) $l_1 = l_x + l_y$; (c) $u_1 = u_x + u_y$ (conditions (ii), (iii), and (iv), respectively, of Definition 3.37).

The concept of strong refinement given by Definition 3.37 requires the super-transition and its strong refinement to have the very same time limits.

Such a concept could have limited practical use from the viewpoint of a design environment, since the high-level description and the implementation perform the same function but typically have different timings and therefore their bounds for the execution time do not coincide. Nonetheless, the notion of strong refinement can be very useful for abstraction purposes. If we relax the requirement of exact correspondence of lower and upper bounds on time, this yields to a weaker notion of refinement.

Definition 3.38 A subnet $N_i = (\mathbf{P}_i, \mathbf{T}_i, \mathbf{ST}_i, \mathbf{I}_i, \mathbf{O}_i, M_{i,0})$ is a *weak refinement* of the super-transition $ST_i \in \mathbf{ST}$ together with its surrounding places in the net $H = (\mathbf{P}, \mathbf{T}, \mathbf{ST}, \mathbf{I}, \mathbf{O}, M_0)$ (or ST_i and its surrounding places is a *weak abstraction* of N_i) if:

(i) N_i is a semi-refinement of ST_i;
(ii) N_i *implements* ST_i;
(iii) The best-case delay τ_i^{bc} of ST_i is less than or equal to the lower bound of the execution time of N_i;
(iv) The worst-case delay τ_i^{wc} of ST_i is greater than or equal to the upper bound of the execution time of N_i. ❑

Given a hierarchical PRES+ net $H = (\mathbf{P}, \mathbf{T}, \mathbf{ST}, \mathbf{I}, \mathbf{O}, M_0)$ and refinements of its super-transitions, it is possible to construct a corresponding non-hierarchical net. For the sake of clarity, in the following definition we consider nets with a single super-transition, nonetheless these concepts can easily be extended to the general case.

Definition 3.39 Let us consider the net $H = (\mathbf{P}, \mathbf{T}, \mathbf{ST}, \mathbf{I}, \mathbf{O}, M_0)$ where $\mathbf{ST} = \{ST_1\}$, and let the subnet $N_1 = (\mathbf{P}_1, \mathbf{T}_1, \mathbf{ST}_1, \mathbf{I}_1, \mathbf{O}_1, M_{1,0})$ be a refinement of ST_1 and its surrounding places. Let $T_{in}, T_{out} \in \mathbf{T}_1$ be unique in-transition and out-transition respectively. Let $\mathbf{inP}_1$ and $\mathbf{outP}_1$ be respectively the sets of in-ports and out-ports of N_1. The net $H' = (\mathbf{P}', \mathbf{T}', \mathbf{ST}', \mathbf{I}', \mathbf{O}', M_0')$ *one level lower in the hierarchy*, is defined as follows:

(i) $\mathbf{ST}' = \mathbf{ST}_1$;
(ii) $\mathbf{P}' = \mathbf{P} \cup (\mathbf{P}_1 \setminus \mathbf{inP}_1 \setminus \mathbf{outP}_1)$;
(iii) $\mathbf{T}' = \mathbf{T} \cup \mathbf{T}_1$;
(iv) $(P, ST) \in \mathbf{I}'$ if $(P, ST) \in \mathbf{I}_1$;
$(P, T) \in \mathbf{I}'$ if $(P, T) \in \mathbf{I}$, or $(P, T) \in \mathbf{I}_1$ and $P \notin \mathbf{inP}_1$;
$(P, T_{in}) \in \mathbf{I}'$ if $(P, ST_1) \in \mathbf{I}$;
(v) $(ST, P) \in \mathbf{O}'$ if $(ST, P) \in \mathbf{O}_1$;
$(T, P) \in \mathbf{O}'$ if $(T, P) \in \mathbf{O}$, or $(T, P) \in \mathbf{O}_1$ and $P \notin \mathbf{outP}_1$;
$(T_{out}, P) \in \mathbf{O}'$ if $(ST_1, P) \in \mathbf{O}$;
(vi) $M_0'(P) = M_0(P)$ for all $P \in \mathbf{P}$;
$M_0'(P) = M_{1,0}(P)$ for all $P \in \mathbf{P}_1 \setminus \mathbf{inP}_1 \setminus \mathbf{outP}_1$. ❑

Definition 3.39 can be used in order to flatten a hierarchical PRES+ model. Given the net of Figure 3.10 and being N_1 (Figure 3.9) a refinement of ST_1, we can construct the equivalent non-hierarchical net as illustrated in Figure 3.11.

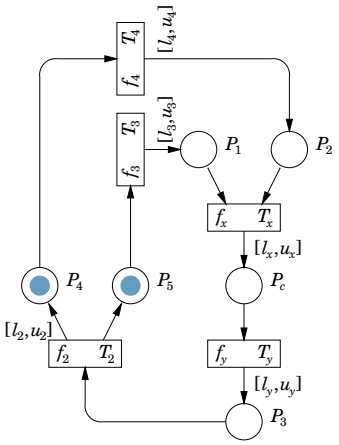


Figure 3.11: A non-hierarchical PRES+ model

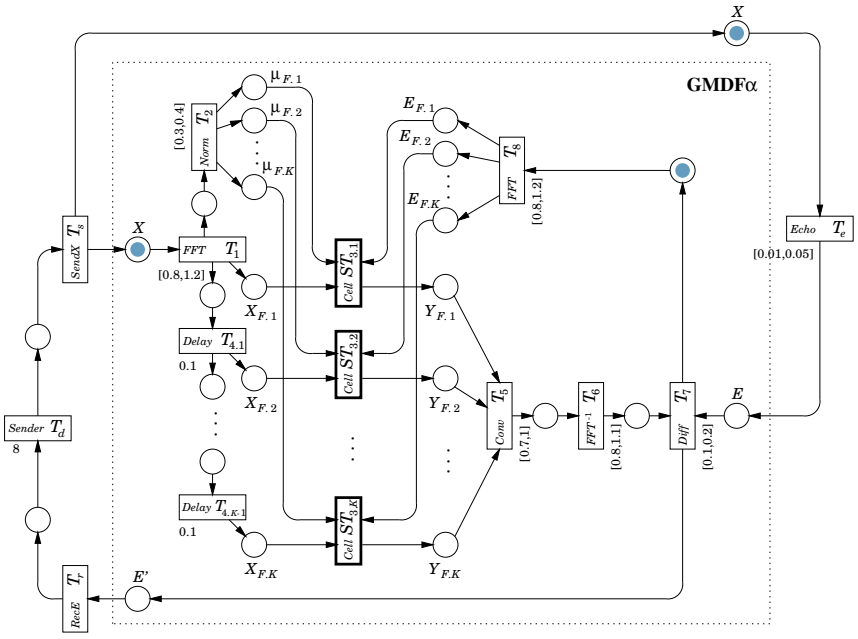
3.6 Modeling Examples

In this section we present two realistic applications that illustrate the modeling of systems using PRES+.

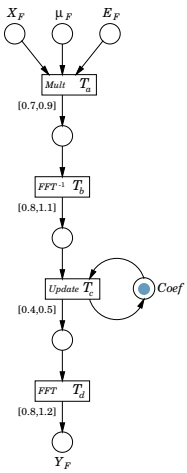
3.6.1 Filter for Acoustic Echo Cancellation

In this subsection we model a Generalized Multi-Delay frequency-domain Filter (GMDFα) [FIR+97] using PRES+. GMDFα has been used in acoustic echo cancellation for improving the quality of hand-free phone and teleconference applications. The GMDFα algorithm is a frequency-domain block adaptive algorithm: a block of input data is processed at one time, producing a block of output data. The impulse response of length L is segmented into K smaller blocks of size N ($K = L/N$), thus leading to better performance. R new samples are processed at each iteration and the filter is adapted α times per block ($R = N/\alpha$).

The filter inputs are a signal X and its echo E, and the output is the reduced or cancelled echo E'. In Figure 3.12 we show the hierarchical PRES+ model of a GMDFα. The transition T_1 transforms the input signal X into the frequency domain by a FFT (Fast Fourier Transform). T_2 corresponds to the normalization block. In each one of the basic cells $ST_{3.i}$ the filter



(a)



(b)

Figure 3.12: GMDFα modeled using PRES+

coefficients are updated. Transitions $T_{4.i}$ serve as delay blocks. T_5 computes the estimated echo in the frequency domain by a convolution product and then it is converted into the time domain by T_6. The difference between the estimated echo and the actual one (signal E) is calculated by T_7 and output

as E'. Such a cancelled echo is also transformed into the frequency domain by T_8 to be used in the next iteration when updating the filter coefficients.

In Figure 3.12(a) we also model the environment with which the GMDFα interacts: T_e models the echoing of signal X, T_s and T_r represent, respectively, the sending of the signal and the reception of the cancelled echo, and T_d is the entity that emits X.

The refinement of the basic cells $ST_{3.i}$ is shown in Figure 3.12(b) where the filter coefficients are computed and thus the filter is adapted by using FFT^{-1} and FFT operations. Transition delays in Figure 3.12 are given in ms.

This example shows how hierarchy allows systems to be structured in an understandable way. It is worth noticing that instances of the same subnet (Figure 3.12(b)) are used as refinements of the different cells $ST_{3.i}$ in Figure 3.12(a). Thus, in cases like this one, the regularity of the system can be exploited in order to obtain a more succinct model.

Later, in Subsection 5.1.2, we show how the verification of this filter is performed and the advantages of modeling it in this way.

3.6.2 Radar Jammer

The example described in this subsection corresponds to a real-life application used in the military industry [LK01]. The function of this system is to deceive a radar apparatus by jamming signals.

The jammer is a system placed on an object (target), typically an aircraft, moving in the area observed by a radar. The radar sends out pulses and some of them are reflected back to the radar by the objects in the area. When a radar receives pulses, it makes use of the received information for determining the distance and direction of the object, and even the velocity and the type of the object. The distance is calculated by measuring the time the pulse has traveled from its emission until it returns to the radar. By rotating the radar antenna lobe, it is possible to find the direction returning maximum energy, that is, the direction of the object. The velocity of the object is found out based on the Doppler shift of the returning pulse. The type of the object can be determined by comparing the shape of the returning pulse with a library of radar signatures for different objects.

The basic function of the jammer is to deceive a radar scanning the area in which the object is moving. The jammer receives a radar pulse, modifies it, and then sends it back to the radar after a certain delay. Based on input parameters, the jammer can create pulses that contain specific Doppler and signature information as well as the desired space and time data. Thus the radar will see a false target. A view of the radar jammer and its environment is shown in Figure 3.13.

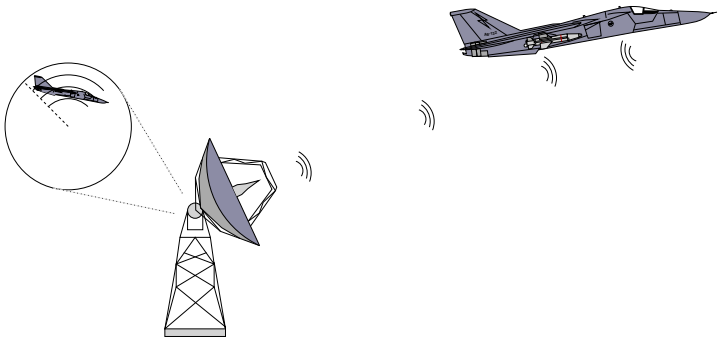


Figure 3.13: Radar jammer and its environment

The jammer has been specified in Haskell using a number of *skeletons* (higher-order functions used to model elementary processes) [LK01]. Using the procedure for translating Haskell descriptions (using skeletons) into PRES+ [CPE01], we obtained the model shown in Figure 3.14. It contains no timing information which can later be annotated as transition delays.

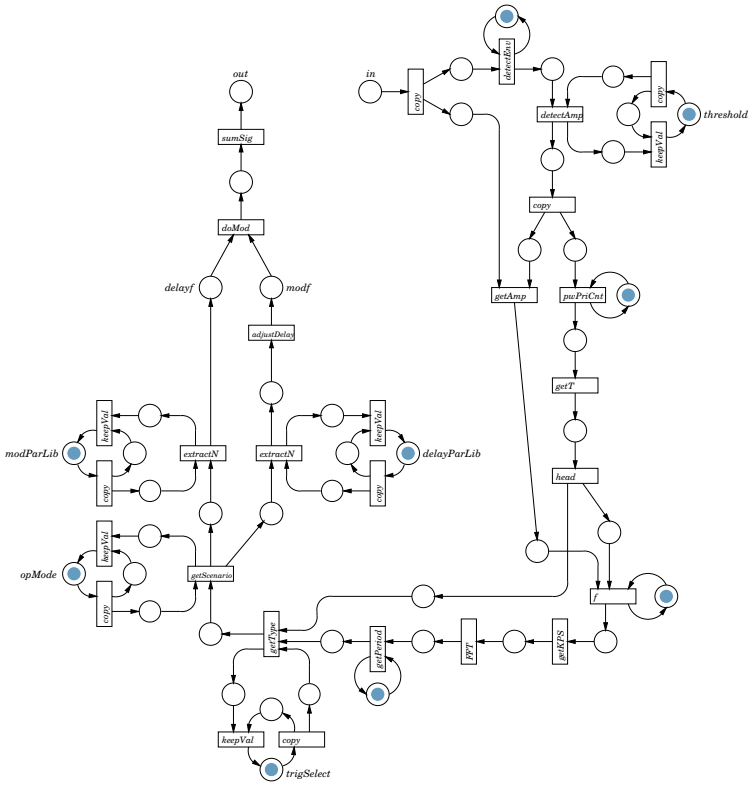


Figure 3.14: A PRES+ model of a radar jammer

We briefly discuss the structure of the PRES+ model of the jammer. We do not intend to provide here a detailed description of each one of the transitions of the model of the radar jammer shown in Figure 3.14 but rather present an intuitive idea about it. When a pulse arrives, it is initially detected and some of its characteristics are calculated by processing the samples taken from the pulse. Such processing is performed by the initial transitions, namely *detectEnv*, *detectAmp*, ..., *getPer*, and *getType*, based on internal parameters like *threshold* and *trigSelect*. Different scenarios are handled by the middle transitions, namely *getScenario*, *extractN*, and *adjustDelay*. The final transitions *doMod* and *sumSig* are the ones that actually alter the pulse to be returned to the radar.

Using the concept of hierarchy, it is possible to obtain a higher-level view of the radar jammer represented in PRES+ as depicted in Figure 3.15. The super-transitions abstract parts of the model given in Figure 3.14. For example, the super-transition ST_5 corresponds to the abstraction of the subnet shown in Figure 3.16. Such a subnet (Figure 3.16) can easily be identified as a portion of the model depicted in Figure 3.14.

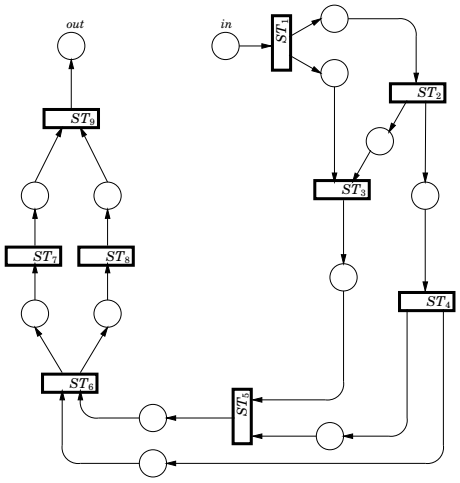


Figure 3.15: Higher-level abstraction of the radar jammer

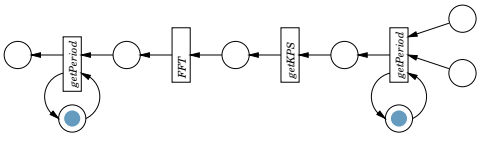


Figure 3.16: Refinement of ST_5 in the model of Figure 3.15

Also, many of the transitions presented in the model of Figure 3.14 could

be refined (for example, during the design process). In order to illustrate this, we show how transition *doMod*, for instance, can be refined according to our concept of hierarchy. Its refinement is presented in Figure 3.17. In this form, hierarchy can conveniently be used to structure the design in a comprehensible manner.

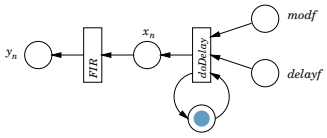


Figure 3.17: Refinement of *doMod* in the model of Figure 3.14

The verification of the radar jammer discussed above is addressed later in Subsection 5.3.2.

Chapter 4

Formal Verification of PRES+ Models

The complexity of electronic systems, among them embedded systems, has increased enormously in the past years. Systems with intricate functionality are now possible due to both advances in the fabrication technology and clever design methodologies. However, as the complexity of systems increases, the likelihood of subtle errors becomes much greater. A way to cope, up to a certain extent, with the issue of correctness is the use of mathematically-based techniques known as *formal methods*. Formal methods offer a rigorous basis for the development of systems because they provide a framework which aims at obtaining provable correct systems along the various steps of the design process.

For the levels of complexity typical to modern embedded systems, traditional validation techniques like simulation and testing are simply not sufficient when it comes to verifying the correctness of the system because, with these methods, it is feasible to cover just a small fraction of the system behavior.

As pointed out in Section 1.1, correctness plays a key role in many embedded systems. One aspect is that, due to the nature of the application (for instance, safety-critical systems like the ones used in transportation, defense, and medical equipment), a failure may lead to catastrophic situations. Another important issue to consider is the fact that bugs found late in prototyping phases have a quite negative impact on the time-to-market of the product. Formal methods are intended to help towards the goal of designing correct systems.

The discipline stimulated by formal methods leads very often to a careful scrutiny of the fundamentals of the system under design, its specification, and the assumptions built around it, which, in turn, leads to a better un-

derstanding of the system and its environment. This represents by itself a benefit when the task is to design complex systems.

In this chapter we introduce our approach to the formal verification of systems represented in PRES+. First, we present some background notions in order to make clearer the presentation of our ideas. Then, we explain our technique and propose a translation procedure from PRES+ into the input formalism of existing verification tools. Finally, we illustrate the approach through the verification of a realistic system.

4.1 Background

The purpose of this section is to present some preliminary concepts that will be needed for the later discussion.

4.1.1 Formal Methods

The weaknesses of traditional validation techniques have stimulated research towards solutions that attempt to prove a system correct. Formal methods are analytical and mathematical techniques intended to prove formally that the implementation of a system conforms its specification. The two well-established approaches to formal verification are *theorem proving* and *model checking* [CW96].

Theorem proving is the process of proving a property or statement by showing that it is a logical consequence of a set of axioms [Fit96]. In theorem proving, when used as verification tool, the idea is to prove a system correct by using axioms and inference and deduction rules, in the same sense that a mathematical theorem is proved correct. Both the system (its rules and axioms) and its desired properties are typically expressed as formulas in some mathematical logic, often first-order logic, because precise formulations are needed for manipulating the statements throughout the proving process. Then, a proof of a given property must be found from axioms and rules of the system. Although there exist computer tools, called theorem provers, that assist the designer in verifying a certain property, theorem proving requires significant interaction with the user and therefore it is a relatively slow and error-prone process. Nonetheless, theorem proving techniques can handle infinite-space systems, which constitutes their major asset.

On the other hand, model checking [CGP99] is an automatic approach to formal verification used to determine whether the model of a system satisfies a set of required properties. In principle, a model checker searches exhaustively the state space. Since the observable behavior of finite-space systems can finitely be represented, such systems can be verified using automatic

approaches. Model checking is fully automatic (the user needs not be an expert in logics or other mathematical disciplines) and can produce counterexamples (it shows why the system fails to satisfy a property that does not hold, giving insight for diagnostic purposes). The main disadvantage of model checking, though, is the state explosion problem. Thus key challenges are the algorithms and data structures that ameliorate the effects of state explosion and allow handling large search spaces.

Formal methods have grown mature and become a practical alternative for ensuring the correctness of designs. They might overcome some of the limitations of traditional validation methods. At the same time, formal verification can give a better understanding of the system behavior, help to uncover ambiguities, and reveal new insights of the system. However, formal methods do have limitations and are not the universal solution to achieve correct systems. We believe that formal verification is to complement, rather than replace, simulation and testing methods.

4.1.2 Temporal Logics

A temporal logic is a logic augmented with temporal modal operators which allow reasoning about how the truth of assertions changes over time [KG99]. Temporal logics are usually employed to specify desired properties of systems. There are different forms of temporal logics depending on the underlying model of time. In this subsection, we focus on CTL (Computation Tree Logic) because it is a representative example of temporal logics and it is one that we use in our verification approach.

Several model checking algorithms have been presented in the literature [CGP99]. Many of them use temporal logics to express the properties of the system. One of the well known algorithms is CTL model checking introduced by Clarke *et al.* [CES86]. CTL is based on propositional logic of branching time, that is, a logic where time may split into more than one possible future using a discrete model of time. Formulas in CTL are composed of atomic propositions, boolean connectors, and temporal operators. Temporal operators consist of forward-time operators (**G** globally, **F** in the future, **X** next time, and **U** until) preceded by a path quantifier (**A** all computation paths, and **E** some computation path). Figure 4.1 illustrates some of the CTL temporal operators. The computation tree represents an unfolded state graph where the nodes are the possible states that the system may reach. The shaded nodes are those states in which property p holds. Thus it is possible to express properties that refer to the root node (initial state) using CTL temporal operators. For instance, **AF** p holds if for every possible path, starting from the initial state, there exists at least one state in which p is satisfied, that is, p will eventually happen.

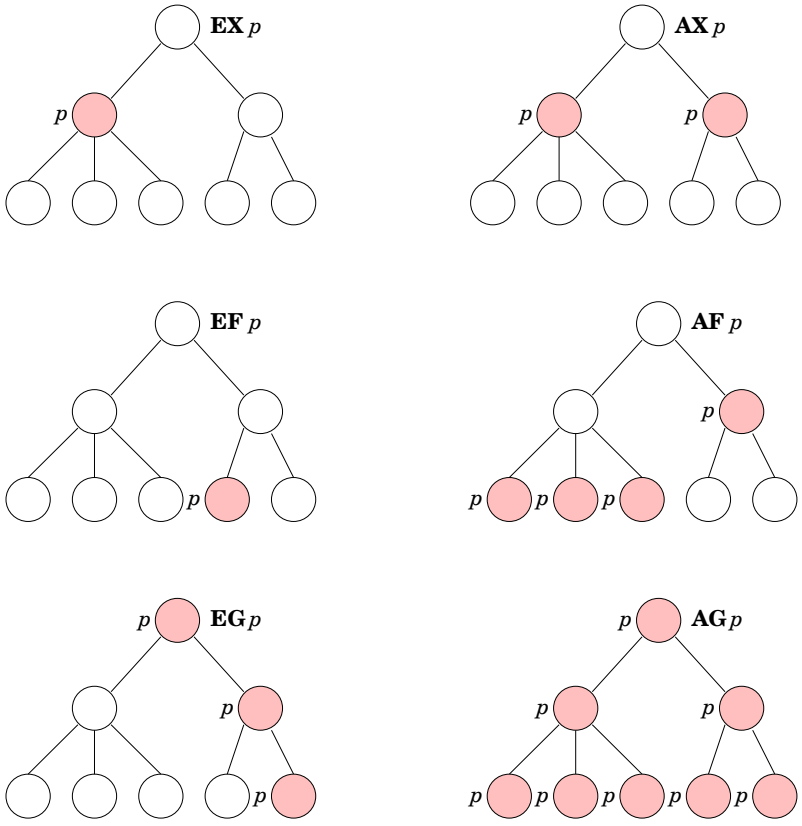


Figure 4.1: CTL temporal operators

CTL does not provide a way to specify quantitatively time. Temporal operators allow only the description of properties in terms of "next time", "eventually", or "always".

TCTL (Timed CTL), introduced by Alur *et al.* [ACD90], is a real-time extension of CTL that allows the inscription of subscripts on the temporal operators in order to limit their scope in time. For instance, $\mathbf{AF}_{<n}\, q$ expresses that, along all computation paths, the property q becomes true within n time units. When using the notion of dense time (time is treated as a continuous quantity) the state space has infinitely many states because of the real-valued clock variables (variables used to count time). However, it is possible to define an equivalence relation over the states such that any two equivalent states are indistinguishable by TCTL formulas [ACD90]. In other words, it is possible to construct a finite representation of the (infinite-space) system that is consistent with TCTL. This makes feasible the model checking of real-time systems when dense-time semantics is considered.

4.1.3 Timed Automata

A timed automaton is a finite automaton augmented with a finite set of real-valued clocks [Alu99]. Timed automata can be thought of as a collection of automata which operate and coordinate with each other through shared variables and synchronization labels. There is a set of real-valued variables, named clocks, all of which change along the time with the same constant rate. There might be conditions over clocks that express timing constraints.

Definition 4.1 A *timed automata* model is a tuple $\vec{\mathcal{T}} = (\mathcal{L}, \mathcal{L}_0, \mathcal{E}, \mathcal{X}, \mathfrak{x}, \mathcal{C}, \mathcal{V}, \mathfrak{c}, \mathfrak{v}, \mathfrak{r}, \mathfrak{a}, \mathfrak{i})$, where:

- $\mathcal{L}$ is a finite set of *locations*;
- $\mathcal{L}_0 \subseteq \mathcal{L}$ is a set of *initial locations*;
- $\mathcal{E} \subseteq \mathcal{L} \times \mathcal{L}$ is a set of *edges*;
- $\mathcal{X}$ is a finite set of *labels*;
- $\mathfrak{x} : \mathcal{E} \to \mathcal{X}$ is a mapping that labels each edge in $\mathcal{E}$ with some label in $\mathcal{X}$;
- $\mathcal{C}$ is a finite set of real-valued *clocks*;
- $\mathcal{V}$ is a finite set of *variables*;
- $\mathfrak{c}$ is a mapping that assigns to each edge $e = (l, l')$ a *clock condition* $\mathfrak{c}(e)$ over $\mathcal{C}$ that must be satisfied in order to allow the automaton to change its location from l to l';
- $\mathfrak{v}$ is a mapping that assigns to each edge $e = (l, l')$ a *variable condition* $\mathfrak{v}(e)$ over $\mathcal{V}$ that must be satisfied in order to allow the automaton to change its location from l to l';
- $\mathfrak{r} : \mathcal{E} \to 2^{\mathcal{C}}$ is a *reset function* that gives the clocks to be reset on each edge;
- $\mathfrak{a}$ is the *activity mapping* that assigns to each edge e a set of *activities* $\mathfrak{a}(e)$;
- $\mathfrak{i}$ is a mapping that assigns to each location l an invariant $\mathfrak{i}(l)$ which allows the automaton to stay in location l as long as its invariant is satisfied. ❑

A timed automaton may stay in its current location if its invariant is satisfied, otherwise it is forced to make a transition and change its location. In order to make a change of location through a particular edge, both its clock condition and its variable condition must be satisfied. When a change of location takes place, the set of activities assigned to the edge occur (for instance, assign to a variable the result of evaluating certain expression) and the clocks corresponding to the edge that are given by the reset function are set to zero.

Let us consider the automata shown in Figure 4.2. We use this simple example in order to illustrate the notation for timed automata presented above. The model consists of two automata where the set of locations and initial locations are $\mathcal{L} = \{a_1, a_2, a_3, b_1, b_2, b_3\}$ and $\mathcal{L}_0 = \{a_1, b_1\}$ respectively.

There are seven edges as drawn in Figure 4.2. For the sake of clarity, only labels shared by different edges are shown. Such labels are called *synchronization labels*. In our example, $T \in \mathcal{X}$ is the only synchronization label, so that a transition from location a_2 to location a_3 in the first automaton must be accompanied by a transition from b_1 to b_2 in the second automaton. The set of clocks and variables are $\mathcal{C} = \{c_a, c_b\}$ and $\mathcal{V} = \{y\}$ respectively. Examples of clock and variable conditions in the model shown in Figure 4.2 are, respectively, $c_b > 4$ and $y == 1$. Thus, for instance, a transition from location b_3 to location b_1 is allowed only if the clock c_b satisfies the condition $c_b > 4$. Similarly, a transition from b_2 to b_3 is allowed if $y = 1$. In Figure 4.2, $c_a := 0$ represents the reset of the clock c_a, that is, $\mathtt{r}((a_2, a_3)) = \{c_a\}$. Also, $y := 2$ represents the activity assigned to the edge (a_3, a_1), that is, when there is a transition from location a_3 to location a_1 the variable y is assigned the value 2. The invariant of location a_3 is $c_a \leq 3$, which means that the automaton may stay in a_3 only as long as $c_a \leq 3$.

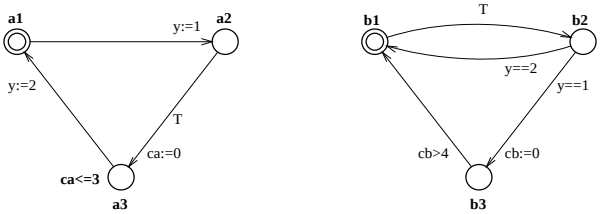


Figure 4.2: A timed automata model

4.2 Verifying PRES+ Models

There are several types of analyses that can be performed on systems represented in PRES+. The absence or presence of tokens in places of the net may represent the state of the system at a certain moment in the dynamic behavior of the net. Based on this, different properties can be studied. For instance, two places marked simultaneously could represent a hazardous situation that must be avoided. This sort of safety requirement might formally be proved by checking that such a dangerous state is never reached. Also, the designer could be interested in proving that the system eventually reaches a certain state, in which the presence of tokens in a particular place represents the completion of a task. This kind of analysis, absence/presence of tokens in places of the net, is termed *reachability analysis*.

Reachability analysis is useful but says nothing about timing aspects nor does it deal with token values. In embedded applications, however, time is an essential factor. Moreover, in hard real-time systems, where deadlines should

not be missed, it is crucial to quantitatively reason about temporal properties in order to ensure the correctness of the design. Therefore, it is needed not only to check that a certain state will eventually be reached but also to ensure that this will occur within some bound on time. In PRES+, time information is attached to tokens so that we can analyze quantitative timing properties. We may prove that a given place will eventually be marked and that its time stamp will be less than a certain time value that represents a temporal constraint. Such sort of analysis is called *time analysis*.

A third type of analysis for systems modeled in PRES+ involves reasoning about values of tokens in marked places and is called *functionality analysis*. In this work we restrict ourselves to reachability and time analyses. In other words, we concentrate on the absence/presence of tokens in the places of the net and their time stamps. Note, however, that in some cases reachability and time analyses are influenced by token values. The way we handle such cases for verification purposes is addressed later in this chapter.

4.2.1 Our Approach to Formal Verification

As discussed in Subsection 4.1.1, model checking is one of the well-established approaches to formal verification: a number of desired properties (called in this context *specification*) are checked against a given model of the system. The two inputs to the model checking problem are the system model and the properties that such a system must satisfy, usually expressed as temporal logic formulas.

The purpose of our verification approach is to formally reason about systems represented in PRES+. For the sake of verification, we restrict ourselves to *safe* PRES+ nets, that is, nets in which each place $P \in \mathbf{P}$ holds at most one token for every marking M reachable from M_0. Otherwise, the formal analysis would become more cumbersome. This is a trade-off between expressiveness and analysis complexity, and avoids excessive verification times for applications of realistic size.

We use model checking in order to verify the correctness of systems modeled in PRES+. In our approach we can determine the truth of formulas expressed in the temporal logics CTL [CES86] and TCTL [ACD90] with respect to a (safe) PRES+ model. In our approach the atomic propositions of CTL/TCTL correspond to the absence/presence of tokens in places in the net. Thus the atomic proposition P holds iff $P \in \mathbf{P}$ is marked.

There exist different tools for the analysis and verification of systems based on the Timed Automata (TA) model, including HyTech [HyT], KRONOS [KRO], and UPPAAL [UPP]. Such tools have been developed along many years and nowadays are quite mature and widely accepted. On the other hand, to the best of our knowledge, there are no tools that support TCTL

model checking of timed Petri nets extended with data information. In order to make use of available tools, we first translate PRES+ models into timed automata and then use one of the existing tools for model checking of timed automata. In Subsection 4.2.2 we propose a systematic procedure for translating PRES+ into timed automata

Figure 4.3 depicts our general approach to formal verification using model checking. A system is described by a PRES+ model and the properties it must satisfy are expressed by CTL/TCTL formulas. Once the PRES+ model has been translated into timed automata, the model checker automatically verifies whether the required properties hold in the model of the system. In case the specification (expressed by CTL/TCTL formulas) is not satisfied, diagnostic information is generated. Given enough computational resources, the procedure will terminate with a *yes/no* answer. However, due to the huge state space of practical systems, it might be the case that it is not feasible to obtain an answer at all, even though in theory the procedure will terminate (probably after a very long time and requiring large amounts of memory).

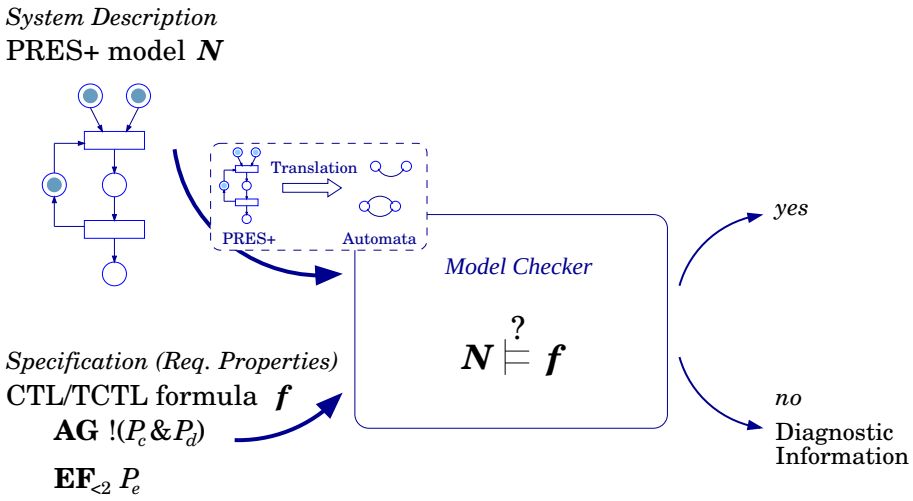


Figure 4.3: Model checking

The verification of hierarchical PRES+ models is done by constructing the equivalent non-hierarchical net as stated in Definition 3.39, and then using the procedure discussed in the next subsection to translate it into timed automata. Note that obtaining the non-hierarchical PRES+ model can be done automatically so that the designer is not concerned with how the net is flattened: he just inputs a hierarchical PRES+ model as well as the properties he is interested in.

4.2.2 Translating PRES+ into Timed Automata

For verification purposes, we translate the PRES+ model into timed automata in order to use existing model checking tools. In the procedure presented in this subsection, the resulting model will consist of one automaton and one clock for each transition in the Petri net. The PRES+ model shown in Figure 4.4 is used to illustrate the proposed translation procedure. The resulting timed automata are shown in Figure 4.5.

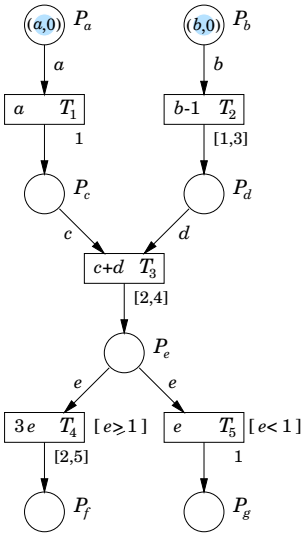


Figure 4.4: PRES+ model to be translated into automata

In the following we describe the different steps of the translation procedure. Given a PRES+ model $N = (\mathbf{P}, \mathbf{T}, \mathbf{I}, \mathbf{O}, M_0)$, we want to construct an equivalent timed automata model $\vec{\mathcal{T}} = (\mathcal{L}, \mathcal{L}_0, \mathcal{E}, \mathcal{X}, \mathfrak{x}, \mathcal{C}, \mathcal{V}, \mathfrak{c}, \nu, \mathfrak{r}, \mathfrak{a}, \mathfrak{i})$ (which is a collection of automata, each denoted $\vec{\mathcal{T}}_i$).

Step 4.1 Define one clock c_i in $\mathcal{C}$ for each transition T_i of the Petri net. Define one variable in $\mathcal{V}$ for each place P_x of the Petri net, corresponding to the token value v_x when P_x is marked. ■

The clock c_i is used to ensure the firing of the transition T_i within its earliest-latest trigger time interval. For the example shown in Figure 4.4, using the short notation x to denote v_x (the value of token K_x in place P_x), the sets of clocks and variables are, respectively, $\mathcal{C} = \{c_1, c_2, c_3, c_4, c_5\}$ and $\mathcal{V} = \{a, b, c, d, e, f, g\}$.

Step 4.2 Define the set $\mathcal{X}$ of labels as the set of transitions in the Petri net. ■

In the resulting automata, at the end of the translation process, the change of location through an edge e labeled $\mathsf{x}(e) = T_i$ will correspond to the firing of transition T_i in the Petri net. For our example, the set of labels is $\mathcal{X} = \{T_1, T_2, T_3, T_4, T_5\}$.

Step 4.3 For every transition T_i in the Petri net, define an automaton $\vec{\mathcal{T}}_i$ with $z+1$ locations named $s_0, s_1, \ldots, s_{z-1}$ and en, where $z = |{}^\circ T_i|$ is number of input places of T_i. ∎

During operation of the timed automata, automaton $\vec{\mathcal{T}}_i$ being in location s_j represents a state in which the transition T_i has j of its input places marked. When $\vec{\mathcal{T}}_i$ is in location en, it corresponds to the situation in which all input places of T_i are marked. The resulting model for the net shown in Figure 4.4 consists of five automata. The automaton $\vec{\mathcal{T}}_3$, for instance, has three locations: s_0 corresponds to no tokens in the input places of T_3, s_1 corresponds to a token in one of the input places, and en corresponds to T_3 having both input places marked.

Step 4.4 Let $pr(T_i) = \{T \in \mathbf{T} \setminus \{T_i\} \mid T^\circ \cap {}^\circ T_i \neq \emptyset\}$ be the set of transitions different from T_i that, when fired, put a token in some place of the pre-set of T_i. Let $cf(T_i) = \{T \in \mathbf{T} \setminus \{T_i\} \mid {}^\circ T \cap {}^\circ T_i \neq \emptyset\}$ be the set of transitions that are in conflict with T_i. Given the automaton $\vec{\mathcal{T}}_i$, corresponding to transition T_i, for every $T_x \in pr(T_i) \cup cf(T_i)$:

(a) If $m = |T_x^\circ \cap {}^\circ T_i| - |{}^\circ T_x \cap {}^\circ T_i| > 0$, define edges (s_0, s_{0+m}), (s_1, s_{1+m}), …, (s_{z-m}, en), each with label T_x;

(b) If $m = |T_x^\circ \cap {}^\circ T_i| - |{}^\circ T_x \cap {}^\circ T_i| < 0$, define edges (en, s_{z+m}), (s_{z-1}, s_{z-1+m}), …, (s_{0-m}, s_0), each with label T_x;

(c) If $m = |T_x^\circ \cap {}^\circ T_i| - |{}^\circ T_x \cap {}^\circ T_i| = 0$, define edges (s_0, s_0), (s_1, s_1), …, (en, en), each with label T_x.

Then define one edge (en, s_n) with synchronization label T_i, where $n = |T_i^\circ \cap {}^\circ T_i|$. ∎

The above step captures how an automaton $\vec{\mathcal{T}}_i$ changes its location, in accordance to how the marking of transition T_i (more precisely the number of its input places that are marked) changes when a transition T_x, that either deposits or removes tokens in/from input places of T_i, fires. For example, the firing of a transition T_x that puts one token in one of the input places of T_i, corresponds to a change of location from s_j to s_{j+1} in the automaton $\vec{\mathcal{T}}_i$, through an edge labeled T_x (recall that s_j in the automaton $\vec{\mathcal{T}}_i$ represents a state corresponding to the situation in which the transition T_i has j of its input places marked).

Let us take, for example, the transition T_3 in the model shown in Figure 4.4. In this case $pr(T_3) = \{T_1, T_2\}$ and $cf(T_3) = \emptyset$. Since

$|T_1^\circ \cap {}^\circ T_3| - |{}^\circ T_1 \cap {}^\circ T_3| = 1$, for the automaton $\vec{\mathcal{T}}_3$, there are two edges (s_0, s_1) and (s_1, en) with label T_1. Since $|T_2^\circ \cap {}^\circ T_3| - |{}^\circ T_2 \cap {}^\circ T_3| = 1$, there are also two edges (s_0, s_1) and (s_1, en) but with label T_2 as shown in Figure 4.5. The one edge that has label T_3 is (en, s_0) (which means that, after firing T_3, all places in its pre-set ${}^\circ T_3$ get no tokens; this is due to the fact that $|T_3^\circ \cap {}^\circ T_3| = 0$).

Let us consider as another example the automaton $\vec{\mathcal{T}}_4$ corresponding to transition T_4. In this case $pr(T_4) = \{T_3\}$ and $cf(T_4) = \{T_5\}$. Corresponding to T_3, since $|T_3^\circ \cap {}^\circ T_4| - |{}^\circ T_3 \cap {}^\circ T_4| = 1$, there is an edge (s_0, en) with label T_3. Corresponding to T_5, since $|T_5^\circ \cap {}^\circ T_4| - |{}^\circ T_5 \cap {}^\circ T_4| = -1$, there is an edge (en, s_0) with label T_5. The automaton $\vec{\mathcal{T}}_4$ must have another edge (en, s_0), this one labeled T_4.

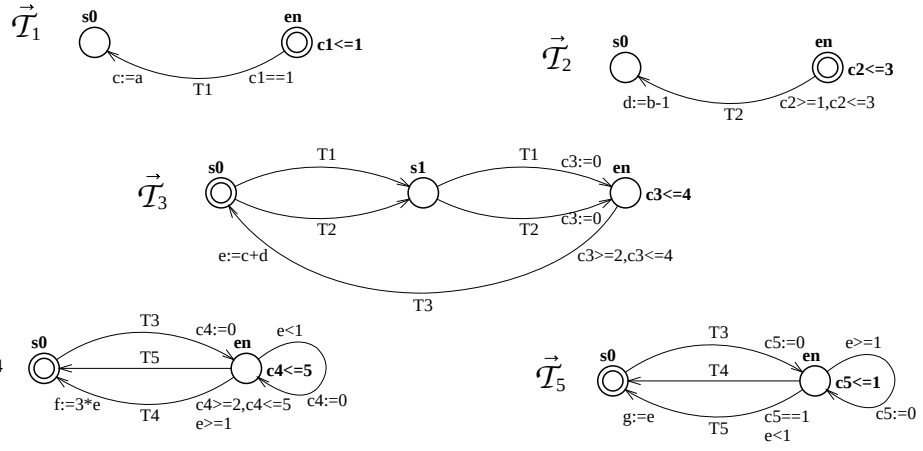


Figure 4.5: Timed automata equivalent to the PRES+ model of Figure 4.4

In the following, let f_i be the transition function associated to T_i, ${}^\circ T_i$ the pre-set of T_i, and τ_i^{bc} and τ_i^{wc} the best-case and worst-case transition delays of T_i.

Step 4.5 Given the automaton $\vec{\mathcal{T}}_i$, for every edge $e_k = (s_j, en)$ define $\mathtt{r}(e_k) = \{c_i\}$. For any other edge e in $\vec{\mathcal{T}}_i$ define $\mathtt{r}(e) = \emptyset$. ∎

This means that the clock c_i will be reset in all edges coming into location en in the automaton $\vec{\mathcal{T}}_i$. In Figure 4.5, the assignment $c_i := 0$ represents the reset of c_i. The two edges (s_1, en) of automaton $\vec{\mathcal{T}}_3$, for example, have $c_3 := 0$ inscribed on them; the clock c_3 is used to take into account the time since T_3 becomes enabled and thus ensure to the firing semantics of PRES+.

Step 4.6 Given the automaton $\vec{\mathcal{T}}_i$, define the invariant $\mathtt{i}(en)$ of location en as $c_i \leq \tau_i^{\text{wc}}$ in order to account for the time semantics of PRES+ in which

the firing of T_i occurs before or at its latest trigger time. For other locations s_j different from en, define their invariant $\mathfrak{i}(s_j)$ as *true* (a condition that is always satisfied). ∎

In Figure 4.5, only the invariants of locations en are shown. For instance, $c_2 \leq 3$ is the location invariant of en in the timed automaton $\vec{\mathcal{T}}_2$.

Step 4.7 Given the automaton $\vec{\mathcal{T}}_i$, assign the clock condition $\tau_i^{\text{bc}} \leq c_i \leq \tau_i^{\text{wc}}$ to the one edge $e = (en, s_n)$ (where $n = |T_i^\circ \cap {}^\circ T_i|$) labeled T_i. Assign the clock condition *true* to all other edges different from (en, s_n). ∎

For example, in the case of automaton $\vec{\mathcal{T}}_2$ the condition $1 \leq c_2 \leq 3$ over the edge (en, s_0) gives the lower and upper limits for the firing of T_2.

Step 4.8 Given the automaton $\vec{\mathcal{T}}_i$, assign to the one edge $e = (en, s_n)$ with label T_i the activities $v_k := f_i$, for every $P_k \in T_i^\circ$. Assign no activities to other edges. ∎

The above step indicates that when changing from location en to location s_n through the one edge labeled T_i in the automaton $\vec{\mathcal{T}}_i$, the variables corresponding to out places of T_i in the Petri net will be updated according to the function f_i. This is in accordance with the firing rule of PRES+ which states that token values of tokens in the post-set of a firing transition T_i are calculated by evaluating its transition function f_i. For instance, in Figure 4.5 the activity $d := b - 1$ expresses that whenever the automaton $\vec{\mathcal{T}}_2$ changes from en to s_0 the value $b - 1$ is assigned to the variable d.

Step 4.9 Given the automaton $\vec{\mathcal{T}}_i$, if the transition T_i in the PRES+ model has guard g_i, assign the variable condition g_i to the one edge (en, s_n) (where $n = |T_i^\circ \cap {}^\circ T_i|$) with label T_i. Then add an edge $e = (en, en)$ with no label, condition $\overline{g_i}$ (the complement of g_i), and $\mathfrak{r}(e) = \{c_i\}$. ∎

When all input places of a transition T_i are marked (the corresponding timed automaton is in location en) but its guard g_i is not satisfied, the transition may not fire. This means that in such a case the corresponding automaton $\vec{\mathcal{T}}_i$ may not change its location through the edge labeled T_i. Note, for example, the condition $e < 1$ assigned to the edge (en, s_0) with label T_5 in the automaton $\vec{\mathcal{T}}_5$: $e < 1$ represents the guard of T_5. Observe also the edge (en, en) with condition $e \geq 1$ and $c_5 := 0$.

Step 4.10 If there are k places initially marked in the pre-set ${}^\circ T_i$ of the transition T_i, make s_k the initial location of $\vec{\mathcal{T}}_i$. If all places in ${}^\circ T_i$ are initially marked, make en the initial location of $\vec{\mathcal{T}}_i$. ∎

For the example discussed in this subsection, *en* is the initial location of $\vec{\mathcal{T}}_1$ because the only input place of transition T_1 is marked for the initial marking of the net. Since no place in ${}^\circ T_3$ is initially marked, the automaton $\vec{\mathcal{T}}_3$ has s_0 as initial location.

In Figures 4.6 and 4.7 we draw a parallel of the dynamic behavior of the PRES+ model used throughout this subsection and its corresponding equivalent time automata model, for a particular firing sequence. Observe how the locations of the automata change according to the given firing sequence.

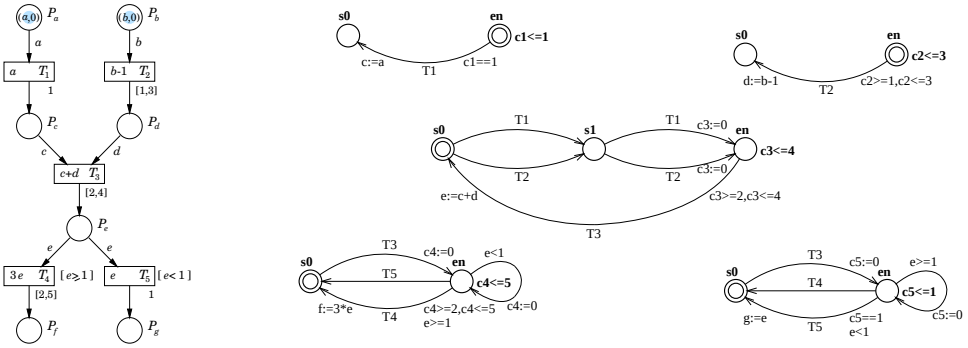
Once we have the equivalent timed automata, we can verify properties against the model of the system. For instance, in the simple system of Figure 4.4 we could check whether, for given values of a and b, there exists a reachable state in which P_f is marked. This property can be expressed by the CTL formula $\mathbf{EF}\, P_f$. If we want to check temporal properties we can express them as TCTL formulas. Thus, for example, we could check whether P_g will possibly be marked and the time stamp of its token be less than 5 time units, expressing this property as $\mathbf{EF}_{<5}\, P_g$.

Some of the model checking tools, namely HyTech [HyT], are capable of performing parametric analyses. Then, for the example shown in Figure 4.4, we can ask the model-checker which values of a and b make a certain property hold in the system model. For instance, we obtain that $\mathbf{EF}\, P_g$ holds if $a + b < 2$.

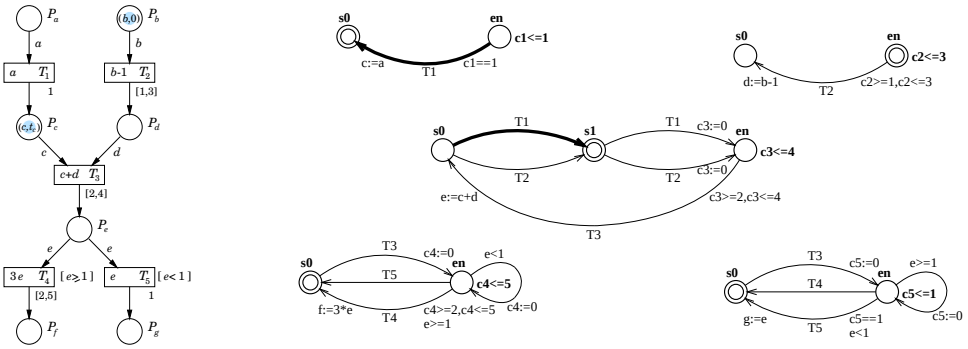
Due to the nature of the model checking tools that we use, the translation procedure introduced above is applicable for PRES+ models in which transition functions are expressed using arithmetic operations and token types of all places are rational. In this case, we could even reason about token values. Recall, however, that we want to focus on reachability and time analyses. From this perspective we can ignore transition functions if they affect neither the absence/presence of tokens nor time stamps. This is the case of PRES+ models that bear no guards and, therefore, they can straightforwardly be verified even if their transition functions are very complex operations, because we simply ignore such functions. Those systems that do include guards in their PRES+ model may also be studied if guard dependencies can be stated by linear expressions. This is the case of the system shown in Figure 4.4. There are many systems in which the transition functions are not linear, but their guard dependencies are, and thus we can inscribe such dependencies as linear expressions and use our method for system verification.

4.3 Verification of an ATM Server

We illustrate in this section the verification of a practical system modeled using PRES+. The net shown in Figure 4.8 represents an ATM-based Vir-



... T_1 fires ...



... T_2 fires ...

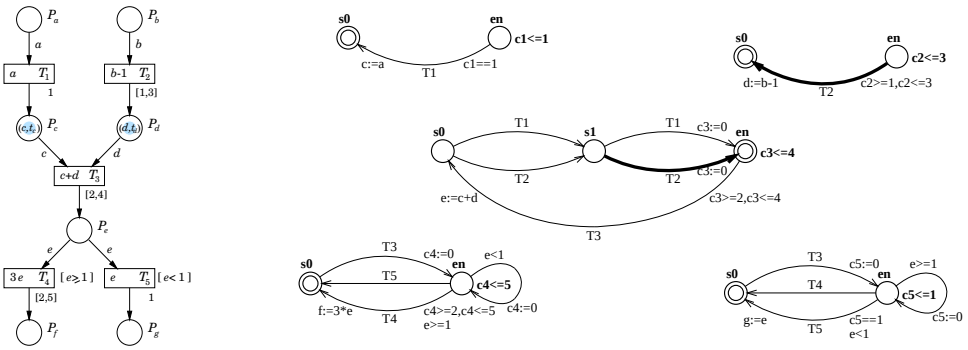
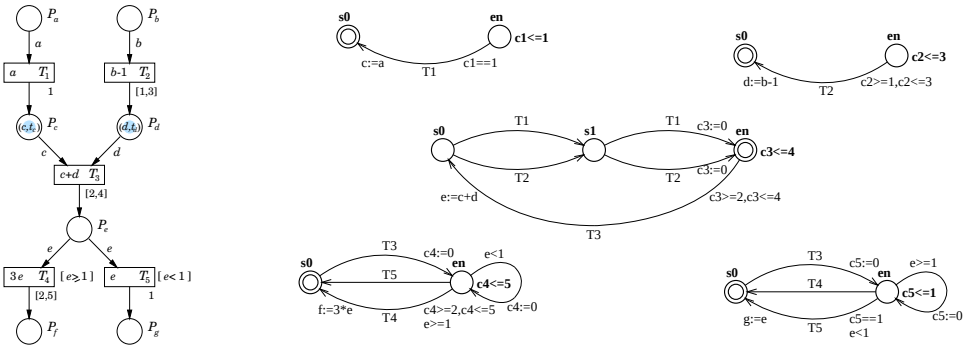
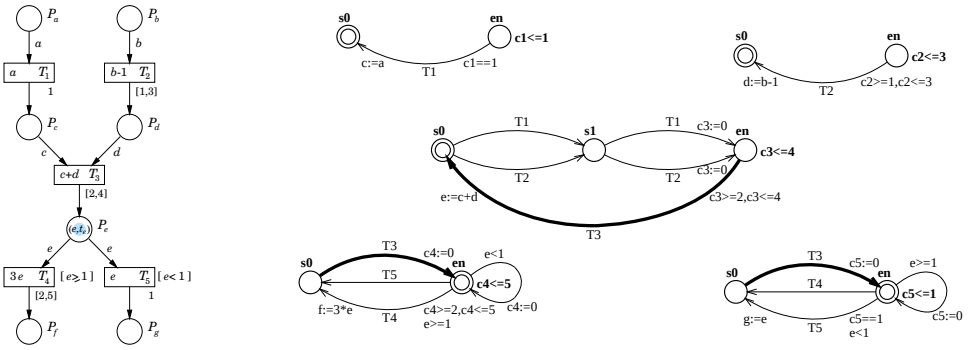


Figure 4.6: Dynamic behavior of a PRES+ model and its equivalent timed automata model

tual Private Network (A-VPN) server [FLL+98]. The behavior of the system can be briefly described as follows. Incoming cells are examined by *Check* in



... T_3 fires ...



... T_4 fires ...

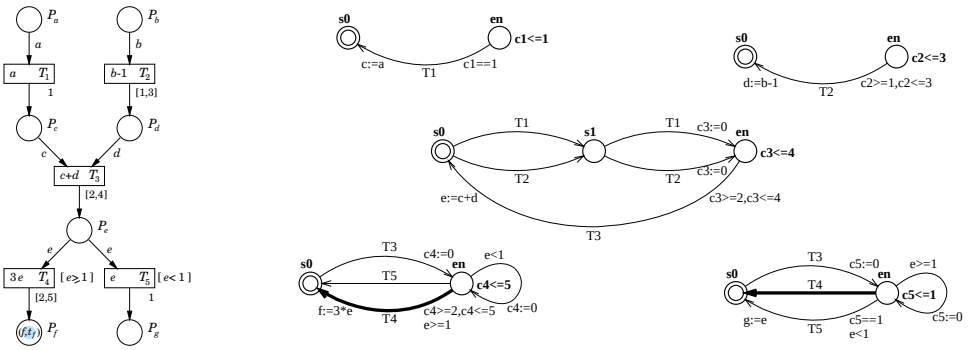


Figure 4.7: Dynamic behavior of a PRES+ model and its equivalent timed automata model

order to determine whether they are faulty. Fault-free cells arrive through the *UTOPIA_Rx* interface and are eventually stored in the *Shared Buffer*.

If the incoming cell is faulty, it goes through the module *Faulty* and then is sent out using the *UTOPIA_Tx* interface without processing. The module *Address Lookup* checks the *Lookup Memory* and, for each non-defective input cell, a compressed form of the Virtual Channel (VC) identifier in the cell header is computed. With this compressed form of the VC identifier, the module *Traffic* checks its internal tables and decides whether to accept the incoming cell or discard it in order to avoid congestion. If the cell is accepted, *Traffic* gives instructions to *Queue Manager* indicating where to store the incoming cell in the buffer. *Traffic* also indicates to *Queue Manager* the cell (stored in Shared Buffer) to be output. *Supervisor* is the module in charge of updating internal tables of *Traffic* and the *Lookup Memory*. The selected outgoing cell is emitted through the module *UTOPIA_Tx*. The specification of the system includes a time constraint given by the rate (155 Mbit/s) of the application: one input cell and one output cell must be processed every 2.7 μs.

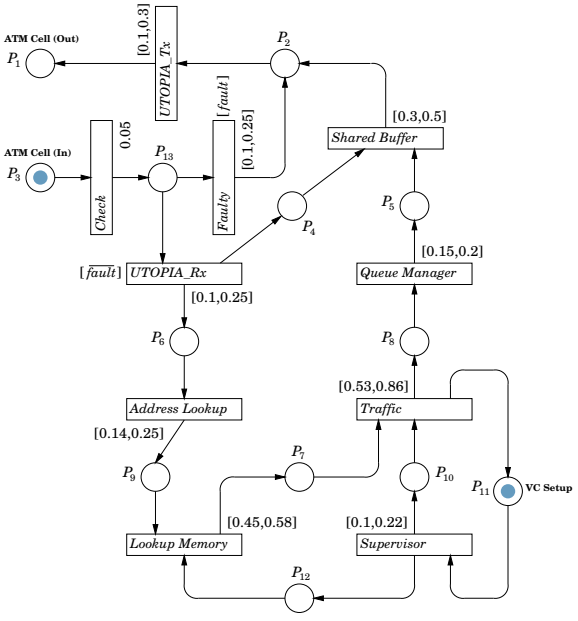


Figure 4.8: PRES+ model of an A-VPN server

In order to verify the correctness of the A-VPN server, we must prove that the system will eventually complete its functionality and that such a functionality will eventually fit within a cell time-slot. The completion of the task of the A-VPN server, modeled by the net in Figure 4.8, is represented by the state (marking) in which the place P_1 is marked. Then we must prove that for all computation paths, P_1 will eventually get a token and its time

stamp will be less than 2.7 μs. These conditions might straightforwardly be specified using CTL and TCTL formulas, namely $\mathbf{AF}\, P_1$ and $\mathbf{AF}_{<2.7}\, P_1$. Note that the first formula is a necessary condition for the second one. Using the translation procedure described in Subsection 4.2.2 and, in this case, the HyTech tool, we found out that the CTL formula $\mathbf{AF}\, P_1$ holds while the TCTL formula $\mathbf{AF}_{<2.7}\, P_1$ does not. Therefore the specification (set of required properties) is not fulfilled by the model shown in Figure 4.8 because it is not guaranteed that the time constraint will be satisfied.

We can consider an alternative solution. To do so, suppose we want to modify *Traffic*, keeping its functional behavior but seeking superior performance: we want to explore the allowed interval of delays for *Traffic* in order to fulfill the system constraints. We can define the best-case and worst-case transition delays of *Traffic* as parameters $\tau^{\mathrm{bc}}_{\mathit{Traffic}}$ and $\tau^{\mathrm{wc}}_{\mathit{Traffic}}$, and then use HyTech in order to perform a parametric analysis and find out the values for which $\mathbf{AF}_{<2.7}\, P_1$ is satisfied. We get that if $\tau^{\mathrm{wc}}_{\mathit{Traffic}} < 0.57$ and, by definition, $\tau^{\mathrm{bc}}_{\mathit{Traffic}} \leq \tau^{\mathrm{wc}}_{\mathit{Traffic}}$ then the property $\mathbf{AF}_{<2.7}\, P_1$ holds. This indicates that the worst-case execution time of the function associated to *Traffic* must be less than 0.57 μs in order to fulfill the system specification.

Running the HyTech tool on a Sun Ultra 10 workstation, both the verification of the TCTL formula $\mathbf{AF}_{<2.7}\, P_1$ for the model given in Figure 4.8 and the parametric analysis described in the above paragraph take roughly 1 s.

We have presented in this chapter an approach to the formal verification of PRES+ models. We studied a practical system for which verification can be performed in reasonable time. Nonetheless, it is possible to improve verification efficiency in different ways, as will be discussed in Chapter 5.

Chapter 5

Improving Verification Efficiency

We presented in Chapter 4 our approach to the formal verification of systems modeled in PRES+. In order to use available model checking tools, a systematic procedure for translating PRES+ models into timed automata was defined in Subsection 4.2.2. In the sequel this method will be referred to as *naive* translation. According to such a translation procedure, the resulting model consists of a collection of timed automata that operate and coordinate with each other through shared variables and synchronization labels: one automaton with one clock variable is obtained for each transition of the PRES+ net. However, since the complexity of model checking of timed automata grows exponentially in the number of clocks, the verification of medium or large systems would take excessive time.

In this chapter we present two different ways of improving verification efficiency: first, applying correctness-preserving transformations in order to simplify the PRES+ model of the system (Section 5.1); second, exploiting the information about the degree of concurrency of the system in order to improve the translation procedure into timed automata (Section 5.2).

5.1 Improvement of Verification Efficiency by Using Transformations

The application of transformations in the verification of systems represented in PRES+ is addressed in this section. The verification efficiency can be improved considerably by using a transformational approach. The model that we use, PRES+, supports such transformations, which is of great benefit during the formal verification phase.

For the sake of reducing the verification effort, we first transform the system model into a simpler but semantically equivalent one, and then verify the simplified model. If a given model is modified using *correctness-preserving* transformations and then the resulting one is proved correct with respect to its specification, the initial model is guaranteed to be correct, and no intermediate steps need to be verified. This simple observation allows us to improve the verification efficiency.

5.1.1 Transformations

As it was argued in Subsection 3.5.2, the concept of hierarchy makes it possible to model systems in a structured way. Thus, using the notion of abstraction/refinement, the system may be broken down into a set of comprehensible nets.

Transformations performed on large and flat systems are, in general, difficult to handle. Hierarchical modeling permits a structural representation of the system in such a way that the composing (sub)nets are simple enough to be transformed efficiently.

We can define a set of transformation rules that make it possible to transform a part of the system model. A simple yet useful transformation is shown in Figure 5.1. It is not difficult to prove the validity of this transformation rule (see Section B.1 of the Appendix B). It is worthwhile to observe that if the net N' is a refinement of a certain super-transition $ST_x \in \mathbf{ST}$ in the hierarchical net $H = (\mathbf{P}, \mathbf{T}, \mathbf{ST}, \mathbf{I}, \mathbf{O}, M_0)$ and N' is transformed into N'', then N'' is also a refinement of ST_x and may be used instead of N'. Such a transformation does not change the overall system at all. First, having tokens with the same token value and token time in corresponding in-ports of the subnets N' and N'' will lead to a marking with the same token value and time in corresponding out-ports, so that the external observer (that is, the rest of the net H) cannot distinguish between N' and N''. Second, once tokens are put in the in-ports of the subnets N' and N'', there is nothing that externally "disturbs" the behavior of N' and N'' (for example, a transition in conflict with the in-transition that could take away tokens from the in-ports) because, by definition, super-transitions may not be in conflict. Thus the overall behavior remains the same when using either N' or N''. Such a transformation rule can therefore be used with the purpose of simplifying PRES+ models and accordingly improving the efficiency of the verification process.

It is worth clarifying the concept of transformation in the context of verification. Along the design flow, the system model is refined to include different design decisions, like architecture selection, partitioning, and scheduling (see Figure 1.1). Such refinements are what we call *vertical transformations*.

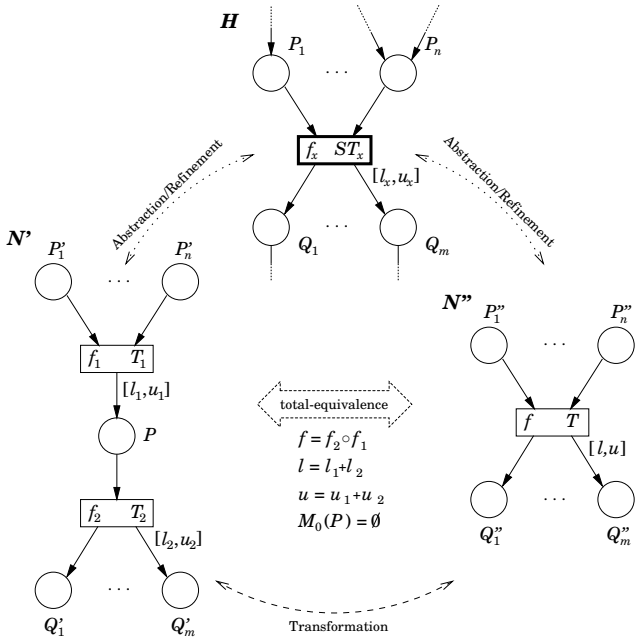


Figure 5.1: Transformation rule TR1

On the other hand, at certain stage of the design flow, the system model can be transformed into another one that preserves certain properties under consideration and, at the same time, makes the verification process easier. These are called *horizontal transformations*.

Horizontal transformations are a mathematical tool for dealing with verification complexity. By simplifying the representation to be model-checked, the verification cost is reduced in a significant manner. We concentrate on horizontal transformations, that is, those that help us improve the efficiency of the verification process.

Figure 5.2(a) depicts how the system model, at a given phase of the design flow, is verified. The model together with the required properties p are input to the model checking tool with the purpose of finding out whether the model conforms to the desired properties. It is possible to do better by trying to apply horizontal transformations in order to get a simpler model, yet semantically equivalent with respect to the properties p. Our transformational approach to verification is illustrated in Figure 5.2(b). If the transformations are p-preserving, only the simplest model is to be verified and there is no need to model-check intermediate steps, thus saving time during verification.

Other transformation rules are presented in Figures 5.3 through 5.7. It is assumed that the nets involved in the transformations refine a certain

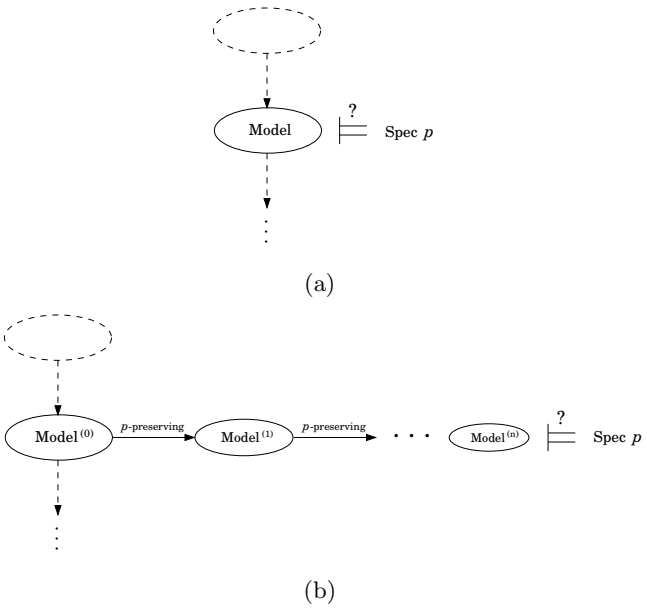


Figure 5.2: Usage of transformations for improving verification efficiency

super-transition.

We may take advantage of transformations aiming at improving verification efficiency. The idea is to get a simpler model using transformations from a library. In the case of total-equivalence transformations, since an external observer cannot distinguish between two total-equivalent subnets (for the same tokens in corresponding in-ports, the observer gets in both cases the same tokens in corresponding out-ports), the global system properties are preserved in terms of reachability, time, and functionality. Therefore such transformations are correctness-preserving: if a property p holds in a net that contains a subnet N'' into which a total-equivalent subnet N' has been transformed, q is also satisfied in the net that contains N'; if q does not hold in the second net, it does not in the first either.

If the system model does not have guards, we can ignore transition functions as reachability and time analyses (which are the focus of our verification approach) will not be affected by token values. In such a case, we can use time-equivalence transformations in order to obtain a simpler model, as they preserve properties related to absence/presence of tokens in the net as well as time stamps of tokens.

Once the system model has been transformed into a simpler but semantically equivalent one, we can formally verify the latter by applying the approach described in Chapter 4.

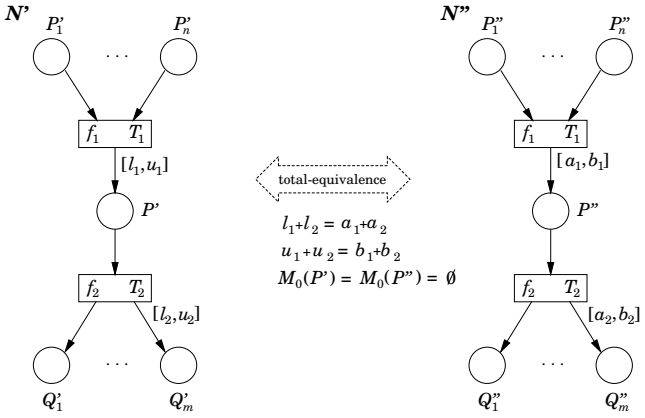


Figure 5.3: Transformation rule TR2

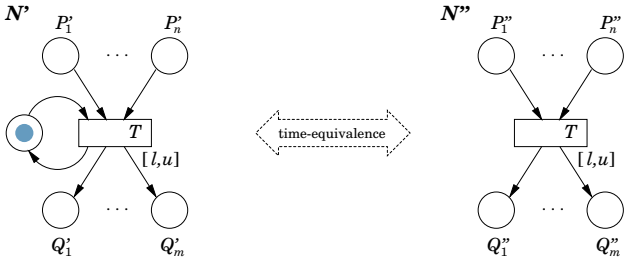


Figure 5.4: Transformation rule TR3

5.1.2 Verification of the GMDFα

In this subsection we verify the GMDFα (Generalized Multi-Delay frequency-domain Filter) modeled using PRES+ in Subsection 3.6.1. We illustrate the benefits of using transformations in the verification of the filter.

We consider two cases of a GMDFα of length 1024: a) with an overlapping factor of 4, we have the following parameters: $L = 1024$, $\alpha = 4$, $K = 4$, $N = 256$, and $R = 64$; b) with an overlapping factor of 2, we have the following parameters: $L = 1024$, $\alpha = 2$, $K = 8$, $N = 128$, and $R = 64$. Having a sampling rate of 8 kHz, the maximum execution time for one iteration is in both cases 8 ms (64 new samples must be processed at each iteration). The completion of one iteration is determined by the marking of the place E'.

We want to prove that the system will eventually complete its functionality. According to the time constraint of the system, it is not sufficient to finish the filtering iteration but also to do so with a bound on time (8 ms). This aspect of the specification is captured by the TCTL formula $\mathbf{AF}_{<8}\, E'$. At this point, our task is to verify that the model of the GMDFα shown in

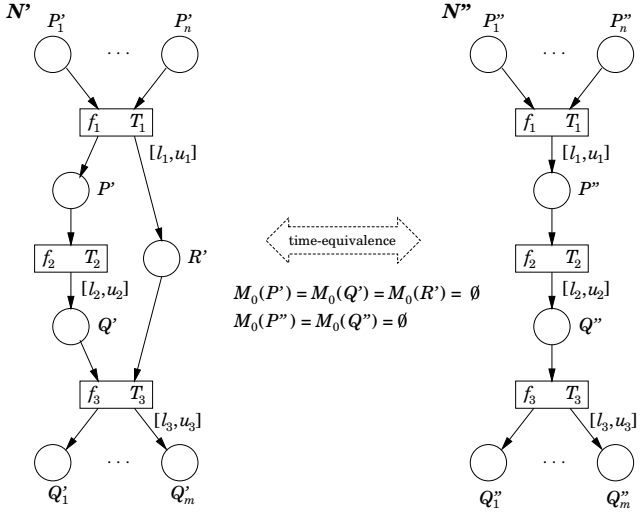


Figure 5.5: Transformation rule TR4

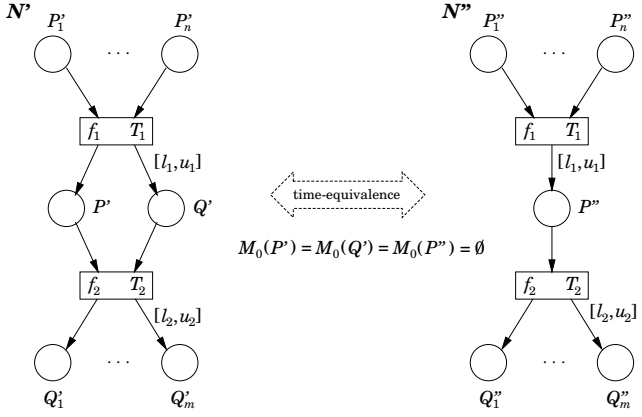


Figure 5.6: Transformation rule TR5

Figure 3.12 satisfies the formula $\mathbf{AF}_{<8}\, E'$.

A straightforward way could be flattening the system model and applying directly the verification technique discussed in Chapter 4. However, a wiser approach would be trying to first simplify the system model by transforming it into an equivalent one, through transformations from a library. Such transformations are a mathematical tool that allows us to improve the verification efficiency. Therefore we try to reduce the model aiming at obtaining a simpler one, still semantically equivalent from the point of view of reachability and time analyses, in such a way that correctness is preserved.

We start by using the transformation rule TR1 illustrated in Figure 5.1 on

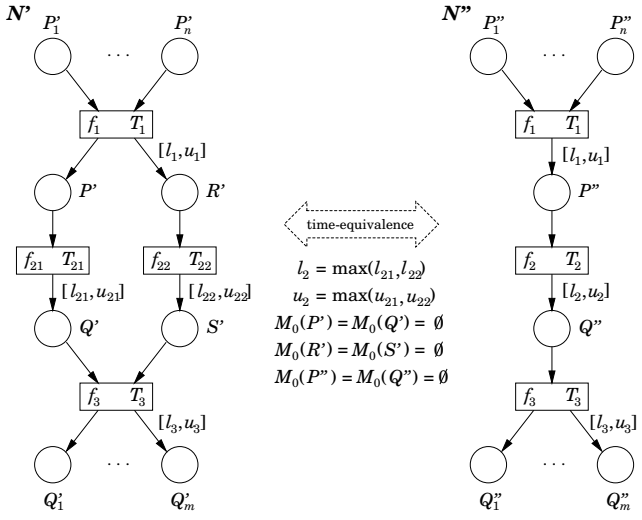


Figure 5.7: Transformation rule TR6

the refinement of the basic cell (Figure 5.8(a)), so that we obtain the subnet shown in Figure 5.8(b). Note that in this transformation step, no time is spent on-line in proving the transformation itself because transformations are proved off-line (only once) and stored in a library. Since the subnets of Figures 5.8(a) and 5.8(b) are total-equivalent, the functionality of the entire GMDFα, so far, remains unchanged. We may also use time-equivalence transformations because the PRES+ model of the GMDFα has no guards. Recall that time-equivalence transformations do not affect reachability and time analyses for models without guards. Using the transformation rule TR3 presented in Figure 5.4, it is possible to obtain a simpler representation of the basic cell as shown in Figure 5.8(c). We apply again the transformation rule TR1, obtaining thus the subnet shown in Figure 5.8(d), and continue until the basic cell refinement is further simplified into the single-transition subnet of Figure 5.8(e). Finally we check the specification against the simplest model of the system, that is, the one in which the refinement of the basic cells $ST_{3.i}$ is the subnet shown in Figure 5.8(e). We have verified the formula $\mathbf{AF}_{<8}\, E'$ and the model of the GMDFα indeed satisfies its specification for both $K = 4$ and $K = 8$. The verification times using the model checking tool UPPAAL are shown in the last row of Table 5.1.

Since the transformations used along the simplification of the GMDFα model are correctness-preserving, the initial model of Figure 3.12 is also correct, that is, it satisfies the system specification, and therefore need not be verified. Nonetheless, in order to illustrate the verification cost (time) at different stages, we have verified the models obtained in the intermediate

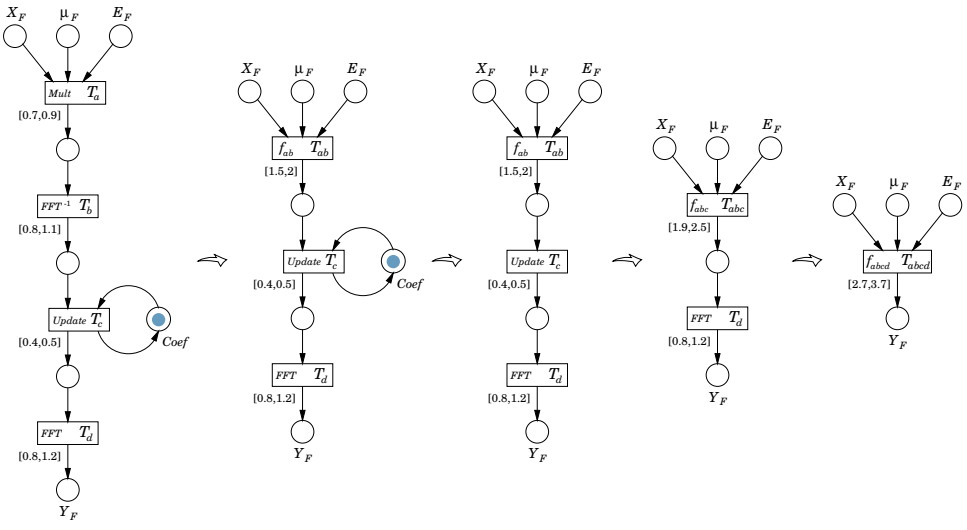


Figure 5.8: Transformations of the GMDFα basic cell

steps (models in which the refinements of the basic cells $ST_{3.i}$ are given by the subnets shown in Figures 5.8(b) through 5.8(d)) as well as the initial model. The results are shown in Table 5.1. Recall, however, that this is not needed as long as the transformation rules preserve the correctness in terms of reachability and time analyses. It can be noted how much effort is saved when the basic cells $ST_{3.i}$ are refined by the simplest net as compared to the original model.

Refinement of the basic cell	Verification time [s]	
	$\alpha = 4$, $K = 4$	$\alpha = 2$, $K = 8$
Figure 5.8(a)	108.9	NA*
Figure 5.8(b)	61.8	8178.9
Figure 5.8(c)	61.1	8177.2
Figure 5.8(d)	9.8	1368.1
Figure 5.8(e)	0.9	9.7

* Not available: out of time

Table 5.1: Verification times for the GMDFα

In this way verification is carried out at low cost (short time) by first using correctness-preserving transformations aiming at simplifying the system representation. If the simpler model is correct (its specification holds), the initial one is guaranteed to be correct and intermediate steps need not be

verified.

5.2 Improvement of Verification Efficiency by Coloring the Concurrency Relation

We proposed in Subsection 4.2.2 a systematic procedure for translating PRES+ models into timed automata. Such a procedure produces a collection of timed automata where one automaton with one clock variable is obtained for each transition of the PRES+ net. In order to improve verification efficiency, the translation method introduced in Subsection 4.2.2 can be enhanced by exploiting the structure of the PRES+ net and, in particular, by extracting the information about the degree of concurrency of the system.

Since the time complexity of model checking of timed automata is exponential in the number of clocks, the translation into timed automata is crucial for our verification approach. We must therefore try to find an optimal or near-optimal solution in terms of number of resulting clocks/automata. This section introduces a technique called *coloring* that utilizes the information on the degree of concurrency of the system, with the aim of obtaining the smallest collection of automata resulting from the translation procedure.

5.2.1 Computing the Concurrency Relation

The first step of the method discussed in this section is to find out the pairs of transitions that may not fire at the same time for any reachable marking. Thus, for example, if we know that there is no reachable marking for which two given transitions may fire in parallel, we can use *one* clock for accounting for the firing time semantics of *both* transitions because they cannot fire simultaneously.

We use the concept of *concurrency relation* (Definition 5.2), a relation that includes the pairs of transitions that can fire concurrently. In order to compute the concurrency relation we work on the Petri net corresponding to a given PRES+ model, that is, we take a regular Petri net (as defined by the classical model—see Definition 3.1) that has the same sets of places and transitions as well as input and output arcs as the original PRES+ model. For example, Figure 5.9 shows the underlying regular Petri net of the PRES+ model shown in Figure 3.2. Recall that in the case of regular Petri nets, the marking M is a function $M : \mathbf{P} \rightarrow \mathbb{N}_0$ from the set of places $\mathbf{P}$ to the set of non-negative integers $\mathbb{N}_0$.

Note that if we find out that two transitions may not fire at the same time in the regular Petri net, it is guaranteed that these two transitions may not fire in parallel in the PRES+ model from which the Petri net was derived.

This is due to the fact that the behavior of a PRES+ model (in terms of transition firings) is a subset of the behavior of its underlying Petri net, because the in former transitions are time-bounded and may have guards.

It is worthwhile to mention that although the focus of our verification approach is safe PRES+ models, the discussion in this subsection is also applicable to non-safe nets.

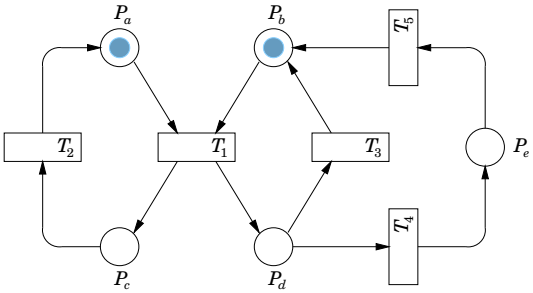


Figure 5.9: Petri net corresponding to the PRES+ model of Figure 3.2

Definition 5.1 [Kov00] Let $N = (\mathbf{P}, \mathbf{T}, \mathbf{I}, \mathbf{O}, M_0)$ be a Petri net and let $\mathbf{X} = \mathbf{P} \cup \mathbf{T}$ be the set of places and transitions. Given $X \in \mathbf{X}$, the marking M_X is defined as follows:

(i) If X is a place, M_X is the marking that puts one token in X and no tokens elsewhere;

(ii) If X is a transition, M_X is the marking that puts one token in every input place of X and no tokens elsewhere. ❑

Definition 5.2 [Kov00] The *concurrency relation* $\| \subseteq \mathbf{X} \times \mathbf{X}$ of a Petri net $N = (\mathbf{P}, \mathbf{T}, \mathbf{I}, \mathbf{O}, M_0)$ is the set of pairs (X, X') such that $M \geq M_X + M_{X'}$ for some reachable marking M, where $\mathbf{X} = \mathbf{P} \cup \mathbf{T}$. ❑

In particular, two places belong to the concurrency relation if they are simultaneously marked for some reachable marking, and two transitions belong to the concurrency relation if they can fire concurrently for some reachable marking. We are specifically interested in the set of transitions that might fire at the same time, as stated by the following definition.

Definition 5.3 The *concurrency relation on* $\mathbf{T}$, denoted $\|_{\mathbf{T}}$, of a Petri net $N = (\mathbf{P}, \mathbf{T}, \mathbf{I}, \mathbf{O}, M_0)$ is the set of pairs of transitions (T, T') such that T and T' can fire concurrently for some reachable marking of N. ❑

Definition 5.4 [Kov00] Let $N = (\mathbf{P}, \mathbf{T}, \mathbf{I}, \mathbf{O}, M_0)$ be a Petri net and let $\mathbf{X} = \mathbf{P} \cup \mathbf{T}$ be the set of places and transitions. The *structural concurrency relation* $\|^{\mathcal{S}} \subseteq \mathbf{X} \times \mathbf{X}$ is the smallest symmetric relation such that:

(i) For all $P, P' \in \mathbf{P}$, $M_0 \geq M_P + M_{P'} \Rightarrow (P, P') \in \|^{\mathcal{S}}$;
(ii) For all $T \in \mathbf{T}$, $(T^\circ \times T^\circ) \setminus i_{\mathbf{P}} \subseteq \|^{\mathcal{S}}$;
(iii) For all $X \in \mathbf{X}$ and for all $T \in \mathbf{T}$, $\{X\} \times {}^\circ T \subseteq \|^{\mathcal{S}} \Rightarrow (X, T) \in \|^{\mathcal{S}}$ and $\{X\} \times T^\circ \subseteq \|^{\mathcal{S}}$,

where $i_{\mathbf{P}}$ denotes the identity relation on $\mathbf{P}$. ❑

The condition (i) of Definition 5.4 states that any two places initially marked are structurally concurrent; condition (ii) states that all output places of a given transition are pair-wise structurally concurrent; condition (iii) states that if a node (place or transition) is structurally concurrent with all input places of a certain transition, then that node is also structurally concurrent with all output places of the transition [KE96].

Two very important theoretical results, proved by Kovalyov [Kov92], in the context of computing the concurrency relation are the following:

(a) For live and extended free-choice Petri nets, $\| = \|^{\mathcal{S}}$;
(b) For any Petri net, $\| \subseteq \|^{\mathcal{S}}$.

There exist polynomial-time algorithms for computing the structural concurrency relation $\|^{\mathcal{S}}$, even in the case of arbitrary Petri nets [Kov00], [KE96]. Therefore, due to the above theoretical results, computing the concurrency relation $\|$ (and consequently $\|_{\mathbf{T}}$) of a live and extended free-choice Petri net can be done in polynomial time. If the net is not live but extended free-choice, the concurrency relation $\|$ can still be computed in polynomial time [Yen91].

If the Petri net is not extended free-choice, computing its concurrency relation $\|$ can take exponential time in the worst case [Esp98]. However, it should be observed that we can compute $\|^{\mathcal{S}}$ (in polynomial time) and exploit the result stating that $\| \subseteq \|^{\mathcal{S}}$: if we find that $(T, T') \notin \|^{\mathcal{S}}$, then we are certain that $(T, T') \notin \|$ (therefore $(T, T') \notin \|_{\mathbf{T}}$). Thus we can still take advantage of this fact for the purpose of reducing the number of automata/clocks resulting from the translation of PRES+ into timed automata, as will be explained in Subsection 5.2.2.

The algorithm that we use for computing the structural concurrency relation of a Petri net is given by Algorithm 5.1. This algorithm has been derived from the notions and results presented in [Kov00] and [KE96].

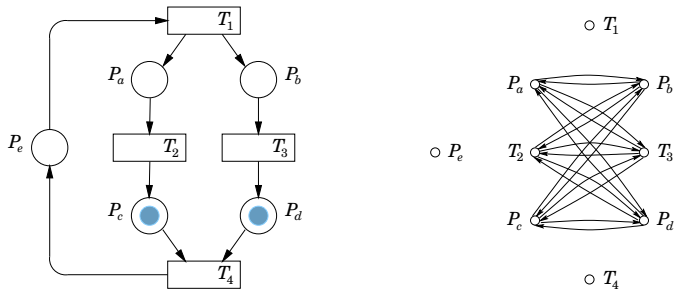
The structural concurrency relation $\|^{\mathcal{S}}$ (as computed by Algorithm 5.1) of the Petri net given in Figure 5.10(a) is shown in Figure 5.10(b). In this case, this also corresponds to the concurrency relation $\|$.

Algorithm 5.1 has a time complexity $\mathcal{O}(|\mathbf{P}|^2 \cdot |\mathbf{T}| \cdot |\mathbf{X}|)$, where $\mathbf{X} = \mathbf{P} \cup \mathbf{T}$. This algorithm computes the structural concurrency relation $\|^{\mathcal{S}}$ of arbitrary Petri nets. In the particular case of live and extended free-choice Petri nets, it is possible to compute $\|^{\mathcal{S}}$ more efficiently in $\mathcal{O}(|\mathbf{P}| \cdot |\mathbf{X}|^2)$ time, as done by Algorithm 5.2. Extended free-choice nets satisfy the property ${}^\circ T_1 = {}^\circ T_2$,

input: A Petri net $N = (\mathbf{P}, \mathbf{T}, \mathbf{I}, \mathbf{O}, M_0)$
output: The structural concurrency relation $\|^{\mathcal{S}}$

1: $\mathbf{R} := \{(P, P') \mid M_0 \geq M_P + M_{P'}\} \cup (\bigcup_{T \in \mathbf{T}} (T^\circ \times T^\circ) \setminus i_{\mathbf{P}})$
2: $\mathbf{E} := \mathbf{R}$
3: **while** $\mathbf{E} \neq \emptyset$ **do**
4: select $(X, P) \in \mathbf{E}$
5: $\mathbf{E} := \mathbf{E} \setminus \{(X, P)\}$
6: **for** each $T \in P^\circ$ **do**
7: **if** $\{X\} \times {}^\circ T \subseteq \mathbf{R}$ **then**
8: $\mathbf{E} := \mathbf{E} \cup ((((\{X\} \times T^\circ) \cup (T^\circ \times \{X\}) \cup \{(T, X)\}) \cap ((\mathbf{P} \cup \mathbf{T}) \times \mathbf{P})) \setminus \mathbf{R})$
9: $\mathbf{R} := \mathbf{R} \cup (\{X\} \times T^\circ) \cup (T^\circ \times \{X\}) \cup \{(X, T)\} \cup \{(T, X)\}$
10: **end if**
11: **end for**
12: **end while**
13: $\|^{\mathcal{S}} := \mathbf{R}$

Algorithm 5.1: StructConcRel(N)



(a) A Petri net (b) $\|^{\mathcal{S}}$ given by Algorithm 5.1

Figure 5.10: Illustration of the concept of concurrency relation

for every two $T_1, T_2 \in P^\circ$. Therefore, in the case of extended free-choice nets, there is no need to check if $\{X\} \times {}^\circ T \subseteq \mathbf{R}$ for each $T \in P^\circ$ (lines 6 and 7 of Algorithm 5.1). It suffices to check just one $T \in P^\circ$ (lines 7 and 8 of Algorithm 5.2). Algorithm 5.2 has also been obtained from the theory introduced in [KE96].

In our approach, we are interested in the concurrency relation *among transitions*. Using $\|^{\mathcal{S}}$ (defined on $\mathbf{P} \cup \mathbf{T}$) makes however the whole process simpler even if, later on, we do not make use of the elements (P, T), (T, P), and (P, P')—with $P, P' \in \mathbf{P}$ and $T \in \mathbf{T}$—that belong to $\|^{\mathcal{S}}$. Once $\|^{\mathcal{S}}$ has been computed we simply obtain the structural concurrency relation on $\mathbf{T}$ as $\|^{\mathcal{S}}_{\mathbf{T}} = \{(T, T') \in \|^{\mathcal{S}} \mid T, T' \in \mathbf{T}\}$. In a similar way, $\|_{\mathbf{T}} = \{(T, T') \in \| \mid T, T' \in \mathbf{T}\}$. In the sequel we work based on the relation on $\mathbf{T}$ and whenever

input: A live and extended free-choice Petri net $N = (\mathbf{P}, \mathbf{T}, \mathbf{I}, \mathbf{O}, M_0)$
output: The structural concurrency relation $\|^{\mathcal{S}}$

1: $\mathbf{R} := \{(P, P') \mid M_0 \geq M_P + M_{P'}\} \cup (\bigcup_{T \in \mathbf{T}} (T^\circ \times T^\circ) \setminus i_{\mathbf{P}})$
2: $\mathbf{A} := \{(P, P') \mid P' \in (P^\circ)^\circ\}$
3: $\mathbf{E} := \mathbf{R}$
4: **while** $\mathbf{E} \neq \emptyset$ **do**
5: select $(X, P) \in \mathbf{E}$
6: $\mathbf{E} := \mathbf{E} \setminus \{(X, P)\}$
7: select $T \in P^\circ$
8: **if** $\{X\} \times {}^\circ T \subseteq \mathbf{R}$ **then**
9: $\mathbf{E} := \mathbf{E} \cup ((((\{(X, P') \mid (P, P') \in \mathbf{A}\} \cup \{(P', X) \mid (P, P') \in \mathbf{A}\} \cup \{(T, X)\}) \cap ((\mathbf{P} \cup \mathbf{T}) \times \mathbf{P})) \setminus \mathbf{R})$
10: $\mathbf{R} := \mathbf{R} \cup \{(X, P') \mid (P, P') \in \mathbf{A}\} \cup \{(P', X) \mid (P, P') \in \mathbf{A}\} \cup \{(X, T)\} \cup \{(T, X)\}$
11: **end if**
12: **end while**
13: $\|^{\mathcal{S}} := \mathbf{R}$

Algorithm 5.2: StructConcRel(N)

we refer to concurrency relation we will mean concurrency relation on $\mathbf{T}$.

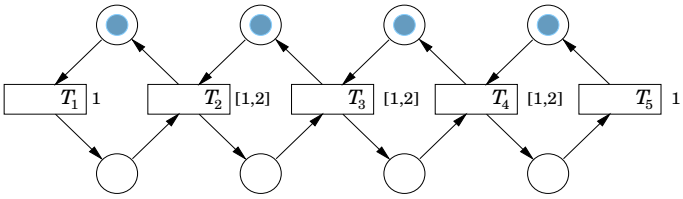
As illustrated by the experimental results of Subsection 5.3.1, the cost of computing the concurrency relation is significantly lower than the cost of the model checking itself.

5.2.2 Grouping Transitions

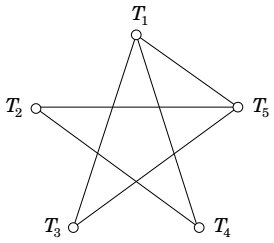
The concurrency relation can be represented as an undirected graph $G = (\mathbf{T}, \mathbf{E})$ where its vertices are the transitions $T \in \mathbf{T}$ and an edge joining two vertices indicates that the corresponding transitions can fire concurrently. For instance, for the PRES+ model of a buffer of capacity 4 [Esp94] shown in Figure 5.11(a), the concurrency relation represented as a graph is depicted in Figure 5.11(b).

With the naive translation procedure (Subsection 4.2.2), we obtain one automaton with one clock for each transition. However, we can do better by exploiting the information given by the concurrency relation. Considering the model and the concurrency relation shown in Figure 5.11, for instance, we may group T_2 and T_3 together since we know that they cannot fire concurrently. This means that the two timed automata $\vec{\mathcal{T}}_2$ and $\vec{\mathcal{T}}_3$ corresponding to these transitions may share the same clock variable. Furthermore, it is possible to construct a single automaton (with one clock) equivalent to the behavior of both transitions.

We aim at obtaining as few groups of transitions as possible so that the



(a) Petri net model



(b) Concurrency relation

Figure 5.11: Buffer of capacity 4

automata equivalent to the PRES+ model have the minimum number of clocks. This problem can be defined as follows:

> Problem 5.1 (Minimum Graph Coloring—MGC) Given the concurrency relation as a graph $G = (\mathbf{T}, \mathbf{E})$, find a *coloring* of $\mathbf{T}$, that is a partitioning of $\mathbf{T}$ into disjoint sets $\mathbf{T}_1, \mathbf{T}_2, \ldots, \mathbf{T}_k$, such that each $\mathbf{T}_i$ is an independent set[1] for G and the size k of the coloring is minimum.

For the example shown in Figure 5.11, the minimum number of colors is 3 and one such optimal coloring is $\mathbf{T}_1 = \{T_1, T_2\}$, $\mathbf{T}_2 = \{T_3, T_4\}$, $\mathbf{T}_3 = \{T_5\}$. This means we can get timed automata with 3 clocks (instead of 5 when using the naive translation) corresponding to the model given in Figure 5.11(a).

Problem 5.1 (MGC) is known to be an NP-hard problem [GJ79]. Though there is no known polynomial-time algorithm that solves MGC, the problem is very well-known and many approximation algorithms have been proposed as well as different heuristics that find reasonably good near-optimal solutions. Note that even a near-optimal solution to MGC implies an improvement in our verification approach because the number of clocks in the resulting timed automata is reduced. There are also algorithms that find the optimal coloring in reasonable time for some instances of the problem. For the systems we address in Subsection 5.2.5 and Section 5.3 we are able

[1] An independent set is a subset $\mathbf{T}_i \subseteq \mathbf{T}$ such that no two vertices in $\mathbf{T}_i$ are joined by an edge in $\mathbf{E}$.

to find the optimal solution in a short time by using an algorithm based on Brélaz's DSATUR [Bré79].

5.2.3 Composing Automata

After the concurrency relation has been colored, we can reduce the number of resulting automata by composing those that correspond to transitions with the same color. Thus we obtain one automaton with one clock for each color.

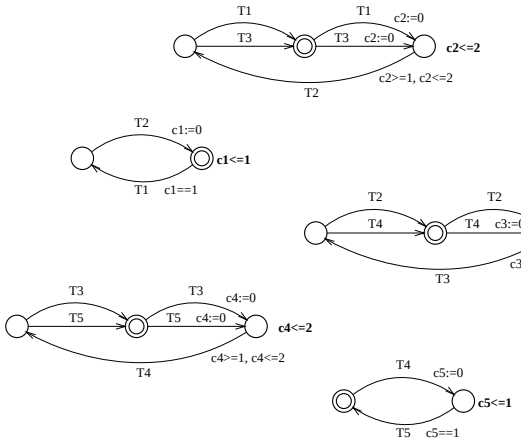
Automata are composed by applying the standard product construction method [HMU01]. In the general case, the product construction suffers from the state-explosion problem, that is the number of locations of the product automaton is an exponential function of the number of components. However, in our approach we do not incur a explosion in the number of states because the automata are tightly linked through synchronization labels and, most importantly, the composing automata are not concurrent. Recall that we do not construct the product automaton of the whole system. We construct one automaton for each color, so that the composing automata (corresponding to that color) cannot occur in parallel.

Figure 5.12(b) depicts the resulting time automata corresponding to the net shown in Figure 5.11(a) when following the translation procedure proposed in this section. Observe and compare with the automata shown in Figure 5.12(a) obtained by applying the naive translation described in Subsection 4.2.2.

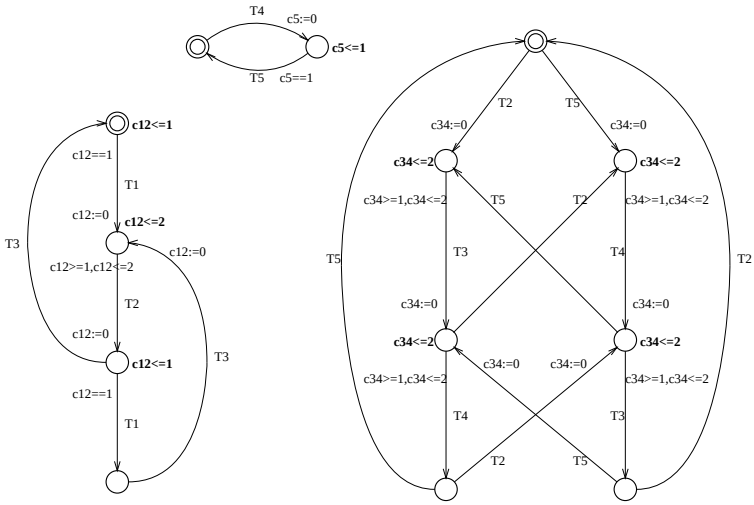
In the example given in Figure 5.11(a), which we used in order to illustrate our coloring technique for improving verification efficiency, for the sake of clarity we have abstracted away transition functions and token values as these do not influence the method described above.

5.2.4 Remarks

For the verification of PRES+ models we initially get a collection of timed automata as given by the naive translation, with one automaton and one clock for each transition in the PRES+ net. One such automaton uses a clock in order to constrain the firing of the corresponding transition in the interval given by its minimum and maximum transition delays. In this way the timing semantics is preserved in the equivalent timed automata. Regarding transition functions and guards in the PRES+ model, these are straightforwardly mapped in the timed automata model as *activities* and *variable conditions* respectively. Thus the naive translation is correct in the sense that the resulting timed automata have a behavior equivalent to the original PRES+ model.



(a) Naive translation



(b) Coloring translation

Figure 5.12: Timed automata equivalent to the Petri net of Figure 5.11(a)

By using the same clock in order to account for the firing time semantics of two (or more) transitions, we do not change the system behavior. We group transitions that cannot fire concurrently in the underlying (regular) Petri net and therefore cannot fire concurrently in the PRES+ model. Consequently, transitions with the same color may share the same clock (so that the system behavior is preserved) because they are pair-wise non-concurrent.

Finally, since automata composition does not change the system be-

havior, the resulting timed automata are indeed equivalent to the original PRES+ model.

5.2.5 Revisiting the GMDFα

In Subsection 3.6.1 we have modeled a GMDFα (Generalized Multi-Delay frequency-domain Filter). In Subsection 5.1.2 such an application has been verified by transforming the system model and using the naive translation procedure described in Subsection 4.2.2.

In this section we revisit the verification of the GMDFα and compare it with the results shown previously in Subsection 5.1.2. We also consider here the two cases of a GMDFα of length 1024: a) with an overlapping factor $\alpha = 4$, $K = 4$; b) with an overlapping factor $\alpha = 2$, $K = 8$. Recall that for a sampling rate of 8 kHz, the maximum execution time for one iteration is 8 ms in both cases. What we want to prove is that the filter eventually completes its functionality and does so within a bound on time (8 ms). This is captured by the TCTL formula $\mathbf{AF}_{<8}\, E'$. As seen in Figure 3.12, K affects directly the dimension of the model and, therefore, the complexity of verification.

The verification times for the GMDFα are shown in Table 5.2. The second column corresponds to the verification using the approach described in Section 4.2 (naive translation of PRES+ into timed automata). The third column shows the results of verification when using the approach discussed in Section 5.1 (transformation of the model into a semantically equivalent and simpler one, followed by the naive translation into timed automata). The verification time for the GMDFα using the coloring method presented in this section is shown in the fourth column of Table 5.2. These results include the time spent in computing the concurrency relation, coloring this relation, and constructing the product automata, as well as model-checking the resulting timed automata. By combining the transformational approach with the coloring one, it is possible to further improve the verification efficiency as shown in the fifth column of Table 5.2.

GMDFα $L = 1024$	Verification time [s]			
	Naive	Transf.	Coloring	Transf. *and* Coloring
$\alpha = 4$, $K = 4$	108.9	0.9	2.2	0.1
$\alpha = 2$, $K = 8$	NA*	9.7	520.8	1.7

* Not available: out of time

Table 5.2: Different verification times for the GMDFα

5.3 Experimental Results

This section presents a scalable example and an industrial design that illustrate our verification approach as well as the proposed improvement techniques.

5.3.1 Ring-Configuration System

This example represents a number n of processing subsystems arranged in a ring configuration. The model for one such subsystem is illustrated in Figure 5.13.

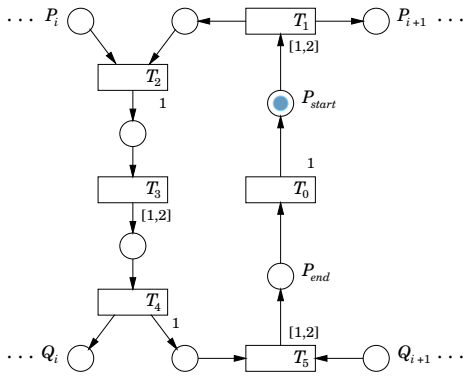


Figure 5.13: Model for one ring-configuration subsystem

Each one of the n subsystems has a bounded response requirement, namely whenever the subsystem gets started it must strictly finish within a time limit, in this case 25 time units. Referring to Figure 5.13, the start of processing for one such subsystem is denoted by the marking of P_{start} while the marking of P_{end} denotes its end. This requirement is expressed by the TCTL formula $\mathbf{AG}(P_{start} \Rightarrow \mathbf{AF}_{<25}\, P_{end})$.

We have used the tool UPPAAL in order to model-check the timing requirements of the ring-configuration system. The results are summarized in Table 5.3. The second column gives the verification time using the naive translation of PRES+ into timed automata. The third column shows the verification time when using the transformational approach (Section 5.1). The fourth, fifth, sixth, and seventh columns correspond, respectively, to the time spent in computing the concurrency relation, finding the optimal coloring of the concurrency relation, constructing the product automata, and model-checking the resulting timed automata. The total verification time, when applying the approach proposed in Section 5.2, is given in the eighth column of Table 5.3. By combining the transformation-based technique and

the coloring one, it is possible to further improve verification efficiency as shown in the last column of Table 5.3: we first apply correctness-preserving transformations in order to simplify the PRES+ model and then translate it into timed automata by using the coloring method. These results have been plotted in Figure 5.14. As can be seen in Figure 5.14, the combination of transformations and coloring outperforms the naive approach by up to two orders of magnitude. Combining such strategies makes it possible to handle ring-configuration systems composed of up to 9 subsystems (whereas with the naive approach we can only verify up to 6 subsystems).

n	Verification time [s]							
	Naive	Trans-formations	Coloring					Transf. *and* Coloring
			Comp. Conc. Relation	Coloring Conc. Relation	Product Autom-ata	Model Check-ing	Total Verif-cation	
2	0.2	0.1	0.002	0.002	0.114	0.1	0.2	0.2
3	2.4	0.6	0.006	0.004	0.153	0.2	0.4	0.2
4	47.3	8.2	0.015	0.014	0.199	1.2	1.4	0.7
5	787.9	114.1	0.026	0.076	0.249	18.2	18.6	5.9
6	13481.2	1200.6	0.046	0.342	0.297	217.8	218.5	55.5
7†	NA*	18702.5	0.076	0.449	0.349	2402.7	2403.6	465.1
8†	NA*	NA*	0.156	0.545	0.405	24705.4	24706.5	3721.7
9†	NA*	NA*	0.259	0.698	0.512	NA*	NA*	28192.7

† Specification does not hold
* Not available: out of time

Table 5.3: Verification of the ring-configuration example

It is interesting to observe that when $n \geq 7$ the bounded response requirement expressed by the TCTL formula $\mathbf{AG}(P_{start} \Rightarrow \mathbf{AF}_{<25}\, P_{end})$ is not satisfied, a fact not obvious at all. An informal explanation is that since transition delays are given in terms of intervals, one subsystem may take longer to execute than another; thus different subsystems can execute "out of phase" and this phase difference may be accumulated depending on the number of subsystems, causing one such subsystem to take eventually longer than 25 time units (for $n \geq 7$). It is also worth mentioning that, although the model has relatively few transitions and places, this example is rather complex because of its large state space which is due to the high degree of parallelism.

5.3.2 Radar Jammer

This subsection addresses the verification of the radar jammer discussed in Subsection 3.6.2. We aim at verifying a pipelined version of the jammer

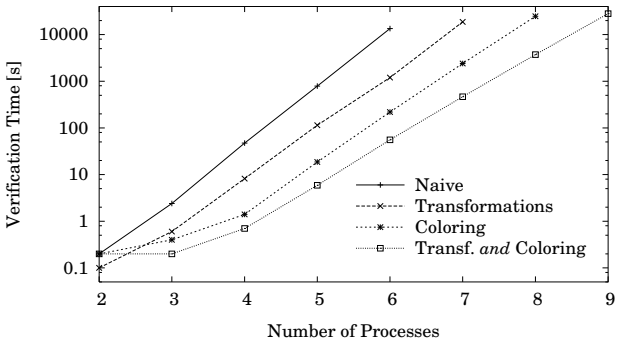


Figure 5.14: Verification of ring-configured subsystems

where the stages correspond precisely to the super-transitions of the model shown in Figure 3.15. In order to represent a pipelined structure, it is necessary to add a number of places and arcs as follows. For every place $P \in \mathbf{P}$ such that there exists $(ST_i, P) \in \mathbf{O}$ and $(P, ST_j) \in \mathbf{I}$ (for $ST_i \neq ST_j$): a) add a place P' initially marked; b) add an input arc (P', ST_i); c) add an output arc (ST_j, P'). In this way, all places but *in* and *out* will hold at most one token, and still several of them might be marked simultaneously, representing the progress of activities along the pipeline.

The model of the pipelined jammer, annotated with timing information, is shown in Figure 5.15. The minimum and maximum transition delays are given in ns. We have included in this model a few more places and transitions that represent the environment, namely transitions *sample* and *emit*, and places *inSig* and *outSig*. The input to the jammer is a radar pulse (actually, a number of samples taken from it). Transition *sample* will fire n times (where n is the number of samples), every PW/n (where PW is the pulse width), depositing the samples in the place *inSig* which are later buffered in the place *in*. In this form, we model the input of the incoming radar pulse. A token in *inSig* means that the input is being sampled. Regarding the emission of the pulse produced by the jammer, the data obtained is buffered in place *out* before being transmitted. After some delay, it is sent out by transition *emit* so that the marking of place *outSig* represents a part of the outgoing pulse being transmitted back to the radar.

We have applied our verification technique to the PRES+ model of the jammer shown in Figure 5.15. There are two properties that are important for the jammer. The first is that there cannot be output while sampling the input. The second requirement is that the whole outgoing pulse must be transmitted before another pulse arrives. These are due to the fact that there is only one physical device for reception and transmission of signals.

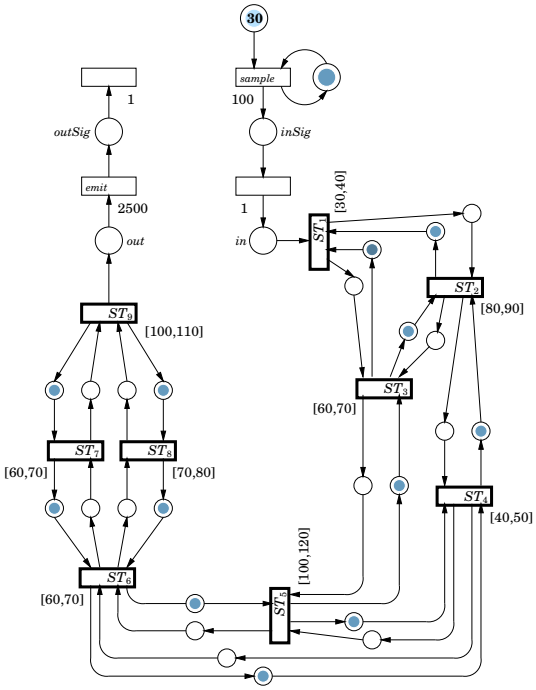


Figure 5.15: Pipelined model of the jammer

The minimum Pulse Repetition Interval (PRI), i.e. the separation in time of two consecutive incoming pulses, for our system is 10 μs, so this is the value we will use for verifying the second property. For a PRI of 10 μs, the Pulse Width PW can vary from 100 ns up to 3 μs. Therefore, we will consider the most critical case, that is, when the pulse width is 3 μs. We assume that the number of samples is $n = 30$ (so that the delay of transition *sample* is 100 ns).

The properties described above can be expressed, respectively, by the formulas $\mathbf{AG}\,\neg(inSig \wedge outSig)$ and $\neg\mathbf{EF}_{>10000}\, outSig$, where *inSig* and *outSig* are places in the Petri net representing the input and output of the jammer respectively. The first formula states that the places *inSig* and *outSig* are never marked at the same time, while the second says that there is no computation path for which *outSig* is marked after 10000 ns. We have verified that both formulas indeed hold in the model of the system. The verification times are given in Table 5.4. Verifying these two formulas takes roughly 20 s when combining the transformational approach and the coloring translation, whereas the naive verification takes almost 10 minutes.

The radar jammer is a realistic example that illustrates how our modeling and verification approach can be used for practical applications. The

Property	Verification time [s]			
	Naive	Transfor-mations	Coloring	Transf. *and* Coloring
$\mathbf{AG}\,\neg(inSig \wedge outSig)$	262.8	68.7	12.4	7.5
$\neg\mathbf{EF}_{>10000}\,outSig$	338.3	89.9	23.8	13.6

Table 5.4: Verification of the radar jammer

verified requirements are very interesting as not only do they impose an upper bound for the completion of the activities but also a lower one, since the emission and sampling of pulses cannot overlap. Although there are few transitions in the model, the state space is very large (5.24×10^7 states in the untimed model) because of the pipeline. Despite such a large state space, the verification of the two studied properties takes relatively short time when applying the techniques addressed in this work.

Part III

Scheduling Techniques

Chapter 6

Introduction and Related Approaches

Part III of this dissertation deals with scheduling techniques for mixed hard/soft real-time systems. The *hard* component comes from the fact that there exist hard deadlines that have to be met in all working scenarios to avoid disastrous consequences. The *soft* component, for the type of systems considered in this Part III, is due to fact that either certain tasks have loose deadlines that may be missed or tasks include optional parts that may be left incomplete, in both cases at the expense of the quality of results. Thus, the soft component provides the flexibility for trading off quality of results with different design metrics.

Real-time systems that include both tasks with hard deadlines and tasks with soft deadlines are studied in Chapter 7. In these cases the quality of results, expressed in terms of *utilities*, is dependent on the completion time of soft tasks.

Real-time systems in which tasks are composed of mandatory and optional parts and for which optional parts can be left incomplete, still subject to hard real-time constraints, are addressed in Chapter 8. In these cases the quality of results, in the form of *rewards*, depends on the amount of computation alloted to tasks.

Both in Chapter 7 and in Chapter 8 we introduce quasi-static scheduling techniques that are able to exploit, with low overhead, the dynamic slack due to variations in actual execution times.

The rest of this chapter is devoted to present approaches related to the scheduling of hard/soft real-time systems as well as quasi-static techniques introduced in different contexts.

6.1 Systems with Hard and Soft Tasks

Scheduling for hard/soft real-time systems has been addressed, for example, in the context of integrating multimedia and hard real-time tasks [KSSR96], [AB98]. The rationale is the need for supporting multimedia soft real-time tasks coexisting together with hard real-time tasks, in a way such hard deadlines are guaranteed while, at the same time, the capability of graceful degradation of the quality of service during system overload is provided.

Most of the scheduling approaches for mixed hard/soft real-time systems consider that hard tasks are periodic while soft tasks are aperiodic. In such a framework, both dynamic and fixed priority systems have been considered. In the former case, the Earliest Deadline First (EDF) algorithm is used for scheduling hard tasks and the response time of soft aperiodic tasks is minimized while guaranteeing hard deadlines [BS99], [RCGF97], [HR94]. Similarly, the joint scheduling approaches for fixed priority systems try to serve soft tasks the soonest possible while guaranteeing hard deadlines, but make use of the Rate Monotonic (RM) algorithm for scheduling hard periodic tasks [DTB93], [LRT92], [SLS95].

It has usually been assumed that it is best to serve a soft task as soon as possible, without making distinction among soft tasks. In many cases, however, distinguishing among soft tasks allows a more efficient allocation of resources. In our approach presented in Chapter 7 we employ utility functions that provide such a distinction. Value or utility functions were first suggested by Locke [Loc86] for representing significance and criticality of tasks. Such functions specify the utility delivered to the system resulting from the termination of a task as a function of its completion time [WRJB04].

Utility-based scheduling [BPB+00], [PBA03] has been addressed before in several contexts, for instance, the QoS-based resource allocation model [RLLS97] and Time-Value-Function scheduling [CM96]. The former is a model that allows the utility to be maximized by allocating resources such that the different needs of concurrent applications are satisfied. The latter considers independent tasks running on a single processor, assumes fixed execution times, and proposes an $O(n^3)$ on-line heuristic for maximizing the total utility; however, such an overhead might be too large for realistic systems.

Earlier work generally uses only the Worst-Case Execution Time (WCET) for scheduling, which leads to an excessive degree of pessimism (Abeni and Buttazzo [AB98] do use mean values for serving soft tasks and WCET for guaranteeing hard deadlines though). In the approach proposed in Chapter 7 we take into consideration the fact that the actual execution time of a task is rarely its WCET. We use instead the expected or mean du-

ration of tasks when evaluating the utility functions associated to soft tasks. Nevertheless, we do consider the worst-case duration of tasks for ensuring that hard time constraints are always met. Moreover, since we consider time intervals rather than fixed execution times for tasks, we are able to exploit the slack due to tasks finishing ahead of their worst-case completion times.

6.2 Imprecise-Computation Systems

Another type of "utility"-based scheduling applies to systems in which tasks produce reward as a function of the amount of computation executed by them. The term reward-based scheduling is thus used in these cases in order to emphasize this aspect (and differentiate from utility-based scheduling where the utility produced is a function of tasks' completion times). Reward-based scheduling has been addressed in the frame of Imprecise Computation (IC) techniques [SLC89], [LSL+94]. These assume that tasks are composed of mandatory and optional parts: both parts must be finished by the deadline but the optional part can be left incomplete at the expense of the quality of results. There is a function associated with each task that assigns a reward as function of the amount of computation allotted to the optional part: the more the optional part executes, the more reward it produces.

In Chapter 8 we study, under the imprecise computation model, real-time systems with reward and energy considerations. Dynamic Voltage Scaling (DVS) techniques permit the trade-off between energy consumption and performance: by lowering the supply voltage quadratic savings in dynamic energy consumption can be achieved, while the performance is degraded in approximately linear fashion. One of the earliest papers in the area is by Yao *et al.* [YDS95], where the case of a single processor with continuous voltage scaling is addressed. The discrete voltage selection for minimizing energy consumption in monoprocessor systems was formulated as an Integer Linear Programming (ILP) problem by Okuma *et al.* [OYI01]. DVS techniques have been applied to distributed systems [LJ03], [KP97], [GK01], and even considering overheads caused by voltage transitions [ZHC03] and leakage energy [ASE+04]. DVS has also been considered under fixed [SC99] and dynamic priority assignments [KL03].

While DVS techniques have mostly been studied in the context of hard real-time systems, IC approaches have until now disregarded the power/energy aspects. Rusu *et al.* [RMM03] proposed recently the first approach in which energy, reward, and deadlines are considered under a unified framework. The goal of the approach is to maximize the reward without exceeding deadlines or the available energy. This approach, however, solves statically at compile-time the optimization problem and therefore consid-

ers only worst cases, which leads to overly pessimistic results. A similar approach (with similar drawbacks) for maximizing the total reward subject to energy constraints, but considering that tasks have fixed priority, was presented in [YK04].

6.3 Quasi-Static Approaches

Tasks in real-time systems may finish ahead of their deadlines. The time difference between deadline and actual completion time is known as slack. Depending on what causes the slack, it can be classified as either static or dynamic. Static slack is due to the fact that at nominal voltage the processor runs faster than needed. Dynamic slack is caused by tasks executing less number of clock cycles than their worst case.

Most of the techniques proposed in the frame of DVS, for instance, are static approaches in the sense that they can only exploit the static slack [YDS95], [OYI01], [RMM03]. Nonetheless, there has been a recent interest in dynamic approaches, that is, techniques aimed at exploiting the dynamic slack [GK01], [SLK01], [AMMMA01]. Solutions by static approaches are computed off-line, but have to make pessimistic assumptions, typically in the form of WCETs. Dynamic approaches recompute solutions at run-time in order to exploit the slack that arises from variations in the execution time, but these on-line computations are typically non-trivial in most interesting cases and consequently their overhead is high. We propose, for the particular problems addressed in Chapters 7 and 8, solutions that belong to the class of the so-called *quasi-static* approaches. Quasi-static approaches attempt to perform the majority of computations at design-time, so that only simple activities are left for run-time.

Quasi-static scheduling has been studied previously, but mostly in the context of formal synthesis and without considering an explicit notion of time, only the partial order of events [SLWSV99], [SH02], [CKLW03]. Recently, in the context of real-time systems, Shih *et al.* have proposed a template-based approach that combines off-line and on-line scheduling for phase array radar systems [SGG+03], where templates for schedules are computed off-line considering performance constraints, and tasks are scheduled on-line such that they fit in the templates. The on-line overhead, though, can be significant when the system workload is high.

The problem of quasi-static voltage scaling for energy minimization in hard real-time systems was recently addressed [ASE+05]. This approach prepares at design-time and stores a number of voltage settings, which are used at run-time for adapting the processor voltage based on actual execution times. We make use of these results in Section 8.3. Another, somehow

similar, approach in which a set of voltage settings is pre-calculated was discussed in [YC03]. It considers that the application is given as a task graph composed of subgraphs, some of which might not execute for a certain activation of the system. The selection of a particular voltage setting is thus done on-line based on which subgraphs will be executed at that activation. For each subgraph, however, WCETs are assumed and consequently no dynamic slack is exploited.

To the best of our knowledge, the quasi-static approaches introduced in Chapters 7 and 8 are the first of their type, that is, they are the first ones that address mixed hard/soft real-time systems in a quasi-static framework. The chief merit of these approaches is their ability to exploit the dynamic slack, caused by tasks completing earlier than in the worst case, at a very low on-line overhead. This is possible because a set of solutions are prepared and stored at design-time, leaving only for run-time the selection of one of them.

Chapter 7

Systems with Hard and Soft Real-Time Tasks

Many real-time systems are composed of tasks which are characterized by distinct types of timing constraints. Some of these real-time tasks correspond to activities that must be completed before a given deadline. These tasks are referred to as *hard* because missing one such deadline might have severe consequences. Such systems can also include tasks that have looser timing constraints and hence are referred to as *soft*. Soft deadline misses can be tolerated at the expense of the quality of results.

As compared to pure hard real-time techniques, scheduling for hard/soft systems permits dealing with a broader range of applications. As pointed out in Section 6.1, most of the previous work on scheduling for hard/soft real-time systems considers that hard tasks are periodic whereas soft tasks are aperiodic. It has usually been assumed that the sooner a soft task is served the better, but no distinction is made among soft tasks, and in this case the problem is to find a schedule such that all hard tasks meet their deadlines and the response time of soft tasks is minimized. However, by differentiating among soft tasks, processing resources can be allocated more efficiently. This is the case, for instance, in videoconference applications where audio streams are deemed more important than the video ones. We make use of utility functions in order to capture the relative importance of soft tasks and how the quality of results is influenced upon missing a soft deadline.

In this chapter we consider systems where both hard and soft tasks are periodic and there might exist data dependencies among tasks. We aim at finding an execution sequence (actually a set of execution sequences as explained later) such that the sum of individual utilities by soft tasks is maximal and, at the same time, satisfaction of all hard deadlines is guaranteed.

Since the actual execution times do not usually coincide with parameters like expected durations or worst-case execution times, it is possible to exploit such information in order to obtain schedules that yield higher utilities, that is, improve the quality of results.

In the frame of the problem discussed in this chapter, *static* or *off-line* scheduling refers to obtaining at design-time one single task execution order that makes the total utility maximal and guarantees the hard constraints. *Dynamic* or *on-line* scheduling refers to finding at run-time, every time a task completes, a new task execution order such that the total utility is maximized, yet guaranteeing that hard deadlines are met, but considering the actual execution times of those tasks which have already completed. On the one hand, static scheduling causes no overhead at run-time but, by producing one static schedule, it can be too pessimistic since the actual execution times might be far off from the time values used to compute the schedule. On the other hand, dynamic scheduling exploits the information about actual execution times and computes at run-time new schedules that improve the quality of results. But, due to the high complexity of the problem, the time and energy overhead needed for computing on-line the schedules can be unacceptable. In order to exploit the benefits of static and dynamic scheduling, and at the same time overcome their drawbacks, we propose in this chapter an approach in which the scheduling problem is solved in two steps: first, we compute a number of schedules at design-time; second, we leave for run-time only the decision regarding which of the precomputed schedules to follow. We call such a solution *quasi-static scheduling*.

7.1 Preliminaries

We consider that the functionality of the system is represented by a directed acyclic graph $G = (\mathbf{T}, \mathbf{E})$ where the nodes $\mathbf{T}$ correspond to tasks and data dependencies are captured by the graph edges $\mathbf{E}$.

The mapping of tasks is defined by a function $m : \mathbf{T} \rightarrow \mathbf{PE}$ where $\mathbf{PE}$ is the set of processing elements. Thus $m(T)$ denotes the processing element on which task T executes. Inter-processor communication is captured by considering the buses as processing elements and the communication activities as tasks. If $T \in \mathbf{C}$ then $m(T) \in \mathbf{B}$, where $\mathbf{C} \subset \mathbf{T}$ is the set of communication tasks and $\mathbf{B} \subset \mathbf{PE}$ is the set of buses. We consider that the mapping of tasks to processing elements (processors and buses) is fixed and already given as input to the problem.

The tasks that make up a system can be classified as non-real-time, hard, or soft. $\mathbf{H}$ and $\mathbf{S}$ denote, respectively, the subsets of hard and soft tasks. Non-real-time tasks are neither hard nor soft, and have no timing

constraints, though they may influence other hard or soft tasks through precedence constraints as defined by the task graph $G = (\mathbf{T}, \mathbf{E})$. Both hard and soft tasks have deadlines. A hard deadline d_i is the time by which a hard task $T_i \in \mathbf{H}$ must be completed, otherwise the integrity of the system is jeopardized. A soft deadline d_i is the time by which a soft task $T_i \in \mathbf{S}$ should be completed. Lateness of soft tasks is acceptable though it decreases the quality of results.

In order to capture the relative importance among soft tasks and how the quality of results is affected when missing a soft deadline, we use a non-increasing utility function $u_i(t_i)$ for each soft task $T_i \in \mathbf{S}$, where t_i is the completion time of T_i. Typical utility functions are depicted in Figure 7.1. We consider that the delivered utility by a soft task decreases after its deadline (for example, in an engine controller, lateness of the task that computes the best fuel injection rate, and accordingly adjusts the throttle, implies a reduced fuel consumption efficiency), hence the use of non-increasing functions. The total utility, denoted U, is given by the expression $U = \sum_{T_i \in \mathbf{S}} u_i(t_i)$.

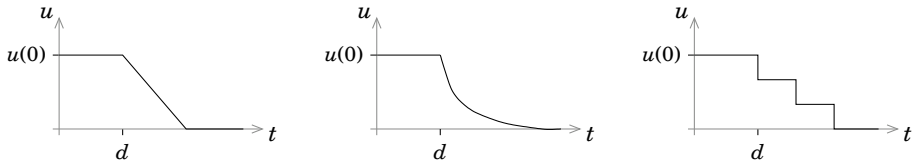


Figure 7.1: Typical utility functions for soft tasks

The actual execution time of a task T_i at a certain activation of the system, denoted τ_i, lies in the interval bounded by the best-case duration τ_i^{bc} and the worst-case duration τ_i^{wc} of the task, in other words, $\tau_i^{\mathrm{bc}} \leq \tau_i \leq \tau_i^{\mathrm{wc}}$ (dense-time semantics, that is, time is treated as a continuous quantity). The expected duration τ_i^{e} of a task T_i is the mean value of the possible execution times of the task. In the simple case of execution times distributed uniformly over the interval $[\tau_i^{\mathrm{bc}}, \tau_i^{\mathrm{wc}}]$, the expected duration is $\tau_i^{\mathrm{e}} = (\tau_i^{\mathrm{bc}} + \tau_i^{\mathrm{wc}})/2$. For an arbitrary continuous execution time probability distribution $f(\nu)$, the expected duration is given by $\tau_i^{\mathrm{e}} = \int_{\tau_i^{\mathrm{bc}}}^{\tau_i^{\mathrm{wc}}} \nu f(\nu) d\nu$.

We use ${}^\circ T$ to denote the set of direct predecessors of task T, that is, ${}^\circ T = \{T' \in \mathbf{T} \mid (T', T) \in \mathbf{E}\}$. Similarly, $T^\circ = \{T' \in \mathbf{T} \mid (T, T') \in \mathbf{E}\}$ denotes the set of direct successors of task T.

We consider that tasks are periodic and non-preemptable. We assume a single-rate semantics, that is, each task is executed exactly once for every activation of the system. Thus a schedule Ω in a system with p processing elements is a set of p bijections $\{\sigma^{(1)} : \mathbf{T}^{(1)} \rightarrow \{1, 2, \ldots, |\mathbf{T}^{(1)}|\}, \sigma^{(2)} : \mathbf{T}^{(2)} \rightarrow \{1, 2, \ldots, |\mathbf{T}^{(2)}|\}, \ldots, \sigma^{(p)} : \mathbf{T}^{(p)} \rightarrow \{1, 2, \ldots, |\mathbf{T}^{(p)}|\}\}$ where $\mathbf{T}^{(i)} = \{T \in$

$\mathbf{T} \mid m(T) = PE_i\}$ is the set of tasks mapped onto the processing element PE_i and $|\mathbf{T}^{(i)}|$ denotes the cardinality of the set $\mathbf{T}^{(i)}$. In the particular case of monoprocessor systems, a schedule is just one bijection $\sigma : \mathbf{T} \rightarrow \{1, 2, \ldots, |\mathbf{T}|\}$. We use the notation $\sigma^{(i)} = T_1 T_2 \ldots T_n$ as shorthand for $\sigma^{(i)}(T_1) = 1, \sigma^{(i)}(T_2) = 2, \ldots, \sigma^{(i)}(T_n) = |\mathbf{T}^{(i)}|$.

In our task model, we assume that the task graph is activated periodically (all the tasks in a task graph have the same period and become ready at the same time) and that, in addition to the deadlines on individual tasks, there exists an implicit hard deadline equal to the period. The latter is easily modeled by adding a hard task, that is successor of all other tasks, which consumes no time and no resources, and which has a deadline equal to the period.

However, if a system contains task graphs with different periods we can still handle it by generating several instances of the task graphs and building a graph that corresponds to a set of task graphs as they occur within their hyperperiod (least common multiple of the periods of the involved tasks), in such a way that the new task graph has one period equal to the aforementioned hyperperiod [Lap04].

For a given schedule, the starting and completion times of a task T_i are denoted s_i and t_i respectively, with $t_i = s_i + \tau_i$. Thus, for $\sigma^{(k)} = T_1 T_2 \ldots T_n$, task T_1 will start executing at $s_1 = \max_{T_j \in {}^\circ T_1}\{t_j\}$ and task T_i, $1 < i \leq n$, will start executing at $s_i = \max(\max_{T_j \in {}^\circ T_i}\{t_j\}, t_{i-1})$. In the sequel, the starting and completion times that we use are relative to the system activation instant. Thus a task T with no predecessor in the task graph has starting time $s = 0$ if $\sigma^{(k)}(T) = 1$. For example, in a monoprocessor system, according to the schedule $\sigma = T_1 T_2 \ldots T_n$, T_1 starts executing at time $s_1 = 0$ and completes at $t_1 = \tau_1$, T_2 starts at $s_2 = t_1$ and completes at $t_2 = \tau_1 + \tau_2$, and so forth.

We aim at finding off-line a set of schedules and the conditions under which the quasi-static scheduler decides on-line to switch from one schedule to another. A *switching point* defines when to switch from one schedule to another. A switching point is characterized by a task and a time interval, as well as the involved schedules. For example, the switching point $\Omega \xrightarrow{T_i;[a,b]} \Omega'$ indicates that, while Ω is the current schedule, when the task T_i finishes and its completion time is $a \leq t_i \leq b$, another schedule Ω' must be followed as execution order for the remaining tasks.

We assume that the system has a dedicated shared memory for storing the set of schedules, which all processing elements can access. There is an exclusion mechanism that grants access to one processing element at a time. The worst-case blocking time on this memory is considered in our analysis as included in the worst-case duration of tasks. Upon finishing a task running

on a certain processing element, a new schedule can be selected (according to the set of schedules and switching points prepared off-line) which will then be followed by all processing elements. Our analysis takes care that the execution sequence of tasks already executed or still under execution is consistent with the new schedule.

7.2 Static Scheduling

In order to address the quasi-static approach for scheduling systems with hard and soft tasks that we propose in this chapter, it is needed to first take up the problem of static scheduling. We present in this section a precise formulation of the problem of static scheduling and provide solutions for it, both for monoprocessor systems and for the general case of multiple processors.

We want to find the schedule that, among all schedules that respect the hard constraints in the *worst case*, maximizes the total utility when tasks last their *expected* duration. Note that an alternative formulation could have been to find the schedule that, among all schedules that respect the hard constraints in the *worst case*, maximizes the total utility when tasks last their *worst-case* duration. For both formulations it is guaranteed that hard deadlines are satisfied. However, in the the former case the schedule is constructed such that the resulting total utility is maximal if tasks execute their expected case, while in the latter case a maximal utility is produced if tasks execute their worst case. Since, by definition, task execution times are such that they are "centered" around the expected value (the mean or expected value is the "center" of the probability distribution, in the same sense as the center of gravity is the mean of the mass distribution as defined in physics [Bai71]), we can obtain better results in the former case.

The problem of static scheduling for real-time systems with hard and soft tasks, in the context of maximizing the total utility, is formulated as follows:

PROBLEM 7.1 (Scheduling to Maximize Utility—SMU) Find a multiprocessor schedule Ω (set of p bijections $\{\sigma^{(1)} : \mathbf{T}^{(1)} \rightarrow \{1, 2, \ldots, |\mathbf{T}^{(1)}|\}$, $\sigma^{(2)} : \mathbf{T}^{(2)} \rightarrow \{1, 2, \ldots, |\mathbf{T}^{(2)}|\}, \ldots, \sigma^{(p)} : \mathbf{T}^{(p)} \rightarrow \{1, 2, \ldots, |\mathbf{T}^{(p)}|\}\}$ with $\mathbf{T}^{(l)}$ being the set of tasks mapped onto the processing element PE_l and p being the number of processing elements) that maximizes $U = \sum_{T_i \in \mathbf{S}} u_i(t_i^{\mathrm{e}})$ where t_i^{e} is the expected completion time of task T_i, subject to: $t_i^{\mathrm{wc}} \leq d_i$ for all $T_i \in \mathbf{H}$, where t_i^{wc} is the worst-case completion time of task T_i; no deadlock is introduced by Ω.

§1. The expected completion time of T_i is given by

$$t_i^{\mathrm{e}} = \begin{cases} \max_{T_j \in {}^\circ T_i}\{t_j^{\mathrm{e}}\} + \tau_i^{\mathrm{e}} & \text{if } \sigma^{(l)}(T_i) = 1, \\ \max(\max_{T_j \in {}^\circ T_i}\{t_j^{\mathrm{e}}\}, t_k^{\mathrm{e}}) + \tau_i^{\mathrm{e}} & \text{if } \sigma^{(l)}(T_i) = \sigma^{(l)}(T_k) + 1. \end{cases}$$

where: $m(T_i) = p_l$; $\max_{T_j \in {}^\circ T_i}\{t_j^{\mathrm{e}}\} = 0$ if ${}^\circ T_i = \emptyset$.

§2. The worst-case completion time of T_i is given by

$$t_i^{\mathrm{wc}} = \begin{cases} \max_{T_j \in {}^\circ T_i}\{t_j^{\mathrm{wc}}\} + \tau_i^{\mathrm{wc}} & \text{if } \sigma^{(l)}(T_i) = 1, \\ \max(\max_{T_j \in {}^\circ T_i}\{t_j^{\mathrm{wc}}\}, t_k^{\mathrm{wc}}) + \tau_i^{\mathrm{wc}} & \text{if } \sigma^{(l)}(T_i) = \sigma^{(l)}(T_k) + 1. \end{cases}$$

where: $m(T_i) = p_l$; $\max_{T_j \in {}^\circ T_i}\{t_j^{\mathrm{wc}}\} = 0$ if ${}^\circ T_i = \emptyset$.

§3. No deadlock introduced by Ω means that when considering a task graph with its original edges together with additional edges defined by the partial order corresponding to the schedule, the resulting task graph must be acyclic.

The items §1 and §2 in Problem 7.1 provide actually the way of obtaining the expected and worst-case completion times respectively. For the expected completion time, for instance, if T_i is the first task in the sequence $\sigma^{(l)}$ of tasks mapped onto PE_l, its completion time t_i^{e} is computed as the maximum among the expected completion times t_j^{e} of its predecessor tasks (which can be mapped onto different processing elements) plus its own expected duration τ_i^{e}; if T_i is not the first task in the sequence $\sigma^{(l)}$ of tasks mapped onto PE_l, we take the maximum among the expected completion times t_j^{e} of its predecessor tasks and then the maximum between this value and the expected completion time t_k^{e} of the previous task in the sequence $\sigma^{(l)}$, and add the expected duration τ_i^{e}.

We also present the formulation of the problem of static scheduling for real-time systems with hard and soft tasks in the special case of monoprocessor systems as we first present solutions for this particular case and then we generalize to multiple processors.

PROBLEM 7.2 (Monoprocessor Scheduling to Maximize Utility—MSMU) Find a monoprocessor schedule σ (a bijection $\sigma : \mathbf{T} \to \{1, 2, \ldots, |\mathbf{T}|\}$) that maximizes $U = \sum_{T_i \in \mathbf{S}} u_i(t_i^{\mathrm{e}})$ where t_i^{e} is the expected completion time of task T_i, subject to: $t_i^{\mathrm{wc}} \leq d_i$ for all $T_i \in \mathbf{H}$, where t_i^{wc} is the worst-case completion time of task T_i; no deadlock is introduced by σ.

§1. The expected completion time of T_i is given by

$$t_i^{\mathrm{e}} = \begin{cases} \tau_i^{\mathrm{e}} & \text{if } \sigma(T_i) = 1, \\ t_k^{\mathrm{e}} + \tau_i^{\mathrm{e}} & \text{if } \sigma(T_i) = \sigma(T_k) + 1. \end{cases}$$

§2. The worst-case completion time of T_i is given by

$$t_i^{\mathrm{wc}} = \begin{cases} \tau_i^{\mathrm{wc}} & \text{if } \sigma(T_i) = 1, \\ t_k^{\mathrm{wc}} + \tau_i^{\mathrm{wc}} & \text{if } \sigma(T_i) = \sigma(T_k) + 1. \end{cases}$$

§3. No deadlock introduced by σ means that $\sigma(T) < \sigma(T')$ for all $(T, T') \in \mathbf{E}$.

We prove in Section B.2 that Problem 7.2 is **NP**-hard. Therefore the

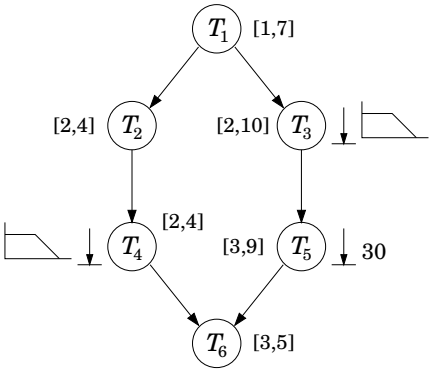
problem of static scheduling for systems with real-time hard and soft tasks discussed in this section is intractable, even in the monoprocessor case.

We discuss in Subsection 7.2.1 solutions to the problem of static scheduling for systems with hard and soft tasks, as formulated above, in the particular case of a single processor. The general case of multiprocessor systems is addressed in Subsection 7.2.2.

7.2.1 Single Processor

In this subsection we address the problem of static scheduling for monoprocessor systems composed of hard and soft tasks (Problem 7.2). We present an algorithm that solves optimally the problem as well as heuristics that find near-optimal solutions at reasonable computational cost.

We first consider as example a system that has six tasks T_1, T_2, T_3, T_4, T_5 and T_6, with data dependencies as shown in Figure 7.2. The best-case and worst-case durations of every task are shown in Figure 7.2 in the form $[\tau^{\mathrm{bc}}, \tau^{\mathrm{wc}}]$. In this example we assume that the execution time of every task T_i is uniformly distributed over its interval $[\tau_i^{\mathrm{bc}}, \tau_i^{\mathrm{wc}}]$ so that $\tau_i^{\mathrm{e}} = (\tau_i^{\mathrm{bc}} + \tau_i^{\mathrm{wc}})/2$. The only hard task in the system is T_5 and its deadline is $d_5 = 30$. Tasks T_3 and T_4 are soft and their utility functions are given in Figure 7.2.



$$u_3(t_3) = \begin{cases} 2 & \text{if } t_3 \leq 10, \\ \frac{24}{7} - \frac{t_3}{7} & \text{if } 10 \leq t_3 \leq 24, \\ 0 & \text{if } t_3 \geq 24. \end{cases}$$

$$u_4(t_4) = \begin{cases} 3 & \text{if } t_4 \leq 9, \\ \frac{9}{2} - \frac{t_4}{6} & \text{if } 9 \leq t_4 \leq 27, \\ 0 & \text{if } t_4 \geq 27. \end{cases}$$

Figure 7.2: A monoprocessor system with hard and soft tasks

Because of the data dependencies, there are only six possible schedules, namely $\sigma_a = T_1T_2T_3T_4T_5T_6$, $\sigma_b = T_1T_2T_3T_5T_4T_6$, $\sigma_c = T_1T_2T_4T_3T_5T_6$, $\sigma_d = T_1T_3T_2T_4T_5T_6$, $\sigma_e = T_1T_3T_2T_5T_4T_6$, and $\sigma_f = T_1T_3T_5T_2T_4T_6$. The schedule σ_a does not guarantee satisfaction of the hard deadline $d_5 = 30$ because the completion time of T_5 is in the worst case $t_5 = \tau_1^{\mathrm{wc}} + \tau_2^{\mathrm{wc}} + \tau_3^{\mathrm{wc}} + \tau_4^{\mathrm{wc}} + \tau_5^{\mathrm{wc}} = 34$. A similar reasoning shows that σ_c and σ_d do not guarantee meeting the hard deadline either. By evaluating the total utility $U = u_3(t_3) + u_4(t_4)$ in the expected case (each task lasts its expected duration) for σ_b, σ_e, and σ_f, we can obtain the optimal schedule. For the example shown in Figure 7.2, σ_e

gives the maximum utility in the expected case ($U_e = u_3(\tau_1^e + \tau_3^e) + u_4(\tau_1^e + \tau_3^e + \tau_2^e + \tau_5^e + \tau_4^e) = u_3(10) + u_4(22) = 2.83$) yet guaranteeing the hard deadline. Therefore $\sigma_e = T_1T_3T_2T_5T_4T_6$ is the optimal static schedule.

The above paragraph suggests a straightforward way to obtaining the optimal static schedule: take all possible schedules that respect data dependencies, check which ones guarantee meeting the hard deadlines, and among those pick the one that yields the highest total utility in the case of expected durations for tasks. Such an approach has a time complexity $\mathcal{O}(|\mathbf{T}|!)$ in the worst case.

7.2.1.1 Optimal Solution

The optimal static schedule can be obtained more efficiently (still in exponential time, recall that the problem is **NP**-hard as demonstrated in Section B.2) by considering permutations of only soft tasks instead of permutations of all tasks. Algorithm 7.1 gives the optimal static schedule, in the case of a single processor, in $\mathcal{O}(|\mathbf{S}|!)$ time. For each one of the possible permutations S_k of soft tasks, the algorithm constructs a schedule σ_k by trying to set the soft tasks in σ_k as early as possible respecting the order given by S_k and the hard deadlines (line 3 in Algorithm 7.1). The schedule σ that, among all σ_k, provides the maximum total utility when considering the expected duration for all tasks is the optimal one.

input: A monoprocessor hard/soft system (see Problem 7.2)
output: The optimal static schedule σ

```
1: U := −∞
2: for k ← 1, 2, ..., |S|! do
3:    σ_k := ConstrSch(S_k)
4:    U_k = Σ_{T_i ∈ S} u_i(t_i^e)
5:    if U_k > U then
6:       σ := σ_k
7:       U := U_k
8:    end if
9: end for
```

Algorithm 7.1: OptStaticSchMonoproc

Algorithm 7.2 constructs a schedule, for a given permutation of soft tasks S, by trying to set the soft tasks, according to the order given by S, as early as possible. The rationale is that, if there exists a schedule which guarantees that all hard deadlines are met and respects the order given by S, the maximum total utility for the particular permutation S is obtained when the soft tasks are set in the schedule as early as possible (the proof is given in Section B.3).

input: A vector S containing a permutation of soft tasks
output: A schedule σ constructed by trying to set soft tasks as early as possible, respecting the order given by S

1: $\mathsf{V} := \mathsf{S}$
2: $\mathbf{R} := \{T \in \mathbf{T} \mid {}^\circ T = \emptyset\}$
3: $\sigma := \epsilon$
4: **while** $\mathbf{R} \neq \emptyset$ **do**
5: $\quad \mathbf{A} := \{T \in \mathbf{R} \mid \mathsf{IsSchedulable}(\sigma T)\}$
6: $\quad \mathbf{B} := \{T \in \mathbf{A} \mid$ there is a path from T to $\mathsf{V}_{[1]}\}$
7: $\quad$ **if** $\mathbf{B} = \emptyset$ **then**
8: $\quad\quad$ select $\widetilde{T} \in \mathbf{A}$
9: $\quad$ **else**
10: $\quad\quad$ select $\widetilde{T} \in \mathbf{B}$
11: $\quad$ **end if**
12: $\quad$ **if** $\widetilde{T}$ is in V **then**
13: $\quad\quad$ remove $\widetilde{T}$ from V
14: $\quad$ **end if**
15: $\quad \sigma := \sigma\widetilde{T}$
16: $\quad \mathbf{R} := \mathbf{R} \setminus \{\widetilde{T}\} \cup \{T \in \widetilde{T}^\circ \mid$ all $T' \in {}^\circ T$ are in $\sigma\}$
17: **end while**

Algorithm 7.2: $\mathsf{ConstrSch}(\mathsf{S})$

Algorithm 7.2 first tries to schedule the soft task $\mathsf{S}_{[1]}$ as early as possible. In order to do so, it will set in first place all tasks from which there exists a path leading to $\mathsf{S}_{[1]}$, taking care of not incurring potential deadlines misses by the hard tasks. Then, a similar procedure is followed for the rest of tasks in S.

Algorithm 7.2 keeps a list $\mathbf{R}$ of tasks that are available at every step and constructs the schedule by progressively concatenating tasks to the string σ (initially $\sigma = \epsilon$, where ϵ is the empty string). $\mathbf{A}$ is the set of available tasks that, at that step, can be added to σ without posing the risk of hard deadline misses. In other words, if we added a task $T \in \mathbf{R} \setminus \mathbf{A}$ to σ we could no longer guarantee that the hard deadlines are met. $\mathbf{B}$ is the set of ready tasks that have a path to the next soft task $\mathsf{V}_{[1]}$ to be scheduled. Once an available task $\widetilde{T}$ is selected, it is concatenated to σ (line 15 in Algorithm 7.2), $\widetilde{T}$ is removed from $\mathbf{R}$, and all its direct successors that become available are added to $\mathbf{R}$ (line 16).

At every iteration of the **while** loop of Algorithm 7.2, we construct the set $\mathbf{A}$ by checking, for every $T \in \mathbf{R}$, whether concatenating T to the schedule prefix σ would imply a possible hard deadline miss. For this purpose we use an algorithm $\mathsf{IsSchedulable}(\varsigma)$, which returns a boolean indicating whether there is a schedule that agrees with the prefix ς and such that hard deadlines

are met.

IsSchedulable(ς) is a simple algorithm that conceptually works as follows: first, in a similar spirit as ConstrSch(S), it constructs a schedule σ (that agrees with the prefix ς) where *hard* tasks are set as early as possible, according to order given by their deadlines (that is, $d_i < d_j \Rightarrow \sigma(T_i) < \sigma(T_j)$); then, it checks if hard deadlines are satisfied when all tasks take their worst-case duration and the execution order given by σ is followed.

It is worthwhile to note that the time complexity of Algorithm 7.2 (ConstrSch(S)) is $\mathcal{O}(|\mathbf{T}|^3)$.

Let us consider again the example given in Figure 7.2. There are two permutations of soft tasks $\mathsf{S}_1 = [T_3, T_4]$ and $\mathsf{S}_2 = [T_4, T_3]$. If we follow Algorithm 7.2 for $\mathsf{S}_2 = [T_4, T_3]$, the illustration of its different steps is shown in Table 7.1. Observe that ConstrSch($[T_4, T_3]$) produces $\sigma_2 = T_1T_2T_3T_5T_4T_6$ and therefore the order given by $[T_4, T_3]$ is not respected ($\sigma_2(T_4) = 5 \not< \sigma_2(T_3) = 3$). This means simply that there is no schedule σ such that $\sigma(T_4) < \sigma(T_3)$ and at the same time hard deadlines are guaranteed. For the former permutation of soft tasks, ConstrSch($[T_3, T_4]$) gives $\sigma_1 = T_1T_3T_2T_5T_4T_6$ which is the optimal static schedule.

Step	**R**	**A**	**B**	$\widetilde{T}$	σ
1	$\{T_1\}$	$\{T_1\}$	$\{T_1\}$	T_1	T_1
2	$\{T_2, T_3\}$	$\{T_2, T_3\}$	$\{T_2\}$	T_2	T_1T_2
3	$\{T_3, T_4\}$	$\{T_3\}$	$\emptyset$	T_3	$T_1T_2T_3$
4	$\{T_4, T_5\}$	$\{T_5\}$	$\emptyset$	T_5	$T_1T_2T_3T_5$
5	$\{T_4\}$	$\{T_4\}$	$\{T_4\}$	T_4	$T_1T_2T_3T_5T_4$
6	$\{T_6\}$	$\{T_6\}$	$\emptyset$	T_6	$T_1T_2T_3T_5T_4T_6$

Table 7.1: Illustration of ConstrSch($[T_4, T_3]$)

7.2.1.2 Heuristics

We have discussed in the previous subsection an algorithm that finds the optimal solution to Problem 7.2. Since Problem 7.2 is intractable, any algorithm that solves it exactly requires exponential time. We present in this subsection several heuristic procedures for finding, in polynomial time, a near-optimal solution to the problem of static scheduling for monoprocessor systems with hard and soft tasks.

The proposed algorithms progressively construct the schedule σ by concatenating tasks to the string σ that at the end will contain the final schedule. The heuristics make use of a list **R** of available tasks at every step. The heuristics differ in how the next task, among those in **R**, is selected

as the one to be concatenated to σ. Note that the algorithms presented in this subsection as well as Algorithm 7.1 introduced in Subsection 7.2.1.1 are applicable only if the system is schedulable in first place (there exists a schedule that satisfies the hard time constraints). Determining the schedulability of a monoprocessor system with non-preemptable tasks can be done in polynomial time [Law73].

The algorithms make use of a list scheduling heuristic. Algorithm 7.3 gives the basic procedure. Initially, $\sigma = \epsilon$ (the empty string) and the list $\mathbf{R}$ contains those tasks that have no predecessor. The **while** loop is executed exactly $|\mathbf{T}|$ times. At every iteration we compute the set $\mathbf{A}$ of ready tasks that do not pose risk of hard deadline misses if concatenated to the schedule prefix σ. If all soft tasks have already been set in σ we select any $\widetilde{T} \in \mathbf{A}$, else we compute a priority for soft tasks (line 8 in Algorithm 7.3). The way such priorities are calculated is what differentiates the proposed heuristics. Among those soft tasks that are not in σ, we take T_k as the one with the highest priority (line 9). Then, we obtain the set $\mathbf{B}$ of ready tasks that cause no hard deadline miss and that have a path leading to T_k. We select any $T \in \mathbf{B}$ if $\mathbf{B} \neq \emptyset$, else we choose any $T \in \mathbf{A}$. Once an available task $\widetilde{T}$ is selected as described above, it is concatenated to σ, $\widetilde{T}$ is removed from the list $\mathbf{R}$, and those direct successors of $\widetilde{T}$ that become available are added to $\mathbf{R}$ (lines 17 and 18).

The first of the proposed heuristics makes use of Algorithm 7.3 in combination with Algorithm 7.4 for computing the priorities of the soft tasks. Algorithm 7.4 (PrioritySingleUtility) assigns a priority to the soft tasks, for a given schedule prefix ς, as follows. If T_i is in ς its priority is $\mathsf{SP}_{[i]} := -\infty$, else we compute the schedule σ that has ς as prefix and sets T_i the earliest (that is, we construct a schedule by concatenating to ς all predecessors of T_i as well as T_i itself before any other task). Then the expected completion time t_i^{e} (as given by σ) is used for evaluating the utility function $u_i(t_i)$ of T_i, that is, we assign to $\mathsf{SP}_{[i]}$ the single utility of T_i evaluated at t_i^{e} (line 6). The rationale behind this heuristic is that it provides a greedy manner to compute the schedule in such a way that, at every step, the construction of the schedule is guided by the the soft task that produces the highest utility.

Since Algorithm 7.4 (PrioritySingleUtility) assigns priorities to soft tasks considering their individual utilities, it can be seen as an algorithm that targets local optima. By considering instead the total utility (that is, the sum of contributions by all soft tasks) we can devise a heuristic that targets the global optimum. This comes at a higher computational cost though. Algorithm 7.5 (PriorityMultipleUtility) also exploits the information of utility functions but, as opposed to Algorithm 7.4 (PrioritySingleUtility), it considers the utility contributions of other soft tasks when computing the priority $\mathsf{SP}_{[i]}$

input: A monoprocessor hard/soft system (see Problem 7.2)
output: A near-optimal static schedule σ

```
1: R := {T ∈ T | °T = ∅}
2: σ := ϵ
3: while R ≠ ∅ do
4:    A := {T ∈ R | IsSchedulable(σT)}
5:    if all T ∈ S are in σ then
6:       select T̃ ∈ A
7:    else
8:       SP := Priority(σ)
9:       take T_k ∈ S such that SP_[k] is the highest
10:      B := {T ∈ A | there is a path from T to T_k}
11:      if B = ∅ then
12:         select T̃ ∈ A
13:      else
14:         select T̃ ∈ B
15:      end if
16:   end if
17:   σ := σT̃
18:   R := R \ {T̃} ∪ {T ∈ T̃° | all T' ∈ °T are in σ}
19: end while
```

Algorithm 7.3: HeurStaticSchMonoproc

input: A schedule prefix ς
output: A vector SP containing the priority for soft tasks

```
1: for i ← 1, 2, ..., |S| do
2:    if T_i is in ς then
3:       SP_[i] := −∞
4:    else
5:       σ := schedule that agrees with ς and sets T_i the earliest
6:       SP_[i] := u_i(t_i^e)
7:    end if
8: end for
```

Algorithm 7.4: PrioritySingleUtility(ς)

of the soft task T_i. If the soft task T_i is not in ς its priority is computed as follows. First, we obtain the schedule σ that agrees with ς and sets T_i the earliest (line 5). Based on σ the expected completion time t_i^{e} is obtained and used as argument for evaluating $u_i(t_i)$. Second, for each soft task T_j, different from T_i, that is not in ς, we compute schedules σ' and σ'' that set T_j the earliest and the latest respectively (lines 9 and 10). The expected completion times t'^{e}_j and t''^{e}_j corresponding to σ' and σ'' respectively are then computed. The average of t'^{e}_j and t''^{e}_j is used as argument for the

utility function $u_j(t_j)$ and this value is considered as part of the computed priority $\mathsf{SP}_{[i]}$ (line 11).

input: A schedule prefix ς
output: A vector SP containing the priority for soft tasks

```
1:  for i ← 1, 2, ..., |S| do
2:    if T_i is in ς then
3:      SP_[i] := −∞
4:    else
5:      σ := schedule that agrees with ς and sets T_i the earliest
6:      sp := u_i(t_i^e)
7:      for j ← 1, 2, ..., |S| do
8:        if T_j ≠ T_i and T_j is not in ς then
9:          σ' := schedule that agrees with ς and sets T_j the earliest
10:         σ'' := schedule that agrees with ς and sets T_j the latest
11:         sp := sp + u_j((t'^e_j + t''^e_j)/2)
12:       end if
13:     end for
14:     SP_[i] := sp
15:   end if
16: end for
```

Algorithm 7.5: PriorityMultipleUtility(ς)

To sum up, we have presented two heuristics that are based on a list scheduling algorithm (Algorithm 7.3) and their difference lies in how the priorities for soft tasks (which guide the construction of the schedule) are calculated. The first heuristic uses Algorithm 7.4 whereas the second uses Algorithm 7.5. We have named the heuristics after the algorithms they use for computing priorities: MonoprocSingleUtility (MSU) and MonoprocTotalUtility (MTU) respectively. The experimental evaluation of the proposed heuristics is presented next.

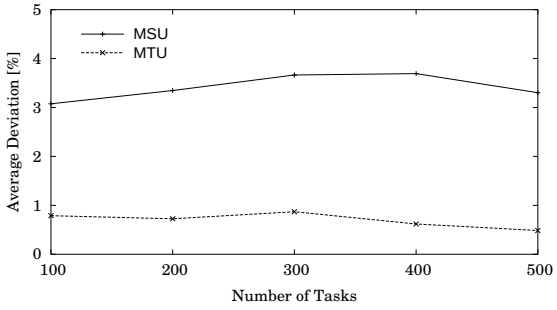
7.2.1.3 Evaluation of the Heuristics

We are initially interested in the quality of the schedules obtained by the heuristics MonoprocSingleUtility (MSU), and MonoprocTotalUtility (MTU) with respect to the optimal schedule as given by the exact algorithm OptStaticSchMonoproc. We use as criterion the deviation $dev = (U_{opt} - U_{heur})/U_{opt}$, where U_{opt} is the total utility given by the optimal schedule and U_{heur} is the total utility corresponding to the schedule obtained with a heuristic.

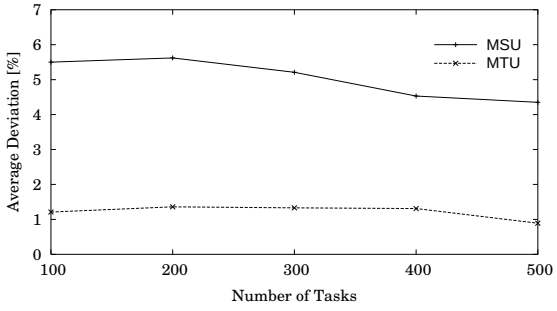
We have randomly generated a large number of task graphs in our experiments. We initially considered graphs with 100, 200, 300, 400, and 500 tasks. For these, we considered systems with 2, 3, 4, 5, 6, 7, and 8 soft tasks. For the case $|\mathbf{T}|$=200 tasks, we considered systems with 25, 50, 75, 100, and

125 hard tasks. We generated 100 graphs for each graph dimension. The graphs were generated in such a way that they all correspond to schedulable systems.

We show the average deviation as a function of the number of tasks in Figures 7.3(a) and 7.3(b). These correspond to systems with 5 and 8 soft tasks respectively. All the systems considered in Figures 7.3(a) and 7.3(b) have 50 hard tasks. These plots consistently show that the MTU gives the best results for the considered cases.



(a) 5 soft tasks



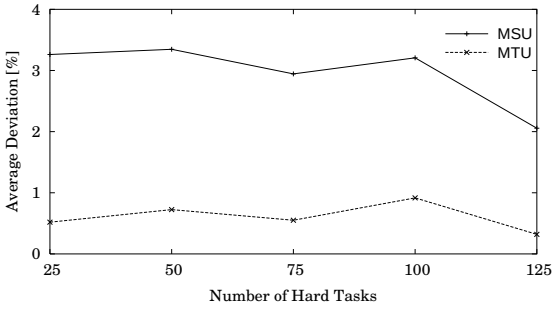
(b) 8 soft tasks

Figure 7.3: Evaluation of the heuristics (50 hard tasks)

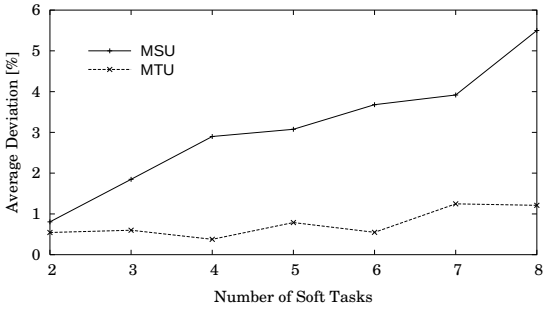
The plot in Figure 7.4(a) depicts the average deviation as a function of the number of hard tasks. In this case, we have considered systems with 200 tasks, out of which 5 are soft. In this graph we observe that the number of hard tasks does not affect significantly the quality the schedules obtained with the proposed heuristics.

We have also studied the average deviation as a function of the number of soft tasks and the results are plotted in Figure 7.4(b). The considered systems have 100 tasks, 50 of them being hard. We again see that the heuristic MTU consistently provides the best results in average. We can also

note that there is a trend showing an increasing average deviation as the number of soft tasks grows, especially for the heuristic MSU. Besides, from Figure 7.4(b), it can be concluded that the quality difference between MSU and MTU grows as the number of soft tasks increases.



(a) 200 tasks, 5 soft tasks



(b) 100 tasks, 50 hard tasks

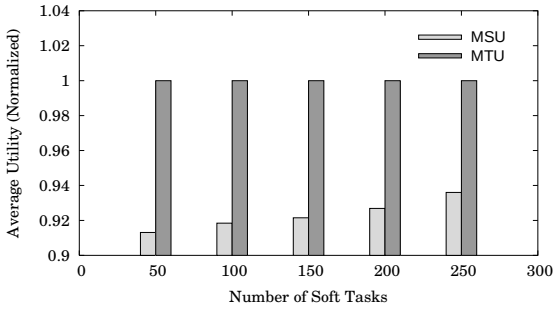
Figure 7.4: Evaluation of the heuristics

Note that, in the experiments we have presented so far, the number of soft tasks is small. Recall that the time complexity of the exact algorithm is $\mathcal{O}(|\mathbf{S}|!)$ and therefore any comparison that requires computing the optimal schedule is infeasible for a large number of soft tasks.

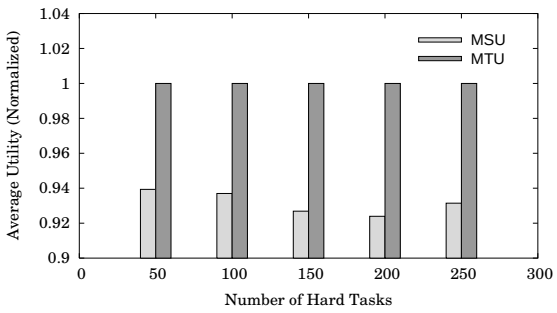
In a second set of experiments, we have compared the two heuristics considering systems with larger numbers of soft and hard tasks. We normalize the utility produced by the heuristics with respect to the total utility delivered by MTU (such a normalized utility is denoted $\|U_{heur}\|$ and is given by $\|U_{heur}\| = U_{heur}/U_{\mathsf{MTU}}$).

We generated, for these experiments, graphs with 500 tasks and considered cases with 50, 100, 150, 200, and 250 hard tasks and 50, 100, 150, 200, and 250 soft tasks. The results are shown in Figures 7.5(a) and 7.5(b), from which we observe that MTU outperforms MSU even for large number of hard

and soft tasks.



(a) 150 hard tasks



(b) 200 soft tasks

Figure 7.5: Comparison among the heuristics (500 tasks)

From the extensive set of experiments that we have performed and its results (Figures 7.3 through 7.5) we conclude that, if one of the proposed heuristics is to be chosen, MTU is the heuristic procedure that should be used for solving the problem of static scheduling for monoprocessor systems with hard and soft tasks. In the cases where it was feasible to compute the optimal schedule, we obtained an average deviation smaller than 2% when using the heuristic MTU. The heuristic MSU gives in average inferior results because, during the process of constructing the schedule, MSU is guided by the individual utility contribution of one soft task whereas MTU takes into account the utility contribution of all soft tasks for computing the individual priority of each particular task.

It must be observed, however, that the heuristic MSU is extremely fast and produces results not far from the optimum. It could be used, for example, in the loop of a design exploration process where it is needed to evaluate quickly several solutions.

Although the superiority of MTU comes at a higher computational cost,

MTU is still fast (for systems with 100 tasks, out of them 30 soft tasks and 50 hard tasks, it takes no longer than 1 s) and hence appropriate for solving efficiently the static scheduling problem discussed in this subsection.

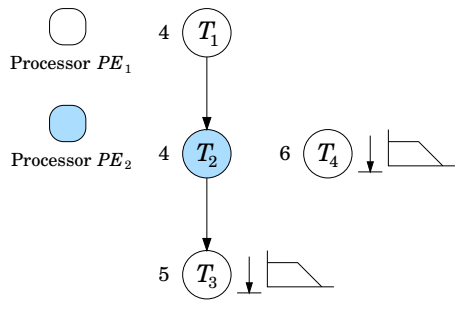
7.2.2 Multiple Processors

We address in this subsection the problem of static scheduling for systems with hard and soft tasks in the general case of multiple processors, according to the formulation of Problem 7.1.

7.2.2.1 Optimal Solution

It was discussed in Subsection 7.2.1 that the problem of static scheduling, in the case of a single processor, could be solved more efficiently in $\mathcal{O}(|\mathbf{S}|!)$ time by considering the permutations of soft tasks, instead of using a method that considers the permutation of all tasks and therefore takes $\mathcal{O}(|\mathbf{T}|!)$ time. However, such a procedure (for each one of the possible permutations S_k of soft tasks, construct schedule σ_k by trying to set the soft tasks in σ_k as early as possible respecting the order given by S_k and the hard deadlines) is no longer valid when the tasks are mapped on several processors. This is illustrated by the following example.

We consider a system with four tasks mapped onto two processors as shown in Figure 7.6 (T_1, T_3, and T_4 are mapped onto PE_1 while T_2 is mapped onto PE_2), where tasks T_3 and T_4 are soft, and there is no hard task. In this particular example $\tau_i^{\mathrm{bc}} = \tau_i^{\mathrm{e}} = \tau_i^{\mathrm{wc}}$ for every task T_i.



$$u_3(t_3) = \begin{cases} 5 & \text{if } t_3 \le 5, \\ \dfrac{25}{4} - \dfrac{t_3}{4} & \text{if } 5 \le t_3 \le 25, \\ 0 & \text{if } t_3 \ge 25. \end{cases}$$

$$u_4(t_4) = \begin{cases} 2 & \text{if } t_4 \le 10, \\ 4 - \dfrac{t_4}{5} & \text{if } 10 \le t_4 \le 20, \\ 0 & \text{if } t_4 \ge 20. \end{cases}$$

Figure 7.6: A multiprocessor system with hard and soft tasks

For the example shown in Figure 7.6 there are two permutations of soft tasks, namely $\mathsf{S}_1 = [T_3, T_4]$ and $\mathsf{S}_2 = [T_4, T_3]$. If we use Algorithm 7.2 (with a slight modification in such a way that we construct a set $\Omega = \{\sigma^{(1)}, \sigma^{(2)}\}$ instead of one σ) we get $\Omega_1 = \{\sigma_1^{(1)} = T_1T_3T_4, \sigma_1^{(2)} = T_2\}$ for S_1 and $\Omega_2 = \{\sigma_2^{(1)} = T_4T_1T_3, \sigma_2^{(2)} = T_2\}$ for S_2. The total utility delivered by Ω_1 is

$U_1 = u_3(13) + u_4(19) = 3.2$ while the total utility delivered by Ω_2 is $U_2 = u_3(19) + u_4(6) = 3.5$. None of these, however, is the optimal schedule. The schedule $\Omega = \{\sigma^{(1)} = T_1T_4T_3, \sigma^{(2)} = T_2\}$ is the one that yields the maximum total utility ($U = u_3(15) + u_4(10) = 4.5$). Hence we conclude that, in case of multiprocessor systems, a procedure that considers only the permutations of soft tasks does not give the optimal schedule. This is so because considering only permutations of soft tasks might lead to unnecessary idle times on certain processors (for instance, $\Omega_1 = \{\sigma_1^{(1)} = T_1T_3T_4, \sigma_1^{(2)} = T_2\}$ obtained out of $\mathsf{S}_1 = [T_3, T_4]$ makes processor PE_1 be idle while T_2 executes on PE_2—and this idle time could be better utilized as done by the optimal schedule $\Omega = \{\sigma^{(1)} = T_1T_4T_3, \sigma^{(2)} = T_2\}$).

For solving optimally Problem 7.1 (SMU) we have to consider the permutations (taking of course into account the data dependencies) of all tasks mapped onto each processor. For instance, for a system with two processors PE_1 and PE_2, we need to consider in the worst case all $|\mathbf{T}^{(1)}|!$ permutations of tasks mapped onto PE_1 combined with all $|\mathbf{T}^{(2)}|!$ permutations of tasks mapped onto PE_2. Then we pick the schedule (among the schedules defined by such permutations) that gives the highest total utility in the expected case and guarantees all deadlines in the worst case. The optimal algorithm is given by Algorithm 7.6.

input: A p-processor hard/soft system (see Problem 7.1)
output: The optimal static schedule Ω

```
1: U := −∞
2: for j ← 1, 2, ..., |T^(1)|! do
3:    for k ← 1, 2, ..., |T^(2)|! do
4:       ⋮
5:       for l ← 1, 2, ..., |T^(p)|! do
6:          Ω_jk...l := {σ_i^(1), σ_j^(2), ..., σ_l^(p)}
7:          if GuaranteesHardDeadlines(Ω_jk...l) then
8:             U_jk...l = Σ_{T_i ∈ S} u_i(t_i^e)
9:             if U_jk...l > U then
10:               Ω := Ω_jk...l
11:               U := U_jk...l
12:            end if
13:         end if
14:      end for
15:   end for
16: end for
```

Algorithm 7.6: OptStaticSchMultiproc

7.2.2.2 Heuristics

The heuristics that we use for multiprocessor systems are based on the same ideas discussed in Subsection 7.2.1 where we presented two heuristics for the particular case of a single processor. We use list scheduling heuristics that rely on lists $\mathbf{R}^{(i)}$ of ready tasks from which tasks are extracted at every step for constructing the schedule. Every list $\mathbf{R}^{(i)}$ contains the tasks that are eligible to be scheduled on processing element PE_i at every step. We solve conflicts among ready tasks by computing priorities for soft tasks, in such a way that the task that has a path leading to the highest priority soft task is selected.

The multiprocessor version of the general heuristic is given by Algorithm 7.7. The reader shall recall that $\mathbf{T}^{(i)}$ denotes the set of tasks mapped onto PE_i. Note that we use a list $\mathbf{R}^{(i)}$ for each processing element instead of a single $\mathbf{R}$ for all processors. At every iteration we compute the set $\mathbf{A}^{(i)}$ (line 8) of tasks that are ready (all predecessors are already scheduled) and that do not make the system non-schedulable (when concatenated to the respective $\sigma^{(i)}$ it is still possible to construct a schedule that guarantees the hard deadlines). If there are soft tasks yet to be scheduled, we compute priorities for soft tasks, take the T_k with the highest priority, and select a task with a path leading to T_k (lines 12 through 19). Once a task $\widetilde{T}$ is selected, it is concatenated to $\sigma^{(i)}$, it is removed from $\mathbf{R}^{(i)}$, and its direct successors that become ready are added to the respective $\mathbf{R}^{(j)}$ (lines 21 through 25).

For computing the priorities of soft tasks, as required by Algorithm 7.7, we use algorithms very similar to Algorithm 7.4 (PrioritySingleUtility) and Algorithm 7.5 (PriorityMultipleUtility), but considering now that a schedule is a set $\Omega = \{\sigma^{(1)}, \sigma^{(2)}, \ldots, \sigma^{(p)}\}$ instead of one σ, and accordingly a schedule prefix is not a single ς but a set $\{\varsigma^{(1)}, \varsigma^{(2)}, \ldots, \varsigma^{(p)}\}$. The heuristics for the case of multiple processors have been named SingleUtility (SU) and TotalUtility (TU).

7.2.2.3 Evaluation of the Heuristics

The computation of an optimal static schedule using the exact algorithm (Algorithm 7.6) is only feasible for small systems. In order to be able to evaluate the quality of the proposed heuristics, we have implemented an algorithm based on simulated annealing. Simulated annealing is a metaheuristic for solving combinatorial optimization problems [vLA87]. It is based on the analogy between the way in which a metal cools and freezes into a minimum energy crystalline structure (the annealing process) and the search for a minimum in an optimization process. Simulated annealing has been applied in diverse areas with good results in terms of the quality of solutions. On the

input: A p-processor hard/soft system (see Problem 7.1)
output: A near-optimal static schedule Ω

for $i \leftarrow 1, 2, \ldots, p$ **do**
  $\mathbf{R}^{(i)} := \{T \in \mathbf{T}^{(i)} \mid {}^\circ T = \emptyset\}$
end for
$\Omega := \{\sigma^{(1)} = \epsilon, \sigma^{(2)} = \epsilon, \ldots, \sigma^{(p)} = \epsilon\}$
while $\mathbf{R} = \bigcup \mathbf{R}^{(i)} \neq \emptyset$ **do**
  for $i \leftarrow 1, 2, \ldots, p$ **do**
    if $\mathbf{R}^{(i)} \neq \emptyset$ **then**
      $\mathbf{A}^{(i)} := \{T \in \mathbf{R}^{(i)} \mid \mathsf{IsSchedulable}(\{\sigma^{(1)}, \ldots, \sigma^{(i)}T, \ldots, \sigma^{(p)}\})\}$
      if all $T \in \mathbf{S}$ are in Ω **then**
        select $\widetilde{T} \in \mathbf{A}^{(i)}$
      else
        $\mathsf{SP} := \mathsf{Priority}(\Omega)$
        take $T_k \in \mathbf{S}$ such that $\mathsf{SP}_{[k]}$ is the highest
        $\mathbf{B}^{(i)} := \{T \in \mathbf{A}^{(i)} \mid$ there is a path from T to $T_k\}$
        if $\mathbf{B}^{(i)} = \emptyset$ **then**
          select $\widetilde{T} \in \mathbf{A}^{(i)}$
        else
          select $\widetilde{T} \in \mathbf{B}^{(i)}$
        end if
      end if
      $\sigma^{(i)} := \sigma^{(i)}\widetilde{T}$
      $\mathbf{R}^{(i)} := \mathbf{R}^{(i)} \setminus \{\widetilde{T}\}$
      for $j \leftarrow 1, 2, \ldots, p$ **do**
        $\mathbf{R}^{(j)} := \mathbf{R}^{(j)} \cup \{T \in \widetilde{T}^\circ \mid T \in \mathbf{T}^{(j)}$ *and* all $T' \in {}^\circ T$ are in $\Omega\}$
      end for
    end if
  end for
end while

Algorithm 7.7: HeurStaticSchMultiproc

other hand, simulated annealing is quite slow.

We compared the solutions produced by SU and TU against the one given by simulated annealing applied to our static scheduling problem. We used as criterion the deviation $dev = (U_{sim-ann} - U_{heur})/U_{sim-ann}$, where $U_{sim-ann}$ is the total utility given by the schedule found using the simulated annealing strategy and U_{heur} is the total utility corresponding to the schedule obtained with a heuristic.

We generated synthetic task graphs (100 systems for each graph dimension) with up to 200 tasks, mapped on architectures consisting of 2 to 10 processing elements. In a first set of experiments, we varied the size of the system keeping constant the number of processing elements. The average deviation as a function of the number of tasks is shown in Figure 7.7. In

these experiments we considered that 25% of the total number of tasks are soft and 25% are hard, and that the architecture has 5 processing elements. As expected, the heuristic TU performs significantly better than SU, with average deviations below 10%. It is interesting to note that the simulated annealing algorithm, which finds the near-optimal solutions that we use as reference point, takes up to 75 minutes, for systems with 200 tasks, while the heuristics have execution times of around 5 s.

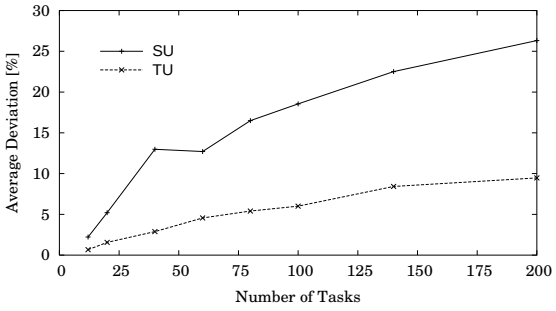


Figure 7.7: Evaluation of the multiprocessor heuristics (25% hard tasks, 25% soft tasks, 5 processors)

In a second set of experiments, we fixed the number of soft, hard, and total number of tasks and varied the number of processing elements. We considered systems with 80 tasks, out of which 20 are soft and 20 are hard. The results are shown in Figure 7.8. A slight decrease in the average deviation can be observed as the number of processing elements increases. This might be explained by the fact that for a larger number of processing elements, while keeping constant the number of tasks, the size of the solution space (number of possible schedules) gets smaller.

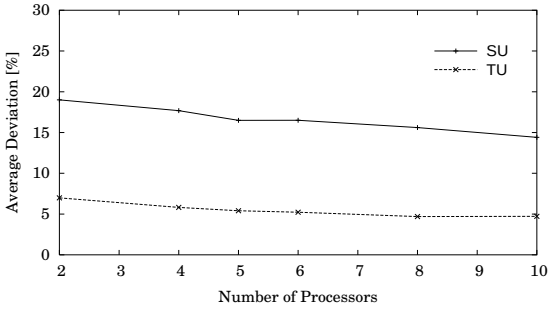


Figure 7.8: Evaluation of the multiprocessor heuristics (80 tasks, 20 soft tasks, 20 hard tasks)

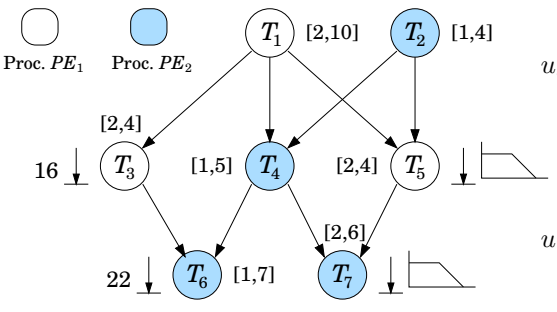
7.3 Quasi-Static Scheduling

It was mentioned in the introduction of this chapter that a single static schedule is overly pessimistic and that a dynamic scheduling approach incurs a high on-line overhead. We propose therefore a quasi-static solution where a number of schedules are computed at design-time, leaving for run-time only the selection of a particular schedule, based on the actual execution times.

This section addresses the problem of quasi-static scheduling for real-time systems that have hard and soft tasks. We start by discussing in Subsection 7.3.1 an example that illustrates various aspects of the approach. We propose a method for computing at design-time a set of schedules such that an ideal on-line scheduler (Subsection 7.3.2) is matched by a quasi-static scheduler operating on this set of schedules (Subsection 7.3.3). Since this problem is intractable, we present heuristics that deal with the time and memory complexity and produce suboptimal good-quality solutions (Subsection 7.3.4).

7.3.1 Motivational Example

Let us consider the system shown in Figure 7.9. Tasks T_1, T_3, T_5 are mapped onto processor PE_1 and tasks T_2, T_4, T_6, T_7 are mapped onto PE_2. For the sake of simplicity, we ignore inter-processor communication in this example. We also assume that the execution time of every task T_i is uniformly distributed over its interval $[\tau_i^{\mathrm{bc}}, \tau_i^{\mathrm{wc}}]$. Tasks T_3 and T_6 are hard and their deadlines are $d_3 = 16$ and $d_6 = 22$ respectively. Tasks T_5 and T_7 are soft and their utility functions are given in Figure 7.9.



$$u_5(t_5) = \begin{cases} 2 & \text{if } t_5 \le 5, \\ 3 - \dfrac{t_5}{5} & \text{if } 5 \le t_5 \le 15, \\ 0 & \text{if } t_5 \ge 15. \end{cases}$$

$$u_7(t_7) = \begin{cases} 3 & \text{if } t_7 \le 3, \\ \dfrac{18}{5} - \dfrac{t_7}{5} & \text{if } 3 \le t_7 \le 18, \\ 0 & \text{if } t_7 \ge 18. \end{cases}$$

Figure 7.9: Motivational multiprocessor example

The optimal static schedule, according to the formulation given by Problem 7.1, corresponds to the task execution order which, among all the schedules that satisfy the hard constraints in the worst case, maximizes the sum of individual contributions by soft tasks when each utility function is eval-

uated at the task's expected completion time. For the system shown in Figure 7.9, the optimal static schedule is $\Omega = \{\sigma^{(1)} = T_1T_3T_5, \sigma^{(2)} = T_2T_4T_6T_7\}$ (in the rest of this section we will use the simplified notation $\Omega = \{T_1T_3T_5, T_2T_4T_6T_7\}$).

Although $\Omega = \{T_1T_3T_5, T_2T_4T_6T_7\}$ is optimal in the static sense, it is still pessimistic because the actual execution times, which are unknown beforehand, might be far off from the ones used to compute the static schedule. This point is illustrated by the following situation. The system starts execution according to Ω, that is T_1 and T_2 start at $s_1 = s_2 = 0$. Assume that T_2 completes at $t_2 = 4$ and then T_1 completes at $t_1 = 6$. At this point, taking advantage of the fact that we know the completion times t_1 and t_2, we can compute the schedule that is consistent with the tasks already executed, maximizes the total utility (considering the actual execution times of T_1 and T_2—already executed—and expected duration for T_3, T_4, T_5, T_6, T_7—remaining tasks), and also guarantees all hard deadlines (even if all remaining tasks execute with their worst-case duration). Such a schedule is $\Omega' = \{T_1T_5T_3, T_2T_4T_6T_7\}$. In the case $\tau_1 = 6$, $\tau_2 = 4$, and $\tau_i = \tau_i^{\mathrm{e}}$ for $3 \leq i \leq 7$, Ω' yields a total utility $U' = u_5(9) + u_7(20) = 1.2$ which is higher than the one given by the static schedule Ω ($U = u_5(12) + u_7(17) = 0.8$). Since the decision to follow Ω' is taken after T_1 completes and knowing its completion time, meeting the hard deadlines is also guaranteed.

A purely on-line scheduler would compute, every time a task completes, a new execution order for the tasks not yet started such that the total utility is maximized for the new conditions (actual execution times of already completed tasks and expected duration for the remaining tasks) while guaranteeing that hard deadlines are met. However, the complexity of the problem is so high that the on-line computation of one such schedule is prohibitively expensive. Recall that such a problem is **NP**-hard, even in the monoprocessor case. In our quasi-static solution, we compute at design-time a number of schedules and switching points, leaving for run-time only the decision to choose a particular schedule based on the actual execution times. Thus the on-line overhead incurred by the quasi-static scheduler is very low because it only compares the actual completion time of a task with that of a predefined switching point and selects accordingly the already computed execution order for the remaining tasks.

We can define, for instance, a switching point $\Omega \xrightarrow{T_1;[2,6]} \Omega'$ for the example given in Figure 7.9, with $\Omega = \{T_1T_3T_5, T_2T_4T_6T_7\}$ and $\Omega' = \{T_1T_5T_3, T_2T_4T_6T_7\}$, such that the system starts executing according to the schedule Ω; when T_1 completes, if $2 \leq t_1 \leq 6$ the tasks not yet started execute in the order given by Ω', else the execution order continues according to Ω. While the solution $\{\Omega, \Omega'\}$, as explained above, guarantees meeting the hard

deadlines and incurs a very low on-line overhead, it provides a total utility which is greater than the one given by the static schedule Ω in 43% of the cases (this figure can be obtained by profiling the system, that is, generating a large number of execution times for tasks according to their probability distributions and, for each particular set of execution times, computing the total utility). Also, for each of the above two solutions, we found that the static schedule Ω yields an average total utility 0.89 while the quasi-static solution $\{\Omega, \Omega'\}$ gives an average total utility of 1.04.

Another quasi-static solution, similar to the one discussed above, is $\{\Omega, \Omega'\}$ but with $\Omega \xrightarrow{T_1;[2,7]} \Omega'$ which actually gives better results (it outperforms the static schedule Ω in 56 % of the cases and yields an average total utility of 1.1, yet guaranteeing no hard deadline miss). Thus the most important question in the quasi-static approach discussed in this section is how to compute, at design-time, the set of schedules and switching points such that they deliver the highest quality (utility). The rest of this section addresses this question and different issues that arise when solving the problem.

7.3.2 Ideal On-Line Scheduler and Problem Formulation

7.3.2.1 Ideal On-Line Scheduler

We use a purely on-line scheduler as reference point in our quasi-static approach to scheduling for real-time systems with hard and soft tasks. This means that, when computing a number of schedules and switching points, our aim is to match an ideal on-line scheduler in terms of the yielded total utility. Such an on-line scheduler solves, after the completion of every task, a problem that is very much alike to Problem 7.1 (SMU), except that actual execution times are considered and the order of completed tasks is taken into account. We rewrite for completeness the problem to be solved by the on-line scheduler.

On-Line Scheduler: Before the activation of the system and every time a task completes, the on-line scheduler solves the following problem:

Problem 7.3 (On-Line SMU) Find a schedule Ω (set of p bijections $\{\sigma^{(1)} : \mathbf{T}^{(1)} \rightarrow \{1, 2, \ldots, |\mathbf{T}^{(1)}|\}, \sigma^{(2)} : \mathbf{T}^{(2)} \rightarrow \{1, 2, \ldots, |\mathbf{T}^{(2)}|\}, \ldots, \sigma^{(p)} : \mathbf{T}^{(p)} \rightarrow \{1, 2, \ldots, |\mathbf{T}^{(p)}|\}\}$ with $\mathbf{T}^{(l)}$ being the set of tasks mapped onto the processing element PE_l and p being the number of processing elements) that maximizes $U = \sum_{T_i \in \mathbf{S}} u_i(t_i^{\mathrm{e}})$ where t_i^{e} is the expected completion time of task T_i, subject to: $t_i^{\mathrm{wc}} \leq d_i$ for all $T_i \in \mathbf{H}$, where t_i^{wc} is the worst-case completion time of task T_i; no deadlock is introduced by Ω; each $\sigma^{(l)}$ has a prefix $\sigma_x^{(l)}$, with $\sigma_x^{(l)}$ being the order of the tasks already executed or under execution on processing element PE_l.

§1. The expected completion time of T_i is given by
$$t_i^{\mathrm{e}} = \begin{cases} \max_{T_j \in {}^\circ T_i}\{t_j^{\mathrm{e}}\} + \mathsf{e}_i & \text{if } \sigma^{(l)}(T_i) = 1, \\ \max(\max_{T_j \in {}^\circ T_i}\{t_j^{\mathrm{e}}\}, t_k^{\mathrm{e}}) + \mathsf{e}_i & \text{if } \sigma^{(l)}(T_i) = \sigma^{(l)}(T_k) + 1. \end{cases}$$
where: $m(T_i) = p_l$; $\max_{T_j \in {}^\circ T_i}\{t_j^{\mathrm{e}}\} = 0$ if ${}^\circ T_i = \emptyset$; $\mathsf{e}_i = \tau_i$ if T_i has completed, $\mathsf{e}_i = \tau_i^{\mathrm{wc}}$ if T_i is executing, else $\mathsf{e}_i = \tau_i^{\mathrm{e}}$.

§2. The worst-case completion time of T_i is given by
$$t_i^{\mathrm{wc}} = \begin{cases} \max_{T_j \in {}^\circ T_i}\{t_j^{\mathrm{wc}}\} + \mathsf{wc}_i & \text{if } \sigma^{(l)}(T_i) = 1, \\ \max(\max_{T_j \in {}^\circ T_i}\{t_j^{\mathrm{wc}}\}, t_k^{\mathrm{wc}}) + \mathsf{wc}_i & \text{if } \sigma^{(l)}(T_i) = \sigma^{(l)}(T_k) + 1. \end{cases}$$
where: $m(T_i) = p_l$; $\max_{T_j \in {}^\circ T_i}\{t_j^{\mathrm{wc}}\} = 0$ if ${}^\circ T_i = \emptyset$; $\mathsf{wc}_i = \tau_i$ if T_i has been completed, else $\mathsf{wc}_i = \tau_i^{\mathrm{wc}}$.

§3. No deadlock introduced by Ω means that when considering a task graph with its original edges together with additional edges defined by the partial order corresponding to the schedule, the resulting task graph must be acyclic.

It can be noted that the differences between SMU (Problem 7.1) and On-Line SMU (Problem 7.3) lie in: a) actual execution times of tasks already completed are used in the latter problem, as seen in §1 and §2; b) the schedule Ω, in the latter problem, must consider the order of tasks already completed or under execution.

Ideal On-Line Scheduler: In an ideal case, where the on-line scheduler solves On-Line SMU in zero time, for any set of execution times $\tau_1, \tau_2, \ldots, \tau_n$ (each known only when the corresponding task completes), the total utility yielded by the on-line scheduler is denoted $U_{\{\tau_i\}}^{ideal}$.

The total utility delivered by the ideal on-line scheduler, as given above, represents an upper bound on the utility that can practically be produced without knowing in advance the actual execution times and without accepting risks regarding hard deadline violations. This is due to the fact that the defined scheduler optimally solves Problem 7.3 (On-Line SMU) in zero time, it is aware of the actual execution times of all completed tasks, and optimizes the total utility assuming that the remaining tasks will run for their expected (which is the most likely) execution time. We note again that, although the optimization goal is the total utility assuming expected duration for the remaining tasks, this optimization is performed under the constraint that hard deadlines are satisfied even in the situation of worst-case duration for the remaining tasks.

7.3.2.2 Problem Formulation

Due to the **NP**-hardness of Problem 7.3, which the on-line scheduler must solve every time a task completes, such an on-line scheduler causes an unacceptable overhead. We propose instead to prepare at design-time schedules

and switching points, where the selection of the actual schedule is done at run-time, at a low cost, by the so-called *quasi-static scheduler*. The aim is to match the utility delivered by an ideal on-line scheduler. This problem is formulated as follows:

PROBLEM 7.4 (Multiple Schedules—MS) Find a set of multiprocessor schedules and switching points such that, for any set of execution times $\tau_1, \tau_2, \ldots, \tau_n$, hard deadlines are guaranteed and the total utility $U_{\{\tau_i\}}$ yielded by the quasi-static scheduler is equal to $U^{ideal}_{\{\tau_i\}}$.

7.3.3 Optimal Set of Schedules and Switching Points

We present in this subsection the systematic procedure for computing the optimal set of schedules and switching points as required by the multiple-schedules problem (Problem 7.4). By optimal, in this context, we mean a solution which guarantees hard deadlines and produces a total utility of $U^{ideal}_{\{\tau_i\}}$. Note that the problem of obtaining such an optimal solution is intractable. Nonetheless, despite its complexity, the optimal procedure described here has also theoretical relevance: it shows that an infinite space of execution times (the execution time of task T_j can be any value in the interval $[\tau_j^{\text{bc}}, \tau_j^{\text{wc}}]$) might be covered optimally by a finite number of schedules, albeit it may be a very large number.

The key idea is to express the total utility, for every feasible task execution order, as a function of the completion time t_k of a particular task T_k. Since different schedules yield different utilities, the objective of the analysis is to pick out the schedule that gives the highest utility and also guarantees no hard deadline miss, depending on the completion time t_k.

We discuss first the case of a single processor and then we generalize the method for multiprocessor systems.

7.3.3.1 Single Processor

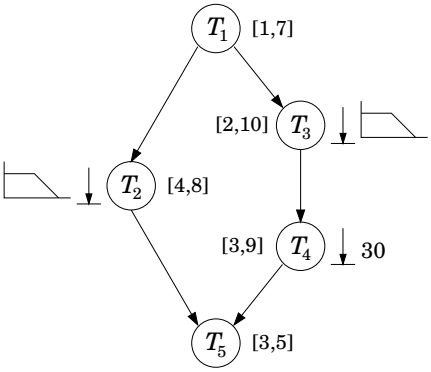
We start by taking the monoprocessor schedule σ that is solution to Problem 7.2 (MSMU). Let us assume that $\sigma(T_1) = 1$, that is, T_1 is the first task of σ. For each one of the schedules σ_i that start with T_1 and satisfy the precedence constraints, we express the total utility $U_i(t_1)$ as a function of the completion time t_1 of task T_1, for the interval of possible completion times of T_1 (in this case $\tau_1^{\text{bc}} \leq t_1 \leq \tau_1^{\text{wc}}$). When computing U_i we consider $\tau_i = \tau_i^{\text{e}}$ for all $T_i \in \mathbf{T} \setminus \{T_1\}$ (expected duration for the remaining tasks). Then, for each possible σ_i, we analyze the schedulability of the system, that is, which values of the completion time t_1 imply potential hard deadline misses when σ_i is followed. For this analysis we consider $\tau_i = \tau_i^{\text{wc}}$ for all $T_i \in \mathbf{T} \setminus \{T_1\}$

(worst-case duration for the remaining tasks). We introduce the auxiliary function $\hat{U}_i$ such that $\hat{U}_i(t_1) = -\infty$ if following σ_i, after T_1 has completed at t_1, does not guarantee the hard deadlines, else $\hat{U}_i(t_1) = U_i(t_1)$.

Once we have computed all the functions $\hat{U}_i(t_1)$, we may determine which σ_i yields the maximum total utility at which instants in the interval $[\tau_1^{\mathrm{bc}}, \tau_1^{\mathrm{wc}}]$. We get thus the interval $[\tau_1^{\mathrm{bc}}, \tau_1^{\mathrm{wc}}]$ partitioned into subintervals and, for each one of these, we obtain the execution order to follow after T_1 depending on the completion time t_1. We refer to this as the *interval-partitioning step*. Observe that such subintervals define the switching points we want to compute.

For each one of the obtained schedules, we repeat the process, this time computing $\hat{U}_j$'s as a function of the completion time of the second task in the schedule and for the interval in which this second task may finish. Then the process is similarly repeated for the third element of the new schedules, and so on. In this manner we obtain the optimal *tree* of schedules and switching points as required by Problem 7.4 (MS).

The process described above is best illustrated by an example. Let us consider the system shown in Figure 7.10. This system has one hard task T_4 with deadline $d_4 = 30$ and two soft tasks T_2 and T_3 whose utility functions are also given in Figure 7.10. Assuming uniform execution time probability distributions, the expected durations of tasks are $\tau_1^{\mathrm{e}} = \tau_5^{\mathrm{e}} = 4$ and $\tau_2^{\mathrm{e}} = \tau_3^{\mathrm{e}} = \tau_4^{\mathrm{e}} = 6$.



$$u_2(t_2) = \begin{cases} 3 & \text{if } t_2 \leq 9, \\ \frac{9}{2} - \frac{t_2}{6} & \text{if } 9 \leq t_2 \leq 27, \\ 0 & \text{if } t_2 \geq 27. \end{cases}$$

$$u_3(t_3) = \begin{cases} 2 & \text{if } t_3 \leq 18, \\ 8 - \frac{t_3}{3} & \text{if } 18 \leq t_3 \leq 24, \\ 0 & \text{if } t_3 \geq 24. \end{cases}$$

Figure 7.10: Motivational monoprocessor example

The optimal static schedule for the system shown in Figure 7.10 is $\sigma = T_1T_3T_4T_2T_5$. Due to the given data dependencies, there are three possible schedules that start with T_1, namely $\sigma_a = T_1T_2T_3T_4T_5$, $\sigma_b = T_1T_3T_2T_4T_5$, and $\sigma_c = T_1T_3T_4T_2T_5$. We want to compute the corresponding functions $U_a(t_1)$, $U_b(t_1)$, and $U_c(t_1)$, $1 \leq t_1 \leq 7$, considering the expected duration for T_2, T_3, T_4, and T_5. For example, $U_b(t_1) = u_2(t_1 + \tau_3^{\mathrm{e}} + \tau_2^{\mathrm{e}}) + u_3(t_1 + \tau_3^{\mathrm{e}}) =$

$u_2(t_1 + 12) + u_3(t_1 + 6)$. We get the following functions:

$$U_a(t_1) = \begin{cases} 5 & \text{if } 1 \le t_1 \le 3, \\ 11/2 - t_1/6 & \text{if } 3 \le t_1 \le 6, \\ 15/2 - t_1/2 & \text{if } 6 \le t_1 \le 7. \end{cases} \tag{7.1}$$

$$U_b(t_1) = 9/2 - t_1/6 \quad \text{if } 1 \le t_1 \le 7. \tag{7.2}$$

$$U_c(t_1) = 7/2 - t_1/6 \quad \text{if } 1 \le t_1 \le 7. \tag{7.3}$$

The functions $U_a(t_1)$, $U_b(t_1)$, and $U_c(t_1)$, as given by Equations (7.1)-(7.3), are shown in Figure 7.11(a). Now, for each one of the schedules σ_a, σ_b, and σ_c, we determine the latest completion time t_1 that guarantees meeting hard deadlines when that schedule is followed. For example, if the execution order given by $\sigma_a = T_1T_2T_3T_4T_5$ is followed and the remaining tasks take their maximum duration, the hard deadline $d_4 = 30$ is met only when $t_1 \le 3$. This is because $t_4 = t_1 + \tau_2^{\text{wc}} + \tau_3^{\text{wc}} + \tau_4^{\text{wc}} = t_1 + 27$ in the worst case and therefore $t_4 \le d_4$ if and only if $t_1 \le 3$. A similar analysis shows that σ_b guarantees meeting the hard deadline only when $t_1 \le 3$ while σ_c guarantees the hard deadline for any completion time t_1 in the interval $[1, 7]$. Thus we get the auxiliary functions as given by Equations (7.4)-(7.6), and depicted in Figure 7.11(b).

$$\hat{U}_a(t_1) = \begin{cases} 5 & \text{if } 1 \le t_1 \le 3, \\ -\infty & \text{if } 3 < t_1 \le 7. \end{cases} \tag{7.4}$$

$$\hat{U}_b(t_1) = \begin{cases} 9/2 - t_1/6 & \text{if } 1 \le t_1 \le 3, \\ -\infty & \text{if } 3 < t_1 \le 7. \end{cases} \tag{7.5}$$

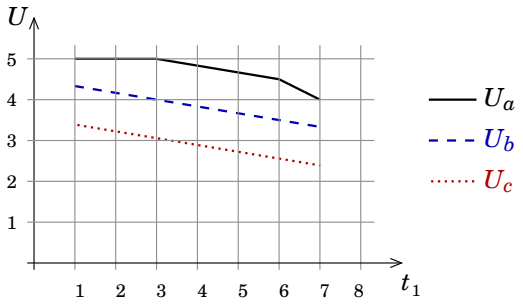
$$\hat{U}_c(t_1) = 7/2 - t_1/6 \quad \text{if } 1 \le t_1 \le 7. \tag{7.6}$$

From the graphic shown in Figure 7.11(b) we conclude that $\sigma_a = T_1T_2T_3T_4T_5$ yields the highest total utility when T_1 completes in the subinterval $[1, 3]$ still guaranteeing the hard deadline, and that $\sigma_c = T_1T_3T_4T_2T_5$ yields the highest total utility when T_1 completes in the subinterval $(3, 7]$ also guaranteeing the hard deadline.

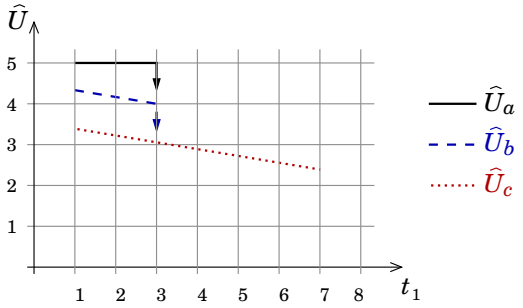
A similar procedure is followed, first for σ_a and then for σ_c, considering the completion time of the second task in these schedules. Let us take $\sigma_a = T_1T_2T_3T_4T_5$. We must analyze the legal schedules having T_1T_2 as prefix. However, since there is only one such schedule, there is no need to continue along the branch originated from σ_a.

Let us take $\sigma_c = T_1T_3T_4T_2T_5$. We make an analysis of the possible schedules σ_j that have T_1T_3 as prefix ($\sigma_d = T_1T_3T_2T_4T_5$ and $\sigma_e = T_1T_3T_4T_2T_5$) and for each of these we obtain $U_j(t_3)$, $5 < t_3 \le 17$ (recall that: σ_c is followed after completing T_1 at $3 < t_1 \le 7$; $2 \le \tau_3 \le 10$). The corresponding functions, when considering expected duration for T_2, T_4, and T_5, are:

$$U_d(t_3) = 7/2 - t_3/6 \quad \text{if } 5 < t_3 \le 17. \tag{7.7}$$



(a) $U_i(t_1)$, $1 \leq t_1 \leq 7$



(b) $\hat{U}_i(t_1)$, $1 \leq t_1 \leq 7$

Figure 7.11: $U_i(t_1)$ and $\hat{U}_i(t_1)$ for the example in Figure 7.10

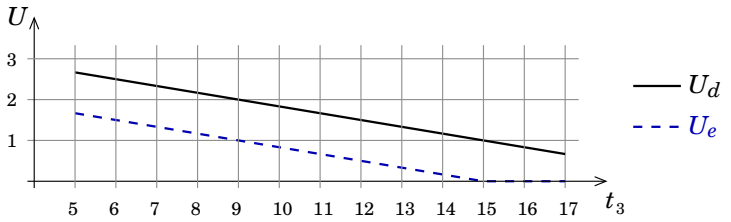
$$U_e(t_3) = \begin{cases} 5/2 - t_3/6 & \text{if } 5 < t_3 \leq 15, \\ 0 & \text{if } 15 \leq t_3 \leq 17. \end{cases} \tag{7.8}$$

$U_d(t_3)$ and $U_e(t_3)$, as given by Equations (7.7) and (7.8), are shown in Figure 7.12(a). Note that there is no need to include the contribution $u_3(t_3)$ by the soft task T_3 in $U_d(t_3)$ and $U_e(t_3)$ because such a contribution is the same for both σ_d and σ_e and therefore it is not relevant when differentiating between $\hat{U}_d(t_3)$ and $\hat{U}_e(t_3)$. After the hard deadlines analysis, the auxiliary utility functions under consideration become:

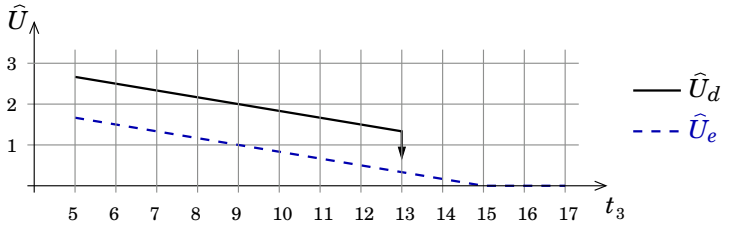
$$\hat{U}_d(t_3) = \begin{cases} 7/2 - t_3/6 & \text{if } 5 < t_3 \leq 13, \\ -\infty & \text{if } 13 < t_3 \leq 17. \end{cases} \tag{7.9}$$

$$\hat{U}_e(t_3) = \begin{cases} 5/2 - t_3/6 & \text{if } 5 < t_3 \leq 15, \\ 0 & \text{if } 15 \leq t_3 \leq 17. \end{cases} \tag{7.10}$$

From the graphic shown in Figure 7.12(b) we conclude: if task T_3 completes in the interval $(5, 13]$, $\sigma_d = T_1T_3T_2T_4T_5$ is the schedule to be followed; if T_3 completes in the interval $(13, 17]$, $\sigma_e = T_1T_3T_4T_2T_5$ is the schedule to be followed. The procedure terminates at this point since there is no other



(a) $U_j(t_3)$, $5 < t_3 \leq 7$



(b) $\hat{U}_j(t_3)$, $5 < t_3 \leq 7$

Figure 7.12: $U_j(t_3)$ and $\hat{U}_j(t_3)$ for the example in Figure 7.10

scheduling alternative after completing the third task of either σ_d or σ_e.

At the end, renaming σ_a and σ_d, we get the set of schedules $\{\sigma = T_1T_3T_4T_2T_5, \sigma' = T_1T_2T_3T_4T_5, \sigma'' = T_1T_3T_2T_4T_5\}$ that works as follows (see Figure 7.13): once the system is activated, it starts following the schedule σ; when T_1 is finished, its completion time t_1 is read, and if $t_1 \leq 3$ the schedule is switched to σ' for the remaining tasks, else the execution order continues according to σ; when T_3 finishes, while σ is the followed schedule, its completion time t_3 is compared with the time point 13: if $t_3 \leq 13$ the remaining tasks are executed according to σ'', else the schedule σ is followed.

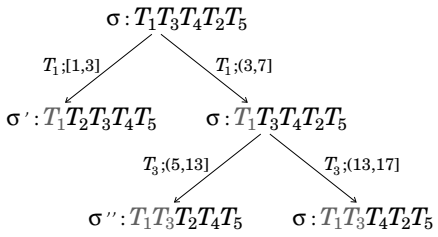


Figure 7.13: Optimal tree of schedules for the example shown in Figure 7.10

It is not difficult to show that, as required by Problem 7.4, the procedure we have described finds a set of schedules and switching points such that the quasi-static scheduler delivers the same utility as the ideal on-line

scheduler defined in Subsection 7.3.2. Both the on-line scheduler and the quasi-static scheduler would start off the system following the same schedule (the optimal static schedule). Upon completion of every task, the on-line scheduler computes a new schedule that maximizes the total utility when taking into account the actual execution times for the already completed tasks and the expected durations for the tasks yet to be executed. Our procedure analyzes off-line, beginning with the first task in the static schedule, the sum of utilities by soft tasks as a function of the completion time of the first task, for each one of the possible schedules starting with that task. For computing the utility as a function of the completion time, our procedure considers expected durations for the remaining tasks. In this way, the procedure determines the schedule that maximizes the total utility at every possible completion time. The process is likewise repeated for the second element of the new schedules, and then the third, and so forth. Thus our procedure solves symbolically the optimization problem for a set of completion times, one of which corresponds to the particular instance solved by the on-line scheduler. Thus, having the tree of schedules and switching points computed in this way, the schedule selected at run-time by the quasi-static scheduler produces a total utility that is equal to that of the ideal on-line scheduler, for any set of execution times.

In the previous discussion regarding the method for finding the optimal set of schedules and switching points (that is, solving Problem 7.4), for instance when T_1 is the first task in the static schedule, we mentioned that we considered each one of the potentially $|\mathbf{T} \setminus \{T_1\}|!$ schedules σ_i that start with T_1 in order to obtain the utilities U_i as a function of the completion time t_1 (interval-partitioning step). This can actually be done more efficiently by considering $|\mathbf{H} \cup \mathbf{S} \setminus \{T_1\}|!$ schedules σ_i, that is, by considering the permutations of hard and soft tasks instead of the permutations of all tasks. In this way the interval-partitioning step, for monoprocessor systems, can be carried out in $\mathcal{O}((|\mathbf{H}| + |\mathbf{S}|)!)$ time instead of $\mathcal{O}(|\mathbf{T}|!)$. The rationale is that the best schedule, for a given permutation HS of hard and soft tasks, is obtained when we try to set the hard and soft tasks in the schedule as early as possible respecting the order given by HS (this is in the same spirit as setting soft tasks the earliest according to a permutation S in order to obtain the highest total utility when solving Problem 7.2, as discussed in Subsection 7.2.1). The proof of this fact is presented in Section B.4 of the Appendix B.

Algorithms 7.8 and 7.9 present the pseudocode for finding the optimal set of schedules and switching points as required by Problem 7.4 (MS) in the particular case of a monoprocessor implementation. First of all, it must be noted that if there is no static schedule that guarantees satisfaction of all

hard deadlines, the system is not schedulable and therefore the problem MS has no solution. According to the previous discussion, when partitioning an interval $\mathcal{I}^i$ of possible completion times t_i, we consider only permutations of hard and soft tasks, instead of permutations of all tasks (lines 5 through 9 in Algorithm 7.9). The procedure $\mathsf{ConstrSch}(\mathsf{HS}_j, \sigma, \mathbf{A})$ (line 6) constructs a schedule σ_j that agrees with σ up to the $|\mathbf{A}|$-th position by trying to set the hard and soft tasks not in $\mathbf{A}$ as early as possible, obeying the order given by HS (its code is very similar to Algorithm 7.2 except that a permutation HS of hard and soft tasks is used instead of S and the resulting schedule is constructed from the prefix corresponding to the first $|\mathbf{A}|$ tasks of σ instead of ϵ). Once the interval $\mathcal{I}^i$ is partitioned, for each one of the obtained schedules σ_k, the process is repeated (lines 11 through 14).

input: A monoprocessor hard/soft system
output: The optimal tree Ψ of schedules and switching points

1: $\sigma :=$ OptStaticSchMonoproc
2: $\Psi :=$ OptTreeMonoproc$(\sigma, \emptyset, -, -)$

Algorithm 7.8: OptTreeMonoproc

input: A schedule σ, the set $\mathbf{A}$ of completed tasks, the last completed task T_l, and the interval $\mathcal{I}^l$ of completion times for T_l
output: The optimal tree Ψ of schedules to follow after completing T_l at $t_l \in \mathcal{I}^l$

1: set σ as root of Ψ
2: $T_i :=$ task after T_l as given by σ
3: $\mathcal{I}^i :=$ interval of possible completion times t_i
4: **if** $|\mathbf{H} \cup \mathbf{S} \setminus \mathbf{A}| > 1$ **then**
5: **for** $j \leftarrow 1, 2, \ldots, |\mathbf{H} \cup \mathbf{S} \setminus \mathbf{A}|!$ **do**
6: $\sigma_j :=$ ConstrSch$(\mathsf{HS}_j, \sigma, \mathbf{A})$
7: compute $\hat{U}_j(t_i)$
8: **end for**
9: partition the interval $\mathcal{I}^i$ into subintervals $\mathcal{I}^i_1, \mathcal{I}^i_2, \ldots, \mathcal{I}^i_K$ s.t. σ_k makes $\hat{U}_k(t_i)$ maximal in $\mathcal{I}^i_k$
10: $\mathbf{A}_i := \mathbf{A} \cup \{T_i\}$
11: **for** $k \leftarrow 1, 2, \ldots, K$ **do**
12: $\Psi_k :=$ OptTreeMonoproc$(\sigma_k, \mathbf{A}_i, T_i, \mathcal{I}^i_k)$
13: add subtree Ψ_k s.t. $\sigma \xrightarrow{T_i; \mathcal{I}^i_k} \sigma_k$
14: **end for**
15: **end if**

Algorithm 7.9: OptTreeMonoproc$(\sigma, \mathbf{A}, T_l, \mathcal{I}^l)$

7.3.3.2 Multiple Processors

The construction of the optimal tree of schedules and switching points in the general case of multiprocessor systems is based on the same ideas of the single processor case: express, for the feasible schedules, the total utility as a function of the completion time t_k of a certain task T_k and then select the schedule that yields the highest utility and guarantees the hard deadlines, depending on t_k.

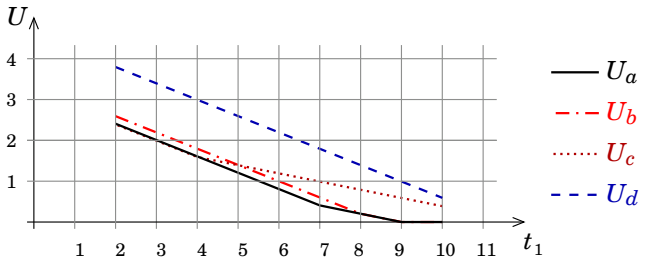
There are though additional considerations to take into account when solving Problem 7.4 for multiple processors:

- Tasks mapped on different processors may be running in parallel at a certain moment. Therefore the "next task to complete" may not necessarily be unique (as it is the case in monoprocessor systems). For example, if n tasks execute concurrently and their completion time intervals overlap, any of them may complete first. In our analysis we must consider separately each of these n cases. For each case the interval of possible completion times can be computed and then it can be partitioned (getting the schedule(s) to follow after completing the task in that particular interval). In other words, the tree also includes the interleaving of possible finishing orders for concurrent tasks.
- In order to partition an interval $\mathcal{I}^k$ of completion times t_k (obtain the schedules that deliver the highest utility, yet guaranteeing the hard deadlines, at different t_k) in the multiprocessor case, it is needed to consider all schedules that satisfy the precedence constraints (all permutations of all tasks in the worst case) whereas in the monoprocessor case, as explained previously, it is needed to consider only feasible schedules defined by the permutations of hard and soft tasks. That is, the interval-partitioning step takes $\mathcal{O}((|\mathbf{H}| + |\mathbf{S}|)!)$ time for monoprocessor systems and $\mathcal{O}((|\mathbf{T}|)!)$ time for multiprocessor systems.
- When a task T_k completes, tasks running on other processors may still be under execution. Therefore the functions $\hat{U}_i(t_k)$ must take into account not only the expected and worst-case durations of tasks not yet started but also the duration of tasks started but not yet completed.

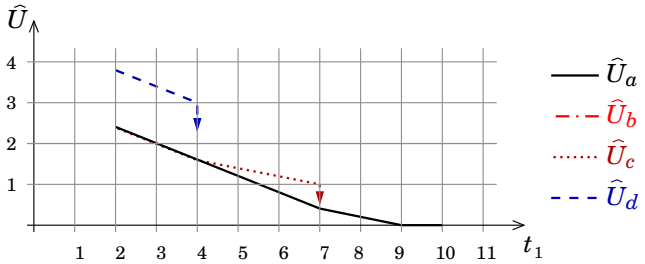
We make use of the example shown in Figure 7.9 in order to illustrate the process of constructing the optimal tree of schedules in the case of multiple processors. The optimal static schedule for this example is $\Omega = \{T_1T_3T_5, T_2T_4T_6T_7\}$. Thus T_1 and T_2 start executing concurrently at time zero and their completion time intervals are $[2, 10]$ and $[1, 4]$ respectively. We initially consider two situations: T_1 completes before T_2 ($2 \leq t_1 \leq 4$); T_2 completes before T_1 ($1 \leq t_2 \leq 4$). For the first one, we compute $\hat{U}_i(t_1)$, $2 \leq t_1 \leq 4$, for each one of the Ω_i that satisfy the precedence constraints,

and we find that $\Omega'' = \{T_1T_5T_3, T_2T_4T_7T_6\}$ is the schedule to follow after T_1 completes (before T_2) at $t_1 \in [2, 4]$ (the details of how this schedule is obtained are skipped at this point). For the second situation, in a similar manner, we find that when T_2 completes (before T_1) in the interval $[1, 4]$, $\Omega = \{T_1T_3T_5, T_2T_4T_6T_7\}$ is the schedule to follow (see Figure 7.15). Details of the interval-partitioning step are illustrated next.

Let us continue with the branch corresponding to T_2 completing first in the interval $[1, 4]$. Under these conditions T_1 is the only running task and its interval of possible completion times is $[2, 10]$. Due to the data dependencies, there are four feasible schedules $\Omega_a = \{T_1T_3T_5, T_2T_4T_6T_7\}$, $\Omega_b = \{T_1T_3T_5, T_2T_4T_7T_6\}$, $\Omega_c = \{T_1T_5T_3, T_2T_4T_6T_7\}$, and $\Omega_d = \{T_1T_5T_3, T_2T_4T_7T_6\}$, and for each of these we compute the corresponding functions $U_a(t_1)$, $U_b(t_1)$, $U_c(t_1)$, and $U_d(t_1)$, $2 \leq t_1 \leq 10$, considering the expected duration for T_3, T_4, T_5, T_6, T_7. For example, $U_d(t_1) = u_5(t_1 + \tau_5^e) + u_7(t_1 + \max(\tau_4^e, \tau_5^e) + \tau_7^e) = u_5(t_1 + 3) + u_7(t_1 + 7)$. We get the functions shown in Figure 7.14(a).



(a) $U_i(t_1)$, $2 \leq t_1 \leq 10$



(b) $\hat{U}_i(t_1)$, $2 \leq t_1 \leq 10$

Figure 7.14: $U_i(t_1)$ and $\hat{U}_i(t_1)$ for the example in Figure 7.9

Now, for Ω_a, Ω_b, Ω_c, and Ω_d, we compute the latest completion time t_1 that guarantees satisfaction of the hard deadlines when that particular schedule is followed. For example, when the execution order is $\Omega_c = \{T_1T_5T_3, T_2T_4T_6T_7\}$, in the worst case $t_3 = t_1 + \tau_5^{wc} + \tau_3^{wc} = t_1 + 8$ and

$t_6 = \max(t_3, t_1 + \tau_4^{\text{wc}}) + \tau_6^{\text{wc}} = \max(t_1 + 8, t_1 + 5) + 7 = t_1 + 15$. Since the hard deadlines for this system are $d_3 = 16$ and $d_6 = 22$, when Ω_c is followed, $t_3 \leq 16$ and $t_6 \leq 22$ if and only if $t_1 \leq 7$. A similar analysis shows the following: Ω_a guarantees the hard deadlines for any completion time $t_1 \in [2, 10]$; Ω_b implies potential hard deadline misses for any $t_1 \in [2, 10]$; Ω_d guarantees the hard deadlines if and only if $t_1 \leq 4$. Thus we get auxiliary functions as shown in Figure 7.14(b).

From the graph in Figure 7.14(b) we conclude that upon completing T_1, in order to get the highest total utility while guaranteeing hard deadlines, the tasks not started must execute according to: $\Omega_d = \{T_1T_5T_3, T_2T_4T_7T_6\}$ if $2 \leq t_1 \leq 4$; $\Omega_c = \{T_1T_5T_3, T_2T_4T_6T_7\}$ if $4 < t_1 \leq 7$; $\Omega_a = \{T_1T_3T_5, T_2T_4T_6T_7\}$ if $7 < t_1 \leq 10$.

The process is then repeated in a similar manner for the newly computed schedules and the possible completion times as defined by the switching points, until the full tree is constructed. The optimal tree of schedules for the system shown in Figure 7.9 is presented in Figure 7.15.

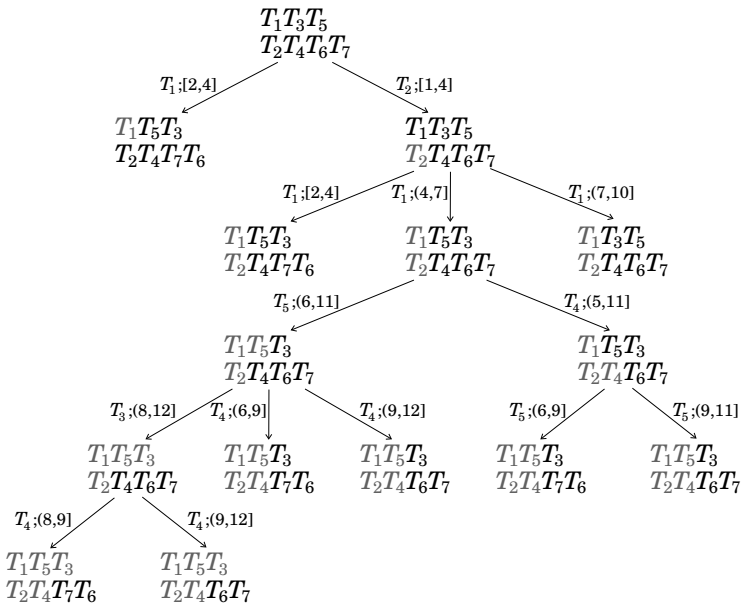


Figure 7.15: Optimal tree of schedules for the example shown in Figure 7.9

When all the descendant schedules of a node (schedule) in the tree are equal to that node, there is no need to store those descendants because the execution order will not change. For example, this is the case of the schedule $\{T_1T_5T_3, T_2T_4T_7T_6\}$ followed after completing T_1 in $[2, 4]$. Also, note that for certain nodes of the tree, there is no need to store the full schedule in

the memory of the target system. For example, the execution order of tasks already completed (which has been taken into account during the preparation of the set of schedules) is clearly unnecessary for the remaining tasks during run-time. Other regularities of the tree can be exploited in order to store it in a more compact way.

It is worthwhile to mention that if two concurrent tasks complete at the very same time (for instance, T_1 and T_2 completing at $t_1 = t_2 = 3$ for the system whose optimal tree is the one in Figure 7.15) the selection by the quasi-static scheduler leads to the same result. In Figure 7.15, if $t_1 = t_2 = 3$ there are two options: a) the branch $T_1; [2, 4]$ is taken and thereafter no schedule change occurs; b) the branch $T_2; [1, 4]$ is taken first followed immediately by the branch $T_1; [2, 4]$. In both cases the selected schedule is $\{T_1T_5T_3, T_2T_4T_7T_6\}$.

The pseudocode for finding the optimal set of schedules and switching points in the case of multiprocessor systems is given by Algorithms 7.10 and 7.11.

input: A multiprocessor hard/soft system (see Problem 7.4)
output: The optimal tree Ψ of schedules and switching points

1: $\Omega :=$ OptStaticSchMultiproc
2: $\Psi :=$ OptTreeMultiproc$(\Omega, \emptyset, -, -)$

Algorithm 7.10: OptTreeMultiproc

The set of schedules is stored in the dedicated shared memory of the system as an ordered tree. Upon completing a task, the cost of selecting at run-time, by the quasi-static scheduler, the execution order for the remaining tasks is $\mathcal{O}(\log n)$ where n is the maximum number of children that a node has in the tree of schedules. Such cost can be included in our analysis by augmenting accordingly the worst-case duration of tasks.

7.3.4 Heuristics and Experimental Evaluation

When computing the optimal tree of schedules and switching points, we partition the interval of possible completion times t for a task T into subintervals which define the switching points and schedules to follow after executing T. As the interval-partitioning step requires in the worst case $\mathcal{O}((|\mathbf{H}| + |\mathbf{S}|)!)$ time for monoprocessor systems and $\mathcal{O}(|\mathbf{T}|!)$ time in general, the multiple-schedules problem (Problem 7.4) is intractable. Moreover, the inherent nature of the problem (finding a tree of schedules) makes it so that it requires exponential time and memory, even when using a polynomial-time heuristic in the interval-partitioning step. Additionally, even if we can afford to compute the optimal tree of schedules (as this is done off-line), the size of

input: A schedule Ω, the set $\mathbf{A}$ of already completed tasks, the last completed task T_l, and the interval $\mathcal{I}^l$ of completion times for T_l
output: The optimal tree Ψ of schedules to follow after completing T_l at $t_l \in \mathcal{I}^l$

```
1: set Ω as root of Ψ
2: compute the set C of concurrent tasks
3: for i ← 1, 2, ..., |C| do
4:   if T_i may complete before the other T_c ∈ C then
5:     compute the interval I^i when T_i may complete first
6:     for j ← 1, 2, ..., |T \ A \ C|! do
7:       if Ω_j is valid then
8:         compute Û_j(t_i)
9:       end if
10:    end for
11:    partition I^i into subintervals I^i_1, I^i_2, ..., I^i_K s.t. Ω_k makes Û_k(t_i)
       maximal in I^i_k
12:    A_i := A ∪ {T_i}
13:    for k ← 1, 2, ..., K do
14:      Ψ_k := OptTreeMultiproc(Ω_k, A_i, T_i, I^i_k)
15:      add subtree Ψ_k s.t. Ω --T_i;I^i_k--> Ω_k
16:    end for
17:  end if
18: end for
```

Algorithm 7.11: $\mathsf{OptTreeMultiproc}(\Omega, \mathbf{A}, T_l, \mathcal{I}^l)$

the tree might still be too large to fit in the available memory resources of the target system. Therefore a suboptimal set of schedules and switching points must be chosen such that the memory constraints imposed by the target system are satisfied. Solutions aiming at tackling different complexity dimensions of the problem, namely the interval-partitioning step and the exponential growth of the tree size, are addressed in this subsection. We discuss the general case of multiprocessor systems, nonetheless the results of the experimental evaluation are shown separately for the single-processor and multiple-processors cases.

7.3.4.1 Interval Partitioning

When partitioning an interval $\mathcal{I}^i$ of possible completion times t_i, the optimal algorithm explores all the permutations of tasks not yet started that define feasible schedules Ω_j and accordingly computes $\hat{U}_j(t_i)$. In order to avoid computing $\hat{U}_j(t_i)$ for all such permutations, we propose a heuristic, called Lim, as given by Algorithm 7.12. This heuristic considers only two schedules Ω_L and Ω_U (line 7 in Algorithm 7.12), computes $\hat{U}_\mathsf{L}(t_i)$ and $\hat{U}_\mathsf{U}(t_i)$

(line 8), and partitions $\mathcal{I}^i$ based on these two functions $\hat{U}_{\mathsf{L}}(t_i)$ and $\hat{U}_{\mathsf{U}}(t_i)$ (line 9). The schedules Ω_{L} and Ω_{U} correspond, respectively, to the solutions to Problem 7.3 for the lower and upper limits t_{L} and t_{U} of the interval $\mathcal{I}^i$. For other completion times $t_i \in \mathcal{I}^i$ different from t_{L}, Ω_{L} is rather optimistic but it might happen that it does not guarantee hard deadlines. On the other hand, Ω_{U} can be pessimistic but does guarantee hard deadlines for all $t_i \in \mathcal{I}^i$. Thus, by combining the optimism of Ω_{L} with the guarantees provided by Ω_{U}, good quality solutions can be obtained.

input: A schedule Ω, the set $\mathbf{A}$ of already completed tasks, the last completed task T_l, and the interval $\mathcal{I}^l$ of completion times for T_l
output: The tree Ψ of schedules to follow after completing T_l at $t_l \in \mathcal{I}^l$

1: set Ω as root of Ψ
2: compute the set $\mathbf{C}$ of concurrent tasks
3: **for** $i \leftarrow 1, 2, \ldots, |\mathbf{C}|$ **do**
4: **if** T_i may complete before the other $T_c \in \mathbf{C}$ **then**
5: compute the interval $\mathcal{I}^i$ when T_i may complete first
6: t_{L} := lower limit of $\mathcal{I}^i$; t_{U} := upper limit of $\mathcal{I}^i$
7: Ω_{L} := solution Prob. 7.3 for t_{L}; Ω_{U} := solution Prob. 7.3 for t_{U}
8: compute $\hat{U}_{\mathsf{L}}(t_i)$ and $\hat{U}_{\mathsf{U}}(t_i)$
9: partition $\mathcal{I}^i$ into subintervals $\mathcal{I}^i_1, \mathcal{I}^i_2, \ldots, \mathcal{I}^i_K$ s.t. Ω_k makes $\hat{U}_k(t_i)$ maximal in $\mathcal{I}^i_k$
10: $\mathbf{A}_i := \mathbf{A} \cup \{T_i\}$
11: **for** $k \leftarrow 1, 2, \ldots, K$ **do**
12: $\Psi_k := \textsc{Lim}(\Omega_k, \mathbf{A}_i, T_i, \mathcal{I}^i_k)$
13: add subtree Ψ_k s.t. $\Omega \xrightarrow{T_i; \mathcal{I}^i_k} \Omega_k$
14: **end for**
15: **end if**
16: **end for**

Algorithm 7.12: $\mathsf{Lim}(\Omega, \mathbf{A}, T_l, \mathcal{I}^l)$

For the example shown in Figure 7.9 (discussed in Subsections 7.3.1 and 7.3.3.2), when partitioning the interval $\mathcal{I}^1 = [2, 10]$ of possible completion times of T_1 (case when T_1 completes after T_2), the heuristic solves Problem 7.3 for $t_{\mathsf{L}} = 2$ and $t_{\mathsf{U}} = 10$. The respective solutions are $\Omega_{\mathsf{L}} = \{T_1T_5T_3, T_2T_4T_7T_6\}$ and $\Omega_{\mathsf{U}} = \{T_1T_3T_5, T_2T_4T_6T_7\}$. Then Lim computes $\hat{U}_{\mathsf{L}}(t_1)$ and $\hat{U}_{\mathsf{U}}(t_1)$ (which correspond, respectively, to $\hat{U}_a(t_1)$ and $\hat{U}_d(t_1)$ in Figure 7.14(b)) and partitions $\mathcal{I}^1$ using only these two functions. In this step, the solution given by Lim is, after T_1: follow Ω_{L} if $2 \leq t_1 \leq 4$; follow Ω_{U} if $4 < t_1 \leq 10$. The reader can note that in this case Lim gives a suboptimal solution (see Figure 7.14(b) and the optimal tree shown in Figure 7.15).

Along with the proposed heuristic we must solve Problem 7.3 (line 7 in Algorithm 7.12), which itself is intractable. We have proposed in Sec-

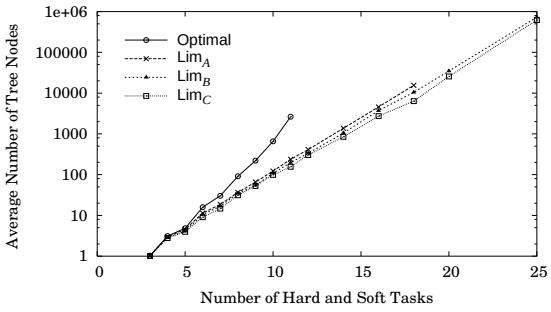
tion 7.2 an exact algorithm and a couple of heuristics for Problem 7.1 that can straightforwardly be applied to Problem 7.3. For the experimental evaluation of Lim we have used the exact algorithm (Subsection 7.2.1.1 for single processor and Subsection 7.2.2.1 for multiple processors) and two heuristics (MTU and MSU in Subsection 7.2.1.2 for single processor, and TU and SU in Subsection 7.2.2.2 for multiple processors) when solving Problem 7.3. Hence we have three heuristics Lim_A, Lim_B, and Lim_C for the multiple-schedules problem. The first uses an optimal algorithm for solving Problem 7.3, the second uses TU (MTU), and the third uses SU (MSU).

Observe that the heuristics presented in this Subsection 7.3.4.1 address only the interval-partitioning step and, in isolation, cannot handle the large complexity of the multiple-schedules problem. These heuristics are to be used in conjunction with the methods discussed in Subsection 7.3.4.2.

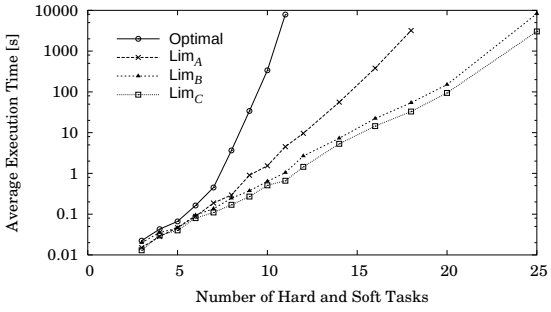
In order to evaluate the quality of the heuristics discussed above, we have generated a large number of synthetic examples. In the case of a single processor we considered systems with 50 tasks among which from 3 up to 25 hard and soft tasks. We generated 100 graphs for each graph dimension. The results are shown in Figure 7.16. In the case of multiple processors we considered that, out of the n tasks of the system, $(n-2)/2$ are soft and $(n-2)/2$ are hard. The tasks are mapped on architectures consisting of between 2 and 4 processors. We also generated 100 synthetic systems for each graph dimension. The results are shown in Figure 7.17. Observe that for a single processor the plots are a function of the number of hard and soft tasks (the total number of tasks is constantly 50) whereas for multiple processors the plots are a function of the total number of tasks. In the former case we can evaluate larger systems because the algorithms tailored for monoprocessor systems are more efficient (for example, the optimal static schedule can be obtained in $\mathcal{O}(|\mathbf{S}|!)$ time for the monoprocessor case and in $\mathcal{O}(|\mathbf{T}|!)$ time for the multiprocessor case).

Figures 7.16(a) and 7.17(a) show the average size of the tree of schedules, for the optimal algorithm as well as for the heuristics. Note the exponential growth even in the heuristic cases which is inherent to the problem of computing a tree of schedules.

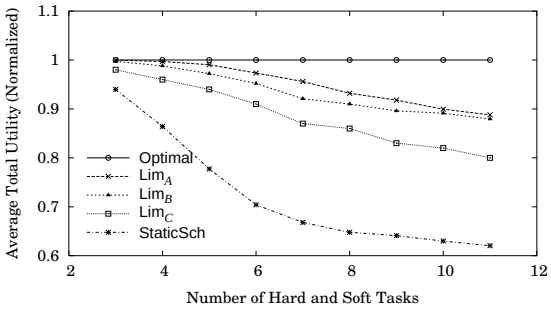
The average execution time of the algorithms is shown in Figures 7.16(b) and 7.17(b). The rapid growth rate of execution time for the optimal algorithm makes it feasible to obtain the optimal tree only in the case of small systems. Observe also that Lim_A takes much longer time than Lim_B and Lim_C, even though they all yield trees with a similar number of nodes. This is due to the fact that, along the construction of the tree, Lim_A solves Problem 7.3 (which is itself intractable) using an exact algorithm while Lim_B and Lim_C make use of polynomial-time heuristics for solving Problem 7.3 during



(a) Tree size



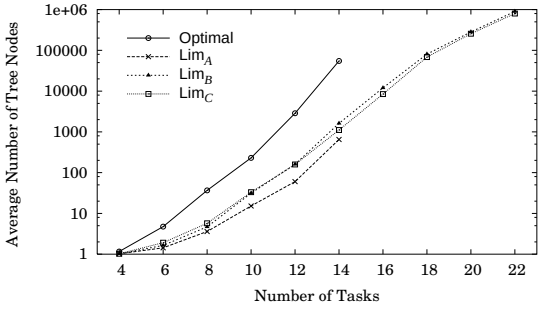
(b) Execution time



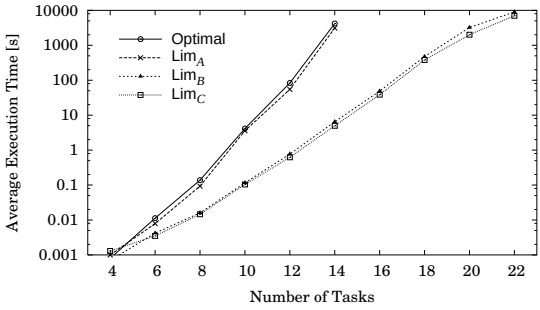
(c) Normalized total utility

Figure 7.16: Evaluation of algorithms for computing a tree of schedules (single processor)

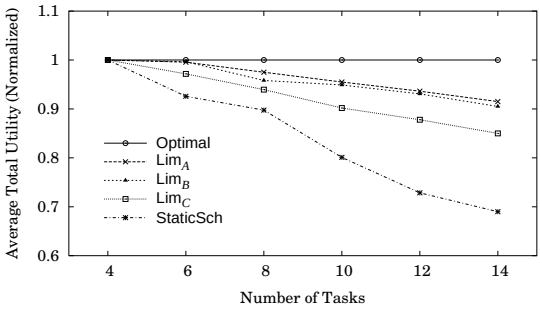
the interval-partitioning step. However, due to the exponential growth of the tree size, even Lim_B and Lim_C require exponential time. It is interesting to note that, for multiprocessor systems (Figure 7.17(b)), the execution times of Lim_A are only slightly smaller than the ones for the optimal algorithm, which is not the case for monoprocessor systems (Figure 7.16(b))



(a) Tree size



(b) Execution time



(c) Normalized total utility

Figure 7.17: Evaluation of algorithms for computing a tree of schedules (multiple processors)

where a significant difference in execution time between Lim_A and the optimal algorithm is noted. This is explained by the fact that, for multiprocessor systems, the interval-partitioning step in the optimal algorithm takes $\mathcal{O}(|\mathbf{T}|!)$ time and the heuristic Lim_A, in the interval-partitioning step, solves exactly a problem that requires also $\mathcal{O}(|\mathbf{T}|!)$ time. On the other hand, for

monoprocessor systems, the interval-partitioning step in the optimal algorithm requires $\mathcal{O}((|\mathbf{H}| + |\mathbf{S}|)!)$ time while the problem solved exactly by the heuristic Lim_A, during the interval-partitioning step takes $\mathcal{O}(|\mathbf{S}|!)$ time.

We evaluated the quality of the trees generated by the different algorithms with respect to the optimal tree. For each one of the randomly generated examples, we profiled the system for a large number of cases. For each case, we obtained the total utility yielded by a given tree of schedules and normalized it with respect to the one produced by the optimal tree: $\|U_{alg}\| = U_{alg}/U_{opt}$. The average normalized utility, as given by trees computed using different algorithms, is shown in Figures 7.16(c) and 7.17(c). We have also plotted the case of a static solution where only one schedule is used regardless of the actual execution times ($\mathsf{StaticSch}$), which is the optimal solution for the static scheduling problem. The plots show Lim_A as the best of the heuristics discussed above, in terms of the total utility yielded by the trees it produces. Lim_B produces still good results, not very far from the optimum, at a significantly lower computational cost. Observe that having one single static schedule leads to a considerable quality loss, even if the static solution is optimal (in the sense as being the best static solution) while the quasi-static is suboptimal (produced by a heuristic).

7.3.4.2 Tree Size Restriction

Even if we could afford to fully compute the optimal tree of schedules (which is not the case for large examples due to the time and memory constraints at design-time), the tree might be too large to fit in the available memory of the target system. Hence we must drop some nodes of the tree at the expense of the solution quality. The heuristics presented in Subsection 7.3.4.1 reduce considerably both the time and memory needed to construct a tree as compared to the optimal algorithm, but still require exponential time and memory. In this subsection, on top of the above heuristics, we propose methods that construct a tree considering its size limit (imposed by the designer) in such a way that we can handle both the time and memory complexity.

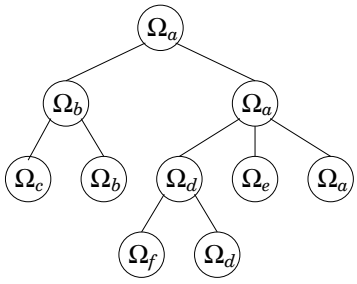
Given a limit for the tree size, only a certain number of schedules can be generated. Thus the question is how to generate a tree of at most M nodes which still delivers a good quality. We explore several strategies which fall under the umbrella of the generic algorithm Restr (Algorithm 7.13). The schedules $\Omega_1, \Omega_2, \ldots, \Omega_K$ to follow after Ω correspond to those obtained in the interval-partitioning step as described in Subsections 7.3.3 and 7.3.4.1. The difference among the approaches discussed in this subsection lies in the order in which the available memory budget is assigned to trees derived from the nodes Ω_k (line 7 in Algorithm 7.13): $\mathsf{Sort}(\Omega_1, \Omega_2, \ldots, \Omega_K)$ gives this order according to different criteria.

input: A schedule Ω and a positive integer M
output: A tree Ψ limited to M nodes whose root is Ω

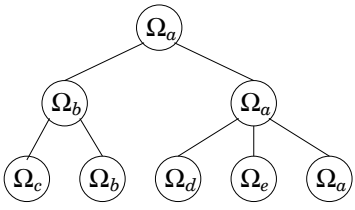
```
1: set Ω as root of Ψ
2: m := M − 1
3: find the schedules Ω_1, Ω_2, ..., Ω_K to follow after Ω (interval-partitioning
   step)
4: if 1 < K ≤ m then
5:    add Ω_1, Ω_2, ..., Ω_K as children of Ω
6:    m := m − K
7:    Sort(Ω_1, Ω_2, ..., Ω_K)
8:    for k ← 1, 2, ..., K do
9:       Ψ_k :=Restr(Ω_k, m + 1)
10:      n_k := size of Ψ_k
11:      m := m − n_k + 1
12:   end for
13: end if
```

Algorithm 7.13: Restr(Ω, M)

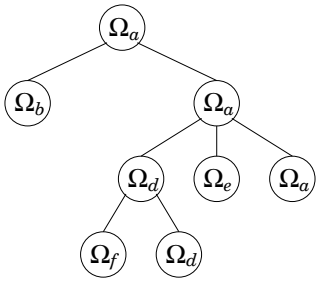
Initially we have studied two simple heuristics for constructing a tree, given a maximum size M. The first one, called Diff, gives priority to subtrees derived from nodes whose schedules differ from their parents. We use a similarity metric, based on the concept of Hamming distance [Lee58], in order to determine how similar two schedules are. For instance, while constructing a tree with a size limit $M = 8$ for the system whose optimal tree is the one given in Figure 7.18(a), we find out that, after the initial schedule Ω_a (the root of the tree), either Ω_b must be followed or the same schedule Ω_a continues as the execution order for the remaining tasks, depending on the completion time of a certain task. Therefore we add Ω_b and Ω_a to the tree. Then, when using Diff, the size budget is assigned first to the subtrees derived from Ω_b (which, as opposed to Ω_a, differs from its parent) and the process continues until we obtain the tree shown in Figure 7.18(b). The second approach, Eq, gives priority to nodes that are equal or more similar to their parents. The tree obtained when using Eq and having a size limit $M = 8$ is shown in Figure 7.18(c). Experimental data (see Figures 7.19(a) and 7.20(a)) shows that in average Eq outperforms Diff. The rationale of the superiority of Eq is that, since no change has yet been operated on the previous schedule, it is likely that several possible alternatives are potentially detected in the future. Hence, it pays to explore the possible changes of schedules derived from such branches. On the contrary, if a different schedule has been detected, it can be assumed that this one is relatively well adapted to the new situation and possible future changes are not leading to dramatic improvements.



(a) Complete tree



(b) Using Diff (max. size $M = 8$)



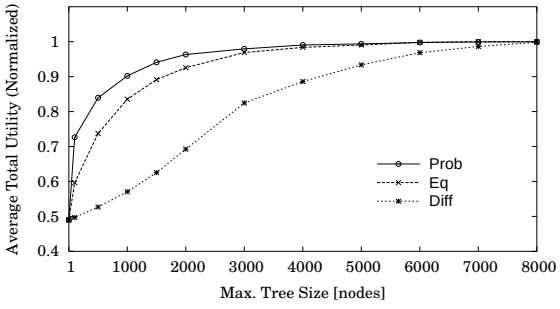
(c) Using Eq (max. size $M = 8$)

Figure 7.18: Trees of schedules

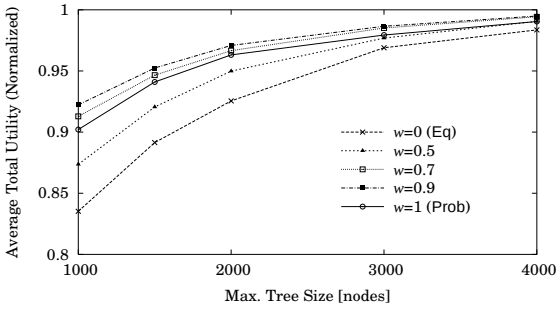
A third, more elaborate, approach brings into the the picture the probability that a certain branch of the tree of schedules is selected during run-time. Knowing the execution time probability distribution of each individual task, we may determine, for a particular execution order, the probability that a certain task completes in a given interval, in particular the intervals defined by the switching points. In this way we can compute the probability for each branch of the tree and exploit this information when constructing the tree of schedules. The procedure Prob gives higher precedence (in terms of size budget) to those subtrees derived from nodes that actually have higher probability of being followed at run-time.

We evaluated the proposed approaches, both for single and multiple processors, by randomly generating 100 systems with a fixed number of tasks and computing for each one of them the complete tree of schedules. Then we constructed the trees for the same systems using the algorithms presented in this subsection, for different size limits. For the experimental evaluation in this section we considered small graphs in order to cope with complete trees: note, for example, that the complete trees for multiprocessor systems with 16 tasks have, in average, around 10.000 nodes when using Lim_B. For each of the examples we profiled the system for a large number of execution times,

and for each of these we obtained the total utility yielded by a restricted tree and normalized it with respect to the utility given by the complete tree (non-restricted): $\|U_{restr}\| = U_{restr}/U_{non-restr}$. Figures 7.19(a) and 7.20(a) show that PROB is the algorithm that gives the best results in average.



(a) Diff, Eq, and Prob

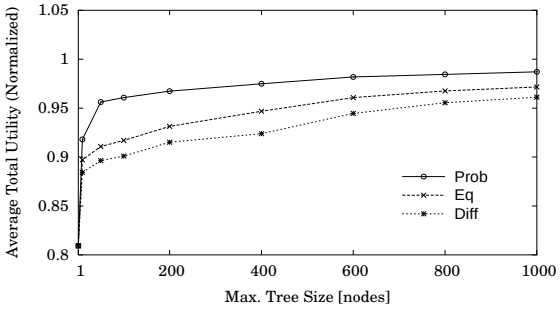


(b) Weighted approach Prob/Eq

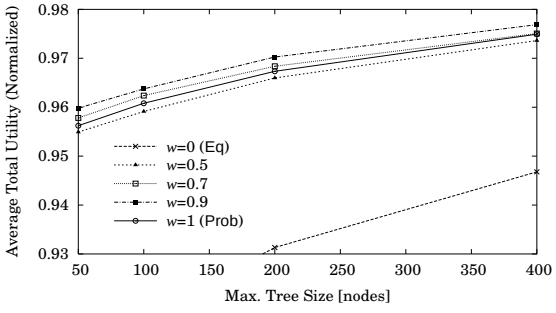
Figure 7.19: Evaluation of the tree size restriction algorithms (single processor)

We investigated further the combination of Prob and Eq through a weighted function that assigns values to the tree nodes. Such values correspond to the priority given to nodes while constructing the tree. Each child of a certain node in the tree is assigned a value given by $wp + (1-w)s$, where p is the probability of that node (schedule) being selected among its siblings and s is a factor that captures how similar that node and its parent are. The particular cases $w = 0$ and $w = 1$ correspond to Eq and Prob respectively. The results of the weighted approach for different values of w are illustrated in Figures 7.19(b) and 7.20(b). It is interesting to note that we can get even better results than Prob for certain weights, with $w = 0.9$ being the one that performs the best. For example, in the case of multiple processors (Figure 7.20(b)), trees limited to 200 nodes (2% of the average

size of the complete tree) yield a total utility that is just 3% off from the one produced by the complete tree. Thus, good quality results and short execution times show that the proposed techniques can be applied to larger systems.



(a) Diff, Eq, and Prob



(b) Weighted approach Prob/Eq

Figure 7.20: Evaluation of the tree size restriction algorithms (multiple processors)

7.3.5 Realistic Application: Cruise Control with Collision Avoidance

Modern vehicles can be equipped with sophisticated electronic aids aiming at assisting the driver, increasing efficiency, and enhancing on-board comfort. One such system is the Cruise Control with Collision Avoidance (CCCA) [GBdSMH01] which assists the driver in maintaining the speed and keeping safe distances to other vehicles. The CCCA allows the driver to set a particular speed. The system maintains that speed until the driver changes the reference speed, presses the break pedal, switches the system off, or the vehicle gets too close to another vehicle or an obstacle. The vehicle may travel

faster than the set speed by overriding the control using the accelerator, but once it is released the cruise control will stabilize the speed to the set level. When another vehicle is detected in the same lane in front of the car, the CCCA will adjust the speed by applying limited braking to maintain a given distance to the vehicle ahead.

The CCCA is composed of four main subsystems, namely Braking Control (BC), Engine Control (EC), Collision Avoidance (CA), and Display Control (DC), each one of them having its own period: $\mathsf{T}_{BC} = 100$ ms, $\mathsf{T}_{EC} = 250$ ms, $\mathsf{T}_{CA} = 125$ ms, and $\mathsf{T}_{DC} = 500$ ms. We have modeled each subsystem as a task graph. Each subsystem has one hard deadline that equals its period. We identified a number of soft tasks in the EC and DC subsystems. The soft tasks in the engine control part are related to the adjustment of the throttle valve for improving the fuel consumption efficiency. Thus their utility functions capture how this efficiency varies as a function of the completion time of the activities that calculate the best fuel injection rate for the actual conditions and, accordingly, control the throttle. For the display control part, the utility of soft tasks is a measure of the time-accuracy of the displayed data, that is, how soon the information on the dashboard is updated.

We have considered an architecture with two processors that communicate through a bus, and assumed that the dedicated memory for storing the schedules has a capacity of 16 kB. We generated several instances of the task graphs of the four subsystems mentioned above in order to construct a graph with a hyperperiod $\mathsf{T} = 500$ ms. The resulting graph, including processing as well as communication activities, contains 126 tasks, out of which 6 are soft and 12 are hard.

Assuming that we need 25 Bytes for storing one schedule, we have an upper limit of 640 nodes in the tree. We have constructed the tree of schedules using the approaches discussed in Subsection 7.3.4.2 combined with one of the heuristics presented in Subsection 7.3.4.1 (Lim_B).

Due to the size of the system, it is infeasible to fully construct the complete tree of schedules. Therefore, we have instead compared the tree limited to 640 nodes with the static, off-line solution of a single schedule. The results are presented in Table 7.2. For the CCCA example, we can achieve with our quasi-static approach a gain of above 40% as compared to a single static schedule. For this example, the weighted approach does not produce further improvements, which is explained by the fact that Eq and Prob give very similar results.

The construction of the tree of schedules, as explained above, for the example discussed in this subsection takes around 70 minutes. Although this time is considerable, the tree is computed only once and this is done

	Average Total Utility	Gain with respect to SingleSch
SingleSch	6.51	—
Diff	7.51	11.42%
Eq	9.54	41.54%
Prob	9.6	42.43%

Table 7.2: Quality of different approaches for the CCCA

off-line.

The CCCA example is a realistic system that illustrates the advantages of exploiting the variations in actual execution times. Our quasi-static solution is able to exploit, with very on-line overhead, this dynamic slack to improve the quality of results (the total utility). By using the heuristics proposed in this chapter, in order to prepare a number of schedules and switching points, important improvements in the quality of results can be achieved.

Chapter 8

Imprecise-Computation Systems with Energy Considerations

In Chapter 7 we studied real-time systems that include both hard and soft tasks and for which the quality of results, expressed in terms of utilities, depends on the completion time of soft tasks.

In this chapter we address real-time systems for which the soft component comes from the fact that tasks have optional parts. In this case, the quality of results, in the form of rewards, depends on the amount of computation alloted to tasks. Also, in contrast to Chapter 7, the dimension of energy consumption is taken into account in this chapter.

There exist some application areas, such as image processing and multimedia, in which approximate but timely results are acceptable. For example, fuzzy images in time are often preferable to perfect images too late. In these cases it is thus possible to trade off precision for timeliness.

Also, power and energy consumption have become very important design considerations for embedded computer systems, in particular for battery-powered devices with stringent energy constraints. The availability of vast computational capabilities at low cost has promoted the use of embedded systems in a wide variety of application areas where power and energy consumption play an important role.

An effective way to reduce the energy consumption in CMOS circuits is to decrease the supply voltage, which however implies a lower operational frequency. The trade-off between energy consumption and performance has extensively been studied under the framework of Dynamic Voltage Scaling (DVS), as pointed out in Section 6.2.

In this chapter we focus on real-time systems for which it is possible to

trade off precision for timeliness and with energy consumption considerations. We study such systems under the Imprecise Computation (IC) model [SLC89], [LSL+94], where tasks are composed of mandatory and optional parts and there are functions that assign reward to tasks depending on the amount of computation allotted to their optional parts.

We discuss in this chapter two different approaches in which energy, reward, and deadlines are considered under a unified framework: the first is maximizing rewards subject to energy constraints (Section 8.2) and the second one is minimizing the energy consumption subject to reward constraints (Section 8.3). In both cases time constraints in the form of deadlines are considered. The goal is to find the voltages at which each task runs and the number of optional cycles, such that the objective function is optimized and the constraints are satisfied. The two approaches introduced in this chapter exploit the dynamic slack, which is caused by tasks executing less number of cycles than their worst case.

In this chapter, *Static V/O assignment* refers to finding at design-time one Voltage/Optional-cycles (V/O) assignment. *Dynamic V/O assignment* refers to finding at run-time, every time a task completes, a new assignment of voltages and optional cycles for those tasks not yet started, but considering the actual execution times by tasks already completed. In a reasoning similar to the one discussed in the introduction of Chapter 7 but applied to the approaches addressed in this chapter, static V/O assignment causes no on-line overhead but is rather pessimistic because actual execution times are typically far off from worst-case values. Dynamic V/O assignment exploits information known only after tasks complete and accordingly computes new assignments, but the energy and time overhead for on-line computations is high, even if polynomial-time algorithms can be used. We propose a *quasi-static* approach that is able to exploit, with low on-line overhead, the dynamic slack: first, at design-time a set of V/O assignments are computed and stored (off-line phase); second, the selection among the precomputed assignments is left for run-time (on-line phase).

8.1 Preliminaries

8.1.1 Task and Architectural Models

In this chapter we consider that the functionality of the system is captured by a directed acyclic graph $G = (\mathbf{T}, \mathbf{E})$ where the nodes $\mathbf{T} = \{T_1, T_2, \ldots, T_n\}$ correspond to the computational tasks and the edges $\mathbf{E}$ indicate the data dependencies between tasks. For the sake of convenience in the notation, we assume that tasks are named according to a particular execution order

(as explained later in this subsection) that respects the data dependencies. That is, task T_{i+1} executes immediately after T_i, $1 \leq i < n$.

Each task T_i consists of a mandatory part and an optional part, characterized in terms of the number of CPU cycles M_i and O_i respectively. The actual number of mandatory cycles M_i of T_i at a certain activation of the system is unknown beforehand but lies in the interval bounded by the best-case number of cycles M_i^{bc} and the worst-case number of cycles M_i^{wc}, that is, $M_i^{\text{bc}} \leq M_i \leq M_i^{\text{wc}}$. The expected number of mandatory cycles of a task T_i is denoted M_i^{e}. The optional part of a task executes immediately after its corresponding mandatory part completes. For each T_i, there is a deadline d_i by which both mandatory and optional parts of T_i must be completed.

For each task T_i, there is a reward function $R_i(O_i)$ that takes as argument the number of optional cycles O_i assigned to T_i; we assume that $R_i(0) = 0$. We consider non-decreasing concave[1] reward functions as they capture the particularities of most real-life applications [RMM03]. Also, as detailed in Subsection 8.2.2, the concavity of reward functions is exploited for obtaining solutions to particular optimization problems in polynomial time. We assume also there is a value $O_i^{\max}$, for each T_i, after which no extra reward is achieved, that is, $R_i(O_i) = R_i^{\max}$ if $O_i \geq O_i^{\max}$. The total reward is denoted $R = \sum_{T_i \in \mathbf{T}} R_i(O_i)$ (sum of individual reward contributions). The reward produced up to the completion of task T_i is denoted RP_i ($RP_i = \sum_{j=1}^{i} R_j(O_j)$). In Section 8.3 we consider a reward constraint, denoted $R^{\min}$, that gives the lower bound of the total reward that must be produced by the system.

We consider that tasks are non-preemptable and have equal release time ($r_i = 0$, $1 \leq i \leq n$). All tasks are mapped onto a single processor and executed in a fixed order, determined off-line according to an EDF (Earliest Deadline First) policy. For non-preemptable tasks with equal release time and running on a single processor, EDF gives the optimal execution order (see Section B.5). T_i denotes the i-th task in this sequence.

The target processor supports voltage scaling and we assume that the voltage levels can be varied in a continuous way in the interval $[V^{\min}, V^{\max}]$. If only a discrete set of voltages are supported by the processor, our approach can easily be adapted by using well-known techniques for determining the discrete voltage levels that replace the calculated continuous one [OYI01].

In our quasi-static approach we compute a number of V/O (Voltage/Optional-cycles) assignments. The set of precomputed V/O assignments is stored in a dedicated memory in the form of lookup tables, one table LUT_i for each task T_i. The maximum number of V/O assignments that can be stored in memory is a parameter fixed by the designer and is denoted $N^{\max}$.

[1] A function $f(x)$ is concave iff $f''(x) \leq 0$, that is, the second derivative is negative.

8.1.2 Energy and Delay Models

The power consumption in CMOS circuits is the sum of dynamic, static (leakage), and short-circuit power. The short-circuit component is negligible. The dynamic power is at the moment the dominating component but the leakage power is becoming an important factor in the overall power dissipation. For the sake of simplicity and clarity in the presentation of our ideas, we consider only the dynamic energy consumption. Nonetheless, the leakage energy and Adaptive Body Biasing (ABB) techniques [ASE+04], [MFMB02] can easily be incorporated into the formulation without changing our general approach. The amount of dynamic energy consumed by task T_i is given by the following expression [MFMB02]:

$$E_i = C_i V_i^2 (M_i + O_i) \tag{8.1}$$

where C_i is the effective switched capacitance, V_i is the supply voltage, and $M_i + O_i$ is the total number of cycles executed by the task. The energy overhead caused by switching from V_i to V_j is as follows [MFMB02]:

$$\mathcal{E}_{i,j}^{\Delta V} = C_r (V_i - V_j)^2 \tag{8.2}$$

where C_r is the capacitance of the power rail. We also consider, for the quasi-static solution, the energy overhead $\mathcal{E}_i^{sel}$ originated by looking up and selecting one of the precomputed V/O assignments. The way we store the precomputed assignments makes the lookup and selection process take $\mathcal{O}(1)$ time. Therefore $\mathcal{E}_i^{sel}$ is a constant value. Also, this value is the same for all tasks ($\mathcal{E}_i^{sel} = \mathcal{E}^{sel}$, for $1 \leq i \leq n$). For consistency reasons we keep the index i in the notation of the selection overhead $\mathcal{E}_i^{sel}$. The energy overhead caused by on-line operations is denoted $\mathcal{E}_i^{dyn}$. In a quasi-static solution the on-line overhead is just the selection overhead ($\mathcal{E}_i^{dyn} = \mathcal{E}_i^{sel}$).

The total energy consumed up to the completion of task T_i (including the energy by the tasks themselves as well as overheads) is denoted EC_i. In Section 8.2 we consider a given energy budget, denoted $E^{\max}$, that imposes a constraint on the total amount of energy that can be consumed by the system.

The execution time of a task T_i executing $M_i + O_i$ cycles at supply voltage V_i is [MFMB02]:

$$\tau_i = k \frac{V_i}{(V_i - V_{th})^\alpha} (M_i + O_i) \tag{8.3}$$

where k is a constant dependent on the process technology, α is the saturation velocity index (also technology dependent, typically $1.4 \leq \alpha \leq 2$), and V_{th} is the threshold voltage. The time overhead, when switching from V_i to V_j, is given by the following expression [ASE+04]:

$$\delta_{i,j}^{\Delta V} = p|V_i - V_j| \tag{8.4}$$

where p is a constant. The time overhead for looking up and selecting

one V/O assignment in the quasi-static approach is denoted δ_i^{sel} and, as explained above, is constant and is the same value for all tasks.

The starting and completion times of a task T_i are denoted s_i and t_i respectively, with $s_i + \delta_i + \tau_i = t_i$ where δ_i captures the different time overheads. $\delta_i = \delta_{i-1,i}^{\Delta V} + \delta_i^{dyn}$ where δ_i^{dyn} is the on-line overhead. Note that in a quasi-static solution this on-line overhead is just the lookup and selection time, that is, $\delta_i^{dyn} = \delta_i^{sel}$.

8.1.3 Mathematical Programming

Mathematical programming is the generic term used to describe methods for solving problems in which an optimal value is sought subject to specified constraints [Vav91]. The general form of a mathematical programming problem is "find the values $x_1, \ldots, x_n$ that minimize the objective function $f(x_1, \ldots, x_n)$ subject to the constraints $g_i(x_1, \ldots, x_n) \leq b_i$ and $l_j \leq x_j \leq u_j$". An optimization problem with linear objective function f as well as linear constraint functions g_i is called *linear programming* (LP) problem. If at least one g_i or f is non-linear, it is called *non-linear programming* (NLP) problem.

A function $f(x_1, \ldots, x_n)$ is *convex* if its Hessian (second derivative matrix) is positive, that is, $\nabla^2 f \geq \mathbf{0}$. When f and g_i are convex functions, the problem is said to be convex. It should be mentioned that LP and convex NLP problems can be solved using polynomial-time methods [NN94] and tools for solving these types of problems are available (for instance, MOSEK [MOS]).

8.2 Maximizing Rewards subject to Energy Constraints

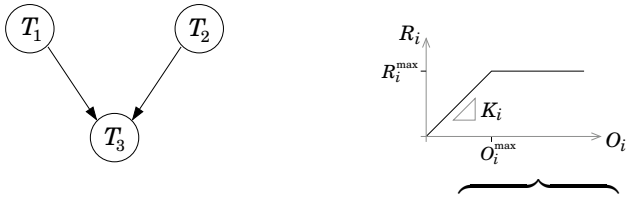
In this section we address the problem of maximizing rewards for real-time systems with energy constraints, in the frame of the Imprecise Computation model.

We present first an example that illustrates the advantages of exploiting the dynamic slack caused by variations in the actual number of execution cycles.

8.2.1 Motivational Example

Let us consider the motivational example shown in Figure 8.1. The non-decreasing reward functions are of the form $R_i(O_i) = K_i O_i$, $O_i \leq O_i^{\max}$. The energy constraint is $E^{\max} = 1$ mJ and the tasks run, according to a

schedule fixed off-line in conformity to an EDF policy, on a processor that permits continuous voltage scaling in the range 0.6-1.8 V. For clarity reasons, in this example we assume that transition overheads are zero.



Task	M_i^{bc}	M_i^{wc}	C_i [nF]	d_i [μs]	K_i	$R_i^{\max}$
T_1	20000	100000	0.7	250	0.00014	7
T_2	70000	160000	1.2	600	0.0002	16
T_3	100000	180000	0.9	1000	0.0001	6

Figure 8.1: Motivational example

The optimal static V/O assignment for this example is given by Table 8.1. It produces a total reward $R^{st} = 3.99$. The assignment gives, for each task T_i, the voltage V_i at which T_i must run and the number of optional cycles O_i that it must execute in order to obtain the maximum total reward, while guaranteeing that deadlines are met and the energy constraint is satisfied.

Task	V_i [V]	O_i
T_1	1.654	35
T_2	1.450	19925
T_3	1.480	11

Table 8.1: Optimal static V/O assignment

The V/O assignment given by Table 8.1 is optimal in the static sense. It is the best possible that can be obtained off-line without knowing the actual number of cycles executed by each task. However, the actual number of cycles, which are not known in advance, are typically far off from the worst-case values used to compute such a static assignment. This point is illustrated by the following situation. The first task starts executing at $V_1 = 1.654$ V, as required by the static assignment. Assume that T_1 executes $M_1 = 60000$ (instead of $M_1^{\text{wc}} = 100000$) mandatory cycles and then its assigned $O_1 = 35$ optional cycles. At this point, knowing that T_1 has completed at $t_1 = \tau_1 = 111.73$ μs and that the consumed energy is $EC_1 = E_1 = 114.97$ μJ, a new V/O assignment can accordingly be computed for the remaining tasks aiming at obtaining the maximum total reward for the new conditions. We consider, for the moment, the ideal case in which such an on-line computation takes zero time and energy. Observe that, for computing

the new assignments, the worst case for tasks not yet completed has to be assumed as their actual number of executed cycles is not known in advance. The new assignment gives $V_2 = 1.446$ V and $O_2 = 51396$. Then T_2 runs at $V_2 = 1.446$ V and let us assume that it executes $M_2 = 100000$ (instead of $M_2^{\text{wc}} = 160000$) mandatory cycles and then its newly assigned $O_2 = 51396$ optional cycles. At this point, the completion time is $t_2 = \tau_1 + \tau_2 = 461.35$ μs and the energy so far consumed is $EC_2 = E_1 + E_2 = 494.83$ μJ. Again, a new assignment can be computed taking into account the information about completion time and consumed energy. This new assignment gives $V_3 = 1.472$ V and $O_3 = 60000$.

For such a situation, in which $M_1 = 60000$, $M_2 = 100000$, $M_3 = 150000$, the V/O assignment computed dynamically (considering $\delta^{dyn} = 0$ and $\mathcal{E}^{dyn} = 0$) is summarized in Table 8.2(a). This assignment delivers a total reward $R^{dyn^{ideal}} = 16.28$. In reality, however, the on-line overhead caused by computing new assignments is not negligible. When considering time and energy overheads, using for example $\delta^{dyn} = 65$ μs and $\mathcal{E}^{dyn} = 55$ μJ, the V/O assignment computed dynamically is evidently different, as given by Table 8.2(b). This assignment delivers a total reward $R^{dyn^{real}} = 6.26$. The values of δ^{dyn} and $\mathcal{E}^{dyn}$ are in practice several orders of magnitude higher than the ones used in this hypothetical example. For instance, for a system with 50 tasks, computing one such V/O assignment using a commercial solver takes a few seconds. Even on-line heuristics, which produce approximate results, have long execution times [RMM03]. This means that a dynamic V/O scheduler might produce solutions that are actually inferior to the static one (in terms of total reward delivered) or, even worse, a dynamic V/O scheduler might not be able to fulfill the given time and energy constraints.

Task	V_i [V]	O_i
T_1	1.654	35
T_2	1.446	51396
T_3	1.472	60000

(a) $\delta^{dyn} = 0$, $\mathcal{E}^{dyn} = 0$

Task	V_i [V]	O_i
T_1	1.654	35
T_2	1.429	1303
T_3	1.533	60000

(b) $\delta^{dyn} = 65$ μs, $\mathcal{E}^{dyn} = 55$ μJ

Table 8.2: Dynamic V/O assignments (for $M_1 = 60000$, $M_2 = 100000$, $M_3 = 150000$)

In our quasi-static solution we compute at design-time a number of V/O assignments, which are selected at run-time by the so-called quasi-static V/O scheduler (at very low overhead) based on the information about completion time and consumed energy after each task completes.

We can define, for instance, a quasi-static set of assignments for the example discussed in this subsection, as given by Table 8.3. Upon completion of each task, V_i and O_i are selected from the precomputed set of V/O assignments, according to the given condition. The assignments were computed considering the selection overheads $\delta^{sel} = 0.3$ μs and $\mathcal{E}^{sel} = 0.3$ μJ.

Task	Condition	V_i [V]	O_i
T_1	—	1.654	35
T_2	if $t_1 \leq 75$ μs $\wedge$ $EC_1 \leq 77$ μJ	1.444	66924
	else if $t_1 \leq 130$ μs $\wedge$ $EC_1 \leq 135$ μJ	1.446	43446
	else	1.450	19925
T_3	if $t_2 \leq 400$ μs $\wedge$ $EC_2 \leq 430$ μJ	1.380	60000
	else if $t_2 \leq 500$ μs $\wedge$ $EC_2 \leq 550$ μJ	1.486	46473
	else	1.480	11

Table 8.3: Precomputed set of V/O assignments

For the situation $M_1 = 60000$, $M_2 = 100000$, $M_3 = 150000$ and the set given by Table 8.3, the quasi-static V/O scheduler would do as follows. Task T_1 is run at $V_1 = 1.654$ V and is allotted $O_1 = 35$ optional cycles. Since, when completing T_1, $t_1 = \tau_1 = 111.73 \leq 130$ μs and $EC_1 = E_1 = 114.97 \leq 135$ μJ, $V_2 = 1.446/O_2 = 43446$ is selected by the quasi-static V/O scheduler. Task T_2 runs under this assignment so that, when it finishes, $t_2 = \tau_1 + \delta_2^{sel} + \tau_2 = 442.99$ μs and $EC_2 = E_1 + \mathcal{E}_2^{sel} + E_2 = 474.89$ μJ. Then $V_3 = 1.486/O_3 = 46473$ is selected and task T_3 is executed accordingly. Table 8.4 summarizes the selected assignment. The total reward delivered by this V/O assignment is $R^{qs} = 13.34$ (compare to $R^{dyn^{ideal}} = 16.28$, $R^{dyn^{real}} = 6.26$, and $R^{st} = 3.99$). It can be noted that the quasi-static solution *qs* outperforms the dynamic one dyn^{real} because of the large overheads of the latter.

Task	V_i [V]	O_i
T_1	1.654	35
T_2	1.446	43446
T_3	1.486	46473

Table 8.4: Quasi-static V/O assignment (for $M_1 = 60000$, $M_2 = 100000$, $M_3 = 150000$) selected from the precomputed set of Table 8.3

8.2.2 Problem Formulation

We tackle the problem of maximizing the total reward subject to a limited energy budget, in the framework of DVS. In what follows we present the pre-

cise formulation of certain related problems and of the particular problem addressed in this section. Recall that the task execution order is predetermined, with T_i being the i-th task in this sequence.

PROBLEM 8.1 (Static V/O Assignment for Maximizing Reward—Static AMR) Find, for each task T_i, $1 \leq i \leq n$, the voltage V_i and the number of optional cycles O_i that

$$\text{maximize } \sum_{i=1}^{n} R_i(O_i) \tag{8.5}$$

$$\text{subject to } V^{\min} \leq V_i \leq V^{\max} \tag{8.6}$$

$$s_{i+1} = t_i = s_i + \underbrace{p|V_{i-1} - V_i|}_{\delta^{\Delta V}_{i-1,i}} + \underbrace{k \frac{V_i}{(V_i - V_{th})^{\alpha}} (M_i^{\text{wc}} + O_i)}_{\tau_i} \leq d_i \tag{8.7}$$

$$\sum_{i=1}^{n} \Big(\underbrace{C_r (V_{i-1} - V_i)^2}_{\mathcal{E}^{\Delta V}_{i-1,i}} + \underbrace{C_i V_i^2 (M_i^{\text{wc}} + O_i)}_{E_i} \Big) \leq E^{\max} \tag{8.8}$$

The above formulation can be explained as follows. The total reward, as given by Equation (8.5), is to be maximized. For each task the voltage V_i must be in the range $[V^{\min}, V^{\max}]$ (Equation (8.6)). The completion time t_i is the sum of the start time s_i, the voltage-switching time $\delta^{\Delta V}_{i-1,i}$, and the execution time τ_i, and tasks must complete before their deadlines (Equation (8.7)). The total energy is the sum of the voltage-switching energies $\mathcal{E}^{\Delta V}_{i-1,i}$ and the energy E_i consumed by each task, and cannot exceed the available energy budget $E^{\max}$ (Equation (8.8)). Note that a static assignment must consider the worst-case number of mandatory cycles M_i^{wc} for every task (Equations (8.7) and (8.8)).

For tractability reasons, when solving the above problem, we consider O_i as a continuous variable and then we round the result down. By this, without generating the optimal solution, we obtain a solution that is very near to the optimal one because one clock cycle is a very fine-grained unit (tasks execute typically hundreds of thousands of clock cycles). We can rewrite the above problem as "minimize $\sum R'_i(O_i)$", where $R'_i(O_i) = -R_i(O_i)$. It can thus be noted that: $R'_i(O_i)$ is convex since $R_i(O_i)$ is a concave function; the constraint functions are also convex[2]. Therefore it corresponds to a convex NLP formulation (see Subsection 8.1.3) and hence the problem can be solved in polynomial time.

[2]Observe that the function *abs* cannot be used directly in mathematical programming because it is not differentiable in 0. However, there exist techniques for obtaining equivalent formulations [ASE+04].

Dynamic V/O Scheduler (AMR): The following is the problem that a dynamic V/O scheduler must solve every time a task T_c completes. It is considered that tasks $T_1, \ldots, T_c$ have already completed (the total energy consumed up to the completion of T_c is EC_c and the completion time of T_c is t_c).

Problem 8.2 (Dynamic AMR) Find V_i and O_i, for $c+1 \leq i \leq n$, that

$$\text{maximize} \sum_{i=c+1}^{n} R_i(O_i) \tag{8.9}$$

$$\text{subject to } V^{\min} \leq V_i \leq V^{\max} \tag{8.10}$$

$$s_{i+1} = t_i = s_i + \delta_i^{dyn} + \delta_{i-1,i}^{\Delta V} + \tau_i \leq d_i \tag{8.11}$$

$$\sum_{i=c+1}^{n} \left(\mathcal{E}_i^{dyn} + \mathcal{E}_{i-1,i}^{\Delta V} + E_i\right) \leq E^{\max} - EC_c \tag{8.12}$$

where δ_i^{dyn} and $\mathcal{E}_i^{dyn}$ are, respectively, the time and energy overhead of computing dynamically V_i and O_i for task T_i.

Observe that the problem solved by the dynamic V/O scheduler corresponds to an instance of the static V/O assignment problem (Problem 8.1 for $c+1 \leq i \leq n$ and taking into account t_c and EC_c), but considering δ_i^{dyn} and $\mathcal{E}_i^{dyn}$. It is worthwhile to note that even the dynamic V/O scheduler must assume worst-case number of cycles M_i^{wc} for tasks T_i yet to be executed. The corresponding explanation is deferred until Subsection 8.3.1.

The total reward R^{ideal} delivered by a dynamic V/O scheduler in the ideal case $\delta_i^{dyn} = 0$ and $\mathcal{E}_i^{dyn} = 0$ represents an upper bound on the reward that can practically be achieved without knowing in advance how many mandatory cycles tasks will execute and without accepting risks regarding deadline or energy violations.

Although the dynamic V/O assignment problem can be solved in polynomial-time, the time and energy for solving it are in practice very large and therefore unacceptable at run-time for practical applications. In our approach we prepare off-line a number of V/O assignments, one of which is to be selected (at very low on-line cost) by the *quasi-static V/O scheduler*.

Every time a task T_c completes, the quasi-static V/O scheduler checks the completion time t_c and the total energy EC_c, and looks up an assignment in the table for T_c. From the lookup table LUT_c it obtains the point (t'_c, EC'_c), which is the closest to (t_c, EC_c) such that $t_c \leq t'_c$ and $EC_c \leq EC'_c$, and selects V'/O' corresponding to (t'_c, EC'_c), which are the voltage and number of optional cycles for the next task T_{c+1}. Our aim is to obtain a reward, as delivered by the quasi-static V/O scheduler, that is maximal. The problem we discuss in the rest of the section is the following:

PROBLEM 8.3 (Set of V/O Assignments for Maximizing Reward—Set AMR) Find a set of N assignments such that: $N \leq N^{\max}$; the V/O assignment selected by the quasi-static V/O scheduler guarantees the deadlines $(s_i + \delta_i^{sel} + \delta_{i-1,i}^{\Delta V} + \tau_i = t_i \leq d_i)$ and the energy constraint $(\sum_{i=1}^{n} \mathcal{E}_i^{sel} + \mathcal{E}_{i-1,i}^{\Delta V} + E_i \leq E^{\max})$, and yields a total reward R^{qs} that is maximal.

As will be discussed in Subsection 8.2.3, for a task T_i, potentially there exist infinitely many values for t_i and EC_i. Therefore, in order to approach the theoretical limit R^{ideal}, it would be needed to compute an infinite number of V/O assignments, one for each (t_i, EC_i). The problem is thus how to select at most $N^{\max}$ points in this infinite space such that the respective V/O assignments produce a reward as close as possible to R^{ideal}.

8.2.3 Computing the Set of V/O Assignments

For each task T_i, there exists a space time-energy of possible values of completion time t_i and energy EC_i consumed up to the completion of T_i (see Figure 8.2). Every point in this space defines a V/O assignment for the next task T_{i+1}, that is, if T_i completed at t^a and the energy consumed was EC^a, the assignment for the next task would be $V_{i+1} = V^a / O_{i+1} = O^a$. The values V^a and O^a are those that an ideal dynamic V/O scheduler would give for the case $t_i = t^a$, $EC_i = EC^a$ (recall that we aim at matching the reward R^{ideal}). Observe that different points (t_i, EC_i) define different V/O assignments. Note also that for a given value t_i there might be different valid values of EC_i, and this is due to the fact that different previous V/O assignments can lead to the same t_i but still different EC_i.

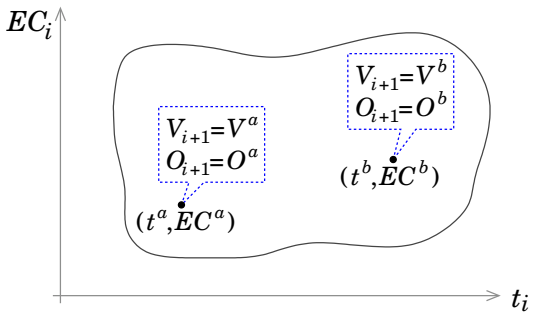


Figure 8.2: Space time-energy

We need first to determine the boundaries of the space time-energy for each task T_i, in order to select N_i points in this space and accordingly compute the set of N_i assignments. N_i is the number of assignments to be

stored in the lookup table LUT_i, after distributing the maximum number $N^{\max}$ of assignments among tasks. A straightforward way to determine these boundaries is to compute the earliest and latest completion times as well as the minimum and maximum consumed energy for task T_i, based on the values $V^{\min}$, $V^{\max}$, M_j^{bc}, M_j^{wc}, and $O_j^{\max}$, $1 \leq j \leq i$. The earliest completion time $t_i^{\min}$ occurs when each of the previous tasks T_j (inclusive T_i) execute their minimum number of cycles M_j^{bc} and zero optional cycles at maximum voltage $V^{\max}$, while $t_i^{\max}$ occurs when each task T_j executes $M_j^{\mathrm{wc}} + O_j^{\max}$ cycles at $V^{\min}$. Similarly, $EC_i^{\min}$ happens when each task T_j executes M_j^{bc} cycles at $V^{\min}$, while $EC_i^{\max}$ happens when each task T_j executes $M_j^{\mathrm{wc}}+O_j^{\max}$ cycles at $V^{\max}$. The intervals $[t_i^{\min}, t_i^{\max}]$ and $[EC_i^{\min}, EC_i^{\max}]$ bound the space time-energy as shown in Figure 8.3. However, there are points in this space that cannot happen. For instance, $(t_i^{\min}, EC_i^{\min})$ is not feasible because $t_i^{\min}$ requires all tasks running at $V^{\max}$ whereas $EC_i^{\min}$ requires all tasks running at $V^{\min}$.

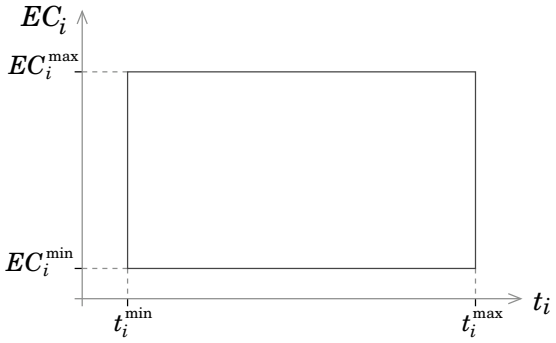


Figure 8.3: Pessimistic boundaries of the space time-energy

8.2.3.1 Characterization of the Space Time-Energy

We take now a closer look at the relation between the energy E_i consumed by a task and its execution time τ_i as given, respectively, by Equations (8.1) and (8.3). In this subsection we consider that the execution time is inversely proportional to the supply voltage ($V_{th} = 0$, $\alpha = 2$), an assumption commonly made in the literature [OYI01]. Observe, however, that we make such an assumption only in order to make the illustration of our point simpler, yet the drawn conclusions are valid in general and do not rely on this assumption. After some simple algebraic manipulations on Equations (8.1) and (8.3) we get the following expression:

$$E_i = \frac{C_i V_i^3}{k} \tau_i \tag{8.13}$$

which, in the space τ_i-E_i, gives a family of straight lines, each for a particular V_i. Thus $E_i = C_i(V^{\min})^3\tau_i/k$ and $E_i = C_i(V^{\max})^3\tau_i/k$ define two boundaries in the space τ_i-E_i. We can also write the following equation:

$$E_i = C_i k^2 (M_i + O_i)^3 \frac{1}{\tau_i^2} \tag{8.14}$$

which gives a family of curves, each for a particular $M_i + O_i$. Thus $E_i = C_i k^2 (M_i^{\text{bc}})^3/\tau_i^2$ and $E_i = C_i k^2 (M_i^{\text{wc}} + O_i^{\max})^3/\tau_i^2$ define other two boundaries, as shown in Figure 8.4. Note that Figure 8.4 represents the energy consumed by *one* task (energy E_i if T_i executes for τ_i time), as opposed to Figure 8.3 that represents the energy by *all tasks up to* T_i (total energy EC_i consumed up to the moment t_i when task T_i finishes).

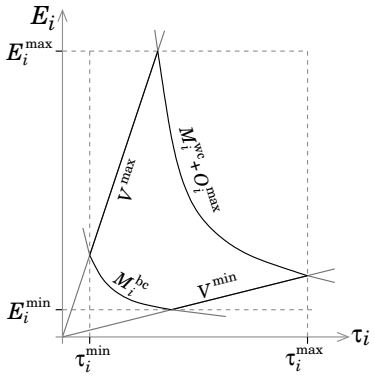


Figure 8.4: Space τ_i-E_i for task T_i

In order to obtain a realistic view of the diagram in Figure 8.3, we must "sum" the spaces τ_j-E_j introduced above. The result of this summation, as illustrated in Figure 8.5, gives the space time-energy t_i-EC_i we are interested in. In Figure 8.5 the space t_2-EC_2 is obtained by sliding the space τ_2-E_2 with its coordinate origin along the boundaries of τ_1-E_1. The "south-east" (SE) and "north-west" (NW) boundaries of the space t_i-EC_i are piecewise linear because the SE and NW boundaries of the individual spaces τ_j-E_j, $1 \leq j \leq i$, are straight lines (see Figure 8.4). Similarly, the NE and SW boundaries of the space t_i-EC_i are piecewise parabolic because the NE and SW of the individual spaces τ_j-E_j are parabolic.

The shape of the space t_i-EC_i is depicted by the solid lines in Figure 8.6. There are, in addition, deadlines d_i to consider as well as energy constraints $E_i^{\max}$. Note that, for each task, the deadline d_i is explicitly given as part of the system model whereas $E_i^{\max}$ is an implicit constraint induced by the overall energy constraint $E^{\max}$. The energy constraint $E_i^{\max}$ imposed upon the completion of task T_i comes from the fact that future tasks will consume at least a certain amount of energy $F_{i+1\to n}$ so that $E_i^{\max} = E^{\max} - F_{i+1\to n}$.

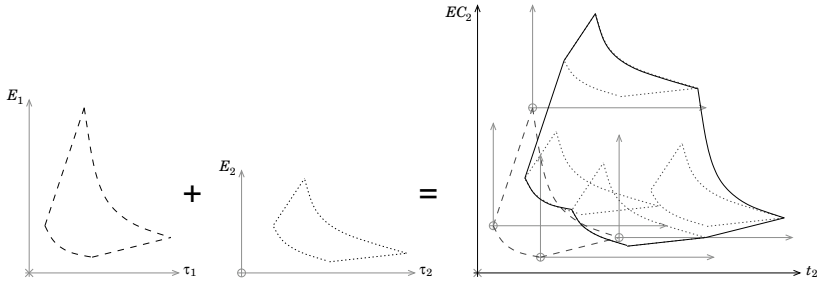


Figure 8.5: Illustration of the "sum" of spaces τ_1-E_1 and τ_2-E_2

The deadline d_i and the induced energy constraint $E_i^{\max}$ further restrict the space time-energy, as depicted by the dashed lines in Figure 8.6.

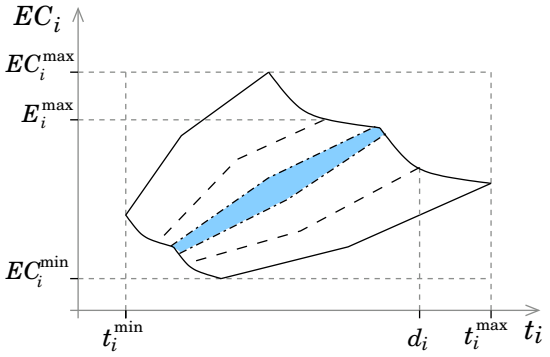


Figure 8.6: Realistic boundaries of the space time-energy

The space time-energy can be narrowed down even further if we take into consideration that we are only interested in points as calculated by the ideal dynamic V/O scheduler. This is explained in the following. Let us consider two different activations of the system. In the first one, after finishing task T_{i-1} at t'_{i-1} with a total consumed energy EC'_{i-1}, task T_i runs under a certain assignment V'_i/O'_i. In the second activation, after T_{i-1} completes at t''_{i-1} with total energy EC''_{i-1}, T_i runs under the assignment V''_i/O''_i. Since the points (t'_{i-1}, EC'_{i-1}) and (t''_{i-1}, EC''_{i-1}) are in general different, the assignments V'_i/O'_i and V''_i/O''_i are also different. The assignment V'_i/O'_i defines in the space t_i-EC_i a segment of straight line L'_i that has slope $C_i(V'_i)^3/k$, with one end point corresponding to the execution of $M_i^{\text{bc}} + O'_i$ cycles at V'_i and the other end corresponding to the execution of $M_i^{\text{wc}} + O'_i$ cycles at V'_i. V''_i/O''_i defines analogously a different segment of straight line L''_i. Solutions to the dynamic V/O assignment problem, though, tend towards letting tasks consume as much as possible of the available energy and finish as late as possible without risking energy or deadline violations: there is no gain by

consuming less energy or finishing earlier than needed as the goal is to maximize the reward. Both solutions V_i'/O_i' and V_i''/O_i'' (that is, the straight lines L_i' and L_i'') will thus converge in the space t_i-EC_i when $M_i' = M_i'' = M_i^{\text{wc}}$ (which is the value that has to be assumed when computing the solutions). Therefore, if T_i under the assignment V_i'/O_i' completes at the same time as under the second assignment V_i''/O_i'' ($t_i' = t_i''$), the respective energy values EC_i' and EC_i'' will actually be very close (see Figure 8.7). This means that in practice the space t_i-EC_i is a narrow area, as depicted by the dash-dot lines and the gray area enclosed by them in Figure 8.6.

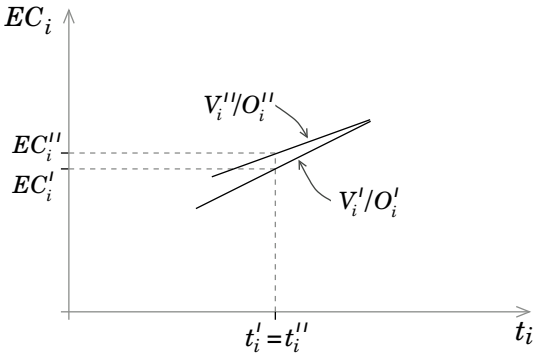


Figure 8.7: V_i'/O_i' and V_i''/O_i'' converge

We have conducted a number of experiments in order to determine how narrow the area of points in the space time-energy actually is. For each task T_i, we consider a segment of straight line, called in the sequel the *diagonal* D_i, defined by the points $(t_i^{\text{s-bc}}, EC_i^{\text{s-bc}})$ and $(t_i^{\text{s-wc}}, EC_i^{\text{s-wc}})$. The point $(t_i^{\text{s-bc}}, EC_i^{\text{s-bc}})$ corresponds to the solution given by the ideal dynamic V/O scheduler in the particular case when every task T_j, $1 \leq j \leq i$, executes its best-case number of mandatory cycles M_j^{bc}. Analogously, $(t_i^{\text{s-wc}}, EC_i^{\text{s-wc}})$ corresponds to the solution in the particular case when every task T_j executes its worst-case number of mandatory cycles M_j^{wc}. We have generated 50 synthetic examples, consisting of between 10 and 100 tasks, and simulated for each of them the ideal dynamic V/O scheduler for 1000 cases, each case S being a combination of executed mandatory cycles $M_1^S, M_2^S, \ldots, M_n^S$. For each task T_i of the different benchmarks and for each set S of mandatory cycles we obtained the actual point (t_i^S, EC_i^S) in the space t_i-EC_i, as given by the ideal dynamic V/O scheduler. Each point (t_i^S, EC_i^S) was compared with the point $(t_i^S, EC_i^{D_i})$ (a point with the same abscissa t_i^S, but on the diagonal D_i so that its ordinate is $EC_i^{D_i}$) and the relative deviation $e = |EC_i^S - EC_i^{D_i}|/EC_i^S$ was computed. We found through our experiments average deviations of around 1% and a maximum deviation of 4.5%. These results

show that the space t_i-EC_i is indeed a narrow area. For example, Figure 8.8 shows the actual points (t_i^S, EC_i^S), corresponding to the simulation of the 1000 sets S of executed mandatory cycles, in the space time-energy of a particular task T_i as well as the diagonal D_i.

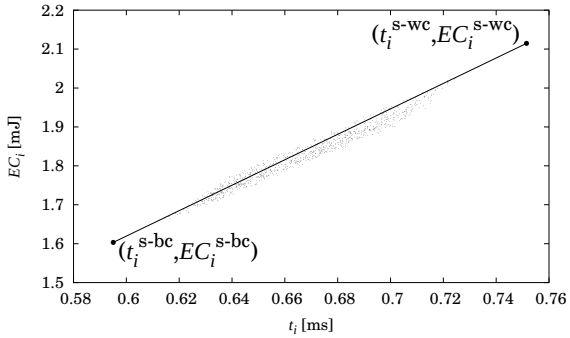


Figure 8.8: Actual points in the space time-energy

8.2.3.2 Selection of Points and Computation of Assignments

From the discussion in Subsection 8.2.3.1 we can draw the conclusion that the points in the space t_i-EC_i are concentrated in a relatively narrow area along a diagonal D_i. This observation is crucial for selecting the points for which we compute at design-time the V/O assignments.

We take points (t_i^j, EC_i^j) along the diagonal D_i in the space t_i-EC_i of task T_i, and then we compute and store the respective assignments V_{i+1}^j/O_{i+1}^j that maximize the total reward when T_i completes at t_i^j and the total energy is EC_i^j. It must be noted that for the computation of the assignment V_{i+1}^j/O_{i+1}^j, the time and energy overheads δ_{i+1}^{sel} and $\mathcal{E}_{i+1}^{sel}$ (needed for selecting assignments at run-time) are taken into account.

Each one of the points, together with its corresponding V/O assignment, covers a region as indicated in Figure 8.9. The quasi-static V/O scheduler selects one of the stored assignments based on the actual completion time and consumed energy. Referring to Figure 8.9, for example, if task T_i completes at t' and the consumed energy is EC', the quasi-static V/O scheduler will select the precomputed V/O assignment corresponding to (t^c, EC^c).

The pseudocode of the procedure we use for computing the set of V/O assignments is given by Algorithm 8.1. First, the maximum number $N^{\max}$ of assignments that are to be stored is distributed among tasks (line 1). A straightforward approach is to distribute them uniformly among the different tasks, so that each lookup table contains the same number of assignments.

Condition	V_{i+1}	O_{i+1}
if $t_i \leq t^a \wedge EC_i \leq EC^a$	V^a	O^a
else if $t_i \leq t^b \wedge EC_i \leq EC^b$	V^b	O^b
else if $t_i \leq t^c \wedge EC_i \leq EC^c$	V^c	O^c
else	V^d	O^d

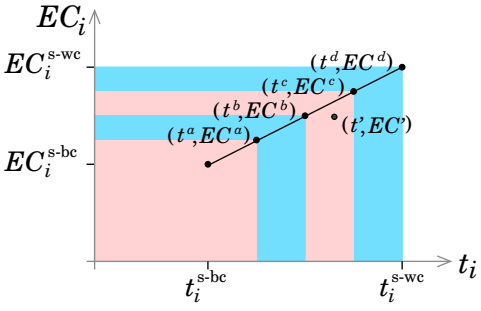


Figure 8.9: Regions

However, as shown by the experimental evaluation of Subsection 8.2.4, it is more efficient to distribute the assignments according to the size of the space time-energy of tasks (we use the length of the diagonal D as a measure of this size), in such a way that lookup tables of tasks with larger spaces get more points.

After distributing $N^{\max}$ among tasks, the solutions $\text{V/O}^{\text{s-bc}}$ and $\text{V/O}^{\text{s-wc}}$ are computed (lines 2-3). $\text{V/O}^{\text{s-bc}}$ ($\text{V/O}^{\text{s-wc}}$) is a structure that contains the pairs $V_i^{\text{s-bc}}/O_i^{\text{s-bc}}$ ($V_i^{\text{s-wc}}/O_i^{\text{s-wc}}$), $1 \leq i \leq n$, as computed by the dynamic V/O scheduler when every task executes its best-case (worst-case) number of cycles. Since the assignment V_1/O_1 is invariably the same, this is the only one stored for the first task (line 5). For every task T_i, $1 \leq i \leq n-1$, we compute: a) $t_i^{\text{s-bc}}$ ($t_i^{\text{s-wc}}$) as the sum of execution times $\tau_k^{\text{s-bc}}$ ($\tau_k^{\text{s-wc}}$)—given by Equation (8.3) with $V_k^{\text{s-bc}}$, M_k^{bc}, and $O_k^{\text{s-bc}}$ ($V_k^{\text{s-wc}}$, M_k^{wc}, and $O_k^{\text{s-wc}}$)—and time overheads δ_k (line 7); b) $EC_i^{\text{s-bc}}$ ($EC_i^{\text{s-wc}}$) as the sum of energies $E_k^{\text{s-bc}}$ ($E_k^{\text{s-wc}}$)—given by Equation (8.1) with $V_k^{\text{s-bc}}$, M_k^{bc}, and $O_k^{\text{s-bc}}$ ($V_k^{\text{s-wc}}$, M_k^{wc}, and $O_k^{\text{s-wc}}$)—and energy overheads $\mathcal{E}_k$ (line 8). For every task T_i, we take N_i equally-spaced points (t_i^j, EC_i^j) along the diagonal D_i (straight line segment from $(t_i^{\text{s-bc}}, EC_i^{\text{s-bc}})$ to $(t_i^{\text{s-wc}}, EC_i^{\text{s-wc}})$) and, for each such point, we compute the respective assignment V_{i+1}^j/O_{i+1}^j and store it accordingly in the j-th position in the particular lookup table LUT_i (lines 10-12).

The set of V/O assignments, prepared off-line, is used on-line by the quasi-static V/O scheduler as outlined by Algorithm 8.2. Upon completing task T_i ($t_i = t$, $EC_i = EC$), the lookup table LUT_i is consulted. If the point (t, EC) lies above the diagonal D_i (line 1) the index j of the table entry is

input: The maximum number $N^{\max}$ of assignments
output: The set of V/O assignments

1: distribute $N^{\max}$ among tasks (T_i gets N_i points)
2: $\text{V/O}^{\text{s-bc}}$:= sol. by dyn. scheduler when $M_k = M_k^{\text{bc}}$, $1 \le k \le n$
3: $\text{V/O}^{\text{s-wc}}$:= sol. by dyn. scheduler when $M_k = M_k^{\text{wc}}$, $1 \le k \le n$
4: $V_1 := V_1^{\text{s-bc}} = V_1^{\text{s-wc}}$; $O_1 := O_1^{\text{s-bc}} = O_1^{\text{s-wc}}$
5: store V_1/O_1 in $\text{LUT}_1[1]$
6: **for** $i \leftarrow 1, 2, \ldots, n-1$ **do**
7: $\quad t_i^{\text{s-bc}} := \sum_{k=1}^{i} (\tau_k^{\text{s-bc}} + \delta_k)$; $\quad t_i^{\text{s-wc}} := \sum_{k=1}^{i} (\tau_k^{\text{s-wc}} + \delta_k)$
8: $\quad EC_i^{\text{s-bc}} := \sum_{k=1}^{i} (E_k^{\text{s-bc}} + \mathcal{E}_k)$; $\quad EC_i^{\text{s-wc}} := \sum_{k=1}^{i} (E_k^{\text{s-wc}} + \mathcal{E}_k)$
9: $\quad$ **for** $j \leftarrow 1, 2, \ldots, N_i$ **do**
10: $\quad\quad t_i^j := [(N_i - j) t_i^{\text{s-bc}} + j\, t_i^{\text{s-wc}}]/N_i$
11: $\quad\quad EC_i^j := [(N_i - j) EC_i^{\text{s-bc}} + j\, EC_i^{\text{s-wc}}]/N_i$
12: $\quad\quad$ compute V_{i+1}^j/O_{i+1}^j for (t_i^j, EC_i^j) and store it in $\text{LUT}_i[j]$
13: $\quad$ **end for**
14: **end for**

Algorithm 8.1: OffLinePhase

simply calculated as in line 2, else as in line 4. Computing directly the index j, instead of searching through the table LUT_i, is possible because the points (t_i^j, EC_i^j) stored in LUT_i are equally-spaced. Finally the V/O assignment stored in $\text{LUT}_i[j]$ is retrieved (line 6). Observe that Algorithm 8.2 has a time complexity $\mathcal{O}(1)$ and therefore the on-line operation performed by the quasi-static V/O scheduler takes constant time and energy. Also, this lookup and selection process is several orders of magnitude cheaper than the on-line computation by the dynamic V/O scheduler.

input: Actual t and EC upon completing T_i, and lookup table LUT_i (contains N_i assignments and the diagonal D_i—defined as $EC_i = A_i t_i + B_i$)
output: The assignment V_{i+1}/O_{i+1} for the next task T_{i+1}

1: **if** $EC > A_i t + B_i$ **then**
2: $\quad j := \lceil N_i (EC - EC_i^{\text{s-bc}})/(EC_i^{\text{s-wc}} - EC_i^{\text{s-bc}}) \rceil$
3: **else**
4: $\quad j := \lceil N_i (t - t_i^{\text{s-bc}})/(t_i^{\text{s-wc}} - t_i^{\text{s-bc}}) \rceil$
5: **end if**
6: return V/O assignment stored in $\text{LUT}_i[j]$

Algorithm 8.2: OnLinePhase

8.2.4 Experimental Evaluation

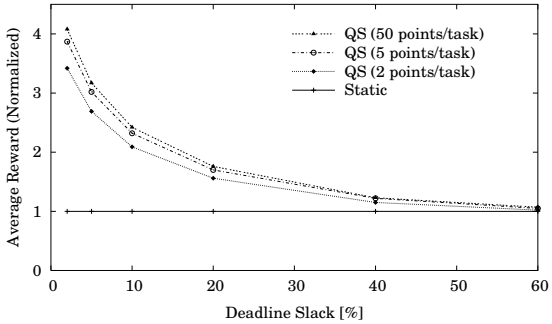
In order to evaluate the presented approach, we generated numerous synthetic benchmarks. We considered task graphs containing between 10 and

100 tasks. Each point in the plots of the experimental results (Figures 8.10, 8.11, and 8.12) corresponds to 50 automatically-generated task graphs, resulting overall in more than 4000 performed evaluations. The technology-dependent parameters were adopted from [MFMB02] and correspond to a processor in a 0.18 μm CMOS fabrication process. The reward functions we used along the experiments are of the form $R_i(O_i) = \alpha_i O_i + \beta_i \sqrt{O_i} + \gamma_i \sqrt[3]{O_i}$, with coefficients α_i, β_i, and γ_i randomly chosen.

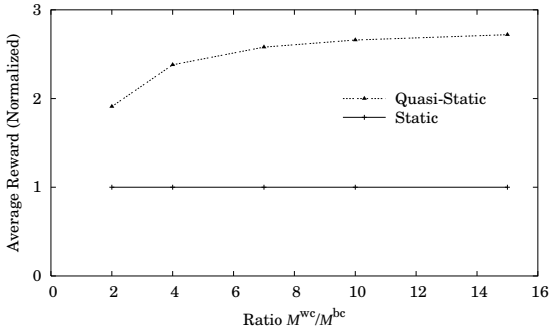
The first set of experiments was performed with the goal of investigating the reward gain achieved by our quasi-static approach compared to the optimal static solution (the approach proposed in [RMM03]). In these experiments we consider that the time and energy overheads needed for selecting the assignments by the quasi-static V/O scheduler are $\delta^{sel} = 450$ ns and $\mathcal{E}^{sel} = 400$ nJ. These are realistic values as selecting a precomputed assignment takes only tens of cycles and the access time and energy consumption (per access) of, for example, a low-power Static RAM are around 70 ns and 20 nJ respectively [NEC]. Figure 8.10(a) shows the reward (normalized with respect to the reward given by the static solution) as a function of the deadline slack (the relative difference between the deadline and the completion time when worst-case number of mandatory cycles are executed at the maximum voltage that guarantees the energy constraint). Three cases for the quasi-static approach (2, 5, and 50 points per task) are considered in this figure. Very significant gains in terms of total reward, up to four times, can be obtained with the quasi-static solution, even with few points per task. The highest reward gains are achieved when the system has very tight deadlines (small deadline slack). This is so because, when large amounts of slack are available, the static solution can accommodate most of the optional cycles (recall there is a value $O_i^{\max}$ after which no extra reward is achieved) and therefore the difference in reward between the static and quasi-static solutions is not big in these cases.

The influence of the ratio between the worst-case number of cycles M^{wc} and the best-case number of cycles M^{bc} has also been studied and the results are presented in Figure 8.10(b). In this case we have considered systems with a deadline slack of 10% and 20 points per task in the quasi-static solution. The larger the ratio $M^{\text{wc}}/M^{\text{bc}}$ is, the more the actual number of execution cycles deviate from the worst-case value M^{wc} (which is the value that has to be considered by a static solution). Thus the dynamic slack becomes larger and therefore there are more chances to exploit such a slack and consequently improve the reward.

The second set of experiments was aimed at evaluating the quality of our quasi-static approach with respect to the theoretical limit that could be achieved without knowing in advance the exact number of execution cy-



(a) Influence of the deadline slack



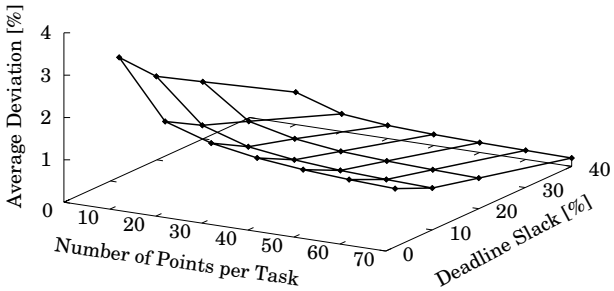
(b) Influence of the ratio $M^{\mathrm{wc}}/M^{\mathrm{bc}}$

Figure 8.10: Comparison of the quasi-static and static solutions

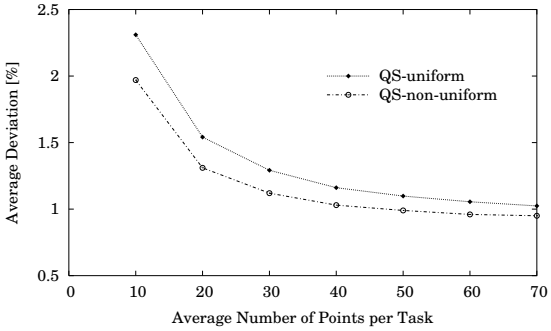
cles (the reward delivered by the ideal dynamic V/O scheduler). For the sake of comparison fairness, we have considered zero time and energy overheads δ^{sel} and $\mathcal{E}^{sel}$ (as opposed to the previous experiments). Figure 8.11(a) shows the deviation $dev = (R^{ideal} - R^{qs})/R^{ideal}$ as a function of the number of precomputed assignments (points per task) and for various degrees of deadline tightness. More points per task produce higher reward, closer to the theoretical limit (smaller deviation). Nonetheless, with relatively few points per task we can get very close to the theoretical limit, for instance, in systems with deadline slack of 20% and for 30 points per task the average deviation is around 1.3%. As mentioned previously, when the deadline slack is large even a static solution (which corresponds to a quasi-static solution with just one point per task) can accommodate most of the optional cycles. Hence, the deviation gets smaller as the deadline slack increases, as shown in Figure 8.11(a).

In the previous experiments it has been considered that, for a given system, the lookup tables have the same size, that is, contain the same number

of assignments. When the number $N^{\max}$ of assignments is distributed among tasks according to the size of their spaces time-energy (more assignments in the lookup tables of tasks that have larger spaces), better results are obtained as shown in Figure 8.11(b). This figure plots the case of equal-size lookup tables (QS-uniform) and the case of assignments distributed non-uniformly among tables (QS-non-uniform), as described above, for systems with a deadline slack of 20%. The abscissa is the *average* number of points per task.



(a) Influence of the deadline slack and number of points



(b) Influence of the distribution of points among lookup tables

Figure 8.11: Comparison of the quasi-static and ideal dynamic solutions

In a third set of experiments we took into account the on-line overheads of the dynamic V/O scheduler (as well as the quasi-static one) and compared the static, quasi-static, and dynamic approaches in the same graph. Figure 8.12 shows the reward normalized with respect to the one by the static solution. It shows that, in a realistic setting, the dynamic approach performs poorly, even worse than the static one. Moreover, for systems with tight deadlines (small deadline slack), the dynamic approach cannot

guarantee the time and energy constraints because of its large overheads (this is why no data is plotted for benchmarks with deadline slack less than 20%). The overhead values that have been considered for the dynamic case correspond actually to overheads by heuristics [RMM03] and not by exact methods, although in the experiments the values produced by the exact solutions were considered. This means that, even in the optimistic case of an on-line algorithm that delivers exact solutions in a time frame similar to the one of existing heuristic methods, the quasi-static approach outperforms the dynamic one.

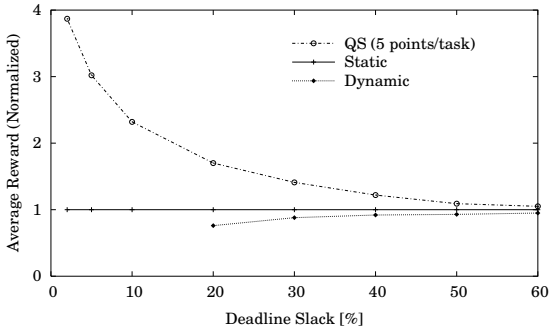


Figure 8.12: Comparison considering realistic overheads

We have also measured the execution time of Algorithm 8.1, used for computing at design-time the set of V/O assignments. Figure 8.13 shows the average execution time as a function of the number of tasks in the system, for different values of $N^{\max}$ (total number of assignments). It can be observed that the execution time is linear in the number of tasks and in the total number of assignments. The time needed for computing the set of assignments, though considerable, is affordable since Algorithm 8.1 is run off-line.

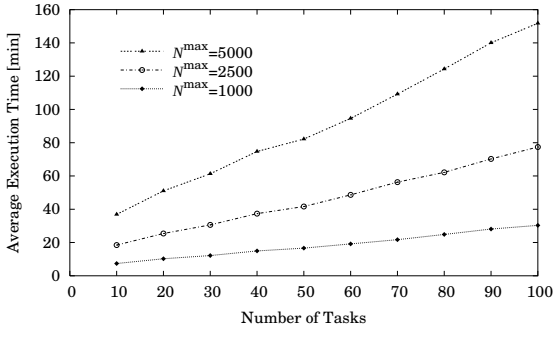


Figure 8.13: Execution time of OffLinePhase

In addition to the synthetic benchmarks discussed above, we have also evaluated our approach by means of a real-life application, namely the navigation controller of an autonomous rover for exploring a remote place [Hul00]. The rover is equipped, among others, with two cameras and a topographic map of the terrain. Based on the images captured by the cameras and the map, the rover must travel towards its destination avoiding nearby obstacles. This application includes a number of tasks described briefly as follows. A *frame acquisition* task captures images from the cameras. A *position estimation* task correlates the data from the captured images with the one from the topographic map in order to estimate the rover's current position. Using the estimated position and the topographic map, a *global path planning* task computes the path to the desired destination. Since there might be impassable obstacles along the global path, there is an *object detection* task for finding obstacles in the path of the rover and a *local path planning* task for adjusting accordingly the course in order to avoid those obstacles. A *collision avoidance* task checks the produced path to prevent the rover from damaging itself. Finally, a *steering control* task commands the motors the direction and speed of the rover.

For this application the total reward is measured in terms of how fast the rover reaches its destination [Hul00]. Rewards produced by the different tasks (all but the steering control task which has no optional part) contribute to the overall reward. For example, higher-resolution images by the frame acquisition task translates into a clearer characterization of the surroundings of the rover, which in turn implies a more accurate estimation of the location and consequently makes the rover get faster to its destination (that is, higher total reward). Similarly, running longer the global path planning task results in a better path which, again, implies reaching the desired destination faster. The other tasks make in a similar manner their individual contribution to the global reward, in such a way that the amount of computation allotted to each of them has a direct impact on how fast the destination is reached.

The navigation controller is activated periodically every 360 ms and tasks have a deadline equal to the period[3]. The energy budget per activation of the controller is 360 mJ (average power consumption 1 W) during the night and 540 mJ (average power 1.5 W) during daytime [Hib01].

Considering that one assignment requires 8 Bytes of memory, one 4-kB memory can store $N^{\max} = 512$ assignments in total. We use two 4-kB memories, one for the assignments used during daytime and the other for the set used during the night (these two sets are different because the energy budget differs). We computed, for both cases, $E^{\max} = 360$ mJ and $E^{\max} =$

[3]At its maximum speed of 10 km/h the rover travels in 360 ms a distance of 1 m, which is the maximum allowed without recomputing the path.

540 mJ, the sets of assignments using Algorithm 8.1. When compared to the respective static solutions, our quasi-static solution delivers rewards that are in average 3.8 times larger for the *night* case and 1.6 times larger for the *day* case. This means that a rover using the precomputed assignments can reach its destination faster than in the case of a static solution and thus explore a larger region under the same energy budget.

The significant difference between the *night* and *day* modes can be explained by the fact that, for more stringent energy constraints, fewer optional cycles can be accommodated by a static solution and therefore its reward is smaller. Thus the relative difference between a quasi-static solution and the corresponding static one is significantly larger for systems with more stringent energy constraints.

8.3 Minimizing Energy Consumption subject to Reward Constraints

We have addressed in Section 8.2 the maximization of rewards subject to energy constraints. In this section—also under the framework of the Imprecise Computation model as well as considering energy, reward, and deadlines—we discuss a different problem, namely minimizing the energy consumption considering that there is a minimum total reward that must be delivered by the system.

8.3.1 Problem Formulation

The static version of the problem addressed in this section has the following formulation.

PROBLEM 8.4 (Static V/O Assignment for Minimizing Energy—Static AME) Find, for each task T_i, $1 \leq i \leq n$, the voltage V_i and the number of optional cycles O_i that

$$\text{minimize} \sum_{i=1}^{n} \Big(\underbrace{C_r (V_{i-1} - V_i)^2}_{\mathcal{E}_{i-1,i}^{\Delta V}} + \underbrace{C_i V_i^2 (M_i^{\mathrm{e}} + O_i)}_{E_i^{\mathrm{e}}} \Big) \tag{8.15}$$

$$\text{subject to } V^{\min} \leq V_i \leq V^{\max} \tag{8.16}$$

$$s_{i+1} = t_i = s_i + \underbrace{p|V_{i-1} - V_i|}_{\delta_{i-1,i}^{\Delta V}} + \underbrace{k \frac{V_i}{(V_i - V_{th})^{\alpha}} (M_i^{\mathrm{wc}} + O_i)}_{\tau_i^{\mathrm{wc}}} \leq d_i \tag{8.17}$$

$$\sum_{i=1}^{n} R_i(O_i) \geq R^{\min} \tag{8.18}$$

In this case the objective function is the total energy, which has to be minimized (Equation (8.15)). The voltage V_i for each task T_i must be in the range $[V^{\min}, V^{\max}]$ (Equation (8.16)). The completion time t_i (sum of s_i, $\delta_{i-1,i}^{\Delta V}$, and τ_i) must be less than or equal to the deadline d_i (Equation (8.17)). The total reward has to be at least $R^{\min}$ (Equation (8.18)). Note that the worst-case number of mandatory cycles has to be assumed in order to guarantee the deadlines (Equation (8.17)).

A dynamic version of the problem addressed in this section is formulated as follows.

Dynamic V/O Scheduler (AME): The following is the problem that a dynamic V/O scheduler must solve every time a task T_c completes. It is considered that tasks $T_1, \ldots, T_c$ have already completed (the reward produced up to the completion of T_c is RP_c and the completion time of T_c is t_c).

Problem 8.5 (Dynamic AME) Find V_i and O_i, for $c+1 \leq i \leq n$, that

$$\text{minimize} \sum_{i=c+1}^{n} \left(\mathcal{E}_i^{dyn} + \mathcal{E}_{i-1,i}^{\Delta V} + \underbrace{C_i V_i^2 (M_i^{\mathrm{e}} + O_i)}_{E_i^{\mathrm{e}}} \right) \tag{8.19}$$

$$\text{subject to } V^{\min} \leq V_i \leq V^{\max} \tag{8.20}$$

$$s_{i+1} = t_i = s_i + \delta_i^{dyn} + \delta_{i-1,i}^{\Delta V} + \underbrace{k \frac{V_i}{(V_i - V_{th})^{\alpha}} (M_i^{\mathrm{wc}} + O_i)}_{\tau_i^{\mathrm{wc}}} \leq d_i \tag{8.21}$$

$$\sum_{i=c+1}^{n} R_i(O_i) \geq \left(R^{\min} - RP_c \right) \tag{8.22}$$

where δ_i^{dyn} and $\mathcal{E}_i^{dyn}$ are, respectively, the time and energy overhead of computing dynamically V_i and O_i for task T_i.

Analogous to Section 8.2, the problem solved by the above dynamic V/O scheduler corresponds to an instance of Problem 8.4, but taking into account δ_i^{dyn} and $\mathcal{E}_i^{dyn}$ and for $c+1 \leq i \leq n$. However, for the case discussed in this section (minimizing energy subject to reward constraints), a *speculative* version of the dynamic V/O scheduler can be formulated as follows. Such a dynamic speculative V/O scheduler produces better results than its non-speculative counterpart, as demonstrated by the experimental results of Subsection 8.3.3.

Dynamic Speculative V/O Scheduler (AME): The following is the problem that a dynamic speculative V/O scheduler must solve every time a task T_c completes. It is considered that tasks $T_1, \ldots, T_c$ have already completed (the reward produced up to the completion of T_c is RP_c and the completion time of T_c is t_c).

PROBLEM 8.6 (Dynamic Speculative AME) Find V_i and O_i, for $c+1 \leq i \leq n$, that

$$\text{minimize} \sum_{i=c+1}^{n} \Big(\mathcal{E}_i^{dyn} + \mathcal{E}_{i-1,i}^{\Delta V} + \underbrace{C_i V_i^2 (M_i^{\mathrm{e}} + O_i)}_{E_i^{\mathrm{e}}} \Big) \tag{8.23}$$

$$\text{subject to } V^{\min} \leq V_i \leq V^{\max} \tag{8.24}$$

$$s_{i+1} = t_i = s_i + \delta_i^{dyn} + \delta_{i-1,i}^{\Delta V} + \underbrace{k \frac{V_i}{(V_i - V_{th})^{\alpha}} (M_i^{\mathrm{e}} + O_i)}_{\tau_i^{\mathrm{e}}} \leq d_i \tag{8.25}$$

$$\sum_{i=c+1}^{n} R_i(O_i) \geq \left(R^{\min} - RP_c \right) \tag{8.26}$$

$$s'_{i+1} = t'_i = s'_i + \delta_i^{dyn} + \delta_{i-1,i}^{\Delta V} + \tau'_i \leq d_i \tag{8.27}$$

$$\tau'_i = \begin{cases} k \dfrac{V_i}{(V_i - V_{th})^{\alpha}} (M_i^{\mathrm{wc}} + O_i) & \text{if } i = c+1 \\ k \dfrac{V^{\max}}{(V^{\max} - V_{th})^{\alpha}} (M_i^{\mathrm{wc}} + O_i) & \text{if } i > c+1 \end{cases} \tag{8.28}$$

where δ_i^{dyn} and $\mathcal{E}_i^{dyn}$ are, respectively, the time and energy overhead of computing dynamically V_i and O_i for task T_i.

Equations (8.23)-(8.26) are basically the same as Equations (8.19)-(8.22) except that the expected number of mandatory cycles M_i^{e} is used instead of the worst-case number of mandatory cycles M_i^{wc} in the constraint corresponding to the deadlines. The constraint given by Equation (8.25) does not guarantee by itself the satisfaction of deadlines because if the actual number of mandatory cycles is larger than M_i^{e} deadline violations might arise. Therefore an additional constraint, as given by Equations (8.27) and (8.28), is introduced. It expresses that: the next task T_{c+1}, running at V_{c+1}, must meet its deadline (T_{c+1} will run at the computed V_{c+1}); the other tasks T_i, $c+1 < i \leq n$, running at $V^{\max}$, must also meet the deadlines (the other tasks T_i might run at a voltage different from the value V_i computed in the current iteration, because solutions obtained upon completion of future tasks might produce different values). Guaranteeing the deadlines in this way is possible because new assignments are similarly recomputed every time a task finishes.

The dynamic speculative V/O scheduler presented above solves the V/O assignment problem speculating that tasks will execute their expected number of mandatory cycles but leaving enough room for increasing the voltage so that future tasks, if needed, run faster and thus meet the deadlines. We consider that the energy E^{ideal} consumed by a system, when the V/O assignments are computed by such a dynamic speculative V/O scheduler in

the ideal case $\delta_i^{dyn} = 0$ and $\mathcal{E}_i^{dyn} = 0$, is the lower bound on the total energy that can practically be achieved without knowing beforehand the number of mandatory cycles executed by tasks.

It is worthwhile to mention at this point that Problem 8.2 (Dynamic AMR) formulated in Section 8.2 does not admit a speculative formulation, as opposed to Problem 8.5 formulated in this section, which does have a speculative version as presented above (Problem 8.6). This is so because, when speculating that tasks execute the expected number of mandatory cycles, there must be enough room for either increasing or decreasing the voltage levels of future tasks. However, if the voltage is increased in order to make tasks run faster and thus meet the deadlines, the energy consumption becomes larger and therefore the constraint on the maximum energy might be violated (Equation (8.12)). If the voltage is decreased in order to make tasks consume less energy and thus satisfy the total energy constraint, the execution times become longer and therefore deadlines might be missed (Equation (8.11)).

In a similar line of thought as in Section 8.2, we prepare at design-time a number of V/O assignments, one of which is selected at run-time (with very low overhead) by the quasi-static V/O scheduler.

Upon finishing a task T_c, the quasi-static V/O scheduler checks the completion time t_c and the reward RP_c produced up to completion of T_c, and looks up an assignment in LUT_c. From the lookup table LUT_c the quasi-static V/O scheduler gets the point (t'_c, RP'_c), which is the closest to (t_c, RP_c) such that $t_c \leq t'_c$ and $RP_c \geq RP'_c$, and selects V'/O' corresponding to (t'_c, RP'_c). The goal in this section is to make the system consume as little energy as possible, when using the assignments selected by the quasi-static V/O scheduler.

PROBLEM 8.7 (Set of V/O Assignments for Minimizing Energy—Set AME) Find a set of N assignments such that: $N \leq N^{\max}$; the V/O assignment selected by the quasi-static V/O scheduler guarantees the deadlines $(s_i + \delta_i^{sel} + \delta_{i-1,i}^{\Delta V} + \tau_i = t_i \leq d_i)$ and the reward constraint $(\sum_{i=1}^{n} R_i(O_i) \geq R^{\min})$, and so that the total energy E^{qs} is minimal.

8.3.2 Computing the Set of V/O Assignments

Analogous to what was discussed in Subsection 8.2.3, there is a space time-reward of possible values of completion time t_i and reward RP_i produced up to completion of T_i, as depicted in Figure 8.14. Each point in this space defines an assignment for the next task T_{i+1}: if T_i finished at t^a and the produced reward is RP^a, T_{i+1} would run at V^a and execute O^a optional cycles.

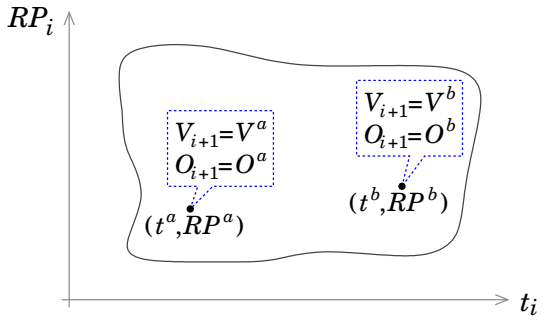


Figure 8.14: Space time-reward

The boundaries of the space t_i-RP_i can be obtained by computing the extreme values of t_i and RP_i considering $V^{\min}$, $V^{\max}$, M_j^{bc}, M_j^{wc}, and $O_j^{\max}$, $1 \leq j \leq i$. The maximum produced reward is $RP_i^{\max} = \sum_{j=1}^{i} R_j(O_j^{\max})$ and the minimum reward is simply $RP_i^{\min} = \sum_{j=1}^{i} R_j(0) = 0$. The maximum completion time $t_i^{\max}$ occurs when each task T_j executes $M_j^{\text{wc}}+O_j^{\max}$ cycles at $V^{\min}$, while $t_i^{\min}$ happens when each task T_j executes M_j^{bc} cycles at $V^{\max}$. The intervals $[t_i^{\min}, t_i^{\max}]$ and $[0, RP_i^{\max}]$ bound the space time-reward as shown in Figure 8.15.

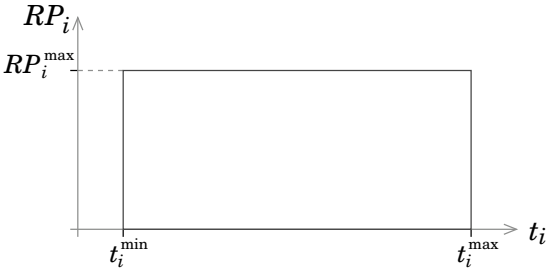


Figure 8.15: Boundaries of the space time-reward

A generic characterization of the space time-reward is not possible because reward functions vary from task to task as well as from system to system. That is, we cannot derive a general expression that relates the reward R_i with the execution time τ_i (as we did in Subsection 8.2.3.1 for E_i and τ_i, resulting in Equations (8.13) and (8.14)).

One alternative for selecting points in the space time-reward would be to consider a mesh-like configuration, in which the space is divided in rectangular areas and each area is covered by one point (the lower-right corner covers the rectangle) as depicted in Figure 8.16. The drawback of this approach is twofold: first, the boundaries in Figure 8.15 define a space time-reward that

include points that cannot happen, for example, the point $(t_i^{\min}, RP_i^{\max})$ is not feasible because $t_i^{\min}$ occurs when no optional cycles are executed whereas $RP_i^{\max}$ requires all tasks T_j executing $O_j^{\max}$ optional cycles; second, the number of required points for covering the space is a *quadratic* function of the granularity of the mesh, which means that too many points might be necessary for achieving an acceptable granularity.

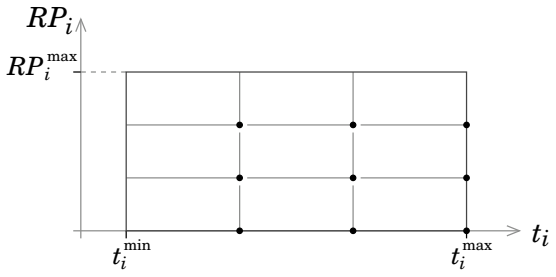


Figure 8.16: Selection of points in a mesh configuration

We have opted for a solution where we "freeze" the assigned optional cycles, that is, for each task T_i we fix O_i to a value $\overline{O}_i$ computed off-line. Thus, for any activation of the system, T_i will invariably execute $\overline{O}_i$ optional cycles. In this way, we transform the original problem into a classical voltage-scaling problem with deadlines since the only variables now are V_i. This means that we reduce the bidimensional space time-reward into a one-dimension space (time is now the only dimension). This approach gives very good results as shown by the experimental evaluation presented in Subsection 8.3.3.

The way we obtain the fixed values $\overline{O}_i$ is the following. We consider the instance of Problem 8.6 that the dynamic speculative V/O scheduler solves at the very beginning, before any task is executed ($c = 0$). The solution gives particular values of V_i and O_i, $1 \leq i \leq n$. For each task, the number of optional cycles given by this solution is taken as the fixed value $\overline{O}_i$ in our approach.

Once the number of optional cycles has been fixed to $\overline{O}_i$, the only variables are V_i and the problem becomes that of *voltage scaling for energy minimization with time constraints*. For the sake of completeness, we include below its formulation. The reward constraint disappears from the formulation because, by fixing the optional cycles as explained above, it is guaranteed that the total reward will be at least $R^{\min}$.

Dynamic Voltage Scheduler: The following is the problem that a dynamic voltage scheduler must solve every time a task T_c completes. It is considered that tasks $T_1, \ldots, T_c$ have already completed (the completion time of T_c is t_c).

PROBLEM 8.8 (Dynamic Voltage Scaling—VS) Find V_i, for $c+1 \leq i \leq n$, that

$$\text{minimize} \sum_{i=c+1}^{n} \left(\mathcal{E}_i^{dyn} + \mathcal{E}_{i-1,i}^{\Delta V} + \underbrace{C_i V_i^2 (M_i^{\mathrm{e}} + \overline{O}_i)}_{E_i^{\mathrm{e}}} \right)$$

$$\text{subject to } V^{\min} \leq V_i \leq V^{\max}$$

$$s_{i+1} = t_i = s_i + \delta_i^{dyn} + \delta_{i-1,i}^{\Delta V} + \underbrace{k \frac{V_i}{(V_i - V_{th})^{\alpha}} (M_i^{\mathrm{e}} + \overline{O}_i)}_{\tau_i^{\mathrm{e}}} \leq d_i$$

$$s'_{i+1} = t'_i = s'_i + \delta_i^{dyn} + \delta_{i-1,i}^{\Delta V} + \tau'_i \leq d_i$$

$$\tau'_i = \begin{cases} k \dfrac{V_i}{(V_i - V_{th})^{\alpha}} (M_i^{\mathrm{wc}} + \overline{O}_i) & \text{if } i = c+1 \\ k \dfrac{V^{\max}}{(V^{\max} - V_{th})^{\alpha}} (M_i^{\mathrm{wc}} + \overline{O}_i) & \text{if } i > c+1 \end{cases}$$

where δ_i^{dyn} and $\mathcal{E}_i^{dyn}$ are, respectively, the time and energy overhead of computing dynamically V_i for task T_i.

The voltage-scaling problem in a quasi-static framework has been addressed and solved by Andrei *et al.* [ASE+05]. In this case a simple static analysis gives, for each task T_i, the earliest and latest completion times $t_i^{\min}$ and $t_i^{\max}$. Thus the question in the quasi-static approach for this problem is to select points along the interval $[t_i^{\min}, t_i^{\max}]$ and compute accordingly the voltage settings that will be stored in memory. The reader is referred to [ASE+05] for a complete presentation of the quasi-static approach to voltage scaling.

In summary, in our quasi-static solution to the problem of minimizing energy subject to time and reward constraints, we first fix off-line the number of optional cycles assigned to each task, by taking the values O_i as given by the solution to Problem 8.6 (instance $c = 0$). Thus the original problem is reduced to quasi-static voltage scaling for energy minimization. Then, in the one-dimension space of possible completion times, we select points and compute the corresponding voltage assignments as discussed in [ASE+05]. For each task, a number of voltage settings are stored in its respective lookup table. Note that these tables contain only voltage values as the number of optional cycles has already been fixed off-line.

8.3.3 Experimental Evaluation

The approach proposed in this section has been evaluated through a large number of synthetic examples. We have considered task graphs that contain

between 10 and 100 tasks.

The first set of experiments validates the claim that the dynamic speculative V/O scheduler outperforms the non-speculative one. Figure 8.17 shows the average energy savings (relative to a static V/O assignment) as a function of the deadline slack (the relative difference between the deadline and the completion time when worst-case number of mandatory cycles are executed at the maximum voltage such that the reward constraint is guaranteed). The highest savings can be obtained for systems with small deadline slack: the larger the deadline slack is, the lower the voltages given by a static assignment can be (tasks can run slower), and therefore the difference in energy consumed as by a static and a dynamic solution is smaller. The experiments whose results are presented in Figure 8.17 were performed considering the ideal case of zero time and energy on-line overheads ($\delta_i^{dyn} = 0$ and $\mathcal{E}_i^{dyn} = 0$).

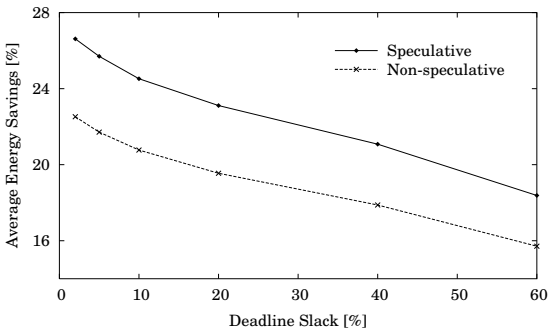
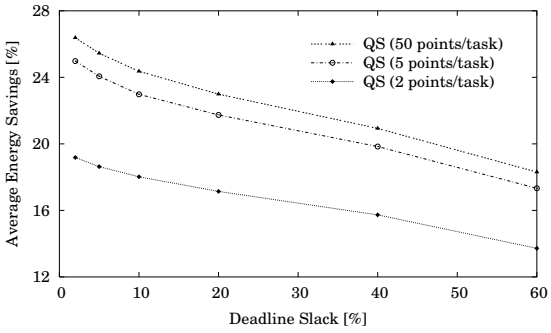


Figure 8.17: Comparison of the speculative and non-speculative dynamic V/O schedulers

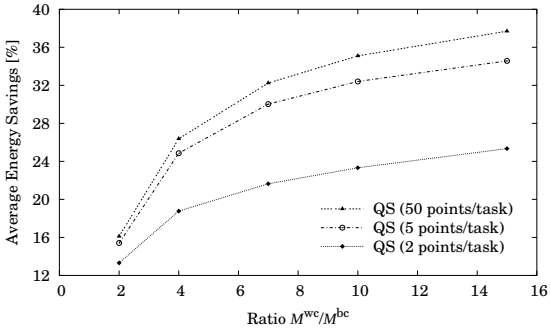
In a second set of experiments we evaluated the quasi-static approach proposed in this section, in terms of the energy savings achieved by it with respect to the optimal static solution. In this set of experiments we did take into consideration the time and energy overheads needed for selecting the voltage settings among the precomputed ones. Figure 8.18(a) shows the energy savings by our quasi-static approach for various numbers of points per task. The plot shows that, even with few points per task, very significant energy savings can be achieved.

Figure 8.18(b) also shows the energy savings achieved by the quasi-static approach, but this time as a function of the ratio between the worst-case number of cycles M^{wc} and the best-case number of cycles M^{bc}. In these experiments we considered systems with a deadline slack of 10%. As the ratio $M^{\text{wc}}/M^{\text{bc}}$ increases, the dynamic slack becomes larger and therefore there is more room for exploiting it in order to reduce the total energy

consumed by the system.



(a) Influence of the deadline slack



(b) Influence of the ratio $M^{\mathrm{wc}}/M^{\mathrm{bc}}$

Figure 8.18: Comparison of the quasi-static and static solutions

In a third set of experiments we evaluated the quality of the solution given by the quasi-static approach presented in this section with respect to the theoretical limit that could be achieved without knowing in advance the actual number of execution cycles (the energy consumed when a dynamic speculative V/O scheduler is used, in the ideal case of zero overheads—$\delta_i^{dyn} = 0$ and $\mathcal{E}_i^{dyn} = 0$). In order to make a fair comparison we considered also zero overheads for the quasi-static approach ($\delta_i^{sel} = 0$ and $\mathcal{E}_i^{sel} = 0$). Figure 8.19 shows the deviation $dev = (E^{qs} - E^{ideal})/E^{ideal}$ as a function of the number of precomputed voltages (points per task), where E^{ideal} is the total energy consumed for the case of an ideal dynamic V/O scheduler and E^{qs} is the total energy consumed for the case of a quasi-static scheduler that selects voltages from lookup tables prepared as explained in Subsection 8.3.2. In this set of experiments we have considered systems with deadline slack of 20%. It must be noted that E^{qs} corresponds to the proposed quasi-static approach in which we fix the number of optional cycles and the precomputed

assignments are only voltage settings, whereas E^{ideal} corresponds to the dynamic V/O scheduler that recomputes both voltage and number of optional cycles every time a task completes. Even so, with relatively few points per task it is possible to get very close to the theoretical limit, for instance, for 20 points per task the average deviation is around 0.4%.

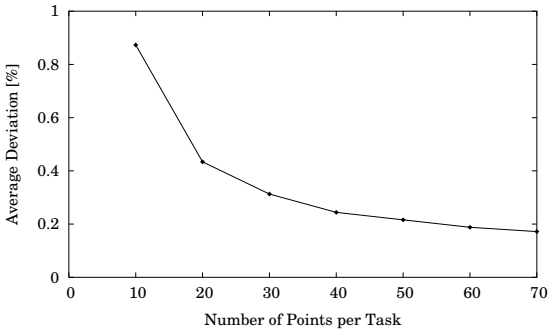


Figure 8.19: Comparison of the quasi-static and ideal dynamic solutions

Finally, in a fourth set of experiments we took into consideration realistic values for the on-line overheads δ_i^{dyn} and $\mathcal{E}_i^{dyn}$ of the dynamic V/O scheduler as well as the on-line overheads δ_i^{sel} and $\mathcal{E}_i^{sel}$ of the quasi-static scheduler. Figure 8.20 shows the average energy savings by the dynamic and quasi-static approaches (taking as baseline the energy consumed when using a static approach). It shows that in practice the dynamic approach makes the energy consumption higher than in the static solution (negative savings), a fact that is due to the high overheads incurred by computing on-line assignments by the dynamic V/O scheduler. Also because of the high overheads, when the system has tight deadlines, the dynamic approach cannot even guarantee the time constraints.

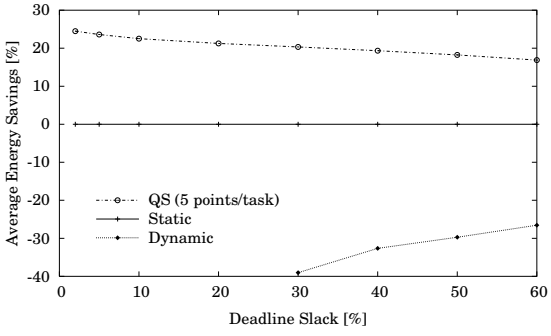


Figure 8.20: Comparison considering realistic overheads

Part IV

Conclusions and Future Work

Chapter 9

Conclusions

This chapter summarizes the principal contributions of the dissertation and presents conclusions drawn from the approaches introduced in the thesis. We first state general remarks on the dissertation as a whole and then we present conclusions organized according to the structure of the thesis, that is, conclusions particular to the approaches addressed in Parts II and III.

Embedded computer systems have become extremely common in our everyday life. They have numerous and diverse applications in a large spectrum of areas. The number of embedded systems as well as their application areas will certainly continue to grow. A very important factor in the widespread use of embedded systems is the vast computation capabilities available at low cost. In order to fully take advantage of this fact, it is needed to devise design methodologies that permit us to use this large amount of computation power in an efficient manner while satisfying the different constraints imposed by the particular types of application.

An essential point in the design of embedded systems is the intended application of the system under design. Design methodologies must thus be tailored to fit the distinctive characteristics of the specific type of system. It is needed to devise design techniques that take into account the particularities of the application domain in order to manage the system complexity, to improve the efficiency of the design process, and to produce high-quality solutions.

In this thesis we have proposed modeling, verification, and scheduling techniques for the class of embedded systems that have real-time requirements. Within this class of systems, two categories have been distinguished: hard real-time systems and soft real-time systems. We have introduced various approaches for hard real-time systems, placing special emphasis on the issue of correctness (Part II). In the case of mixed hard/soft real-time systems, we have focused on exploiting the flexibility provided by the soft com-

ponent which allows the designer to trade off quality of results and different design goals (Part III).

A distinguishing feature common to the different approaches introduced in this dissertation is the consideration of varying execution times for tasks. Instead of assuming always worst-case values, we have considered task execution times varying within a given interval. This model is more realistic and permits exploiting variations in actual execution times in order to, for example, improve the quality of results or reduce the energy consumption. At the same time, constraints dependent on execution times, such as deadlines, can be guaranteed.

The presented techniques have been studied and evaluated through a large number of experiments, including synthetic examples as well as realistic applications. The relevance of these techniques has been demonstrated by the corresponding experimental results.

In the rest of this chapter we summarize contributions and present conclusions that are particular to Parts II and III.

Modeling and Verification

In the second part of the thesis, we have dealt with modeling and verification of hard real-time systems.

A model of computation with precise mathematical semantics is essential in any systematic design methodology. A sound model of computation supports an unambiguous representation of the system, the use of formal methods to verify its correctness, and the automation of different tasks along the design process.

We have formally defined a model of computation that extends Petri nets. PRES+ allows the representation of systems at different levels of granularity and supports hierarchical constructs. It may easily capture both sequential and concurrent activities as well as non-determinism. In our model of computation tokens carry information and transitions perform transformation of data when fired, characteristics that are quite important in terms of expressiveness. Overall, PRES+ is simple, intuitive, and can easily be handled by the designer. It is also possible to translate textual-based specifications, such as Haskell descriptions, into the PRES+ model.

Several examples, including an industrial application, have been studied in order to demonstrate the applicability of our modeling technique to different systems.

Correctness is an aspect of prime importance for safety-critical, hard real-time systems. The cost of an error can be extremely high, in terms of loss of both human lives and money. Solutions that attempt to prove the

system correct are therefore essential when dealing with this type of systems.

We have proposed an approach to the formal verification of systems represented in PRES+. Our approach makes use of model checking in order to prove whether certain properties, expressed as CTL and TCTL formulas, hold with respect to the system model. We have introduced a systematic procedure to translate PRES+ models into timed automata so that it is possible to use available model checking tools.

Additionally, two strategies have been proposed in this thesis in order to improve the efficiency of the verification process.

First, we apply transformations to the initial system model, aiming at simplifying it, while still preserving the properties under consideration. This is a transformational approach that tries to reduce the model, and therefore improve the efficiency of verification, by using correctness-preserving transformations. Thus if the simpler model is proved correct, the initial one is guaranteed to be correct.

Second, we exploit the structure of the system and extract information regarding its degree of concurrency. We improve accordingly the translation procedure from PRES+ into timed automata by obtaining a reduced collection of automata and clocks. Since the time complexity of model checking of timed automata is exponential in the number of clocks, this technique improves considerably the verification efficiency.

Moreover, experimental results have shown that, by combining the transformational approach with the one for reducing the number of automata and clocks, the verification efficiency can be improved even further.

Scheduling Techniques

In the third part of the dissertation, we have dealt with scheduling techniques for mixed hard/soft real-time systems.

Approaches for hard/soft real-time systems permit dealing with tasks with different levels of criticality and therefore tackling a broad range of applications.

We have studied real-time systems composed of both hard and soft tasks. We make use of utility functions in order to capture the relative importance of soft tasks as well as how the quality of results is influenced upon missing a soft deadline. Differentiating among soft tasks gives an additional degree of flexibility as it allows the processing resources to be allocated more efficiently.

We have also studied real-time systems for which approximate but timely results are acceptable. We have considered the Imprecise Computation framework in which there exist functions that assign reward to tasks depending on how much they execute. Having different reward functions for

different tasks permits also distinguishing tasks and thus denoting their comparative significance.

Both for systems in which the quality of results (in the form of utilities) depends on task completion times and for systems in which the quality of results (in the form of rewards) depends on the amount of computation alloted to tasks, we have proposed quasi-static approaches.

The chief merit of the quasi-static techniques introduced in this thesis is their ability to exploit the dynamic slack, caused by tasks completing earlier than in the worst case, at a very low on-line overhead.

Real-time applications exhibit large variations in execution times and considering only worst-case values is typically too pessimistic, hence the importance of exploiting the dynamic slack for improving different design metrics (such as higher quality of results or lower energy consumption). However, dynamic approaches that recompute solutions at run-time in order to take advantage of such a slack incur a large overhead as these on-line computations are very complex in many cases. Even when the problems to be solved on-line admit polynomial-time solutions, or even when heuristics that produce approximate solutions are employed, the overhead is so high that it actually has a counterproductive effect.

Therefore, in order to efficiently exploit the dynamic slack, we need methods with low on-line overhead. The quasi-static approaches proposed in this dissertation succeed in exploiting the dynamic slack, yet having small on-line overhead, because the complex time- and energy-consuming parts of the computations are performed off-line, at design-time, leaving for run-time only simple lookup and selection operations.

In a quasi-static solution a number of schedules/assignments are computed and stored at design-time. This number of schedules/assignments that can be stored is limited by the resources of the target system. Therefore, a careful selection of schedules/assignments is crucial because it has a large impact on the quality of the solution.

Numerous experiments considering realistic settings have demonstrated the advantages of our quasi-static techniques over their static and dynamic counterparts.

Chapter 10

Future Work

There are certainly many possible extensions that can be pursued on the basis of the techniques proposed in this dissertation. This chapter discusses future directions of our research by pointing out some of the possible ways to improve and extend the work presented in this thesis.

- The verification approach introduced in this thesis is applicable to safe PRES+ models, that is, nets in which, for every reachable marking, each place holds at most one token. It would be desirable to extend the approach in such a way that it can also handle non-safe PRES+ models. A more general approach comes at the expense of verification complexity though.
- Along our verification approach we have proposed two strategies for improving verification efficiency. Future work in this line includes finding more efficient techniques that further improve the verification process, in terms of both time and memory. This can be achieved, for example, by identifying the parts of the system that are irrelevant for a particular property; in this way, when verifying that property, it is needed to consider only a fraction of the original model and thus the verification process is simplified.
- Our verification approach has concentrated on the presence/absence of tokens in the places of a PRES+ model and their time stamps. An interesting direction is to extend the techniques in such a way that reasoning about token values is also possible.
- The problem of mapping tasks onto processing resources is of particular interest. We have considered that the mapping is fixed and given as input to the scheduling problems addressed in the thesis. By considering the mapping of tasks as part of the problem, the designer can explore a larger portion of the design space and therefore search for better solutions. For instance, in relation to the problem of maximizing utility for real-

time systems with hard and soft tasks, not only does the task execution order affect the total utility but also the way tasks are mapped onto the available processing elements. Tools supporting both mapping and scheduling activities assist the designer in taking decisions that may lead to better results. There are also other steps of the design flow well worth considering, such as architecture selection.

- For the techniques discussed in this thesis in the frame of the Imprecise Computation model, we concentrated on the monoprocessor case. A natural extension is to explore similar approaches, in which energy, reward, and deadlines are considered under a unified framework, for the general case of multiprocessor systems.

Bibliography

[AB98] L. Abeni and G. Buttazzo. Integrating Multimedia Applications in Hard Real-Time Systems. In *Proc. Real-Time Systems Symposium*, pages 4–13, 1998.

[ACD90] R. Alur, C. Courcoubetis, and D. L. Dill. Model Checking for Real-Time Systems. In *Proc. Symposium on Logic in Computer Science*, pages 414–425, 1990.

[ACHH93] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems, LNCS 736*, pages 209–229, Berlin, 1993. Springer-Verlag.

[AHH96] R. Alur, T. A. Henzinger, and P.-H. Ho. Automatic Symbolic Verification of Embedded Systems. *IEEE Trans. on Software Engineering*, 22(3):181–201, March 1996.

[Alu99] R. Alur. Timed Automata. In D. A. Peled and N. Halbwachs, editors, *Computer-Aided Verification, LNCS 1633*, pages 8–22, Berlin, 1999. Springer-Verlag.

[AMMMA01] H. Aydin, R. Melhem, D. Mossé, and P. Mejía-Alvarez. Dynamic and Aggressive Scheduling Techniques for Power-Aware Real-Time Systems. In *Proc. Real-Time Systems Symposium*, pages 95–105, 2001.

[ASE+04] A. Andrei, M. Schmitz, P. Eles, Z. Peng, and B. Al-Hashimi. Overhead-Conscious Voltage Selection for Dynamic and Leakage Energy Reduction of Time-Constrained Systems. In *Proc. DATE Conference*, pages 518–523, 2004.

[ASE+05] A. Andrei, M. Schmitz, P. Eles, Z. Peng, and B. Al-Hashimi. Quasi-Static Voltage Scaling for Energy Minimization with Time Constraints. 2005. Submitted for publication.

[Bai71] D. E. Bailey. *Probability and Statistics.* John Wiley & Sons, New York, NY, 1971.

[BCG+97] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach.* Kluwer, Norwell, MA, 1997.

[Ben99] L. P. M. Benders. Specification and Performance Analysis of Embedded Systems with Coloured Petri Nets. *Computers and Mathematics with Applications*, 37(11):177–190, June 1999.

[BHJ+96] F. Balarin, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. Formal Verification of Embedded Systems based on CFSM Networks. In *Proc. DAC*, pages 568–571, 1996.

[BPB+00] A. Burns, D. Prasad, A. Bondavalli, F. Di Giandomenico, K. Ramamritham, J. A. Stankovic, and L. Strigini. The Meaning and Role of Value in Scheduling Flexible Real-Time Systems. *Journal of Systems Architecture*, 46(4):305–325, January 2000.

[Bré79] D. Brélaz. New Methods to Color the Vertices of a Graph. *Communications of the ACM*, 22(4):251–256, April 1979.

[BS99] G. Buttazzo and F. Sensini. Optimal Deadline Assignment for Scheduling Soft Aperiodic Tasks in Hard Real-Time Environments. *IEEE. Trans. on Computers*, 48(10):1035–1052, October 1999.

[But97] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Kluwer, Dordrecht, 1997.

[CEP99] L. A. Cortés, P. Eles, and Z. Peng. A Petri Net based Model for Heterogeneous Embedded Systems. In *Proc. NORCHIP Conference*, pages 248–255, 1999.

[CEP00a] L. A. Cortés, P. Eles, and Z. Peng. Definitions of Equivalence for Transformational Synthesis of Embedded Systems. In *Proc. Intl. Conference on Engineering of Complex Computer Systems*, pages 134–142, 2000.

[CEP00b] L. A. Cortés, P. Eles, and Z. Peng. Formal Coverification of Embedded Systems using Model Checking. In *Proc. Euromicro Conference (Digital Systems Design)*, volume 1, pages 106–113, 2000.

[CEP00c] L. A. Cortés, P. Eles, and Z. Peng. Verification of Embedded Systems using a Petri Net based Representation. In *Proc. Intl. Symposium on System Synthesis*, pages 149–155, 2000.

[CEP01] L. A. Cortés, P. Eles, and Z. Peng. Hierarchical Modeling and Verification of Embedded Systems. In *Proc. Euromicro Symposium on Digital System Design*, pages 63–70, 2001.

[CEP02a] L. A. Cortés, P. Eles, and Z. Peng. An Approach to Reducing Verification Complexity of Real-Time Embedded Systems. In *Proc. Euromicro Conference on Real-Time Systems (Work-in-progress Session)*, pages 45–48, 2002.

[CEP02b] L. A. Cortés, P. Eles, and Z. Peng. Verification of Real-Time Embedded Systems using Petri Net Models and Timed Automata. In *Proc. Intl. Conference on Real-Time Computing Systems and Applications*, pages 191–199, 2002.

[CEP03] L. A. Cortés, P. Eles, and Z. Peng. Modeling and Formal Verification of Embedded Systems based on a Petri Net Representation. *Journal of Systems Architecture*, 49(12-15):571–598, December 2003.

[CEP04a] L. A. Cortés, P. Eles, and Z. Peng. Combining Static and Dynamic Scheduling for Real-Time Systems. In *Proc. Intl. Workshop on Software Analysis and Development for Pervasive Systems*, pages 32–40, 2004. Invited paper.

[CEP04b] L. A. Cortés, P. Eles, and Z. Peng. Quasi-Static Scheduling for Real-Time Systems with Hard and Soft Tasks. In *Proc. DATE Conference*, pages 1176–1181, 2004.

[CEP04c] L. A. Cortés, P. Eles, and Z. Peng. Static Scheduling of Monoprocessor Real-Time Systems composed of Hard and

Soft Tasks. In *Proc. Intl. Workshop on Electronic Design, Test and Applications*, pages 115–120, 2004.

[CEP05a] L. A. Cortés, P. Eles, and Z. Peng. Quasi-Static Assignment of Voltages and Optional Cycles for Maximizing Rewards in Real-Time Systems with Energy Constraints. 2005. Submitted for publication.

[CEP05b] L. A. Cortés, P. Eles, and Z. Peng. Quasi-Static Scheduling for Multiprocessor Real-Time Systems with Hard and Soft Tasks. 2005. Submitted for publication.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CGH+93] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. A Formal Specification Model for Hardware/Software Codesign. Technical Report UCB/ERL M93/48, Dept. EECS, University of California, Berkeley, June 1993.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[CKLW03] J. Cortadella, A. Kondratyev, L. Lavagno, and Y. Watanabe. Quasi-Static Scheduling for Concurrent Architectures. In *Proc. Intl. Conference on Application of Concurrency to System Design*, pages 29–40, 2003.

[CM96] K. Chen and P. Muhlethaler. A Scheduling Algorithm for Tasks described by Time Value Function. *Real-Time Systems*, 10(3):293–312, May 1996.

[CPE01] L. A. Cortés, Z. Peng, and P. Eles. From Haskell to PRES+: Basic Translation Procedures. SAVE Project Report, Dept. of Computer and Information Science, Linköping University, Linköping, April 2001. Available from `http://www.ida.liu.se/~eslab/save`.

[CW96] E. M. Clarke and J. M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[DeM97] G. De Micheli. Hardware/Software Co-Design. *Proc. IEEE*, 85(3):349–365, March 1997.

[DFL72] J. B. Dennis, J. B. Fosseen, and J. P. Linderman. Data flow schemas. In *Proc. Intl. Symposium on Theoretical Programming*, pages 187–216, 1972.

[Dil98] D. L. Dill. What's Between Simulation and Formal Verification? In *Proc. DAC*, pages 328–329, 1998.

[Dit95] G. Dittrich. Modeling of Complex Systems Using Hierarchical Petri Nets. In J. Rozenblit and K. Buchenrieder, editors, *Codesign: Computer-Aided Software/Hardware Engineering*, pages 128–144, Piscataway, NJ, 1995. IEEE Press.

[DTB93] R. I. Davis, K. W. Tindell, and A. Burns. Scheduling Slack Time in Fixed Priority Pre-emptive Systems. In *Proc. Real-Time Systems Symposium*, pages 222–231, 1993.

[EKP+98] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, and P. Pop. Scheduling of Conditional Process Graphs for the Synthesis of Embedded Systems. In *Proc. DATE Conference*, pages 132–138, 1998.

[ELLSV97] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vicentelli. Design of Embedded Systems: Formal Models, Validation, and Synthesis. *Proc. IEEE*, 85(3):366–390, March 1997.

[Esp94] J. Esparza. Model Checking using Net Unfoldings. *Science of Computer Programming*, 23(2-3):151–195, December 1994.

[Esp98] J. Esparza. Decidability and complexity of Petri net problems—an introduction. In P. Wolper and G. Rozenberg, editors, *Lectures on Petri Nets: Basic Models, LNCS 1491*, pages 374–428, Berlin, 1998. Springer-Verlag.

[ETT98] R. Esser, J. Teich, and L. Thiele. CodeSign: An Embedded System Design Environment. *IEE Proc. Computers and Digital Techniques*, 145(3):171–180, May 1998.

[FFP04] L. Formaggio, F. Fummi, and G. Pravadelli. A Timing-Accurate HW/SW Co-simulation of an ISS with SystemC. In *Proc. CODES+ISSS*, pages 152–157, 2004.

[FIR+97] L. Freund, M. Israel, F. Rousseau, J. M. Bergé, M. Auguin, C. Belleudy, and G. Gogniat. A Codesign Experiment in Acoustic Echo Cancellation: GMDFα. *ACM Trans. on Design Automation of Electronic Systems*, 2(4):365–383, October 1997.

[Fit96] M. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer-Verlag, New York, NY, 1996.

[FLL+98] E. Filippi, L. Lavagno, L. Licciardi, A. Montanaro, M. Paolini, R. Passerone, M. Sgroi, and A. Sangiovanni-Vincentelli. Intellectual Property Re-use in Embedded System Co-design: an Industrial Case Study. In *Proc. ISSS*, pages 37–42, 1998.

[FPF+03] F. Fummi, G. Pravadelli, A. Fedeli, U. Rossi, and F. Toto. On the Use of a High-Level Fault Model to Check Properties Incompleteness. In *Proc. Intl. Conference on Formal Methods and Models for Co-Design*, pages 145–152, 2003.

[Gal87] J. H. Gallier. *Foundations of Automatic Theorem Proving.* John Wiley & Sons, New York, NY, 1987.

[GBdSMH01] A. R. Girard, J. Borges de Sousa, J. A. Misener, and J. K. Hedrick. A Control Architecture for Integrated Cooperative Cruise Control with Collision Warning Systems. In *Proc. Conference on Decision and Control*, volume 2, pages 1491–1496, 2001.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, San Francisco, CA, 1979.

[GK01] F. Gruian and K. Kuchcinski. LEneS: Task Scheduling for Low-Energy Systems Using Variable Supply Voltage Processors. In *Proc. ASP-DAC*, pages 449–455, 2001.

[GR94] D. D. Gajski and L. Ramachandran. Introduction to High-Level Synthesis. *IEEE Design & Test of Computers*, 11(4):44–54, 1994.

[GVNG94] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1994.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[Has] Haskell. `http://www.haskell.org`.

[HG02] P.-A. Hsiung and C.-H. Gau. Formal Synthesis of Real-Time Embedded Software by Time-Memory Scheduling of Colored Time Petri Nets. *Electronic Notes in Theoretical Computer Science*, 65(6), June 2002.

[Hib01] B. D. Hibbs. Mars Solar Rover Feasibility Study. Technical Report NASA/CR 2001-210802, NASA/AeroVironment, Inc., Washington, DC, March 2001.

[HMU01] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Boston, MA, 2001.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[HR94] N. Homayoun and P. Ramanathan. Dynamic Priority Scheduling of Periodic and Aperiodic Tasks in Hard Real-Time Systems. *Real-Time Systems*, 6(2):207–232, March 1994.

[Hsi99] P.-A. Hsiung. Hardware-Software Coverification of Concurrent Embedded Real-Time Systems. In *Proc. Euromicro Conference on Real-Time Systems*, pages 216–223, 1999.

[Hul00] D. L. Hull. *An Environment for Imprecise Computations*. PhD thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, January 2000.

[HyT] HyTech. `http://www-cad.eecs.berkeley.edu/~tah/HyTech`.

[IAJ94] T. B. Ismail, M. Abid, and A. A. Jerraya. COSMOS: A CoDesign Approach for Communicating Systems. In *Proc. CODES/CASHE*, pages 17–24, 1994.

[Jan03] A. Jantsch. *Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation*. Morgan Kaufmann, San Francisco, CA, 2003.

[Jen92] K. Jensen. *Coloured Petri Nets*. Springer-Verlag, Berlin, 1992.

[JO95] A. A. Jerraya and K. O'Brien. SOLAR: An Intermediate Format for System-Level Modeling and Synthesis. In J. Rozenblit and K. Buchenrieder, editors, *Codesign: Computer-Aided Software/Hardware Engineering*, pages 145–175, Piscataway, NJ, 1995. IEEE Press.

[Joh98] H. Johnson. Keeping Up with Moore. *EDN Magazine*, May 1998.

[JR91] K. Jensen and G. Rozenberg, editors. *High-level Petri Nets.* Springer-Verlag, Berlin, 1991.

[Kah01] A. B. Kahng. Design Technology Productivity in the DSM Era. In *Proc. ASP-DAC*, pages 443–448, 2001.

[KE96] A. Kovalyov and J. Esparza. A Polynomial Algorithm to Compute the Concurrency Relation of Free-choice Signal Transition Graphs. In *Proc. Intl. Workshop on Discrete Event Systems*, pages 1–6, 1996.

[KG99] C. Kern and M. R. Greenstreet. Formal Verification in Hardware Design: A Survey. *ACM Trans. on Design Automation of Electronic Systems*, 4(2):123–193, April 1999.

[KL03] C. M. Krishna and Y.-H. Lee. Voltage-Clock-Scaling Adaptive Scheduling Techniques for Low Power in Hard Real-Time Systems. *IEEE Trans. on Computers*, 52(12):1586–1593, December 2003.

[Kop97] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Kluwer, Dordrecht, 1997.

[Kov92] A. Kovalyov. Concurrency Relations and the Safety Problem for Petri Nets. In K. Jensen, editor, *Application and Theory of Petri Nets, LNCS 616*, pages 299–309, Berlin, 1992. Springer-Verlag.

[Kov00] A. Kovalyov. A Polynomial Algorithm to Compute the Concurrency Relation of a Regular STG. In A. Yakovlev, L. Gomes, and L. Lavagno, editors, *Hardware Design and Petri Nets*, pages 107–126, Dordrecht, 2000. Kluwer.

[Koz97] D. C. Kozen. *Automata and Computability.* Springer-Verlag, New York, NY, 1997.

[KP97] D. Kirovski and M. Potkonjak. System-Level Synthesis of Low-Power Hard Real-Time Systems. In *Proc. DAC*, pages 697–702, 1997.

[Kro] Kronos. `http://www-verimag.imag.fr/TEMPORISE/kronos`.

[KSSR96] H. Kaneko, J. A. Stankovic, S. Sen, and K. Ramamritham. Integrated Scheduling of Multimedia and Hard Real-Time Tasks. In *Proc. Real-Time Systems Symposium*, pages 206–217, 1996.

[Lap04] P. A. Laplante. *Real-Time Systems Design and Analysis*. John Wiley & Sons, Hoboken, NY, 2004.

[Law73] E. L. Lawler. Optimal Sequencing of a Single Machine subject to Precedence Constraints. *Management Science*, 19:544–546, 1973.

[Lee58] C. Y. Lee. Some Properties of Nonbinary Error-Correcting Codes. *IEEE Trans. on Information Theory*, 2(4):77–82, June 1958.

[LJ03] J. Luo and N. K. Jha. Power-profile Driven Variable Voltage Scaling for Heterogeneous Distributed Real-Time Embedded Systems. In *Proc. Intl. Conference on VLSI Design*, pages 369–375, 2003.

[LK01] P. Lind and S. Kvist. Jammer Model Description. Technical Report, Saab Bofors Dynamics AB, Linköping, April 2001.

[LM87] E. A. Lee and D. G. Messerschmitt. Synchronous Data Flow. *Proc. IEEE*, 75(9):1235–1245, September 1987.

[Loc86] C. D. Locke. *Best-Effort Decision Making for Real-Time Scheduling*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, May 1986.

[LP95] E. A. Lee and T. M. Parks. Dataflow Process Networks. *Proc. IEEE*, 83(5):773–799, May 1995.

[LPY95] K. G. Larsen, P. Pettersson, and W. Yi. Model-Checking for Real-Time Systems. In H. Reichel, editor, *Fundamentals of Computation Theory, LNCS 965*, pages 62–88, Berlin, 1995. Springer-Verlag.

[LRT92] J. P. Lehoczky and S. Ramos-Thuel. An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems. In *Proc. Real-Time Systems Symposium*, pages 110–123, 1992.

[LSL+94] J. W. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung. Imprecise Computations. *Proc. IEEE*, 82(1):83–94, January 1994.

[LSVS99] L. Lavagno, A. Sangiovanni-Vincentelli, and E. Sentovich. Models of Computation for Embedded System Design. In A. A. Jerraya and J. Mermet, editors, *System-Level Synthesis*, pages 45–102, Dordrecht, 1999. Kluwer.

[MBR99] P. Maciel, E. Barros, and W. Rosenstiel. A Petri Net Model for Hardware/Software Codesign. *Design Automation for Embedded Systems*, 4(4):243–310, October 1999.

[McC99] S. McCartney. *ENIAC: The Triumphs and Tragedies of the World's First Computer*. Walker Publishing, New York, NY, 1999.

[MF76] P. M. Merlin and D. J. Farber. Recoverability of Communication Protocols–Implications of a Theoretical Study. *IEEE Trans. on Communications*, COM-24(9):1036–1042, September 1976.

[MFMB02] S. M. Martin, K. Flautner, T. Mudge, and D. Blaauw. Combined Dynamic Voltage Scaling and Adaptive Body Biasing for Low Power Microprocessors under Dynamic Workloads. In *Proc. ICCAD*, pages 721–725, 2002.

[Moo65] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 1965.

[MOS] MOSEK. `http://www.mosek.com`.

[Mur89] T. Murata. Petri Nets: Analysis and Applications. *Proc. IEEE*, 77(4):541–580, April 1989.

[NEC] NEC Memories. `http://www.necel.com/memory/index_e.html`.

[NN94] Y. Nesterov and A. Nemirovski. *Interior-Point Polynomial Algorithms in Convex Programming*. SIAM, Philadelphia, 1994.

[OYI01] T. Okuma, H. Yasuura, and T. Ishihara. Software Energy Reduction Techniques for Variable-Voltage Processors. *IEEE Design & Test of Computers*, 18(2):31–41, March 2001.

[PBA03] D. Prasad, A. Burns, and M. Atkins. The Valid Use of Utility in Adaptive Real-Time Systems. *Real-Time Systems*, 25(2-3):277–296, September 2003.

[Pet81] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[RCGF97] I. Ripoll, A. Crespo, and A. García-Fornes. An Optimal Algorithm for Scheduling Soft Aperiodic Tasks in Dynamic-Priority Preemptive Systems. *IEEE. Trans. on Software Engineering*, 23(6):388–400, October 1997.

[RLLS97] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A Resource Allocation Model for QoS Management. In *Proc. Real-Time Systems Symposium*, pages 298–307, 1997.

[RMM03] C. Rusu, R. Melhem, and D. Mossé. Maximizing Rewards for Real-Time Applications with Energy Constraints. *ACM Trans. on Embedded Computing Systems*, 2(4):537–559, November 2003.

[Row94] J. A. Rowson. Hardware/Software Co-Simulation. In *Proc. DAC*, pages 439–440, 1994.

[SC99] Y. Shin and K. Choi. Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems. In *Proc. DAC*, pages 134–139, 1999.

[SGG+03] C.-S. Shih, S. Gopalakrishnan, P. Ganti, M. Caccamo, and L. Sha. Template-Based Real-Time Dwell Scheduling with Energy Constraints. In *Proc. Real-Time and Embedded Technology and Applications Symposium*, pages 19–27, 2003.

[SH02] F.-S. Su and P.-A. Hsiung. Extended Quasi-Static Scheduling for Formal Synthesis and Code Generation of Embedded Software. In *Proc. CODES*, pages 211–216, 2002.

[SJ04] I. Sander and A. Jantsch. System modeling and transformational design refinement in forsyde. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 23(1):17–32, January 2004.

[SLC89] W.-K. Shih, J. W. S. Liu, and J.-Y. Chung. Fast Algorithms for Scheduling Imprecise Computations. In *Proc. Real-Time Systems Symposium*, pages 12–19, 1989.

[SLK01] D. Shin, S. Lee, and J. Kim. Intra-Task Voltage Scheduling for Low-Energy Hard Real-Time Applications. *IEEE Design & Test of Computers*, 18(2):20–30, March 2001.

[SLS95] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. *IEEE Trans. on Computers*, 44(1):73–91, January 1995.

[SLSV00] M. Sgroi, L. Lavagno, and A. Sangiovanni-Vincentelli. Formal Models for Embedded System Design. *IEEE Design & Test of Computers*, 17(2):14–27, April 2000.

[SLWSV99] M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli. Synthesis of Embedded Software Using Free-Choice Petri Nets. In *Proc. DAC*, pages 805–810, 1999.

[STG+01] K. Strehl, L. Thiele, M. Gries, D. Ziegenbein, R. Ernst, and J. Teich. *FunState*–An Internal Design Representation for Codesign. *IEEE Trans. on VLSI Systems*, 9(4):524–544, August 2001.

[Sto95] E. Stoy. A Petri Net Based Unified Representation for Hardware/Software Co-Design. Licentiate Thesis, Dept. of Computer and Information Science, Linköping University, Linköping, 1995.

[TAS93] D. E. Thomas, J. K. Adams, and H. Schmit. A Model and Methodology for Hardware-Software Codesign. *IEEE Design & Test of Computers*, 10(3):6–15, 1993.

[Tur02] J. Turley. The Two Percent Solution. *Embedded Systems Programming*, 15(12), December 2002.

[Upp] UPPAAL. `http://www.uppaal.com`.

[VAH01] M. Varea and B. Al-Hashimi. Dual Transitions Petri Net based Modelling Technique for Embedded Systems Specification. In *Proc. DATE Conference*, pages 566–571, 2001.

[VAHC+02] M. Varea, B. Al-Hashimi, L. A. Cortés, P. Eles, and Z. Peng. Symbolic Model Checking of Dual Transition Petri Nets. In *Proc. Intl. Symposium on Hardware/Software Codesign*, pages 43–48, 2002.

[Vav91] S. A. Vavasis. *Nonlinear Optimization: Complexity Issues*. Oxford University Press, New York, NY, 1991.

[VG02] F. Vahid and T. Givargis. *Embedded Systems Design: A Unified Hardware/Software Introduction*. John Wiley & Sons, New York, NY, 2002.

[vLA87] P. J. M. van Laarhoven and E. H. L. Aarts. *Simulated Annealing: Theory and Applications*. Kluwer, Dordrecht, 1987.

[WRJB04] H. Wu, B. Ravindran, E. D. Jensen, and U. Balli. Utility Accrual Scheduling under Arbitrary Time/Utility Functions and Multi-unit Resource Constraints. In *Proc. Intl. Conference on Real-Time Computing Systems and Applications*, pages 80–98, 2004.

[YC03] P. Yang and F. Catthoor. Pareto-Optimization-Based Run-Time Task Scheduling for Embedded Systems. In *Proc. CODES+ISSS*, pages 120–125, 2003.

[YDS95] F. Yao, A. Demers, and S. Shenker. A Scheduling Model for Reduced CPU Energy. In *Proc. Symposium on Foundations of Computer Science*, pages 374–382, 1995.

[Yen91] H.-C. Yen. A Polynomial Time Algorithm to Decide Pairwise Concurrency of Transitions for 1-bounded Conflict-Free Petri Nets. *Information Processing Letters*, 38:71–76, April 1991.

[YK04] H.-S. Yun and J. Kim. Reward-Based Voltage Scheduling for Fixed-Priority Hard Real-Time Systems. In *Proc. Intl. Workshop on Power-Aware Real-Time Computing*, pages 1–4, 2004.

[ZHC03] Y. Zhang, X. S. Hu, and D. Z. Chen. Energy Minimization of Real-Time Tasks on Variable Voltage Processors with Transition Overheads. In *Proc. ASP-DAC*, pages 65–70, 2003.

Appendices

Appendix A

Notation

Petri Nets and PRES+

Notation	*Description*
$\parallel$	concurrency relation
$\parallel_{\mathbf{T}}$	concurrency relation on $\mathbf{T}$
$\parallel^{\mathcal{S}}$	structural concurrency relation
$\parallel^{\mathcal{S}}_{\mathbf{T}}$	structural concurrency relation on $\mathbf{T}$
B_i	binding of transition T_i
et_i	enabling time of transition T_i
f_i	transition function of transition T_i
	high-level function of super-transition ST_i
g_i	guard of transition T_i
H	abstract PRES+ net
inP	set of in-ports
I	input (place-transition) arc
I	set of input arcs
K	token
K	set of all possible tokens in a net
$\mathbf{K}_P$	set of possible tokens in place P
$m_i(P)$	number of tokens in place P, for marking M_i
M	marking
M_0	initial marking
$M(P)$	marking of place P
$M_0(P)$	initial marking of place P
N	Petri net
	PRES+ net

Notation	*Description*
outP	set of out-ports
O	output (transition-place) arc
O	set of output arcs
P	place
${}^\circ P$	set of input transitions of place P
P°	set of output transitions of place P
P	set of places
$\mathscr{R}(N)$	reachability set of net N
ST	super-transition
${}^\circ ST$	set of input places of super-transition ST
ST°	set of output places of super-transition ST
ST	set of super-transitions
t_i	token time of token K_i
tt_i^{bc}	earliest trigger time of transition T_i
tt_i^{wc}	latest trigger time of transition T_i
τ_i^{bc}	best-case transition delay of transition T_i
	best-case delay of super-transition ST_i
τ_i^{wc}	worst-case transition delay of transition T_i
	worst-case delay of super-transition ST_i
T	transition
${}^\circ T$	set of input places of transition T
T°	set of output places of transition T
T	set of transitions
v_i	token value of token K_i
$\zeta(P)$	token type associated to place P
$\boldsymbol{\zeta}$	set of all token types in a net

Timed Automata

Notation	*Description*
$\mathtt{a}(e)$	activities assigned to edge e
c	clock
$\mathtt{c}(e)$	clock condition over edge e
$\mathcal{C}$	set of clocks
e	edge
$\mathcal{E}$	set of edges
$\mathtt{i}(l)$	invariant of location l
l	location

Notation	*Description*
$\mathcal{L}$	set of locations
$\mathcal{L}_0$	set of initial locations
$\mathrm{r}(e)$	set of clocks to reset on edge e
$\vec{\mathcal{T}}$	timed automaton
$\mathrm{v}(e)$	variable condition over edge e
$\mathcal{V}$	set of variables
$\mathrm{x}(e)$	label of edge e
$\mathcal{X}$	set of labels

Systems with Real-Time Hard and Soft Tasks

Notation	*Description*
d_i	deadline of task T_i
E	edge
$\mathbf{E}$	set of edges
G	dataflow graph
$\mathbf{H}$	set of hard tasks
$\mathcal{I}^i$	interval of possible completions times t_i
$m(T)$	mapping of task T
PE	processing element
$\mathbf{PE}$	set of processing elements
s_i	starting time of task T_i
$\mathbf{S}$	set of soft tasks
$\sigma^{(i)}$	task execution order on processing element PE_i
Ω	schedule (set of task execution orders $\sigma^{(i)}$)
t_i	completion time of task T_i
τ_i	actual execution time of task T_i
τ_i^{bc}	best-case duration of task T_i
τ_i^{e}	expected duration of task T_i
τ_i^{wc}	worst-case duration of task T_i
T	task
${}^\circ T$	set of direct predecessors of task T
T°	set of direct successors of task T
$\mathbf{T}$	set of tasks
$\mathbf{T}^{(i)}$	set of tasks mapped onto processing element PE_i
$u_i(t_i)$	utility function of soft task T_i

Notation	*Description*
U	total utility

Imprecise-Computation Systems

Notation	*Description*
C_i	effective switched capacitance corresponding to task T_i
d_i	deadline of task T_i
$\delta_{i,j}^{\Delta V}$	time overhead by switching from V_i to V_j
δ_i^{sel}	time overhead by selecting assignments for task T_i
δ_i^{dyn}	time overhead by on-line operations (upon completing T_i)
$\mathbf{E}$	set of edges
$E^{\max}$	upper limit in the energy consumed by the system
E_i	dynamic energy consumed by task T_i
EC_i	total energy consumed up to the completion of task T_i
$\mathcal{E}_{i,j}^{\Delta V}$	energy overhead by switching from V_i to V_j
$\mathcal{E}_i^{sel}$	energy overhead by selecting assignments for task T_i
$\mathcal{E}_i^{dyn}$	energy overhead by on-line operations (upon completing T_i)
G	dataflow graph
LUT_i	lookup table corresponding to task T_i
M_i	actual number of mandatory cycles of task T_i
M_i^{bc}	best-case number of mandatory cycles of task T_i
M_i^{e}	expected number of mandatory cycles of task T_i
M_i^{wc}	worst-case number of mandatory cycles of task T_i
$N^{\max}$	maximum number of V/O assignments that can be stored
O_i	number of optional cycles of task T_i
$O_i^{\max}$	num. of opt. cycles of T_i after which no extra reward is gained
$R_i(O_i)$	reward function of task T_i
R	total reward
$R^{\min}$	lower limit in the reward produced by the system
$R_i^{\max}$	maximum reward for task T_i
RP_i	reward produced up to the completion of task T_i
s_i	starting time of task T_i
t_i	completion time of task T_i
τ_i	execution time of task T_i
T	task
$\mathbf{T}$	set of tasks

Notation	*Description*
$V^{\min}$	minimum voltage of the target processor
$V^{\max}$	maximum voltage of the target processor
V_i	voltage at which task T_i runs

Appendix B

Proofs

B.1 Validity of Transformation Rule TR1

The validity of the transformation rule TR1 introduced in Subsection 5.1.1 (see Figure 5.1) is proved by showing that the nets N' and N'' in Figure B.1 are total-equivalent, provided that $f = f_2 \circ f_1$, $l = l_1 + l_2$, $u = u_1 + u_2$, and $M_0(P) = \emptyset$.

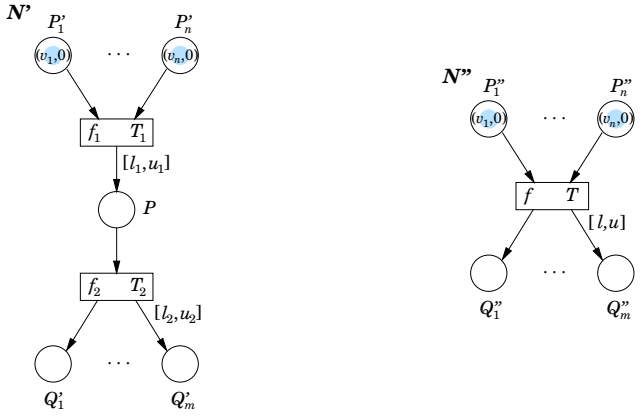


Figure B.1: Proving the validity of TR1

As defined in Subsection 3.5.1, the idea behind total-equivalence is as follows: a) there exist bijections that define one-to-one correspondences between the in(out)-ports of N' and N''; b) having initially identical tokens in corresponding in-ports, there exists a firing sequence which leads to the same marking (same token values and same token times) in corresponding out-ports.

The sets of in-ports and out-ports of N' are $\mathbf{inP'} = \{P'_1, \ldots, P'_n\}$ and

$\mathbf{outP}' = \{Q'_1, \ldots, Q'_m\}$ respectively. Similarly, the sets of in-ports and out-ports of N'' are $\mathbf{inP}'' = \{P''_1, \ldots, P''_n\}$ and $\mathbf{outP}'' = \{Q''_1, \ldots, Q''_m\}$ respectively. Let $h_{in} : \mathbf{inP}' \to \mathbf{inP}''$ and $h_{out} : \mathbf{outP}' \to \mathbf{outP}''$ be, respectively, bijections defining one-to-one correspondences between in(out)-ports of N' and N'', such that $h_{in}(P'_i) = P''_i$ for all $1 \leq i \leq n$, and $h_{out}(Q'_j) = Q''_j$ for all $1 \leq j \leq m$.

By firing the transition T in N'', we get a marking M'' where $M''(Q''_j) = \{(v''_j, t''_j)\}$ for all $Q''_j \in \mathbf{outP}''$. In such a marking the token $K''_j = (v''_j, t''_j)$ in Q''_j has token value $v''_j = f(v_1, \ldots, v_n)$ and token time t''_j, where $l \leq t''_j \leq u$.

Since $M_0(P) = \emptyset$, by firing T_1 in N', we obtain a marking M where P is the only place marked in N' ($M(P) = \{(v, t)\}$) with a token that has token value $v = f_1(v_1, \ldots, v_n)$ and token time t, where $l_1 \leq t \leq u_1$. Then, by firing T_2 in N'', we obtain a marking M' where $M'(Q'_j) = \{(v'_j, t'_j)\}$ for all $Q'_j \in \mathbf{outP}'$. In this marking the token $K'_j = (v'_j, t'_j)$ in Q'_j has token value $v'_j = f_2(v) = f_2(f_1(v_1, \ldots, v_n))$ and token time t'_j, where $l_2 + t \leq t'_j \leq u_2 + t$, that is, $l_1 + l_2 \leq t'_j \leq u_1 + u_2$. Since $f = f_2 \circ f_1$, $l = l_1 + l_2$, and $u = u_1 + u_2$, we have $v'_j = f(v_1, \ldots, v_n)$ and $l \leq t'_j \leq u$.

Therefore, for all $K'_j = (v'_j, t'_j)$ in $Q'_j \in \mathbf{outP}'$ and all $K''_j = (v''_j, t''_j)$ in $Q''_j \in \mathbf{outP}''$: a) $v'_j = v''_j$; b) for every t'_j there exists t''_j such that $t'_j = t''_j$, and vice versa. Hence the nets N' and N'' shown in Figure B.1 are total-equivalent. □

B.2 NP-hardness of MSMU (Monoprocessor Scheduling to Maximize Utility)

In order to prove that the problem of static scheduling for monoprocessor real-time systems with hard and soft tasks (Problem 7.2 as formulated in Section 7.2) is **NP**-hard, we first turn such an optimization problem into a decision problem as follows.

PROBLEM B.1 (Monoprocessor Scheduling to Maximize Utility—MSMU) Given a set $\mathbf{T}$ of tasks, a directed acyclic graph $G = (\mathbf{T}, \mathbf{E})$ defining precedence constraints for tasks, expected and worst-case durations τ_i^{e} and τ_i^{wc} for each task $T_i \in \mathbf{T}$, subsets $\mathbf{H} \subseteq \mathbf{T}$ and $\mathbf{S} \subseteq \mathbf{T}$ of hard and soft tasks respectively ($\mathbf{H} \cap \mathbf{S} = \emptyset$), a hard deadline d_i for each task $T_i \in \mathbf{H}$, a non-increasing utility function $u_i(t_i)$ for each task $T_i \in \mathbf{S}$ (t_i is the completion time of T_i), and a constant K; does there exist a monoprocessor schedule σ (a bijection $\sigma : \mathbf{T} \to \{1, 2, \ldots, |\mathbf{T}|\}$) such that $\sum_{T_i \in \mathbf{S}} u_i(t_i^{\mathrm{e}}) \geq K$, $t_i^{\mathrm{wc}} \leq d_i$ for all $T_i \in \mathbf{H}$, and $\sigma(T) < \sigma(T')$ for all $(T, T') \in \mathbf{E}$?

In order to prove the **NP**-hardness of MSMU, we transform a known **NP**-hard problem into an instance of MSMU. We have selected the problem Scheduling to Minimize Weighted Completion Time (SMWCT) [GJ79] for this purpose. The formulation of SMWCT is shown below.

PROBLEM B.2 (Scheduling to Minimize Weighted Completion Time—SMWCT) Given a set $\mathbf{T}$ of tasks, a partial order $\lessdot$ on $\mathbf{T}$, a duration τ_i and a weight w_i for each task $T_i \in \mathbf{T}$, and a constant K; does there exist a monoprocessor schedule σ (a bijection $\sigma : \mathbf{T} \to \{1, 2, \ldots, |\mathbf{T}|\}$) respecting the precedence constraints imposed by $\lessdot$ such that $\sum_{T_i \in \mathbf{T}} w_i t_i \leq K$ (t_i is the completion time of T_i)?

We prove that MSMU is **NP**-hard by transforming SMWCT (known to be **NP**-hard [GJ79]) into MSMU. Let $\Pi = \{\mathbf{T}, \lessdot, \{\tau_1, \ldots, \tau_{|\mathbf{T}|}\}, \{w_1, \ldots, w_{|\mathbf{T}|}\}, K\}$ be an arbitrary instance of SMWCT. We construct an instance $\Pi' = \{\mathbf{T}', G(\mathbf{T}', \mathbf{E}'), \{\tau_1'^{\mathrm{e}}, \ldots, \tau'^{\mathrm{e}}_{|\mathbf{T}'|}\}, \{\tau_1'^{\mathrm{wc}}, \ldots, \tau'^{\mathrm{wc}}_{|\mathbf{T}'|}\}, \mathbf{H}', \mathbf{S}', \{d'_1, \ldots, d'_{|\mathbf{H}'|}\}, \{u'_1(t'_1), \ldots, u'_{|\mathbf{S}'|}(t'_{|\mathbf{S}'|})\}, K'\}$ of MSMU as follows:

- $\mathbf{H}' = \emptyset$
- $\mathbf{S}' = \mathbf{T}' = \mathbf{T}$
- $(T'_i, T'_j) \in \mathbf{E}'$ iff $T_i \lessdot T_j$
- $\tau_i'^{\mathrm{e}} = \tau_i'^{\mathrm{wc}} = \tau_i$ for each $T'_i \in \mathbf{T}'$
- the utility function $u'_i(t'_i)$ for each $T'_i \in \mathbf{T}'$ is defined as $u'_i(t'_i) = w_i(C - t'_i)$, where $C = \sum_{T_i \in \mathbf{T}} \tau_i$
- $K' = (C \sum_{T_i \in \mathbf{T}} w_i) - K$

To see that this transformation can be performed in polynomial time, it suffices to observe that $\mathbf{T}'$, $\{\tau_1'^{\mathrm{e}}, \ldots, \tau'^{\mathrm{e}}_{|\mathbf{T}'|}\}$, $\{\tau_1'^{\mathrm{wc}}, \ldots, \tau'^{\mathrm{wc}}_{|\mathbf{T}'|}\}$, and K' can be obtained in $\mathcal{O}(|\mathbf{T}|)$ time, $G(\mathbf{T}', \mathbf{E}')$ can be constructed in $\mathcal{O}(|\mathbf{T}| + |\lessdot|)$ time, and all the utility functions $u'_i(t'_i)$ can be obtained in $\mathcal{O}(|\mathbf{T}|)$ time. What remains to be shown is that Π has a schedule for which $\sum_{T_i \in \mathbf{T}} w_i t_i \leq K$ if and only if Π' has a schedule for which $\sum_{T'_i \in \mathbf{T}'} u'_i(t'_i) \geq K'$.

We show next that the schedule that minimizes $\sum_{T_i \in \mathbf{T}} w_i t_i$ for Π is exactly the one that maximizes $\sum_{T'_i \in \mathbf{T}'} u'_i(t'_i)$ for Π'. Note that, due to the transformation we described above, the set of tasks is the same for Π and Π', and the precedence constraints for tasks is precisely the same in both cases. We assume that σ is the schedule respecting the precedence constraints in Π that minimizes $\sum_{T_i \in \mathbf{T}} w_i t_i$ and that K is such a minimum. Observe that σ also respects the precedence constraints in Π'. Moreover, since $\tau_i'^{\mathrm{e}} = \tau_i$ for each $T'_i \in \mathbf{T}'$, the completion time t'_i of every task T'_i, when we use σ as schedule in Π', is the very same as t_i and thus:

$$\sum_{T'_i \in \mathbf{T}'} u'_i(t'_i) = \sum_{T_i \in \mathbf{T}} u'_i(t_i)$$

$$
\begin{aligned}
&= \sum_{T_i \in \mathbf{T}} w_i (C - t_i) \\
&= C \sum_{T_i \in \mathbf{T}} w_i - \sum_{T_i \in \mathbf{T}} w_i t_i \\
&= C \sum_{T_i \in \mathbf{T}} w_i - K \\
&= K'
\end{aligned}
$$

Since $C \sum_{T_i \in \mathbf{T}} w_i$ is a constant value that does not depend on σ and $\sum_{T_i \in \mathbf{T}} w_i t_i = K$ is the minimum for Π, we conclude that $(C \sum_{T_i \in \mathbf{T}} w_i) - K = K'$ is the maximum for Π', in other words, σ maximizes $\sum_{T'_i \in \mathbf{T}'} u'_i(t'_i)$.

Hence MSMU is **NP**-hard. □

B.3 MSMU (Monoprocessor Scheduling to Maximize Utility) Solvable in $\mathcal{O}(|\mathbf{S}|!)$ Time

The optimal solution to Problem 7.2 can be obtained in $\mathcal{O}(|\mathbf{S}|!)$ time by considering only permutations of soft tasks (recall $\mathbf{S}$ is the set of soft tasks). This is so because a schedule that sets soft tasks as early as possible according to the order given by a particular permutation S of soft tasks is the one that produces the maximal utility among all schedules that respect the order given by S.

The proof of the fact that by setting soft tasks as early as possible according to the order given by S (provided that there exists at least one schedule that respects the order in S and guarantees hard deadlines) we get the maximum total utility for S is as follows.

Let σ be the schedule that respects the order of soft tasks given by S (that is, $1 \leq i < j \leq |\mathsf{S}| \Rightarrow \sigma(\mathsf{S}_{[i]}) < \sigma(\mathsf{S}_{[j]})$) and such that soft tasks are set as early as possible (that is, for every schedule σ', different from σ, that obeys the order of soft tasks given by S and respects all hard deadlines in the worst case, $\sigma'(\mathsf{S}_{[i]}) > \sigma(\mathsf{S}_{[i]})$ for some $1 \leq i \leq |\mathsf{S}|$). Take one such σ'. For at least one soft task $T_j \in \mathbf{S}$ it holds $\sigma'(T_j) > \sigma(T_j)$, therefore $t'^{\mathrm{e}}_j > t^{\mathrm{e}}_j$ (t'^{e}_j is the completion time of T_j when we use σ' as schedule while t^{e}_j is the completion time of T_j when σ is used as schedule, considering in both cases expected duration for all tasks). Thus $u_j(t'^{\mathrm{e}}_j) \leq u_j(t^{\mathrm{e}}_j)$ because utility functions for soft tasks are non-increasing. Consequently $U' \leq U$, where U' and U are the total utility when using, respectively, σ' and σ as schedules. Hence we conclude that no schedule σ', which respects the order for soft tasks given by S, will yield a total utility greater than the one by σ.

Since the schedule that sets soft tasks as early as possible according to the order given by S gives the highest utility for S, it is needed to consider only

permutations of soft tasks in order to solve optimally MSMU (Problem 7.2). Hence MSMU is solvable in $\mathcal{O}(|\mathbf{S}|!)$ time. □

B.4 *Interval-Partitioning Step* Solvable in $\mathcal{O}((|\mathbf{H}|+|\mathbf{S}|)!)$ Time for Monoprocessor Systems

The *interval-partitioning step* is an important phase in the process of finding the optimal set of schedules and switching points, as formulated by Problem 7.4 and discussed in Subsection 7.3.3.

For monoprocessor systems, the interval-partitioning step can be carried out in $\mathcal{O}((|\mathbf{H}| + |\mathbf{S}|)!)$ time, with $\mathbf{H}$ and $\mathbf{S}$ denoting, respectively, the set of hard tasks and the set of soft tasks. The rationale is that the best schedule, for a given permutation HS of hard and soft tasks, is obtained when we try to set the hard and soft tasks in the schedule as early as possible respecting the order given by HS.

The proof of the fact that by setting hard and soft tasks as early as possible according to the order given by HS (provided that there exists at least one schedule that respects the order in HS and guarantees hard deadlines) we get the best schedule for HS is as follows

Let σ be the schedule that respects the order of hard and soft tasks given by HS (that is, $1 \leq i < j \leq |\mathsf{HS}| \Rightarrow \sigma(\mathsf{HS}_{[i]}) < \sigma(\mathsf{HS}_{[j]})$) and such that hard and soft tasks are set as early as possible (that is, for every schedule σ', different from σ, that obeys the order of hard and soft tasks given by HS and guarantees meeting all hard deadlines, $\sigma'(\mathsf{HS}_{[i]}) > \sigma(\mathsf{HS}_{[i]})$ for some $1 \leq i \leq |\mathsf{HS}|$). Take one such σ'. For at least one task $T_j \in \mathbf{H} \cup \mathbf{S}$ it holds $\sigma'(T_j) > \sigma(T_j)$. We study two situations:

(a) $T_j \in \mathbf{S}$: in this case $t'^{\mathrm{e}}_j > t^{\mathrm{e}}_j$ (t'^{e}_j is the completion time of T_j when we use σ' as schedule while t^{e}_j is the completion time of T_j when σ is used as schedule, considering in both cases expected duration for the remaining tasks). Thus $u_j(t'^{\mathrm{e}}_j) \leq u_j(t^{\mathrm{e}}_j)$ because utility functions for soft tasks are non-increasing. Consequently $\hat{U}'(t) \leq \hat{U}(t)$ for every possible completion time t, where $\hat{U}'(t)$ and $\hat{U}(t)$ correspond, respectively, to σ' and σ.

(b) $T_j \in \mathbf{H}$: in this case $t'^{\mathrm{wc}}_j > t^{\mathrm{wc}}_j$ (t'^{wc}_j is the completion time of T_j when we use σ' as schedule while t^{wc}_j is the completion time of T_j when σ is used as schedule, considering in both cases worst-case duration for the remaining tasks). Thus there exists some $t^{\times}$ for which σ guarantees meeting hard deadlines whereas σ' does not. Recall that we include the information about potential hard deadline misses in the form $\hat{U}'(t) = -\infty$ if following σ', after completing the current task at t, implies

potential hard deadline violations. Accordingly $\hat{U}'(t) \leq \hat{U}(t)$ for every possible completion time t.

We conclude from the preceding facts that every schedule σ', which respects the order for hard and soft tasks given by HS, yields a function $\hat{U}'(t)$ such that $\hat{U}'(t) \leq \hat{U}(t)$ for every t, and therefore σ is the best schedule for the given permutation HS. This means therefore it is needed to consider only permutations of hard and soft tasks during the interval-partitioning step (for monoprocessor systems) and the problem is hence solvable in $\mathcal{O}((|\mathbf{H}|+|\mathbf{S}|)!)$ time. □

B.5 Optimality of EDF for Non-Preemptable Tasks with Equal Release Time on a Single Processor

With regard to the problems addressed in Chapter 8, an EDF policy gives the optimal execution order for non-preemptable tasks with equal release time and running on a single processor. In order to prove this statement, we show that an EDF execution order is the one that least constrains the space of solutions.

The task execution order affects only the time constraints (see Equations (8.7) and (8.17) in Problems 8.1 and 8.4 respectively), that is, the constraints $t_i \leq d_i$, where t_i is the completion time of task T_i and d_i is its deadline.

Let us assume that $d_i \leq d_j$ if $i < j$. The execution order according to an EDF policy is thus $T_1T_2 \ldots T_i \ldots T_j \ldots T_n$ and the corresponding time constraints for tasks T_i and T_j are $t_i \leq d_i$ and $t_j \leq d_j$ respectively.

We consider now a non-EDF execution order $T_1T_2 \ldots T_j \ldots T_i \ldots T_n$ with similar time constraints $t_j \leq d_j$ and $t_i \leq d_i$. It must be noted, however, that according to this non-EDF order T_j executes before T_i and hence it follows that $t_j \leq d_i$. Since $d_i \leq d_j$, the constraint $t_j \leq d_j$ is redundant with respect to the constraint $t_j \leq d_i$. Therefore the time constraints for tasks T_j and T_i are actually $t_j \leq d_i$ and $t_i \leq d_i$.

Thus a non-EDF execution order ($t_j \leq d_i$, $t_i \leq d_i$) imposes more stringent constraints than the EDF order ($t_i \leq d_i$, $t_j \leq d_j$) because $d_i \leq d_j$. This is illustrated in Figure B.2. Note that Figure B.2 refers to the space of possible Voltage/Optional-cycles assignments (given a fixed task execution order) and not to the space of possible execution orders.

The space of possible solutions for any non-EDF execution order is contained in the space of solutions corresponding to the EDF order, which means that EDF is the least restrictive task execution order. Since the so-

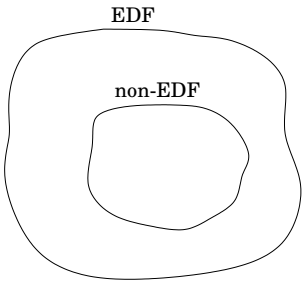


Figure B.2: Space of solutions (V/O assignments)

lution space generated by the EDF order is the largest, we conclude that an execution order fixed according the EDF policy is optimal. □