UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE
Spécialité : **Informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Jean-François Kempf

Thèse dirigée par **Oded Maler**
et codirigée par **Marius Bozga**

préparée au sein **Verimag**
et de **École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

On Computer-Aided Design-Space Exploration for Multi-Cores

Thèse soutenue publiquement le **29 Octobre 2012**,
devant le jury composé de :

Kim G. Larsen
Aalborg University, Rapporteur
Bruce Krogh
Carnegie Mellon, Rapporteur
Boudewijn R. Haverkort
University of Twente, Examinateur
Eugene Asarin
Université Paris Diderot Paris 7, Examinateur
Fahim Rahim
Atrenta, Examinateur
Oded Maler
CNRS, Directeur de thèse
Marius Bozga
CNRS, Co-Directeur de thèse

On Computer-Aided Design-Space Exploration for Multi-Cores

Jean-François KEMPF

29 Octobre 2012

Acknowledgements

I want to thank all the people who have contributed to the achievement of this thesis. I thank especially, Oded Maler, my supervisor who give me the opportunity to carry out this work in his team. His thoughtful advice, his support as well as the freedom to work he gave me, allowed me to work in excellent conditions. I also want to thank Marius Bozga, my co-supervisor, for his support, his availability and patience during our fruitful discussions. I thank Olivier for his contribution to this work, especially for all advice he gave me during the development of DeSpEx. I also thank the members of the ATHOLE project who contributed to the achievement of this work. Thanks to all other members of Verimag who provide a very good working atmosphere. Finally I would also like to thank my family, without whose help I certainly would not be here today.

Contents

Chapter 1

Introduction

This thesis is concerned with models, analysis techniques and tools intended to aid hardware and software designers in exploring their systems design space and finding efficient deployments of application programs on multi-processor platforms. The thesis can be viewed as a confluence point between several academic and industrial research currents in a domain which is very important practically and for which no agreed upon unified theoretical framework exists. To better understand the context of this thesis we mention two of the major driving forces behind the thesis.

Formal Analysis of Timed Systems

Formal verification is a kind of exhaustive simulation of abstract automaton-based models of software and hardware that capture mostly concurrency and synchronization features. Timed models such as timed automata, add quantitative timing information to the models and allow to reason about delays, execution times, deadlines and other *performance*-related measures. Over the years, a lot of work, at Verimag and elsewhere, explored the applicability of these models to scheduling for embedded (and other) systems and circuits. Although the expressivity of timed automata can be used to model more complex situations than what is possible using classical real-time models, their standard analysis technique (forward computation of reachable states and zones) does not scale up and for the the time being cannot be applied to systems beyond a rather low threshold of size and complexity. Moreover, this type of analysis is *worst-case* oriented and is not always suitable for *soft* real-time applications where we care more about the *average* performance. There is a recent trend in verification, sometimes called *statistical model checking*, which replaces verification-style exhaustive exploration by Monte-Carlo simulation. In the timed context this means, implicitly or explicitly, to refine temporal uncertainty intervals into distributions supported by those intervals, resulting in some kind of continuous-time “non Markovian” stochastic processes.[1] We use such models in this thesis in two ways. First we do statistical simulation on high-level models of applications running on multi-core platforms and secondly, we develop new semi-analytic techniques for performance evaluation and optimization for duration probabilistic automata that can model a class problems of scheduling under stochastic uncertainty.

The ATHOLE Project

The thesis was aligned, temporally and conceptually, with the ATHOLE project, coordinated by ST Microelectronics, with the participation of CEA-LETI, THALES and CWS. Initially the project was focused on the xSTream architecture but around mid-time, xSTream development has been freezed and the project shifted to Plateform 2012 (P2012). The role of Verimag in the project

[1] Tradition aside, the term non-Markovian is misleading. These processes are Markovian (state-based) with respect to extended states that include clock values.

was to apply its expertise in timed systems to help automating deployment decisions (mapping, scheduling) and ease the difficult task of exploiting multi-cores efficiently. The exposure to industrial and quasi-industrial practices and cultures was an opportunity to compare the modeling and analysis methods used by *system builders* (hardware designers and programmers) and those used by *model builders* (in verification and performance analysis). The tool described in this thesis, which brings abstract modeling concepts and analysis techniques to these application domains, was a *major deliverable* in the ATHOLE project.

The thesis is organized as follows:

- Chapter 2 is a very non-exhaustive survey on multi-core embedded systems and the modeling approaches used in various levels of abstraction during the design flow, with emphasis on performance evaluation;
- Chapter 3 is a short introduction to timed automata, symbolic reachability computation on zones and the IF toolset on top of which our tool is implemented;
- Chapter 4 develops a new technique for computing the performance (expected termination time) of schedulers for acyclic scheduling problems (such as job-shop or task-graph) where the duration of each task is considered to be uniformly distributed over a finite interval;
- Chapter 5 attacks the more ambitious *synthesis* problem and develops a dynamic programming approach to derive expected-time optimal schedulers. These two sections constitute the major theoretical contribution of the thesis and are accompanied by a prototype implementation;
- Chapter 6 is the core of the thesis. It presents our abstract modeling framework for *application programs* (modeled as extended task-graphs), *input generators* (the processes that generate new tasks instances subject to temporal and logical constraints), *execution platforms* (very abstract models of processors, data transfer mechanisms and memories) and *deployment* (mapping and scheduling policies). Each modeled object is transformed into a timed automaton and the composition of these automata represents all the possible systems behavior;
- Chapter 7 describes the tool *DESPEX, the DEsign-SPace EXplorer*, its architecture and the type of analysis it provides, namely symbolic reachability computation and Monte-Carlo discrete event simulation, followed by statistical and visual trace analysis;
- Chapter 8 describes three case studies treated by the tool. The first is a toy problem used to demonstrate the advantage of timed automata verification over corner-case simulation. The second is a video application provided by ST, evaluated for power and performance on P2012 using simulation, and the third is a signal processing application provided by Thales and evaluated on xSTream;
- Chapter 9 concludes the thesis and sketches some ideas for future work.

Chapter 2

Multi-core embedded system

Current embedded applications necessitate intensive computation and data communication which are difficult to handle by a single processor architecture. The performance demanded by these applications, like for example multimedia applications, requires the use of multi-processors architectures in a single chip endowed with complex communication infrastructures, such as hierarchical buses or networks on chips (NoCs). That's why Multiprocessors System on Chip (MPSoC) architectures have become a very attractive solution for the consumer multimedia embedded market [197].

System on Chip (SoC) represents the integration of different computing elements and other electronics subsystems into a single integrated circuit. MPSoC [196] are SoC that may contains one or more types of computing subsystems, memories, I/O devices and other peripherals. Additionally, heterogeneous components are exploited to meet the tight performance and cost constraints. This trend of building heterogeneous MPSoC will be even accelerated due to current embedded application requirements. ITRS (International Technology Roadmap for Semiconductors) roadmap [1] predicts that the number of processing engines on future MPSoC platforms will increase rapidly (Fig-2.1), which will introduce a huge complexity into the software development process for such complex platforms. Making the potential parallelism of the applications more explicit so as to exploit the available processing engines, and then efficiently deploying it on the underlying hardware are the new and big challenges for MPSoC software developers.

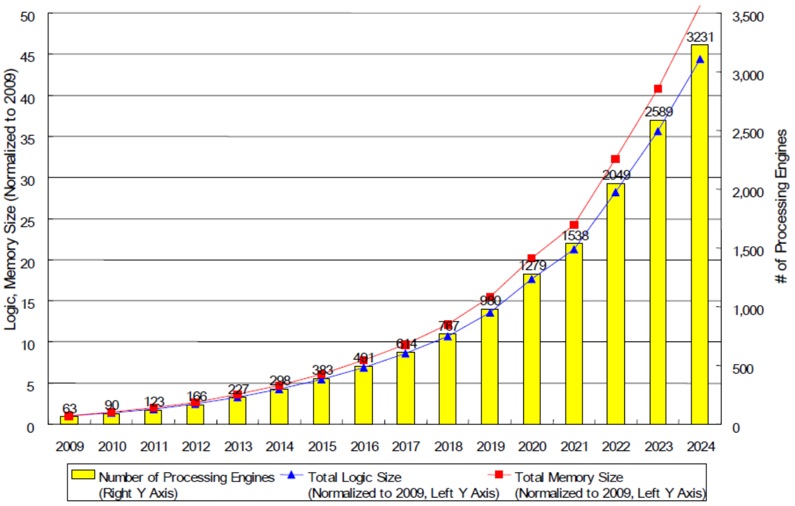


Figure 2.1: Consumer Portable Design Complexity Trends [1]

We will discuss current MPSOC architectures and their different components. Then we will survey the typical embedded systems design flow and see that performance evaluation is done at different level of granularity. Here we are more interested in performance estimation at early design stage. We will present typical modeling techniques and discuss their utility for performance estimation.

2.1 MPSOC Architecture

Heterogeneous MPSoC architecture can be represented as a set of *processing elements* (PE) or components which interact via a communication network (Fig-2.2). The PE can be of different type like processors (DSP, microcontroller, ASIP, . . .) or memory elements (caches, DRAM . . .) connected trough different communication schemes. This type of heterogeneous architecture provides highly concurrent computation and flexible programmability. Heterogeneous MPSoC are composed of different kind of PE as opposed to homogeneous MPSoC where the same type of element is instantiated several times.

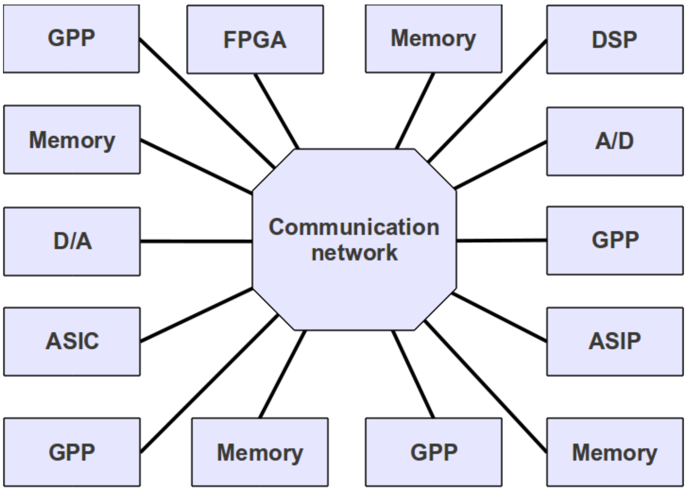


Figure 2.2: MPSOC Architecture

Some heterogeneous platforms from the major semiconductor vendors such as NXP Nexperia [157], TI OMAP [185] or ST Nomadik [178] are already available. On the other hand, homogeneous ones were pionnered by the Lucent Daytona architecture [3, 196].

The literature relates mainly two kinds of organizations for multiprocessor architectures, shared memory and message passing [68]. Heterogeneous MPSoCs generally combine both models and integrate a massive number of processors on a single chip. Future heterogeneous MPSoC will be made of few heterogeneous subsystems, where each subsystem includes a massive number of identical processors to run a specific software stack [113].

We will now give an overview of the different hardware components in MPSoC.

2.1.1 Processing Elements

Processing elements range in a spectrum between generality and specificity. We can classify them into two major type: General Purpose Processor (GPP) and Application Specific Processors (ASP). GPPs are flexible because they are built to be used in a variety of applications with different specifications. Changing functionalities or improving a system becomes relatively easy when you only need to change a software program. On the other hand ASPs are designed to execute exactly one program, increasing performance and reducing power consumption. However, flexibility and reprogramming is limited because it is designed as a custom digital circuit dedicate to restricted application range. We can distinguish different sub-classes of ASP, such as ASIC (Application Specific Integrated Circuit) where algorithms are completely implemented in hardware and programmable microprocessors like DSP (Digital Signal Processor) for extensive numerical real-time computation, or ASIP (Application Specific Instruction Set Processor) where hardware and instruction set are designed together for one particular application.

2.1.2 Memory Organization

The memory subsystem is an important component of system designs that can benefit from customization. Unlike general purpose processors where a standard cache hierarchy is employed, the memory hierarchy of embedded systems can be tailored in various ways. The memory can be selectively cached. The cache line size can be determined by the application. The designer can opt to discard the cache completely and choose specialized memory configurations such as FIFOs and stream buffers and so on.

The memory is a bottleneck in a computer system since the memory speed cannot keep up with the processor speed and the gap is becoming larger and larger. Memory hierarchy issues are among the most important concerns in designing application-driven embedded systems. A typical embedded system architecture consists of processor cores, reconfigurable hardware, instruction cache, data cache, on-chip scratch memory and on-chip or off-chip DRAM.

The cache is a special high-speed, low volume memory that is in close proximity to the processing hardware that it is reserved for. It can be seen as an interface between the processor and the off-chip memory. Embedded architectures include both data and instruction caches.

Scratch-Pad memory refers to data memory residing on-chip, that is mapped into an address space disjoint from the off-chip memory, but connected to the same address and data buses. Both the cache and Scratch-Pad memory (usually SRAM) allow fast access to their residing data, much faster than accessing off-chip memory. Off-chip memory (usually DRAM) refers to a highest volume memory residing far from the processing element. The main difference between the Scratch-Pad SRAM and data cache is that the SRAM guarantees a single-cycle access time, whereas an access to the cache is subject to cache misses.

2.1.3 Interconnect

The interconnect is a resource shared between various hardware components. Its role is to transmit data from a *source* to a *destination* component, thus implementing the communication network. Basically the network component is characterized by two metrics:

- Latency: total time to transfer a quantity of data from the source to the destination component;

- Bandwidth: amount of data that can be transmitted per time unit.

On-chip communication architectures can be divided into the following three main classes [126]:

1. Point to point interconnect: pairs of processing elements communicate directly over dedicated physically-wired connections [33];

2. Bus architectures [151] : long wires are grouped together to form a single physical entity. One can find different bus based architecture:

    - FPGA-like Bus [60] with programmable interconnects using static network;

- Arbitrated Bus [110] with time-shared multiple core connectivity;
- Hierarchical Bus [11, 12, 195] combining multiple buses using bus bridges.

3. Network on Chip (NoC): Communication is achieved by sending message packets between blocs using an on-chip packet-switched network. NoC is a relatively new chip design paradigm concurrently proposed by many research group [172, 127, 28]. A survey of research and practices of Network-on-Chip can be found in [40].

A component which can be seen as part of the interconnect is Direct Memory Access (DMA). It is a device that can control the memory system without using the CPU. On a specified stimulus, the module will move blocks of data from one memory location to another. DMA is essential for embedded systems since it allows large quantities of information to be transferred to or from memory, while the processor can be doing something else.

2.2 Design Flow

Systems-on-chip require the creation and use of radically new design methodologies because some of the key problems in SoC design lie at the *boundary* between *hardware* and *software*.

Classic SoC design flows imply a long design cycle because most of them rely on a sequential approach where complete hardware architecture should first be developed before software could be designed on top of it. This long design cycle is not acceptable because of time to market constraints.

Due to their complexity, the design of embedded systems requires modeling phases at different abstraction levels (Fig-2.3). At each level, many different activities are required during the design flow: specification and functional modeling, performance modeling, design and synthesis, validation and verification. Because the real product is not available before the development task is completed, all activities operate on models. According to [112]:

> *A model is a simplification of another entity, which can be a physical thing or another model. The model contains exactly those characteristics and properties of the modeled entity that are relevant for a given task. A model is minimal with respect to a task if it does not contain any other characteristics than those relevant for the task.*

A model is therefore an abstraction of one entity and may be defined differently according to its use (functional validation, performance evaluation). There is an increasing use of early system-level modeling, even if it would not contain the entire hardware architecture, but only a subset of components which are sufficient to allow some level of software verification on the hardware before the full hardware is available, thus reducing the sequential nature of the design process. The use of high-level programming models to abstract hardware/software interfaces is the key enabler for concurrent hardware and software designs. This abstraction allows to separate low-level implementation issues from high-level application programming (Fig-2.3). It also smoothens the design flow and eases the interaction between hardware and software designers. It acts as a contract between hardware and software teams that may work concurrently. Additionally, this scheme eases the integration phase since both hardware and software have been developed to comply with a well-defined interface.

Furthermore programming an application-specific heterogeneous multi-processor architectures becomes one of the key issues for MPSoC, because of two *contradictory* requirements:

- Reducing software development cost and overall design time: needs high level abstract models;
- Improving performance: needs accurate and precise low level models.

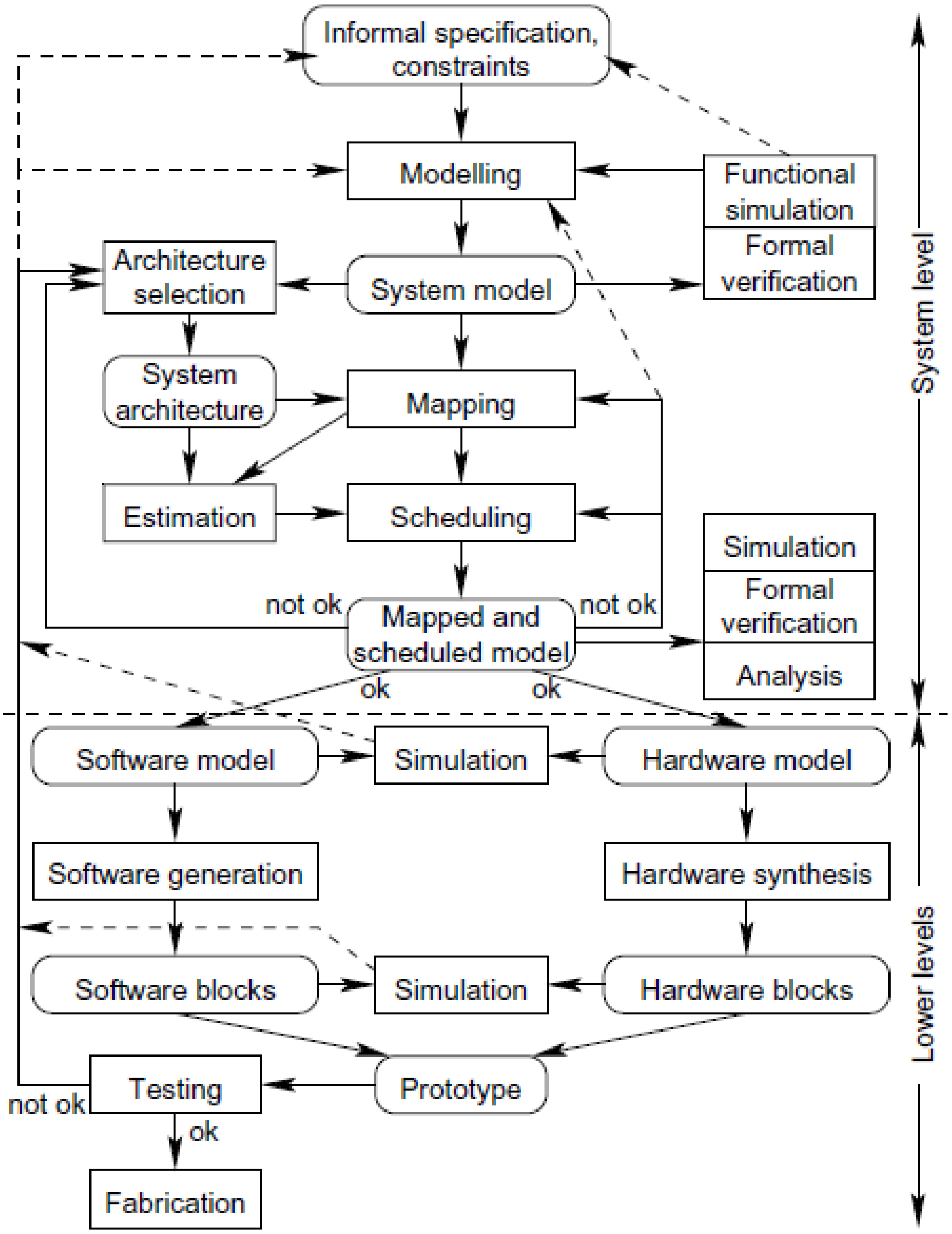


Figure 2.3: Generic Embedded System Design Flow

All models are simplification of reality, an exact copy of a real product can only be the real product itself. So there is always a trade-off as to what level of detail is included in the model, too little detail implies a risk of missing relevant informations and giving wrong predictions, too much detail makes models overly complicated and thus difficult to understand or analyze.

2.2.1 Low Level Modeling

Modeling the hardware is an important phase in the design of an embedded system. It is achieved by developing virtual prototypes that can be fully functional software models of the physical hardware, allowing accurate simulation, design verification and automatic layout generation. The advantage of virtual prototypes lies in their early availability in the development cycle. This way software developers can begin earlier with development and verification of the hardware dependent software.

Platform

To give an overview about the different levels of abstraction and to illustrate their interrelation to the different specification domains, the Y-Chart of Fig-2.4 proposed by Gajeski is commonly used [96].

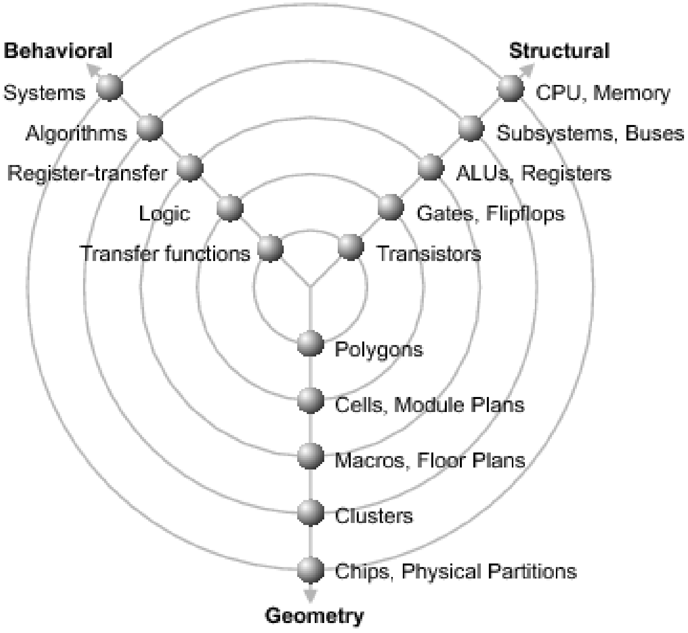


Figure 2.4: Gajski-Kuhn Y-chart

The Y-chart model distinguishes between 3 abstraction domains

- Behavioral (functional): describes the temporal and functional behavior of a system without any reference to the particular way in which it is implemented.
- Structural: deals with how the system is composed of interconnected hierarchical subsystems.
- Physical/Geometrical: specifies how the system is laid out in terms of physical placement in space and physical characteristics without any elements of functionality.

Each of these domains can also be divided into levels of abstraction:

- Transistor level
- Logical level
- Register-transfer level (RTL)
- Algorithmic level
- System level

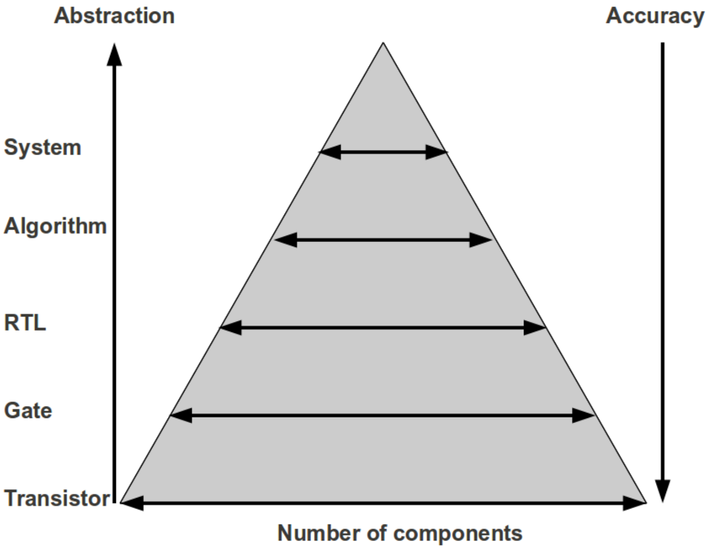


Figure 2.5: Abstraction levels

Each level defines an abstract platform, called *virtual prototype* on which simulation can be done more or less accurately (cycle accurate, instruction accurate, transaction accurate . . .). As depicted in Fig-2.5, highest accuracy is obtained by lowest abstraction level but implies a higher modeling effort as the number of components is huge. Simulation or timing analysis on such models is also a major constraint. Typically a cycle-accurate processor model can be simulated at a rate of between 50 and 1000 instructions per second, while execution on the real hardware is on the order of millions of instructions per second.

Hardware Description Languages (HDL) are used to model systems at a low level of abstraction, typically at RTL, and are used to do simulation with high precision or to automatically generate the hardware layer. Tools like Verilog [184] or VHDL [176] are used to specify hardware in a textual format.

Applications

Software too can be described at different levels of abstraction. The lowest level is binary machine-executable code which is typically generated from a program written in a higher level programming language such as C. An application is then evaluated on a real platform if such exists, or by using a virtual platform. C code is compiled and translated automatically to assembly language and machine code.

More generally, applications are written using a programming model. The programming model specifies how application components are running in parallel and how they communicate including synchronization operations that are available to coordinate their activities. The programming model is usually embodied in a parallel language or a programming environment [68].

Examples of parallel programming models are as follows:

- Shared address space: communication is performed by posting data into shared memory locations, accessible by all the communicating processing elements;
- Data-parallel programming: several processing elements perform the same operations simultaneously, but on separate parts of the same data set;
- Message passing: when the communication is performed between a specific sender and a specific receiver.

Examples of such programming models are briefly described in the following:

- StreamIt is an example of programming model for streaming applications [183];
- MPI (message-passing interface) [154] is a message-passing library interface specification. It includes the specification of a set of primitives for point-to-point communication with message passing, collective communications, process creation and management, one-sided communications, and external interfaces;
- MCAPI (multi-core communications APIs) [65] defines a set of communication APIs for multi-core communications, to support lightweight, high performance implementations typically required in embedded applications;
- YAPI (Y-chart application programmer's interface) is an interface to write signal and stream processing applications as process networks, developed by Philips Research [75]. TTL (task transaction level interface) proposed in [188] is derived from YAPI;
- OpenCL (Open computing language) is an open standard for writing applications that execute across heterogeneous platforms consisting of CPUs, GPUs, and other processors, introduced by Khronos working group [136]. OpenCL provides parallel computing using task-based and data-based parallelism;
- OpenMP (open multi-processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran on many architectures [59].

Both hardware and software abstraction levels described previously are defined with too much detail for rapid evaluation at early stage of the design flow. It implies a high development effort before performances estimation can be done. The abstraction level close to our work is situated on top of what is generally defined as *system-level design*.

2.2.2 System-Level Design

In modern system-level design methodology, known as HW/SW co-design, the development of hardware platform and application software is done in parallel. Design space exploration needs separate application and architecture specifications. An explicit *mapping* step maps application components to architecture. HW/SW co-design includes various design problems including system specification, design space exploration, performance estimation, HW/SW co-verification, and system synthesis. This can be achieved at system level for early estimation and then models are refined at lower level for more accurate performance evaluation.

Modeling and simulation at high abstraction levels are used to increase and to simplify the development and validation of MPSoC at early design stage. For that we need abstract models of both software and hardware components. Defining such models, requires knowledge about hardware and software architecture details as well as execution environment constraints (timing, power consumption ...) at early design stages. This can be achieve by hardware-software co-designer based on profiling or past experiences.

Description languages have been developed as well at higher levels of abstraction. At system level, these language can describe abstractions of software and a high level component view of hardware. One can cite for example:

- SystemVerilog [180] is an extension of Verilog inheriting capabilities for synthesizable modules description (Verilog) and object oriented language abstraction, that allow the verification of complex systems;

- SpecC [88] is built on top of ANSI-C programming language and is intended for the specification and design of digital embedded systems, including hardware and software portions and supports concepts like behavioral and structural hierarchy, concurrency, communication, synchronization, state transitions, exception handling, and timing;
- SystemC transaction-level modeling (TLM) is another standardization work [92] which offers a set of standard APIs and a library, built on top of C++ programming language, that implements a foundation layer upon which interoperable transaction-level models can be built;
- AADL (Architecture Analysis and Design Language)[87] defines a language for describing both the software architecture and the execution platform architecture of performance-critical, embedded, real-time systems. It describes a system as a hierarchy of components with their interfaces and their interconnections, specifying both functional interfaces and aspects critical for performance;
- SysML [194] is based on an extension of the Unified Modeling Language (UML) and has views that deal with multiple aspects of the system: functional and behavioral, structural, performance, and slew of other models like cost and safety.

Underlying the specification of embedded systems there is the notion of *Model of computation and communication* which defines more formally how concurrent components interact with each other. A multitude of modeling formalisms have been applied to embedded system design. Typically, these formalisms strive for a maximum of precision, as they rely on a mathematical (formal) model. We will presents next the main classes of models used in the design of MPSoC.

2.3 Models of Computation for Embedded Systems Design

Modeling formalisms for embedded system design have been widely studied, and several reviews and textbooks about models of computation (MoCs) can be found in the literature [131, 173, 80, 187, 94]. Usage of formal models in embedded system design allows (at least) one of the following [173]:

- Unambiguously capture functionality of the required system;
- Verification of functional specification correctness with respect to its desired properties;
- Support synthesis onto a specific architecture and communication resources;
- Use different tools based on the same model (supporting communication among teams involved in design, producing, and maintaining the system).

Two basic types of MoCs can be differentiated: process based and state based MoCs [89].

Process-based MoCs describe the system behavior as a set of concurrent processes communicating with each other through message passing channels or shared memory facilities such as:

- Kahn process networks [116]: process are independent of each other and execute in parallel. Communication is done through uni-directional message passing channels incorporating buffers, which enable asynchrony;
- Dataflow model, processes are broken down into atomic blocks of execution, called actors, executing once all their inputs are available. On every execution, an actor consumes the required number of tokens on all of its inputs and produces resulting tokens on all of its outputs. In the same way as KPNs, actors are connected into a network using unbounded,

uni-directional FIFOs with tokens of arbitrary type. More formally, a dataflow network is a directed graph where nodes are actors and edges are infinite queues. Dataflow networks are deterministic and have the same termination semantics as KPNs. It is a basis for many commercial tools such as LabView [39] and Simulink [149];

- Synchronous DataFlow (SDF) [134] is a specialization of dataflow modeling, where production and consumption values on edges are not necessarily the same. Unlike dataflow, states do not influence the number of tokens that are produced and consumed in each firing cycle and these numbers can be specified a priori;
- Process Calculi :
  - Communicating Sequential Processes (CSP) [107, 52] allows the description of systems in terms of component processes that operate independently, and interact with each other solely through message-passing communication. CSP uses explicit channels for message passing, whereas actor systems transmit messages to named destination actors;
  - Calculus of Communicating Systems (CCS) [150]: the fundamental model of interaction is synchronous and symmetric, i.e. the partners act at the same time performing complementary actions;
  - Algebra of Communicating Processes (ACP) [34] focus on the algebra of processes, and sought to create an abstract, generalized axiomatic system for processes.

State based models are defined by a set of states and transitions, called state machines. States explicitly represents the memory state of a program and transitions which can be guarded over specific boolean conditions, represents the changes in the system behavior. Finite State Machines (FSM, automata) is one of the basic model for describing various type of application and is usually sequential, i.e it can only be in one state at a time. Two types of FSM exists: Moore type in which the outputs are determined solely by its current state [153] and Mealy type in which outputs are determined both by its current state and the current inputs.

Several extension of FSM have been proposed:

- Finite State Machine with DataPath (FSMD) introduces a set of variable in order to reduce the number of states;
- Hierarchical and concurrent finite states machines (HCFSM) incorporate notions of hiearchy and concurrency. Hierarchy is defined through the notion of superstates representing an enclosed state machine and communicating through shared variables, events or signals. Statecharts [104] are the most well known formalism;
- Co-Design Finite State Machines (CFSM) [61] connect individual sequentials elements in a global asynchronous network.

Combinations of different models have also been developed such as Program State Machine (PSM) [90] which can be seen as a combination of KPN and HCFSM. Petri nets [161] are directed graphs, similar to dataflows, with two types of nodes: places and transitions. Places correspond to the states of the program and transitions are the computational entities. A firing of the transition implies the consumption of tokens in the input places and output tokens in the output places.

The notion of time is extremely important in many of the modeling formalisms for embedded systems. Untimed MoC, like Petri nets, is based on data dependency only and neither a transition or the transfer of a token from one place to another takes a particular amount of time. Using a timed MoC reflects the intention of capturing the timing behavior of a component which could also influence its functional behavior. We can distinguish between continuous-time and discrete-time system according to the type of the time variables.

In discrete event MoC, events are sorted by a time stamp stating when they occur and are treated in chronological order. Transaction Level Modeling (TLM) is a discrete-event MoC built on top of SystemC, where modules interchange events with time stamps.

Synchronous models of computation divide the time axis into totally ordered slots and everything inside a slot occurs at the same time. This type of MoC is more suited for programming control and real-time systems. Esterel [36], Lustre [102] and Signal [30] are some existing synchronous languages.

Many of the untimed MoC presented previously have also been extended with timing information. One can cite for example, Timed automata [6] , that extend finite automata with clock variables which give timing constraints on the behavior of the system, or Timed Petri Nets [37] where a timed interval is associated with each transition.

Modeling implies also the notion of determinism. Deterministic systems produce one unambiguous output for a given input, but a system may not always react with the same outputs when confronted with the same inputs or inputs may not be precisely defined. For example communication times in distributed systems are hard to predict and may vary due to physical effects or interferences. This leads to non-deterministic models where systems may produce different behaviors. Quantitative properties concerning the different outputs are sometimes captured with stochastic systems. We will discuss non determinism and uncertainty in more detail in the next chapters.

2.4 Performance Evaluation

Performance analysis aims to assess and understand some quantitative properties at an early stage of the product development and is as important as functionality. In [148] Marwedel indicates five metrics for the evaluation of the efficiency of an embedded system:

- Power consumption
- Code size
- Run time efficiency
- Weight
- Cost

All these metrics can be subject to design requirements of the system, that is appropriate predictions of these characteristics are necessary in early design stage and can be considered objectives of early performance analysis. According to [152], design space exploration is the process of analyzing several implementation alternatives to identify an optimal solution. For efficient system-level design space exploration of complex embedded systems the separation-of-concerns concept has been introduced by [121]. Therefore, Y-chart design methodology [19, 122], depicted on Fig-2.6, became a popular basis for early design space exploration.

To perform quantitative performance analysis, application models are mapped onto the architecture model under consideration, whereafter performance of each application-architecture combination can be evaluated. Subsequently, the resulting performance numbers may inspire the designer to improve the architecture, adapt the application or modify its mapping.

Performance is often focus on the analysis of timing aspects e.g. how fast a system can react to events. However power consumption is, in some situation, as important as execution time, and we will focus in this thesis on performance analysis on these two metrics.

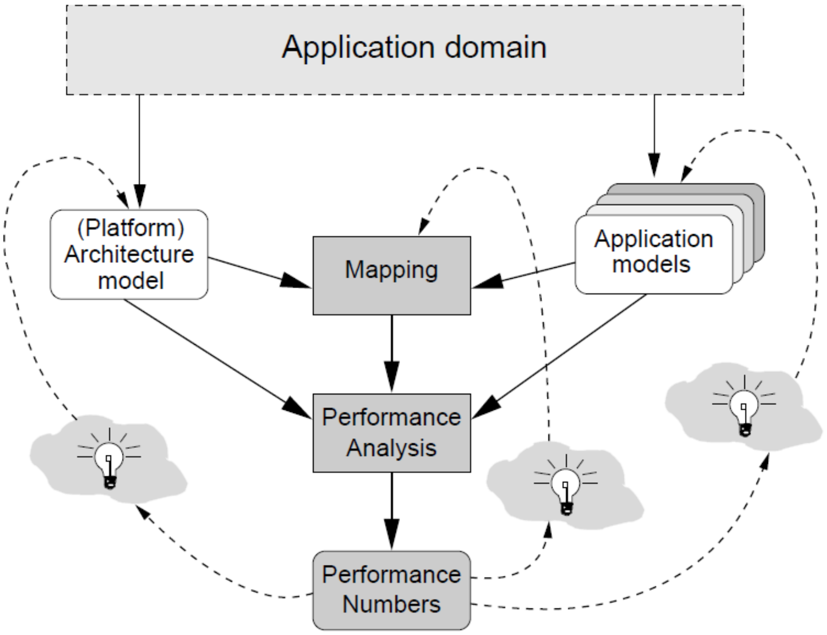


Figure 2.6: Y-chart based design space exploration (obtained from [162])

Very often, the timing behavior of an embedded system can be described by the time interval between a specified pair of events. Depending on the application domain timing properties can be more or less critical. Many embedded systems must meet real time constraints, that is, they must react to events within a specific time interval, called *deadline*. A real time constraint is called hard if its violation is considered a system failure, and it is called soft otherwise. First of all it is necessary to distinguish between the following terms (taken from [182]):

- Worst case and best case. The worst case and the best case are the maximal and minimal time interval between the arrival and termination events under all admissible system and environment states. The execution time may vary largely, owing to different input data and interference between concurrent system activities.

- Upper and lower bounds. Upper and lower bounds are quantities that bound the worst- and best-case behavior. These quantities are usually computed offline, that is, not during the runtime of the system.

- Statistical measures. Instead of computing bounds on the worst- and best-case behavior, one may also determine a statistical characterization of the runtime behavior of the system, for example, distributions, expected values, variances, and quantiles.

In hard real time systems, performance is hardwired into correctness, for example requiring that some deadline is *never* violated. For such systems we are more interested in upper bounds and worst case behavior whereas in soft real time system lower and upper-bounds represent very extreme cases and a more quantitative estimation will be more useful.

Performance evaluation is a key challenge in the analysis of MPSoC and can be broadly divided in two main approaches: formal methods and simulation-based approaches. Formal verification is the process of checking whether some properties are satisfied by using mathematical proofs. There are different type of formal verification:

- Model checking [63]: given an abstract model and a property to verify, typically expressed as a temporal logic formula, a model checker performs exhaustive exploration of the set of all possible states.

- Theorem proving: the system is modeled as a set of mathematical definitions in some formal mathematical logic. Properties of the system are derived as theorems that follow from these definitions by using standard results in mathematical logic [160].
- Equivalence checking: formulas for both the specification and the implementation are reduced to some canonical form (if one exists) by applying mathematical transformations. If their canonical forms are identical, then the specification and the implementation are said to be equivalent.

Simulation consists in exploring a model interactively or randomly, possibly using heuristics for choosing the visiting states. It is a technique of partial validation, i.e if no error is detected, this method increase the confidence in program correctness but can never ensure that all properties are satisfied. Discrete event simulation [56, 140] is widely used for evaluating performance of MPSoCs.

Timing Analysis

Timing requirements have been widely studied in the real time community where schedulability analysis techniques have been developed such as Rate Monotonic Analysis [141]. Most of these techniques are devoted to single-processor systems but have been extended to distributed systems [186] and more recently for fixed priority multiprocessors scheduling [97].

In the domain of communication networks, abstractions have been developed to model flow of data through a network. In particular Network Calculus [132] provides the means to deterministically reason about timing properties of data flows in queuing networks, and can be viewed as a *deterministic* queuing theory. Modular performance analysis [191] and Real-Time Calculus [181] extends the concepts of Network Calculus to the domain of real-time embedded systems.

Some other model-based solutions for timing design, performance optimization and timing verification are symbolic timing analysis for systems (SymTA/S) [105] or schedulability analysis provided by the tool TIMES [10], which is based on timed automata, a model we will discuss later in detail as it underlies our models.

Power Consumption Evaluation

Accurate power consumption estimation can be done at transistor or gate level, but due to the complexity of working at this level of abstraction, this is very costly. Thus, to accelerate power estimation analysis, many abstract models have been proposed including TLM approaches [17, 137, 25, 199] or algorithmic descriptions [147, 83, 85, 108, 124] .

Generally, high-level models use a reference design model to retrieve more accurate power estimation with RTL power estimation tools. Nevertheless, simulation time and memory requirement of these tools are considerable, making their use impracticable when exploring large design spaces.

2.5 Our Modeling Framework

The models presented in this thesis are much more abstract than those used in the *development* of the software and hardware in the sense that they do not represent the actual computations but attempt to capture the essential features which are relevant for predicting performance. For the application models we abstract away from the actual lines of code and view an application as a *task graph*, a collection of high-level procedures, each characterized by its execution cost (number of processor cycles), its precedences (to which tasks it needs to wait and which tasks wait for it to terminate) and the amount of data it sends/receives to/from other tasks.

The modeling of the processing elements is even more abstract compared to their real complexity. The state of a processor at a given instant is characterized by its speed (assuming processors that can switch between voltage/frequency levels), whether it is turned on, and which task is executing on it. The speed of the processor is used to translate the quantity of work of a task into a *duration*. In addition we use rough model of static and dynamic power consumption for the processors indicating the power per time unit in either of its speeds, in execution and idling. The same high-level modeling style is applied to other architectural components such as interconnect and memory

In this modeling framework, a task is viewed as a simple *timed automaton* which leaves its active state some *time* after entering it. The advantages of such abstract models in terms of how hard it is to simulate or analyze them are evident: to advance a clock in a discrete event simulation is orders of magnitude faster than a cycle-accurate simulation of the underlying software/herdware system, and even much faster than simulation at the operational semantics level of C programs. And of course, such models need not wait for the *complete* hardware and software to be realized. However the question about the relation between such models and any reality should not be ignored, and we phrase it explicitly: *Where do the numbers that decorate the model come from?*

The answer may vary depending on how developed the system in question is at the time of analysis. If the code is fully written and the architecture fully developed, one may profile the tasks and measure the execution times. In fact, if the systems is fully operational, testing it on the real hardware can be more efficient than any simulation. If a new application is to be deployed on a variant of an existing architecture, numbers can be derived based on a combination of profiling and designers know-how from similar applications. These numbers can be very imprecise and there may also be a large real-life variability among execution times of instances of the same task. To compensate for the imprecision we invoke a very attractive feature of our models, inherited from the tradition of formal verification: unlike "executable" models needed for simulation and for implementation, we use models that are not necessarily *deterministic*: they may exhibit non-determinism (or under-determination in the sense of [146]) in power consumption, in size of data items and in task *durations* as well as in their *arrival* patterns. The methods applied to handle this non determinism vary from Monte-Carlo simulation where the uncertainty space is randomly sampled to generate statistics, via verification methods that attempt to prove that something such as deadline miss never happens under all possible choices in the uncertainty space, to more sophisticated methods that compute the *expected* performance of a system.

To recapitulate our approach: we replace overly detailed models at a very low-level of abstraction (some of which might be inexistent at the time of the analysis) by very coarse grained models that compensate for their imprecision by taking the under-determination more seriously and explicitly in the analysis method. It seems that software and especially hardware developers bring to performance analysis too many low-level details (which are indispensable for implementation and functional correctness, though) while investing less effort in modeling the external environment of the system, such as the arrival model of tasks, which can have more effect on the performance than the low-level implementation details. This is understood because the external environment is not part of the system that they are responsible for developing and whose detailed implementation model cannot be avoided.

Since our approach originates historically from the tradition of formal verification it is worth making the premises of verification explicit in order to assess both its potential contribution and its current inadequacy for performance evaluation. Algorithmic formal verification is concerned with proving functional correctness of systems such as communication protocols and digital hardware. This is often done on models that abstract away from *data* and focus on *control* (synchronization).

However, functional correctness in the strict sense often used in verification is not a necessary nor sufficient condition for the usefulness of a system. A correct system with an extremely slow response is not likely be ever used, while systems that work well *most* of the time are all around us. To apply the insights of formal verification to system design beyond the very narrow context in which it is currently used, one should rethink some the following premises of the field:

1. System models, at least traditionally, are qualitative/logical without quantitative information;

2. The questions posed to the verification tool are of a qualitative yes/no nature: is the system correct or not;

3. There is an implicit universal quantification over the possible behaviors of the system: a system is correct if *all* its behaviors do not violate the specifications.

The first premise is relaxed by models of automata augmented with numerical variables are used extensively in software verification as well as in hybrid systems. Timed automata [6], the model most relevant to the present thesis, have been invented to model delays and execution times in a quantitative way. Relaxing the second premise has been promoted as *quantitative* analysis/synthesis [57, 42] and it consists of decorating states and transitions with numerical *costs* and tracking their evolution along system behaviors. Such costs typically admit a simpler dynamics than more general numerical variables in programs or hybrid systems. For example, the model of linearly-priced timed automata [46, 128], which are timed automata augmented with costs that can grow at different rates at different states, is simpler to analyze than other hybrid systems with constant slopes [120, 119, 15] because the cost variables are passive observers of the dynamics. The relaxation of universal quantification is what underlies *statistical model checking* [201] [62] [72] and can be viewed as a compromise between the rigor of formal verification and the scalability constraints for real systems. We demonstrate in this thesis that a combination of all these relaxations has a great potential in solving a central problem related to the multi-core revolution: how to evaluate and optimize the performance of application programs on such execution platforms.

Functional correctness and good performance are complementary and sometimes conflicting evaluation criteria. In *hard* real-time systems, performance is hardwired into correctness: a feedback function of a controller should be computed between every two consecutive sensor readings which puts a deadline constraint on its computation time. Using a timed model of the software/hardware architecture, which represents the execution times of the tasks as well as the scheduling policy, one can verify that such a deadline is never missed. In certain simple situations studied extensively by the real-time community [54, 142, 125] one can do the calculation [141] without invoking an explicit dynamic “executable” model at all. For embedded systems where the real-time constraints are *softer* the system is expected to give a *best effort* performance depending on the system load and resource availability. A typical example would be video streaming where a good trade-off between response time and image quality is sought. For such systems, the actual response time is a performance measure of the system, together with additional criteria such as system price or power consumption. Unlike what is common in verification, the quantitative measures are not Booleanized via predicates/constraints into a yes/no answer but remain quantitative and can be used to compare the relative performance of different designs.

Unlike safety-critical verification, soft systems are not evaluated according to their *worst-case* behavior but in a more probabilistic fashion. The traditional verification approach to the problem of performance evaluation based on “classical” timed automata technology [202, 82, 48, 130, 192, 38, 10] is *exhaustive*: it can compute performance measures such as termination time and other costs for *all* possible values of the uncertainty space, thus compute lower- and upper-bounds on termination time. For soft real-time systems this is, at the same time, too much and too little. The lower and upper-bounds represent very extreme cases which are realized only when all the tasks take their extremal duration values.

Under very reasonable assumptions these extreme values are less likely than termination times that admit many realizations (as 7 is more likely than 12 in dice). In contrast with the exhaustive approach, in Monte-Carlo simulation the uncertainty space is finitely sampled according to some distribution and each sampling point induces a single deterministic behavior whose performance is evaluated by (cheap) simulation. Such an approach is weaker than formal verification because it does not cover all behaviors: it can, at most, put bounds on the probability of error or a deadline miss. On the other hand it is stronger as it can give an estimation of the distribution and *expected value* of the termination times, which can be much more useful for this type of applications than the very conservative bounds computed by the exhaustive approach.

2.6 Related tools

Timed automata are a common and theoretically well-founded formalism for real-time systems. Reachability analysis of timed automata has been implemented in several tools, including KRONOS [73], UPPAAL [130], IF [48] or RABBIT [38]. Literature relates many tools for design-space exploration, based on timed automata or other formalisms. We present in the sequel some of them, close to our work.

TIMES **[10]**

It is a tool suite designed mainly for symbolic schedulability analysis and synthesis of executable code with predictable behaviours for real-time systems. Given a system design model consisting of a set of application tasks whose executions may be required to meet mixed timing, precedence, and resource constraints, a network of timed automata describing the task arrival patterns and a preemptive or non-preemptive scheduling policy, Times will generate a scheduler, and calculate the worst case response times for the tasks. The design model may be further validated using the UPPAAL timed model checker.

PTOLEMY **[53]**

The Ptolemy project studies heterogeneous modeling, simulation and design of concurrent systems with a focus on systems that mix computationnal domains [84] It uses tokens as the underlaying communication mechanism and controllers regulate how actors fire and how tokens are sent between each actors. Actors are software components that execute concurrently and communicate through messages sent via interconnected ports. A model is a hierarchical interconnection of actors. The semantics of a model is not determined by the framework, but rather by a software component in the model called a director, which implements a model of computation including process networks, discrete-events, dataflow, synchronous/reactive, rendezvous-based models, 3-D visualization, and continuous-time models. Each level of the hierarchy in a model can have its own director, and distinct directors can be composed hierarchically. A major emphasis of the project has been on understanding the heterogeneous combinations of models of computation realized by these directors. Directors can be combined hierarchically with state machines to make modal models [133]. For example, a hierarchical combination of continuous-time models with state machines yields hybrid systems [135].

OCTOPUS **[20]**

The octopus toolset supports model-driven design-space exploration based on high level modeling with a clear separation of application, platform and mapping. It provides formal analysis of functional correctness and performance and semi-automatic exploration of alternatives and synthesis of optimized designs. Analysis process works with different type of models:

- Timed automata model checking using UPPAAL [130]
- Petri nets simulation using CPNTools [164]
- Synchronous dataflow for trade-off analysis using SDF3 [179]

HOPES **[99]**

It is a model based framework for MPSoC software development. The application model is based on actor-orientation and is described in UML using PeaCE model [100]:

- Process network: specify execution condition of each task and define interaction between them.
- Synchronous piggybacked dataflow [101]: specify signal processing
- Flexible FSM [123]: specify control tasks.

The hardware platform is separately specified in a block diagram with a set of architectures and constraints parameters described in an xml-style. Application is manually partitioned into the abstract PEs that compose the hardware platform.

SESAME *(Simulation of Embedded System Architectures for Multilevel Exploration)* **[162]**

It is a modeling and simulation environment for system-level design based on the Y-chart design approach [122] which allows application and architecture to be modeled separately. The application model can be mapped onto the platform model and both are co-simulated via trace-driven simulation [163].

BIP *(Behavior, Interaction, Priority)* [22]

It is a component framework intended for rigorous system design. BIP allows the construction of composite hierarchically structured systems from atomic components characterized by their behavior and their interface. Components are composed by layered application of interactions and of priorities. Interactions express synchronization constraints between actions of the composed components while priorities are used to filter amongst possible interactions and to steer system evolution so as to meet performance requirements e.g. to express scheduling policies. Interactions are described in BIP as the combination of two types of protocols: rendez-vous, to express strong symmetric synchronization and broadcast, to express triggered asymmetric synchronization.

BIP has a rigorous operational semantics which has been implemented by specific execution engines for centralized, distributed and real-time execution. BIP is used as a unifying semantic model in a rigorous system design flow [21]. Rigor is ensured by two kinds of tools: verification tools such as D-Finder [29] for checking safety properties (and deadlock-freedom in particular) and source-to-source transformers [50], [43] that allow progressive refinement of (purely functional) application software towards platform-dependent implementations.

Chapter 3

Timed automata

In order to talk about systems behavior in a formal manner, it is necessary to represent them as some kind of mathematical structure. The simplest way to represent behavior is by means of automata or labelled transition system. These are simply graphs containing nodes and directed, labelled edges. Nodes represent the possible states of the system and edges (or transitions) represent activities as moves between two nodes.

In [70] authors have identified a handful of semantic concepts which are well-established in the context of computer-aided verification and modelling formalisms for discrete event systems:

- **Action nondeterminism**: From a given state several transitions may exist.
- **Probabilistic branching**: From a given state several transitions may exists and the choice is based on some probability distribution.
- **Clocks**: A way to represent real time constraints and to specify the dynamics of a model in relation to a physical, quantitive notion of time.
- **Delay nondeterminism**: allows one to leave the precise timing of events under specified.
- **Random variables**: give quantitative information about the likelihood of a certain event to happen within a given time frame.

Working with high level models, implies taking into account uncertainty in order to compensate for the lack of precision. Non-determinism can be modeled in many ways and labeled transition systems possess only action non-determinism. Other formalisms associate some kind of quantitative informations with action non-determinism.

Probabilistic automata have been used [170, 171, 169] for the purpose of modeling and analyzing asynchronous, concurrent systems with *discrete* probabilistic choice in a formal and precise way. Basically, a probabilistic automaton is a labeled transition system where the target of a transition is a probabilistic choice over several next states. Stochastic processes [79] is another formalism which is often used to represent the evolution of some random value, or system, over time. This formalism will be introduced in the next chapters.

In the present chapter we are interested in *timing uncertainty*. Timed automata [6] provide a theory for modeling and verification of real time systems. They provide the ability to constraint the execution of a transition to occur anywhere within a time interval. Timed automata introduce a dense non-determinism which is a very useful modeling feature when we have uncertain information about process durations. Other formalisms with the same purpose include timed Petri Nets [37], timed process algebra [165, 200, 156] and real time logics [9, 58].

We will present next, the timed automaton formalism, which will be used as a basis for our modeling framework presented in chapter 6.

3.1 Clocks, time constraints, zones

We use $\mathbb{Z}$ and $\mathbb{R}$ to denote, respectively the integer and real numbers, while $\mathbb{N}$ and $\mathbb{R}+$ will stand for their respective non-negative restrictions. We will use $\mathbb{R}+$ as the time domain on which clock variables range. We use $\mathbb{R}_\perp$ to denote $\mathbb{R}+\cup\perp$ were $\perp$ is a special symbol meaning inactive or irrelevant. We extend the addition operation to $\mathbb{R}_\perp$ by letting $\perp + d = \perp$.

Clocks and Valuations

Let $C = \{c_1, ..., c_n\}$ be a finite set of variables called clocks, each ranging over $\mathbb{R}_\perp$. A clock valuation is a function $v : C \to \mathbb{R}_\perp$ assigning to each clock $c \in C$ its value $v(c)$. The set of possible valuations of C is then $\mathbb{R}^n_\perp$. A clock c is said to be active in valuation v iff $v(c) \neq \perp$, otherwise it is inactive. In timed automata, clock valuations change due to two types of activities: time progress which happens inside a discrete state and clock assignment which take place during discrete transitions:

Time progress

Let d be a non-negative real. We say that clock valuation v' is the result of applying d-time-progress to clock valuation v, denoted by $v' = v + d$, if for every clock $c, v'(c) = v(c) + d$. Note that by the definition of addition on $\mathbb{R}_\perp$, all the clocks inactive in v do not change their value while all the other clocks advance uniformly.

Clocks assignment

A clock assignment is a function $\gamma : \mathbb{R}_\perp \to \mathbb{R}_\perp$ indicating a transformation of clock values which occurs during a transition. $v' = \gamma(v)$ denotes the fact that v' is the result of applying assignment γ to clock valuation v. The type of assignments that we allow in the definition of timed automata is restricted to compositions of one or more of the following basic assignments:

- Resetting to zero: $c_i = 0$
- Deactivation of a clock: $c_i = \perp$
- Clock copying: $c_i = c_j$

Clock constraints

Clock constraints are used to express the influence of clock values on the discrete dynamics (invariants and transition guards). We restrict ourselves to a family of constraints that we denote by Ψ_C, defined by the following grammar:

$$\psi ::= true \mid c_i \prec k \mid c_i - c_j \prec k \mid \psi \wedge \psi$$

where $c_i, c_j \in C, k \in \mathbb{N}$ and $\prec \in \{<, \leq, =, \geq, >\}$.

Timed zones

Clock constraints define naturally the set of clock valuations that satisfy them. These are subsets of $\mathbb{R}_+^n$. Every constraint $\psi \in \Psi_C$ is a conjunction of atomic constraints. Knowing that the set of valuations satisfying an atomic constraint defines a half-space, every constraint $\psi \in \Psi_C$ will define a convex polyhedron which is the intersection of those half-spaces. Z_ψ denote this polyhedron and it is called the timed zone associated with ψ. The set of all the zones defined on C will be denoted Z_C. Since the half-spaces are either orthogonal ($c_i \prec k$) or diagonal ($c_i - c_j \prec k$) with $k \in \mathbb{N}$, the vertices of these polyhedra are integer points and there is a finite number of zones in any bounded subset of $\mathbb{R}^n$. The most important property of zones is that they can be canonically represented as matrices i.e. DBMs (Difference Bound Matrices) [78].

In the following we will define some useful operations on zones that will be used throughout this chapter. Let $C' \subseteq C$ be a set of clocks, and let $Z_1, Z_2 \in Z_C$ be two timed zones defined on C, then:

$Z_1 \cap Z_2$ is the intersection of two zones Z_1 and Z_2, which is a zone (fig3.1-(b))

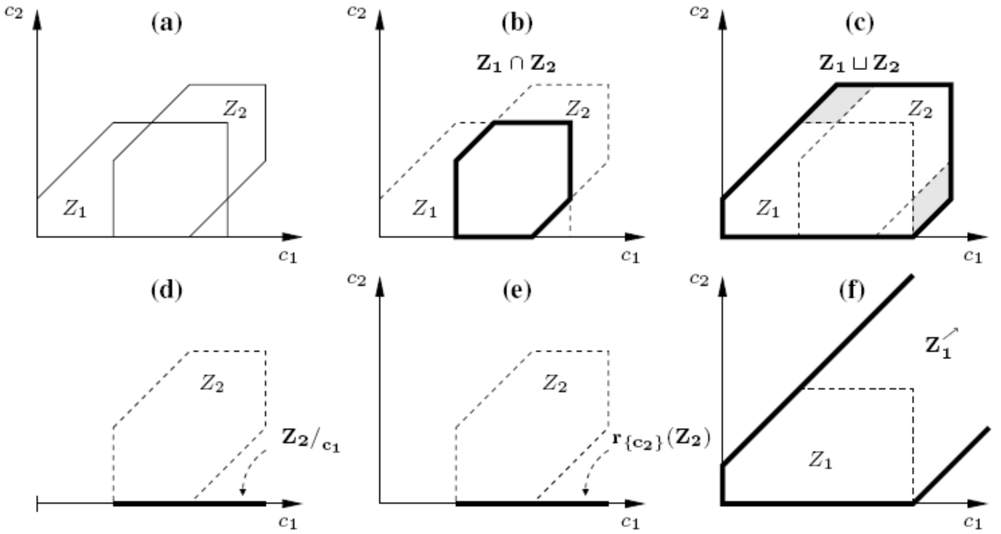
$Z_1 \sqcup Z_2$ is the convex hull of two zones Z_1 and Z_2 defined as
$Z_1 \sqcup Z_2 = \min\{Z \in Z_C \mid (Z_1 \subseteq Z) \wedge (Z_2 \subseteq Z)\}$
that is the smallest (in terms of inclusion) zone containing both Z_1 and Z_2, see (fig3.1-(c)). Since zones are not closed under union, $Z_1 \sqcup Z_2$ is used as an over-approximation of $Z_1 \cup Z_2$.

$Z^{\nearrow}$ is the forward projection of Z, i.e. all clock valuations that can result by applying time progress to element of Z (fig 3.1-(f)):
$Z^{\nearrow} = \{v \in V_C \mid \exists d \geq 0, v - d \in Z\}$

$Z_{/C'}$ is the projection of a zone Z on a clock subset $C' \subseteq C$:
$Z_{/C'} = \{v_{/C'} \mid v \in Z\}$
This operation is related to clocks deactivation, (fig 3.1-(d))

$\gamma(Z)$ is the result of applying the clock assignment function γ to all element of Z:
$\gamma(Z) = \{\gamma(v) \mid v \in Z\}$

Figure 3.1: Operations on timed zones

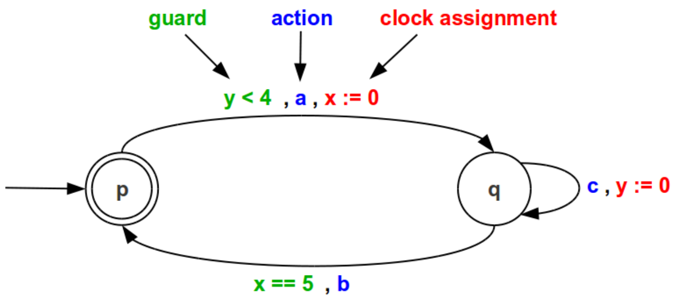


All these operations can be computed efficiently on a DBM representation of timed zones. More details can be found, for example, in [203].

3.2 Timed Automata: Syntax and Semantics

Timed automata have been introduced in [8, 6] as finite state *Büchi automata* (a variation of finite automaton that runs on infinite, rather than finite, inputs) extended with a set of real valued variables modeling clocks. Constraints on the clock variables are used to restrict the behavior of an automaton and Büchi accepting conditions are used to enforce progress properties. A simplified version, namely *Timed Safety Automata*, has been later introduced in [106] to specify progress properties using local invariant conditions.

Figure 3.2: Example of timed automaton



A timed automaton is presented as a discrete structure which is essentially a finite automaton (i.e a graph containing a finite set of nodes or locations and a finite set of labeled edges) extended with clock variables. Locations are supposed to capture all information about the current status of the system, except for timing information. Edges represents events or transitions which change the discrete state of the system. Time progress takes place *inside* the discrete states and is not expressed explicitly in this structure. Actually, time passage is recorded using clocks. All clocks of the system increase synchronously at the same rate. These clocks can be set to zero, or deactivated when a transition is taken.

Clocks constraints are used to restrict the behavior of the automaton by forcing it to leave a state or forbidding it from taking a certain transition. In [174, 44] these constraints are associated with the transitions, while in [106] they are divided between states and transition. Constraints on states denote staying conditions (called invariant). The automaton may stay in a state (while the active clocks are progressing) as long as the staying condition holds, otherwise it has to leave the state via one of the enabled transitions. Constraints on transitions are called *guards*, and a transition can be taken only if its guard constraints are satisfied.

Definition 3.2.1. ***(Timed automaton)*** *A timed automaton is a tuple* $A = (Q, q_0, C, \Sigma, I, \Delta)$ *where* Q *is a finite set of discrete states,* $q_0 \in Q$ *is the initial state,* C *is a finite set of clocks and* Σ *is a finite set of labels.* $I \in Q \to \Psi_C$ *is a function associating a staying condition (invariant) with every state* q*. The automaton is permitted to stay at* q *only as long as the clock constraint* I_q *is satisfied.*
$\Delta \subseteq Q \times \Psi_C \times \Sigma \times \Gamma_C \times Q$ *is the transition consisting of elements of the form* $e = (q, g, a, \gamma, q')$ *where:*

$q, q' \in Q$ *are, respectively, the source and the target of the transition*
$g \in \Psi_C$ *is an enabling condition called the transition guard.It restricts the execution of the transition to clock valuations that satisfy it.*
$a \in \Sigma$ *is the transition label,*
$\gamma \in \Gamma_C$ *is a clock assignment function which takes place during a transition.*

We assume, without loss of generality, that from every state q there is at most one transition labeled by a for every $a \in \Sigma$.

Parallel Composition of Timed Automata

A timed automaton is often considered to be an element in a network of components running in parallel and communicating with each other. The global behavior of such a network is captured by the global timed automaton, called the product. There are many variations of composition depending mainly on the interaction mechanisms through which the automata influence each other. At this point we use a definition based on a distributed alphabet [77] where each component A^i has its alphabet Σ^i. The alphabets of the components may have non-empty intersections and any global transition labeled by a must involve a local a-transition in every automaton A^i such that $a \in \Sigma^i$. Independent local transitions (transitions with different labels) enabled at the same global state can be executed in any order (interleaving).

Definition 3.2.2. ***(Parallel composition of timed automata)***
Let $N = \{A^i = (Q^i, q_0^i, C^i, \Sigma^i, I^i, \Delta^i) \mid i \in \{1,..,n\}\}$ be a network of timed automata. We assume the sets of clocks of each pair of automata to be disjoint and denote by $J(a)$ the indices i such that $a \in \Sigma^i$. The composition of these automata, denoted by $A^1 \parallel \ldots \parallel A^n$ is a timed automaton $A = (Q, q_0, C, \Sigma, I, \Delta)$ where $Q = Q^1 \times \ldots \times Q^n$ is the set of global discrete states of the form $q = (q^1, \ldots, q^n)$, $q_0 = (q_0^1, \ldots, q_0^n)$ is the initial state, $C = \bigcup_{i=1}^n C^i$ is the global set of clocks, $\Sigma = \bigcup_{i=1}^n \Sigma^i$ is the global alphabet and I is the global invariant $I(q) = \bigwedge_{i\in\{1,\ldots,n\}} I^i(q^i)$. The global transition relation Δ consists of tuples of the form $((q^1, \ldots, q^n), g, a, \gamma, (q'^1, \ldots, q'^n))$ such that:

- $\forall i \notin J(a), q'^i = q^i$
- $\forall i \in J(a), (q^i, g^i, a, \gamma^i, q'^i) \in \Delta^i$
- $g = \bigcap_{i\in J(a)} g^i$
- $\gamma = \circ_{i\in J(a)}\gamma^i$

Steps

Timed automata define infinite transition systems whose states are configurations of the form (q, v) consisting of a discrete state q and a clock valuation v. The initial configuration is $s_0 = (q_0, \bot)$ with all clocks inactive and the transitions are either discrete transitions of the automaton or time-passage transitions. This is formalized by the notion of a step.

Definition 3.2.3. ***(Steps)*** *A step of a timed automaton A is one of the following :*

- *A discrete step: $(q, v) \longrightarrow^a (q', v')$, for some transition $(q, g, a, \gamma, q') \in \Delta$ such that $v \models g$ and $v' = \gamma(v)$*
- *A time step:$(q, v) \longrightarrow^d (q', v + d)$ for some $d \in \mathbb{R}_+$ such that $v + d$ satisfies $I(q)$*

Note that the concatenation of two time steps is a time step:

$$(q,v) \longrightarrow^{d_1} (q,v+d_1) \longrightarrow^{d_2} (q,v+d_1+d_2) \equiv (q,v) \longrightarrow^{d_1+d_2} (q,v+d_1+d_2)$$

Conversely, due to the dense nature of the real numbers, a time step can be split into any number of smaller time steps. A compound step is a discrete step followed by a time step (possibly of zero duration):

$$(q,v) \longrightarrow^{a,d} (q',v'+d) \equiv (q,v) \longrightarrow^{a} (q',v') \longrightarrow^{d} (q',v'+d)$$

A run of the automaton A starting from a configuration (q,v) is a finite sequence of compound steps (possibly starting by a pure time step). These definitions apply to products as well. Note that a global time step in a global state $q=(q^1,\ldots,q^n)$ is just a local (and uniform) time step for each component A^i. The global invariant requires that all local invariants hold at $v+d$. On the other hand a global discrete step labeled by a is a local discrete step for all components A^i such that $a \in \Sigma^i$.

3.3 Reachability Analysis

Given timed automata models we would like to verify them, that is, to see what the possible runs of a given automaton are, and whether they satisfy a given property. The enumeration of possible runs is done in a set-based fashion, computing in one step all the successors of a set of configurations by an arbitrary passage of time and by transitions. Verification based on timed automaton can be done in different manners:

- Reachability analysis: does there exists a run of A which ends in some specific state?
- Verification of temporal timed logic: does the timed automaton A satisfies a temporal logic property φ ?
- Language inclusion, simulation: can A_1 produce all behaviors that A_2 does?
- Equivalence checking, timed bisimulation of timed automata: are two automaton A_1, A_2 equivalent or bisimilar?

In the context of this thesis, timed automata are used as the basis modeling formalism and we use only reachability analysis to check whether some error state is reachable or not. All the property that we use are bounded-horizon timed properties that can be reduced to safety reachability on the composition of the system and the property monitor. We will recall next how reachability analysis works.

The original decidability proof for verification of timed automata [8] was based on partitioning the state space into a finite number of equivalence classes called regions. Regions are the "atomic" zones from which all other zones can be constructed. Two configurations (q,v) and (q,v') are region-equivalent if for every transition guard g, $v \models g$ iff $v' \models g$, and if by letting time pass they reach the same region. Hence for every sequence of regions visited by a run from (q,v) there is a run from (q,v') visiting the same sequence. The timed automaton can thus be reduced to a finite automaton whose states are regions with discrete transitions and special transitions that correspond to the passage of time. Region equivalence is guaranteed to capture all the qualitative behaviors of any automaton, but its force is also its weakness because the large number of regions renders this approach impractical.

The most popular approach is on-the-fly forward search based on zones. This approach has the following advantages:

- It does not explore the parts of the state space which are not reachable from the initial state;
- It does not refine zones beyond what is necessary and will typically result in a number of generated zones much smaller than the number of regions;
- It uses an efficient data structure, the DBM, to store and manipulate zones.

The principle of symbolic reachability computation for discrete systems is to take a representation of a set P of states reachable after k steps, and compute from it the set $succ(P)$ of its successors by all transitions, that is, the set

$$succ(P) = \bigcup_{q \in P} \bigcup_{a \in \Sigma} \{succ^a(q)\}$$

The basic element in this computation is an object consisting of one discrete state and a set of clock valuations. Successors by different transitions are treated separately and enumeratively, while the symbolic treatement is reserved for time passage and clocks valuations.

Definition 3.3.1. *A* ***symbolic state*** *of a timed automaton $A = (Q, q_0, C, \Sigma, I, \Delta)$ is a pair (q, Z) with $q \in Q$ a discrete state and Z a zone.*

Symbolic states are closed under the following operations:

- The time successor of a symbolic state (q, Z) is the symbolic state (q, Z') where Z' is the set of clock valuations reachable from Z by letting time progress without violating the staying condition $I(q)$:

  $$post^t(q, Z) = \{(q, v + d) | (v \in Z) \wedge (d \geq 0) \wedge ((v + d) \models I(q))\} = (q, (Z^{\nearrow} \cap I(q)))$$

  We say that (q, Z) is time-closed if $(q, Z) = post^t(q, Z)$.

- Let $(q, g, a, \gamma, q') \in \Delta$ be a transition. The a-transition successor of a symbolic state (q, Z) is the set of configurations reached by taking this transition. Only clock valuations of Z that satisfy the guard g are concerned with this transition. These clock valuations will be transformed according to the assignment function σ while taking this transition:

  $$post^a(q, Z) = \{(q', v') | \exists v \in Z, v \models g \wedge v' = \gamma(v)\} = (q', (\gamma(Z \wedge g)))$$

- The a-successor of a symbolic state (q, Z) is the set of configurations reached from (q, Z) by an a-transition followed by passage of time:

  $$succ^a(q, Z) = post^t(post^a(q, Z)) = (q', (\gamma(Z \wedge g))^{\nearrow} \cap I(q'))$$

Proposition 3.3.1. *Let $(q', Z') = succ^a(q, Z)$ for a transition (q, g, a, γ, q'). A configuration (q', v') belongs to (q', Z') if and only if it is the endpoint of a compound step*

$$(q, v) \longrightarrow^{a,d} (q', v')$$

for some $(q, v) \in (q, Z)$ and some $d \geq 0$

Equipped with these operators under which the set of symbolic states is closed, we can now introduce the forward reachability algorithm for timed automata which computes this way all the runs of A. Algorithm-1 presents a breadth-first version of reachability computation but other exploration orders are possible. The algorithm terminates because there are finitely many zones in any bounded subset of $\mathbb{R}^n_{\perp}$.

Algorithm 1 Forward reachability algorithm (breadth first).

```
Explored ← $\emptyset$
New ← $\emptyset$
Waiting ← $\{(q_0, \bot)\}$
**while** Waiting $\neq \emptyset$ **do**
    **for all** $(q, Z) \in$ Waiting **do**
        **for all** $(q, g, a, \gamma, q') \in \Delta$ **do**
            New ← New $\cup succ^a(q, Z)$
        **end for**
        Explored ← Explored $\cup(q, Z)$
    **end for**
    Waiting ← New $\setminus$ Explored
    New ← $\emptyset$
**end while**
    **return** Explored
```

Authors in [74] introduce the notion of clock activity in order to reduce the number of clocks and the complexity of timed verification. A clock is said to be active at some discrete state when its value is relevant for the future evolution of the system, for example, when the clock appears in the state invariant or in a guard of a transition outgoing from the state. The work of [74] had two major parts. The first was the (approximate) detection of such clock inactivity by performing a syntactic data-flow analysis of the automaton. Then this information was used to reduce the dimensionality of the clock space, by manipulating in each state polyhedra whose dimension is equal to the number of clocks active in this state. In our modeling framework we express clock deactivation explicitly by assigning a clock value to $\bot$ and, in the class of automata that we use for modeling, clock activity tracking is self-evident and we can benefit from the dimensionality reduction without performing the analysis.

3.4 Implementation

In the context of this thesis, timed automata models presented thereafter are defined using the IF language [48]. We give in this section, an overview of this formalism as described in [49].

IF is a notation for systems of components (called *processes*), running in parallel and interacting either through shared variables or asynchronous signals. Processes describe sequential behaviors including data transformations, communications and process creation. Furthermore, the behavior of a process may be subject to timing constraints. The number of processes may change over time: they may be created and deleted dynamically. The semantics of a system is the labelled transition system (LTS) obtained by interleaving the behavior of its processes. To enforce scheduling policies, the set of runs of the LTS can be further restricted using dynamic priorities. This representation is expressive enough to describe the basic concepts of modeling and programming languages for distributed real time systems.

Processes

The behavior of a *process* is described as a timed automaton, extended with data. A process has a unique process identifier (*pid*) and local memory consisting of variables (including clocks), control states and a queue of pending messages (received and not yet consumed).

A process can move from one control state to another by executing some *transition*. As for StateCharts [104, 103], control states can be hierarchically structured to factorize common behaviors. A sequence of transitions between two stable states defines a *step*. The execution of a step is *atomic*, meaning that it corresponds to a single transition in the LTS representing the semantics. Notice that several transitions may be enabled at the same time, in which case the choice is made non-deterministically.

Transitions can be either *triggered* by signals in the input queue or be *spontaneous*. Transitions can also be *guarded* by predicates on variables, where a guard is the conjunction of a data guard and a time guard. A transition is enabled in a state if its trigger signal is present and its guard evaluates to true. Signals in the input queue are a priori consumed in a FIFO fashion, but one can specify in transitions which signals should be "*saved*" in the queue for later use.

Transition *bodies* are *sequential programs* consisting of elementary actions (variable or clock assignments, message sending, process creation/destruction, resource requirement/release, etc) and structured using elementary control-flow statements (like if-then-else, while-do, etc). In addition, transition bodies can use external functions/procedures, written in an external programming language (C/C++).

Signals

Signals are typed and can have data parameters. Signals can be addressed directly to a process (using its *pid*) and/or to a signal route which will deliver it to one or more processes. The destination process stores received signals in a FIFO buffer.

Data

The IF notation provides the *predefined basic types* bool, integer, real, pid and clock, where *clock* is used for variables measuring time progress. Structured data types are built using the *type constructors* enumeration, range, array, record and abstract. Abstract data types can be used for manipulating external types and code.

Composition

The semantics associates with a system a global LTS. At any point of time, its state is defined as the tuple of the states of its living components: the states of a process are the possible evaluations of its attributes (control state, variables and signal queue content). The transitions of the global LTS representing a system are steps of processes and signal deliveries to queues where in any global state there is an outgoing transition for all enabled transitions of all components (interleaving semantics). The formal definition of the semantics can be found in [47].

System models may be highly nondeterministic, due to the non-determinism of the environment which is considered as open and to the concurrency between their processes. For the validation of functional properties, leaving this second type of non-determinism non resolved is important in order to verify correctness independently of any particular execution order. Nevertheless, going towards an implementation means resolving a part of this non determinism and choosing an execution order satisfying time related and other nonfunctional constraints.

In IF, such additional restrictions can be enforced by dynamic priorities defined by rules specifying that whenever for two process instances some condition (state predicate) holds, then one has less priority than the other.

Time

The time model of IF is that of *timed automata with urgency* [44]. The execution of a transition is an event defining an instant of state change, whereas time is progressing in states. The progress of time is controlled by urgencies of the enabled transitions. Transitions are annotated with urgency types that are *eager*, *lazy* and *delayable*. Eager transitions prevent time progress. Lazy transition have no impact on the time progress. Delayable transitions are considered lazy unless their guard is disabled by the progress of time, and then they are considered eager.

Example

As an example, consider a multi-threaded server which can handle at most N simultaneous requests. Thus, if possible, for a request message (received from the environment) a thread is created. The server keeps in the thc variable the number of running threads. Threads processes are quite simple: once created, they work during a time interval $[l, u]$, and when finished they send a *done* message back to the server. A graphical description is shown in Fig-3.3 and the corresponding IF code is shown in Listing-3.1.

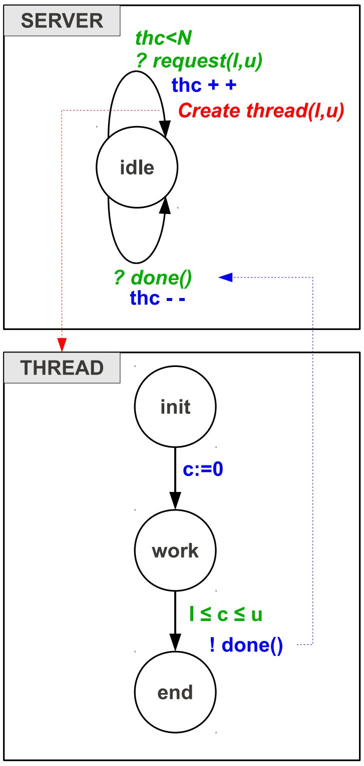


Figure 3.3: Example of a multi-threaded server

```
//request for [l,u] amount of work
signal request(l,u);
signal done();

process server(1);
 var thc integer;

 state idle #start ;
  deadline lazy;
  provided thc < N;
  input request(l,u);
   fork thread(self, l, u);
   task thc := thc + 1;
    nextstate idle;

  deadline eager;
  input done();
   task thc := thc - 1;
    nextstate idle;
 endstate;
endprocess;

process thread(0);
 fpar parent pid, l integer, u integer;
 var c clock;

 state init #start ;
  set c := 0;
   nextstate work;
 endstate;

 state work;
  deadline delayable;
  when c>=l and c<=u;
   output done() to parent;
   stop;
 endstate;
endprocess;
```

Listing 3.1: IF description

Observers

Observers express in an operational way safety properties of a system by characterizing its acceptable execution sequences. They also provide a simple and flexible mechanism for controlling model generation. They can be used to select parts of the model to explore and to cut off execution paths that are irrelevant with respect to given criteria.

Observers are described as IF processes i.e., as extended timed automata. They differ from IF processes in that they can react *synchronously* to events and conditions occurring in the observed system. Observers are classified into:

- *Pure* observers - which express requirements to be checked on the system.
- *Cut* observers - which in addition to monitoring, guide simulation by selecting execution paths. For example, they are used to restrict the behavior of the environment
- *Intrusive* observers - which may also alter the system's behavior by sending signals and changing variables.

For monitoring the system *state*, observers can use primitives for retrieving values of *variables*, the *current state* of the processes, the contents of *queues*, etc. For monitoring *actions* performed by a system, observers use constructs for retrieving events together with data associated with them. Events are generated whenever the system executes one of the following actions: signal output, signal delivery, signal input, process creation and destruction and informal statements. Observers can also monitor time progress, by using their own clocks or by monitoring the clocks of the system.

In order to express properties, observer states can be marked as *ordinary*, *error* or *success*. *Error* and *success* are both terminating states. Reaching a success state (an error state) means satisfaction (violation). *Cut* observers use a *cut* action which stops exploration.

Model exploration

IF toolset includes an *exploration platform* that is a component which has an API providing access to the LTS corresponding to IF description. The interface offers primitives for representing and accessing states and labels as well as basic primitives for traversing LTS enabling implementation of any on-the-fly forward enumerative exploration or validation algorithm. The exploration platform composes all active processes and computes global states and the corresponding system behavior.

Simulation time is handled by a specialized process managing allocation and deallocation of clocks, computing time progress conditions and firing timed transitions. There are two implementations available, one for discrete time and one for dense time. For discrete time, clock values are explicitly represented by integers. Time progress is computed with respect to the next enabled deadline. For dense time, clock valuations are represented using variable-size Difference Bound Matrices (DBMs) as in tools dedicated to timed automata such as KRONOS [202] and UPPAAL [130].

Chapter 4

Duration Probabilistic Automata: Analysis

The next two chapters constitute the major theoretical contribution of the thesis. They are concerned with a class of stochastic processes, Duration Probabilistic Automata (DPA) [145], that can model scheduling problems such as job-shop, where task durations are uncertain and distributed *uniformly* over a *bounded interval*. DPAs can also be viewed as timed automata where the intervals of temporal uncertainty are interpreted probabilistically, an assumption which is implicit in the Monte-Carlo simulation provided by the tool. In this chapter we develop a piecewise-analytic approach to compute performance measures for such systems, while in the next chapter we study the synthesis of schedulers which are expected-time optimal. Naturally, these new results and computational techniques are first developed on clean abstract models, less loaded with real-life details than the models used in the more applied parts of the thesis.

4.1 Scheduling under Stochastic Uncertainty

Scheduling, the allocation of limited reusable resources over time to competing tasks, is a universal activity. It is performed routinely in domains of very different scales in terms of time, space and energy. These include the allocation of airways and runways to flights, allocating machines to different product lines in a factory, and the efficient allocation of computation and communication resources to information-processing tasks. This latter activity is becoming of prime importance in many scales, ranging from world-wide cloud computing, via the realization of multiple distributed control loops, down to mapping and scheduling tasks on multi-core computers. In all such situations one wants to synthesize schedulers which are optimal or good in some sense, or at least to be able to compare the performance of proposed schedulers and choose the better ones. Performance and optimality of such schedulers are typically based on the quantity of work performed over time, which in the case of a finite amount of work can be expressed as *termination* time. Good schedules are typically associated with intensive, almost idle-free, utilization of critical bottleneck resources.

In a *deterministic* setting one assumes that everything is known *in advance* about the demand for work, including the tasks to be executed, their arrival times and the durations for which they occupy resources. In other words, once the scheduling policy itself is determined, the system admits a *unique* execution scenario (run, realization). Evaluating a scheduler based on this unique run is straightforward – just simulate it – while finding an optimal scheduler for any non-trivial scheduling problem (such as job-shop) is NP-hard or worse. However, determinism is rarely the case in real life and exact duration of tasks, their arrival times and many other features may vary to large extents. Each instance in this uncontrollable space yields a *different schedule* and the overall evaluation of a scheduler or a scheduling policy, which can be viewed as a strategy in a two-person timed game [42] with uncertainty viewed as an adversary, should be based on some *quantification* over all possible behaviors it induces [144].

This adversarial time-optimality problem has been tackled in [14, 2] using a *worst-case* approach on models of different types of uncertainty. In [14], using, the general model of *timed game automaton* [16] where the adversary is discrete, the following problem was proved to be decidable: synthesize a controller which is worst-case time-optimal in the sense that the maximal (over all possible runs induced by the adversary) time to reach a goal state is minimal. In [2] the case of job-shop scheduling with uncertain task *durations* each ranging over a bounded interval was treated. For this problem, worst-case optimality is defined trivially by the optimal solution to a *deterministic* scheduling problem associated with the worst case where all tasks take their respective maximal duration. One has to define a new notion of optimality (d-future optimal strategies) to make the optimal synthesis problem meaningful, resulting in a synthesis algorithm based on value iteration over sets of clock valuations (zones) which can be seen as an *offline* version of some kind of *model-predictive control.*

The use of worst-case reasoning is to some extent a residue of the safety-critical banner under which formal verification has been argued for, but in many (if not most) real-life situations, temporal uncertainty is modeled probabilistically as a distribution over the durations of each task and scheduler quality is measured accordingly, for example by the *expected* completion time or by its maximum over all but a small fraction of the runs. In this chapter we develop and implement a computational framework in order to evaluate and optimize the performance of such schedulers, modeled by automata similar in structure to those used in [2] but whose durations are probabilistic. Such automata are sufficiently rich to express stochastic variants of well-known scheduling problems such as job-shop or task-graph.

The study of continuous-time stochastic processes has been going on for many years in other branches of mathematics where simple computational questions like those we pose are not typically asked, as well as in closer domains such as probabilistic verification and performance evaluation [51, 45]. A well-studied class of such processes are *continuous-time Markov chains* (CTMC) where durations are distributed *exponentially*. Such distributions are memoryless in the sense that time spent waiting for a task to terminate does not influence the distribution on the remaining time. As a result they are easy to compute with and problems such as model-checking against qualitative [5] and quantitative [18] temporal properties or optimal controller synthesis for finite-horizon problems [2] are well understood. This forgetfulness assumption may be realistic and useful for modeling request arrivals in queuing models, but seems inappropriate for modeling the durations of several instances of the same computational task.

In this work we assume task durations to be *uniform* over a *bounded* interval, which is a natural “stochastization” of the set-theoretic temporal uncertainty of timed automata. Handling such systems we find ourselves in the realm of the so-called *generalized semi-Markov processes* (GSMP), a class of continuous-time stochastic processes [91, 93, 56, 118]. Similar computational studies of GSMPs include [7, 35], [189, 55] and [145]. The former are concerned with verifying temporal properties for some classes of GSMPs and develop techniques to determine whether the probability of a property-violating behavior is zero. The work of [189, 55] is concerned with stochastic Petri nets for which a computational framework is developed for propagating densities in the marking graph. This work, as well as [145] on duration probabilistic automata, use densities on clocks which are auxiliary state variables. At each reachable state and zone in the clock space, the distribution over clock values is maintained and used to compute the distribution after the next transition. In contrast, the approach presented in the this chapter works directly on the space of the *duration* random variables and does not use clocks explicitly. Similar ideas were developed in [115] to compute the probability of test cases in timed systems.

The rest of the chapter is organized as follows. Section 4.2 defines single and parallel processes, their behaviors (timed and qualitative) and presents a useful coordinate transformation between durations and time stamps. Section 4.3 shows how to derive the timing constraints associated with a qualitative behavior when processes execute independently without resource conflicts, and how to compute the volumes of the polytopes they define. Section 4.4 extends the framework to the more interesting case of resource conflicts which are resolved by dynamic scheduling strategies. Finally we present very preliminary experimental results and discuss future directions.

4.2 Preliminaries

We consider a composition $S = P^{\mathbf{1}}||\cdots||P^{\mathbf{n}}$ of n *sequential stochastic processes*, each consisting of a sequence of steps. Each step has a probabilistic duration and cannot start before its predecessor terminates. We consider two execution frameworks:

1. *Independent execution*: all processes start simultaneously and each process starts a step as soon as its preceding step has terminated, regardless of the state of other processes;
2. *Coordinated execution*: the initiation of a step is controlled by a *scheduler* which may hold a step of one process in a waiting state until the termination of a step of another process that uses the same resource.

The second framework will allow us to compare schedulers but we start with the first because it is simpler and does not require knowledge of timed automata. On this simpler model we will develop the basic computational machinery that will allow us to compute the probabilities of different *qualitative behaviors*, each corresponding to an equivalence class of timed behaviors associated with a particular *order* in which steps of different processes terminate.

Definition 4.2.1 (Uniform Distribution)**.** *A* uniform *distribution inside an interval $I = [a, b]$ is characterized by a density ψ defined as*

$$\psi(y) = \begin{cases} 1/(b-a) & \textit{if } a \leq y < b \\ 0 & \textit{otherwise} \end{cases}$$

and in terms of distribution as

$$F(y) = \int_0^y \psi(\tau)d\tau = \begin{cases} 0 & \textit{if } y < a \\ (y-a)/(b-a) & \textit{if } a \leq y \leq b \\ 1 & \textit{if } b \leq y \end{cases}$$

Definition 4.2.2 (Process)**.** *A sequential stochastic process is a pair $P = (\mathcal{I}, \Psi)$ where $\mathcal{I} = \{I_j\}_{j \in K}$ is a sequence of duration intervals and $\Psi = \{\psi_j\}_{j \in K}$ is a matching sequence of densities with ψ_j being the uniform density over $I_j = [a_j, b_j]$, indicating the duration of step j.*

We consider *finite* processes with $K = \{1, \ldots, k\}$. Probabilistically speaking, step durations can be viewed as a finite sequence of *independent* uniform random variables $\{y_j\}_{j \in K}$ that we denote as vectors $y = (y_1, \ldots, y_k)$ ranging over a *duration space*

$$D = I_1 \times \cdots \times I_k \subseteq \mathbb{R}^k$$

with density $\psi(y_1, \ldots, y_k) = \psi_1(y_1) \cdots \psi_k(y_k)$. Each point y in the duration space induces a *unique* behavior of the system written as a *time-event sequence* of the form

$$\xi_y = y_1\, e_1\, y_2\, e_2\, \cdots\, y_k\, e_k. \tag{4.1}$$

Time event sequences are alternations between time elapses represented by real numbers and discrete events that take no time. In the case of a single process $y_j \in I_j$ is the *duration* of step

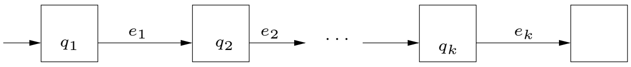


Figure 4.1: An automaton view of a process.

j and e_j is the *event* of terminating that step. The *timed language*[1] associated with the process consists of all the timed behaviors it may generate, namely $L = \{\xi_y : y \in D\}$. The *untimed language* associated with the process is $\underline{L}$, obtained by projecting away durations and retaining events and their order. In the case of a single process $\underline{L}$ is simply the singleton language $\{w\}$ where $w = e_1\, e_2 \cdots\, e_k$.

Mechanically speaking the process behaviors can be viewed as generated by the automaton of Fig. 4.1 in which being at state q_j corresponds to executing step j. Each run of the automaton is associated with a point y in the duration space. Upon entering q_j an auxiliary clock variable x is reset to zero and the termination transition labeled by e_j is taken exactly when $x = y_j$.

Suppose we want to characterize the probability of a certain subset of L. For example those behaviors in which for every j the actual duration of step j is in some sub-interval $I'_j = [a'_j, b'_j] \subseteq I_j$. The total probability of these behaviors is simply the *volume* of the rectangle $I'_1 \times \cdots \times I'_k$ divided by the volume of the whole rectangle D. Probabilities of other subsets of the language can be more interesting but harder to compute. For example, the probability that the whole process terminates before some deadline r is simply the volume of the subset of D satisfying $y_1 + \cdots + y_k < r$ divided by the volume of D. Our technique is based on computing such volumes for a system of several parallel processes as described in the sequel.

It turns out to be easier to compute volumes after a coordinate transformation from the space of durations to the space of *time stamps* consisting of vectors $t = (t_1, \ldots, t_k)$ where t_j is the absolute occurrence time of event e_j, defined as $t_j = y_1 + y_2 + \cdots + y_j$. A behavior ξ_y can thus be written also as a sequence of time-stamped events[2]

$$\xi_t = (e_1, t_1), (e_2, t_2), \ldots, (e_k, t_k).$$

Assuming that all durations admit a positive lower bound $a_j > 0$, all time stamps satisfy *precedence constraints* of the form $t_j < t_{j+1}$.

Converting y to t and vice versa is done by the linear transformations $t = Ty$ and $y = T't$ where T and T' are matrices of the form

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \qquad T' = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 1 \end{pmatrix}$$

These matrices are lower triangular (the value of t_j cannot depend on a duration $y_{j'}$ with $j' > j$) and their diagonal entries are equal to 1. The determinant of a triangular matrix is equal to the product of the diagonal entries which is 1 and hence the transformations are *volume preserving*. This means that the volume of the duration space D is equal to the volume of the *time-stamp space* C defined by the constraints

$$\varphi_C : \bigwedge_{j \in K} a_j \leq t_j - t_{j-1} \leq b_j$$

[1] In the computer science tradition the term *language* is often used to denote a set of sequences or other objects that define dynamic behaviors.

[2] These are the *timed traces* used originally in [6] to give semantics to timed automata. More about the relation between semantic models of timed behaviors can be found in [13].

and computing the volume of any subset $C' \subseteq C$ amounts to computing the volume of its T' image $D' \subseteq D$. Let us remark that the density of t_j is the *convolution* of the densities $\psi_1, \ldots, \psi_j$ and its support is the Minkowski sum of $I_1, \ldots, I_j$.

The time-stamp space C and its subsets that we will encounter are defined as conjunctions of inequalities of the form $x \prec c$ or $x - x' \prec c$ where $\prec \in \{<, \leq, =, \geq, >\}$ and c is an integer constant. They define polytopes which are called *zones* (or timed polyhedra). Zones are used extensively in the analysis of timed automata [106, 73, 130]. They admit an efficient representation by *difference-bound matrices* (DBM) [78] and efficient algorithms based on shortest-path to remove redundant constraints [67].

Definition 4.2.3 (Process System)**.** *A process system consists of n processes*

$$S = P^{\mathbf{1}} || \cdots || P^{\mathbf{n}} = \{(\mathcal{I}^{\mathbf{i}}, \Psi^{\mathbf{i}})\}_{i=1}^{n}$$

We use notations $P^{\mathbf{i}}_j$ to refer to step j of process i and $I^{\mathbf{i}}_j = [a^{\mathbf{i}}_j, b^{\mathbf{i}}_j]$ and $\psi^{\mathbf{i}}_j$ for the respective intervals and densities. To ease notation we assume all processes to have the same number k of steps. The event alphabet of the system is

$$\Sigma = \{e^{\mathbf{1}}_1, e^{\mathbf{1}}_2, \ldots, e^{\mathbf{n}}_{k-1}, e^{\mathbf{n}}_k\}$$

consisting of all the termination events of the steps of the various processes.

A behavior of the system is induced by a point in the global duration space

$$y = (y^{\mathbf{1}}_1, y^{\mathbf{1}}_2, \ldots, y^{\mathbf{n}}_{k-1}, y^{\mathbf{n}}_k) \in \mathcal{D} = \prod_{i=1}^{n} \prod_{j=1}^{k} I^{\mathbf{i}}_j \subset \mathbb{R}^{nk},$$

which can be transformed into a point t in the time-stamp space

$$t = (t^{\mathbf{1}}_1, t^{\mathbf{1}}_2, \ldots, t^{\mathbf{n}}_{k-1}, t^{\mathbf{n}}_k) \in \mathcal{C} = T\mathcal{D}$$

where T is the appropriate block diagonal matrix.

When all processes start simultaneously, the time stamps are taken from the same global time reference and one can view a global run as merging local runs and sorting the events according to their time stamps, as illustrated in Fig. 4.2. The set of all such global behaviors is denoted by

$$L = L^{\mathbf{1}} || \cdots || L^{\mathbf{n}}.$$

All timed behaviors that admit the same *order* of events are said to exhibit the same *qualitative behavior*. This can be formalized as an operation among the untimed local languages. Let $\underline{L}^{\mathbf{i}} = \{e^{\mathbf{i}}_1\, e^{\mathbf{i}}_2 \cdots e^{\mathbf{i}}_k\}$ be the untimed language associated with process $P^{\mathbf{i}}$: it consists of the unique qualitative behavior which satisfies the precedence constraints of $P^{\mathbf{i}}$. The potential qualitative behaviors of S constitute the language

$$\underline{L} = \underline{L}^1 || \cdots || \underline{L}^n$$

which is the *shuffle* of these languages, that is, the set of sequences consisting of one occurrence of each event in Σ and respecting the local precedence constraints for each process. Mathematically speaking, a qualitative behavior corresponds to a linear order[3] which is consistent with the partial order defined by the union of the precedence relations of all the tasks. Such an order is also known as *interleaving* in the theory of *concurrency* (readers can refer to [77] or [86]).

[3] Since we are dealing with volumes, our neglect of the possibility of events occurring at *exactly* the same time and not paying too much attention to the distinction between strict and non strict inequalities is justified.

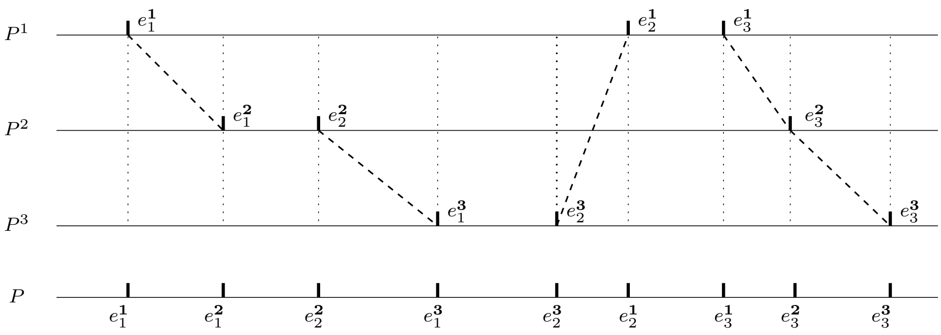


Figure 4.2: A global behavior $w = e_1^{\mathbf{1}}\, e_1^{\mathbf{2}}\, e_2^{\mathbf{2}}\, e_1^{\mathbf{3}}\, e_3^{\mathbf{2}}\, e_2^{\mathbf{1}}\, e_3^{\mathbf{1}}\, e_2^{\mathbf{3}}\, e_3^{\mathbf{3}}$ obtained by merging local behaviors. The dashed line indicate the minimal set of additional inter-process constraints that characterize w.

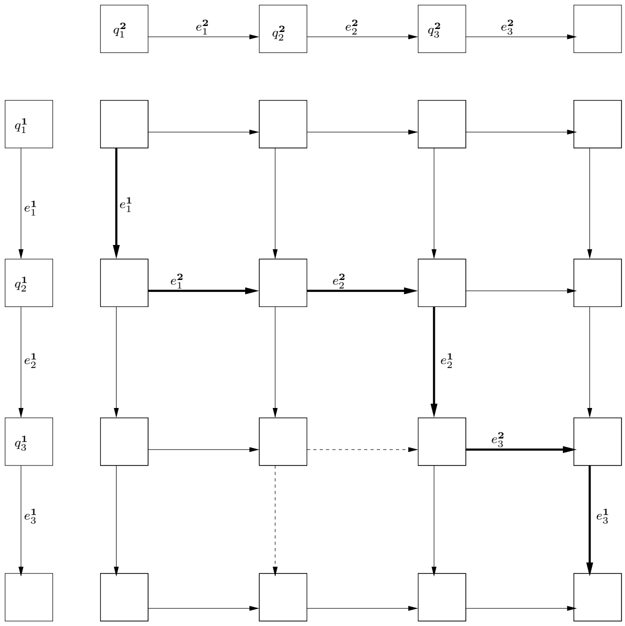


Figure 4.3: The product automaton for a process system with $n = 2$, $k = 3$. The thick arrows indicate the path corresponding to the qualitative behavior $w = e_1^{\mathbf{1}}\, e_1^{\mathbf{2}}\, e_2^{\mathbf{2}}\, e_1^{\mathbf{3}}\, e_3^{\mathbf{2}}\, e_2^{\mathbf{1}}\, e_3^{\mathbf{1}}\, e_2^{\mathbf{3}}\, e_3^{\mathbf{3}}$. The race between $e_3^{\mathbf{1}}$ and $e_2^{\mathbf{2}}$ in state $(q_3^{\mathbf{1}}, q_2^{\mathbf{2}})$ is indicated by the dashed arrows.

We use the term qualitative behavior also for any *prefix* of a sequence in $\underline{L}$. Such a prefix corresponds naturally to an *incomplete* run where not all processes have finished all their steps. From the standpoint of automata, qualitative behaviors correspond to *paths* in the transition graph of the *global automaton* associated with the system which is the (Cartesian) product $\mathcal{A} = \mathcal{A}^{\mathbf{1}}||\cdots||\mathcal{A}^{\mathbf{n}}$ of the automata associated with the individual processes as illustrated in Fig. 4.3. Unfortunately, these extremely important objects are not easy to draw for non-trivial dimensions. Incomplete behaviors correspond to paths not reaching the final state.

In a global *state* of the form $(q_{j_1}^{\mathbf{1}}, \ldots, q_{j_n}^{\mathbf{n}})$ each process i is busy executing its step j_i and there is a *race* between the termination transitions. The transition $e_{j_i}^{\mathbf{i}}$ that will win will be the first to satisfy the condition $x^{\mathbf{i}} = y_{j_i}^{\mathbf{i}}$. Since $x^{\mathbf{i}}$ has been reset to zero at $t_{j_i-1}^{\mathbf{i}}$ this condition will be fulfilled at time $t_{j_i-1}^{\mathbf{i}} + y_{j_i}^{\mathbf{i}} = t_{j_i}^{\mathbf{i}}$. The outcomes of all these races are completely determined by the value of y, and this determines the qualitative behavior which is exhibited. Had there been no timing constraints on task durations, that is, $I^{\mathbf{i}}_j = [0, \infty)$, the system would be completely *asynchronous* and all interleavings would, in principle, be possible. When durations are bounded, some qualitative behaviors may become strictly impossible due to the arithmetics of timing constraints while others will occur at low probability. In the sequel we develop methods for computing these probabilities.

4.3 Computing Volumes

The computation of the probability of a qualitative behavior w is performed in two steps. First we associate with it a zone $Z_w \subseteq \mathcal{C}$ consisting of all instances of t that yield this behavior. Then we integrate over this zone to find its volume.

Let $\varphi_{\mathcal{C}}$ be the constraint describing the whole time-stamp space:

$$\varphi_{\mathcal{C}} : \bigwedge_{i \in N} \bigwedge_{j \in K} a^{\mathbf{i}}_j \leq t^{\mathbf{i}}_j - t^{\mathbf{i}}_{j-1} \leq b^{\mathbf{i}}_j$$

with $t^{\mathbf{i}}_0 = 0$ for every i. The zone Z_w for the qualitative behavior of Fig. 4.2 can be characterized by adding constraints that specify the particular order of events in w:

$$\varphi_w : \varphi_{\mathcal{C}} \wedge t^{\mathbf{1}}_1 < t^{\mathbf{2}}_1 < t^{\mathbf{2}}_2 < t^{\mathbf{3}}_1 < t^{\mathbf{2}}_3 < t^{\mathbf{1}}_2 < t^{\mathbf{1}}_3 < t^{\mathbf{3}}_2 < t^{\mathbf{3}}_3.$$

Some of these constraints appear already in $\varphi_{\mathcal{C}}$ and some are implied via transitivity by other constraints. After eliminating these redundant constraints one obtains the following description:

$$\varphi_w : \varphi_{\mathcal{C}} \wedge (t^{\mathbf{1}}_1 < t^{\mathbf{2}}_1) \wedge (t^{\mathbf{2}}_2 < t^{\mathbf{3}}_1) \wedge (t^{\mathbf{3}}_2 < t^{\mathbf{1}}_2) \wedge (t^{\mathbf{1}}_3 < t^{\mathbf{2}}_3) \wedge (t^{\mathbf{2}}_3 < t^{\mathbf{3}}_3).$$

As illustrated in Fig. 4.2, the constraints that remain in φ_w are the inter-process constraints that are sufficient to characterize w. These constraints can be computed *incrementally* as we move along the prefix of a qualitative behavior. Let us follow the first two steps. Initially we have the empty word whose associated zone is $\mathcal{C}$ and hence its probability is 1. After the occurrence of the first event $e^{\mathbf{1}}_1$ we know that $P^{\mathbf{1}}_1$ terminated before $P^{\mathbf{2}}_1$ and $P^{\mathbf{3}}_1$. This leads to the constraints:

$$\varphi_{e^{\mathbf{1}}_1} : \varphi_{\mathcal{C}} \wedge (t^{\mathbf{1}}_1 < t^{\mathbf{2}}_1) \ \wedge \ (t^{\mathbf{1}}_1 < t^{\mathbf{3}}_1) \tag{4.2}$$

After this first event we have a competition between $e^{\mathbf{2}}_1$, $e^{\mathbf{3}}_1$ and $e^{\mathbf{1}}_2$. The winner of the race is the next event of w, $e^{\mathbf{2}}_1$ and hence we add the constrains $t^{\mathbf{2}}_1 < t^{\mathbf{3}}_1$ and $t^{\mathbf{2}}_1 < t^{\mathbf{1}}_2$ and remove the constraint $t^{\mathbf{1}}_1 < t^{\mathbf{3}}_1$ which becomes redundant, yielding:

$$\varphi_{e^{\mathbf{1}}_1 e^{\mathbf{2}}_1} : \varphi_{\mathcal{C}} \wedge (t^{\mathbf{1}}_1 < t^{\mathbf{2}}_1) \ \wedge \ (t^{\mathbf{2}}_1 < t^{\mathbf{3}}_1) \ \wedge \ (t^{\mathbf{2}}_1 < t^{\mathbf{1}}_2).$$

In general whenever event $e^{\mathbf{i}}_j$ occurs, we add a constraint stating that $t^{\mathbf{i}}_j$ is smaller than the time stamps associated with all the pending events in the other processes. The incremental process is illustrated in Fig. 4.4.

This procedure is probabilistically correct in the following sense. For every $w \in \underline{L}$ the probability of all behaviors having w as a prefix is the relative volume of the corresponding zone Z_w, namely, $p(w) = |Z_w|/|\mathcal{C}|$. This holds trivially for the empty behavior when there are no constraints. For the inductive step observe that any qualitative behavior of the form $w\ e$ which extends w has to satisfy φ_w due to *causality* as well as additional constraints that guarantee that e is indeed the next event to win the race. The constraints associated with all the extension of w form a *partition* of Z_w and all the probabilistic mass $p(w)$ is split among them, satisfying

$$\sum_e p(w\ e) = p(w).$$

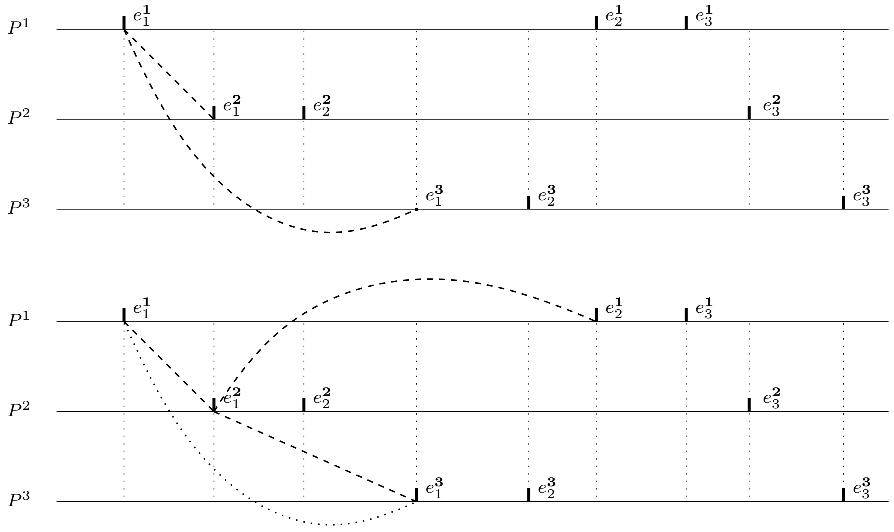


Figure 4.4: Incremental constraint construction: constraints for $e_1^{\mathbf{1}}$ and then for $e_1^{\mathbf{1}}\ e_1^{\mathbf{2}}$. The constraint $t_1^{\mathbf{1}} < t_1^{\mathbf{3}}$ becomes redundant after the second event.

In [145] a similar incremental approach that goes from a path/prefix to its successors has been developed using the clock auxiliary variables. The use of clocks required the concept of *density transformers* to account for the distribution of clock values before and after transitions (see also [35, 7, 189, 55]). These are not needed in the clock-free approach presented here. Those acquainted with the verification of timed automata using a forward computation of the simulation/reachability graph [73, 130] may notice that for every w the zone Z_w in the time-stamp space is empty exactly when its associated clock space zone in the reachability graph becomes empty. This suggests an alternative clock-free analysis algorithm for timed automata which is immediately applicable to acyclic systems but will require more work to be adapted to the cyclic case.

Having labeled qualitative behaviors by constraints we need to compute the volume of the zones. We illustrate this procedure on a concrete example with $n = 3$ and $k = 1$, hence $\mathcal{D} = \mathcal{C} = I_1^{\mathbf{1}} \times I_1^{\mathbf{2}} \times I_1^{\mathbf{3}}$, with concrete values

$$[a_1^{\mathbf{1}}, b_1^{\mathbf{1}}] = [2,5], \quad [a_1^{\mathbf{2}}, b_1^{\mathbf{2}}] = [3,4], \quad \text{and} \quad [a_1^{\mathbf{3}}, b_1^{\mathbf{3}}] = [4,7].$$

The constraints associated with all qualitative behaviors where process $P^{\mathbf{1}}$ wins the first race are

$$\varphi_{e_1^{\mathbf{1}}} : \quad (2 \le t_1^{\mathbf{1}} \le 5) \wedge (3 \le t_1^{\mathbf{2}} \le 4) \wedge (4 \le t_1^{\mathbf{3}} \le 7) \wedge (t_1^{\mathbf{1}} < t_1^{\mathbf{2}}) \wedge (t_1^{\mathbf{1}} < t_1^{\mathbf{3}}).$$

We pick an integration order $t_1^{\mathbf{3}} \prec t_1^{\mathbf{2}} \prec t_1^{\mathbf{1}}$, that is, the inside-out order of variable elimination, and rewrite $\varphi_{e_1^{\mathbf{1}}}$ as

$$\varphi_{e_1^{\mathbf{1}}} : \quad (2 \le t_1^{\mathbf{1}} \le 5) \wedge (\max(3, t_1^{\mathbf{1}}) \le t_1^{\mathbf{2}} \le 4) \wedge (\max(4, t_1^{\mathbf{1}}) \le t_1^{\mathbf{3}} \le 7)$$

Then we split $I_1^{\mathbf{1}}$ into maximal segments where both $\max(3, t_1^{\mathbf{1}})$ and $\max(4, t_1^{\mathbf{1}})$ are uniform. In our example $[2,5]$ splits into $[2,3]$, $[3,4]$ and $[4,5]$ and the volume of the set can be written as

$$\left[\int_2^3\int_3^4\int_4^7 + \int_3^4\int_{t_1^1}^4\int_4^7 + \int_4^5\int_{t_1^1}^4\int_{t_1^1}^7\right] dt_1^{\mathbf{3}} dt_1^{\mathbf{2}} dt_1^{\mathbf{1}} = 3 + \frac{3}{2} + 0 = \frac{9}{2}$$

which after dividing by $|\mathcal{C}| = 9$ gives a probability of $1/2$ for $e_1^{\mathbf{1}}$ winning the first race. Figure 4.5 illustrates two possible splits of a 2-dimensional zone into integration domains. The number of case splits and the forms of the integration domains may vary a lot depending on the chosen order.

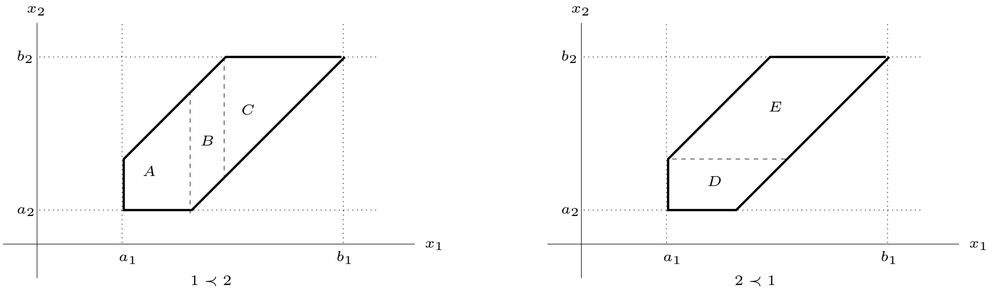


Figure 4.5: Zone volume computation by splitting into integration domains in two different integration orders which yield 3 and 2 domains, respectively.

Theorem 1 (Probability of Qualitative Behaviors)**.** *Given a system of stochastic sequential processes as in Def. 5.3.1 the probability of any of its qualitative behaviors is computable.*

The global termination time (makespan) of a behavior is $\Theta = \max\{t_k^{\mathbf{1}}, \ldots, t_k^{\mathbf{n}}\}$. For all behaviors that are qualitatively equivalent the maximum is attained by the same variable, namely $t_k^{\mathbf{i}}$ for any behavior whose last event is $e_k^{\mathbf{i}}$. To compute the expected termination time we integrate $t_k^{\mathbf{i}}$ over Z_w and sum up over all w:

$$\mathbb{E}(\Theta) = \frac{1}{|\mathcal{C}|} \sum_{i=1}^{n} \sum_{w=w'e_k^{\mathbf{i}}} \int_{Z_w} t_k^{\mathbf{i}}.$$

Before moving to the coordinated execution framework let us mention some useful observations. So far we have treated qualitative behaviors in their finest granularity, taking note of the ordering between any pair of events. In many situations we are interested in *sets* of qualitative behaviors and their probability can often be computed more efficiently than summing up the probabilities of individual qualitative behaviors.

Suppose we want to characterize the set of all qualitative behaviors that pass through a global state $q = (q_{j_1}^{\mathbf{1}}, \ldots, q_{j_n}^{\mathbf{n}})$. Let $\underline{L}_j^{\mathbf{i}} = \{e_1^{\mathbf{i}} \cdots e_{j-1}^{\mathbf{i}}\}$ be the qualitative behavior of $P^{\mathbf{i}}$ that leads to $q_j^{\mathbf{i}}$. Then the set of qualitative behaviors that lead to q is

$$\underline{L}(q) = \underline{L}_{j_1}^{\mathbf{1}} || \cdots || \underline{L}_{j_n}^{\mathbf{n}}.$$

The constraints that characterize $\underline{L}(q)$ may forget the specific interleaving, that is, the specific order in which *past* events have occurred. The only constraints that are relevant are those that guarantee that the entrance of each process into its respective local state preceded the exit of all other processes from their respective states, that is,

$$\varphi_q : \varphi_{\mathcal{C}} \wedge \bigwedge_{i=1}^{n} \bigwedge_{i' \neq i} t_{j_i - 1}^{\mathbf{i}} < t_{j_{i'}}^{\mathbf{i}'}.$$

Thus, to compute the expected termination time it suffices to partition the set of qualitative behaviors into n classes according to the identity of the *last* transition, letting Z^i be the zone defined by

$$\varphi^i : \varphi_{\mathcal{C}} \wedge \bigwedge_{i' \neq i} t_k^{\mathbf{i}'} < t_k^{\mathbf{i}}.$$

Then the expected termination time is

$$\mathbb{E}(\Theta) = \frac{1}{|\mathcal{C}|} \sum_{i=1}^{n} \int_{Z^i} t_k^{\mathbf{i}}. \tag{4.3}$$

A similar observation, made in the context of zone-based verification of timed automata, underlies the fact that the union of zones reached by interleavings of the same set of events is convex [204, 143, 26].

4.4 Conflicts and Schedulers

Now we adapt the framework to the case where steps of different processes may be at conflict due to requiring the same resource and hence cannot be executed simultaneously. Naturally, this situation is more intuitively expressed using automata, states and runs. We explain the automaton-based modeling very informally and delegate the formal definitions of DPA to Chapter 5 where they are used for deriving a state-based optimal scheduler.

As a running example consider a system of two processes with three steps each, admitting a resource conflict between their respective second steps $P^{\mathbf{1}}_2$ and $P^{\mathbf{2}}_2$. Conflicts are modeled in automata using *forbidden states* in the global automaton, state $(q^{\mathbf{1}}_2, q^{\mathbf{2}}_2)$ in our example. To be able to prevent the automaton from entering this state[4] we refine the process model so that the initiation of step $P^{\mathbf{i}}_j$ does not occur *automatically* upon the termination of step $P^{\mathbf{i}}_{j-1}$. We thus modify the process automaton shown in Fig. 4.3 by inserting a *waiting state* $\bar{q}^{\mathbf{i}}_j$ between $q^{\mathbf{i}}_{j-1}$ and $q^{\mathbf{i}}_j$. The automaton can leave this state only when it receives a *start* command $s^{\mathbf{i}}_j$ from a scheduler as illustrated in Fig. 4.6-(a).

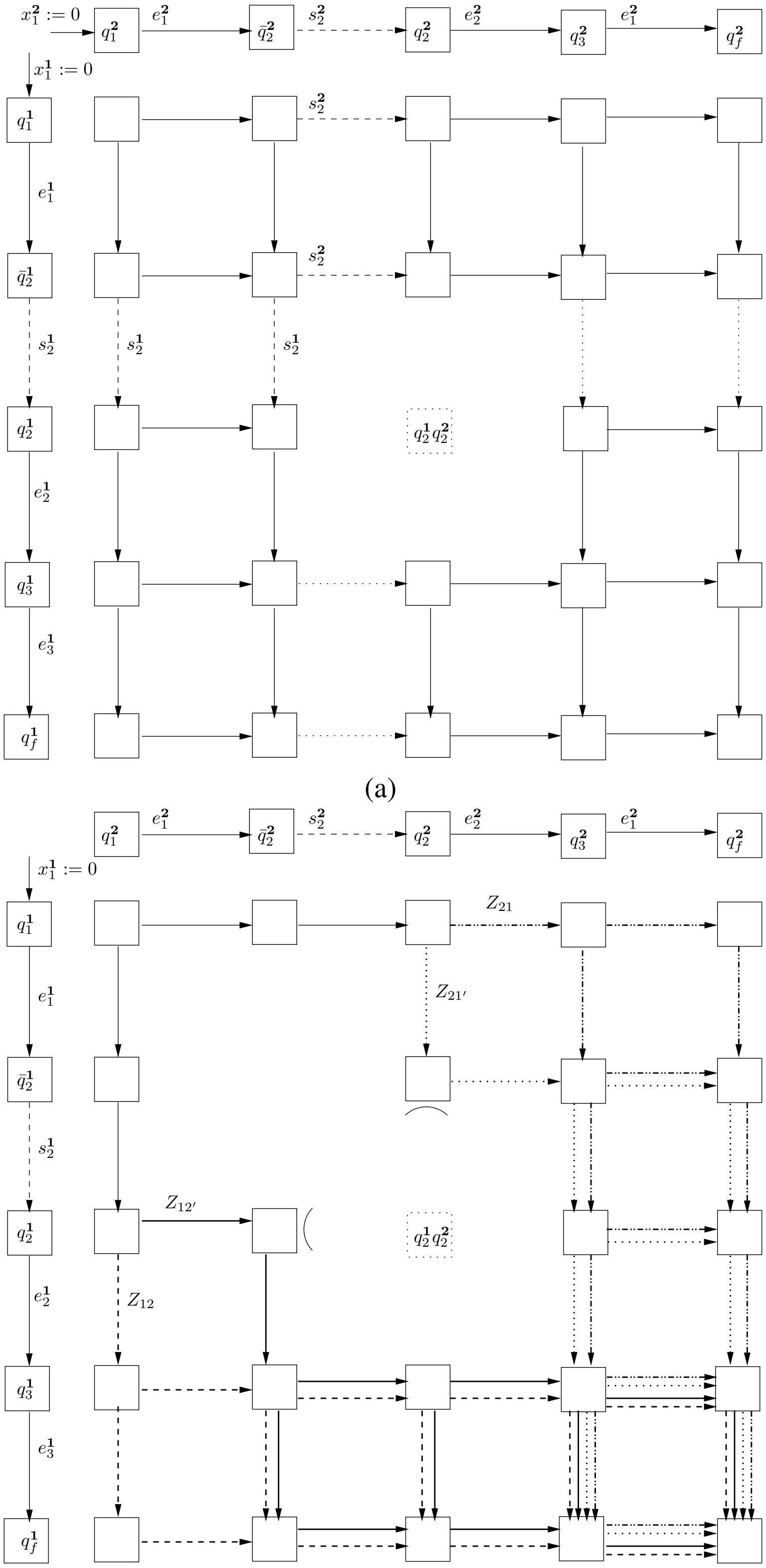
As long as the scheduler is not completely specified the system is *open* or using another terminology, admits both *probabilistic* and *set-theoretic* non-determinism. For example in state $(\bar{q}^{\mathbf{1}}_2, q^{\mathbf{2}}_1)$ process $P^{\mathbf{1}}$ may either start its second step $(\bar{q}^{\mathbf{1}}_2, q^{\mathbf{2}}_1) \rightarrow (q^{\mathbf{1}}_2, q^{\mathbf{2}}_1)$ or wait until step $P^{\mathbf{2}}_1$ terminates and let $P^{\mathbf{2}}$ take the resource first $(\bar{q}^{\mathbf{1}}_2, q^{\mathbf{2}}_1) \rightarrow (\bar{q}^{\mathbf{1}}_2, \bar{q}^{\mathbf{2}}_2) \rightarrow (\bar{q}^{\mathbf{1}}_2, q^{\mathbf{2}}_2)$. A scheduler resolves this type of non-determinism by telling each process in a waiting state whether to take the resource and proceed to execution or wait until the resource is taken and released by another process.[5] Once such a scheduler is defined, the set-theoretic non-determinism is eliminated and the only non-determinism that remains is the one associated with task durations and thus it becomes possible to compute probabilities. To be more precise, probabilities can be computed also for non-deterministic schedulers that make a probabilistic choice, but we do not consider them here.

A scheduler is thus a mechanism which may observe the state of the system and decide whether to grant a resource to a process, possibly based on the level of progress of other processes. The most passive scheduler grants the resource to the *first* process whose corresponding step becomes enabled. Under such a FIFO scheduling policy it is the result of the race between $e^{\mathbf{1}}_1$ and $e^{\mathbf{2}}_1$ which determines the resource granting decision. The automaton obtained by composing the system with such a scheduler is shown in Figure 4.6-(b) where we have chosen to ignore the zero-measure situation when both processes terminate *exactly* at the same time (alternatively this situation can be handled by assigning an arbitrary priority when this is the case).

More active schedulers interfere with the execution order by imposing additional conditions upon the *start* transitions. Suppose that the duration of step $P^{\mathbf{1}}_3$ is much longer than that of $P^{\mathbf{2}}_3$ hence it would be reasonable to give $P^{\mathbf{1}}_2$ a priority over $P^{\mathbf{2}}_2$ even if the latter becomes enabled earlier. This priority can have different degrees of rigidity. A *strict* priority scheduler allows $s^{\mathbf{2}}_2$ only in global states where $P^{\mathbf{1}}_2$ has terminated, a condition that we write as $\mathcal{A}^{\mathbf{1}} > q^{\mathbf{1}}_2$. The automaton obtained by composing the system with such a scheduler is shown in Fig. 4.7. Note that strict priority schedulers make the automaton always "bypass" a conflict state from the same side.

[4] We consider schedulers that by construction cannot make the system enter a forbidden state.

[5] Note that we restrict ourselves to *non-lazy* schedulers: if they do not issue an $s^{\mathbf{i}}_j$ command at some point, they will not issue it later unless another process has utilized the resource. This class has been shown [2] to contain the optimal schedulers for deterministic problems and its extension to our setting is discussed in Chapter 5.



(a)

(b)

Figure 4.6: (a) Two parallel processes admitting a resource conflict and their product automaton. The dashed arrows indicate *start* transitions which should be under the control of a scheduler while the dotted arrows indicate post-conflict *start* transitions; (b) The automaton resulting from composition with a FIFO scheduler and the 4 potential conflict resolution and resource utilization scenarios.

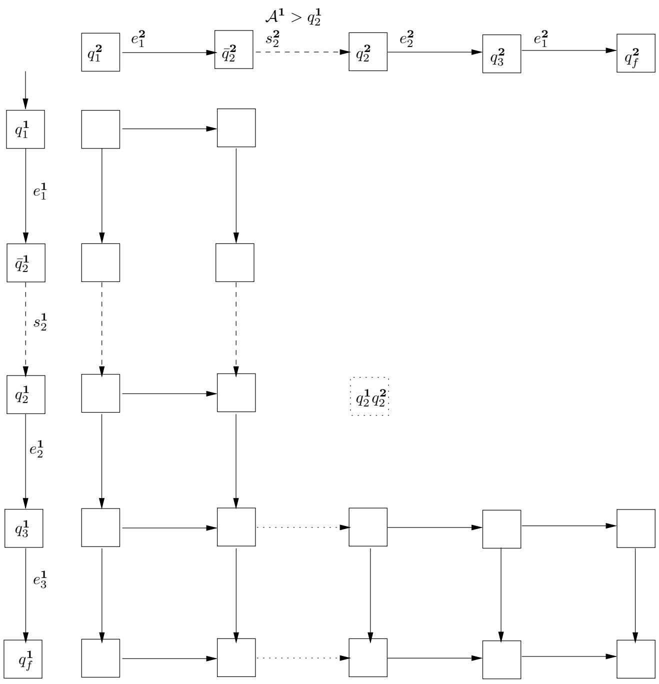


Figure 4.7: (a) A scheduler that gives strict priority to P^1. This is realized by the condition $\mathcal{A}^1 > q_2^1$ which allows P_2^2 to start only after P_2^1 terminates.

Strict priority schedulers can be unnecessarily rigid for tasks with a significant variability in durations as they do not adapt to the actual evolution of the schedule. As an example for such adaptability consider the case where P_1^2 terminates very early so that we can start P_2^2 so that it will surely terminate before P_2^1 becomes enabled and hence will not block it. Even if this is not guaranteed with certainty, a scheduler might want to start P_2^2 if the expected delay incurred to P^1 is small. Technically, the knowledge of the relative timing of e_1^2 at decision time is encoded by the value of clock x^1 reset upon starting P_1^1. The larger is the value of x^1, the more we are likely to block P^1 and for a longer period. Hence the condition for issuing s_2^2 by such a state-dependent scheduler will be of the form $(\mathcal{A}^1 > q_2^1) \vee (\mathcal{A}^1 < \bar{q}_2^1 \wedge x^1 < d)$ for some constant d.

The labeling of states and qualitative behaviors with constraints in order to compute volumes, probabilities and expected termination times can be extended to handle all these types of schedulers. As an illustration consider the FIFO scheduler of Fig. 4.6-(b) which admits 4 classes of qualitative behaviors (scenarios) that correspond to the outcomes of the conflict between P^1 and P^2 on the shared resource. These scenarios are characterized by the identity of the winner (for this scheduler it depends on the relation between t_1^1 and t_1^2) and by whether the loser termination time is delayed (depending on whether the winner releases the resource before the loser becomes enabled). These cases are summarized in Table 4.1 and depicted in Fig. 4.6-(b).

The transformation T from the duration space to the time-stamp space is different from the independent execution framework. It can nevertheless be shown to be volume preserving along the following lines. First, one can show that after adding inter-process precedence constraints causality is preserved and there is always a rearrangement of the indices such that the transformation matrix remains lower triangular. Secondly the notion of volume preservation can be easily generalized from linear to piecewise-linear transformations.

The above analysis can be generalized to m distinct resources and to multi-party conflicts on each of them. For each resource l one can compute the set U_l of all the utilization scenarios for this resource and their respective zones. A scenario corresponds to a particular order of resource

winner	loser delayed		loser not delayed
$P^{\mathbf{1}}$	$\begin{array}{rcl} \multicolumn{3}{c}{Z_{12'}} \\ \hline t_1^{\mathbf{1}} < & t_1^{\mathbf{2}} & \\ t_1^{\mathbf{1}} + a_2^{\mathbf{1}} < & t_2^{\mathbf{1}} & < t_1^{\mathbf{1}} + b_2^{\mathbf{1}} \\ & t_1^{\mathbf{2}} & < t_2^{\mathbf{1}} \\ t_2^{\mathbf{1}} + a_2^{\mathbf{2}} < & t_2^{\mathbf{2}} & < t_2^{\mathbf{1}} + b_2^{\mathbf{2}} \end{array}$		$\begin{array}{rcl} \multicolumn{3}{c}{Z_{12}} \\ \hline t_1^{\mathbf{1}} < & t_1^{\mathbf{2}} & \\ t_1^{\mathbf{1}} + a_2^{\mathbf{1}} < & t_2^{\mathbf{1}} & < t_1^{\mathbf{1}} + b_2^{\mathbf{1}} \\ t_2^{\mathbf{1}} < & t_1^{\mathbf{2}} & \\ t_1^{\mathbf{2}} + a_2^{\mathbf{2}} < & t_2^{\mathbf{2}} & < t_1^{\mathbf{2}} + b_2^{\mathbf{2}} \end{array}$
$P^{\mathbf{2}}$	$\begin{array}{rcl} \multicolumn{3}{c}{Z_{21'}} \\ \hline & t_1^{\mathbf{2}} & < t_2^{\mathbf{1}} \\ t_1^{\mathbf{2}} + a_2^{\mathbf{2}} < & t_2^{\mathbf{2}} & < t_1^{\mathbf{2}} + b_2^{\mathbf{2}} \\ t_2^{\mathbf{2}} < & t_1^{\mathbf{1}} & \\ t_1^{\mathbf{2}} + a_2^{\mathbf{2}} < & t_2^{\mathbf{2}} & < t_1^{\mathbf{2}} + b_2^{\mathbf{2}} \end{array}$		$\begin{array}{rcl} \multicolumn{3}{c}{Z_{21}} \\ \hline & t_1^{\mathbf{2}} & < t_2^{\mathbf{1}} \\ t_1^{\mathbf{2}} + a_2^{\mathbf{2}} < & t_2^{\mathbf{2}} & < t_1^{\mathbf{2}} + b_2^{\mathbf{2}} \\ & t_1^{\mathbf{1}} & < t_2^{\mathbf{2}} \\ t_1^{\mathbf{2}} + a_2^{\mathbf{2}} < & t_2^{\mathbf{2}} & < t_1^{\mathbf{2}} + b_2^{\mathbf{2}} \end{array}$

Table 4.1: The zones corresponding to the four possible outcomes of the resource conflict of Fig. 4.7-(b). Constraints on $t_3^{\mathbf{1}}$ and $t_3^{\mathbf{2}}$ as well as the bounding constraints on $t_1^{\mathbf{1}}$ and $t_1^{\mathbf{2}}$ are omitted.

utilization by conflicting steps and to the waiting delays incurred to these steps. Then the classes of potential qualitative behaviors of interest are the combinations of those, that is, $U = U_1 \times \cdots \times U_m$ with zones defined by intersection. While this sounds like a recipe for a severe combinatorial explosion, note that many scenarios will lead to empty zones, either for logical reasons (inter-process ordering of conflicting steps is incompatible with local precedence constraints) or due to the arithmetics of timing constraints (two conflicting tasks, one at the beginning and one at the end of their respective processes, are likely to be executed in one order). Naturally, for priority schedulers there will be fewer scenarios to analyze.

4.5 Implementation and Experimental Results

We have implemented a prototype tool which computes expected termination times as described in this chapter. As input it takes a system description consisting of processes, steps, duration intervals and conflicts as well as a definition of a scheduling policy. Then for every utilization scenario it derives the corresponding zone, using the DBM library of IF [49] to normalize constraints and detect empty zones. Then it performs integration over the non-empty zones to compute probability and expected termination time. The integration uses the *GNU Multiple Precision Arithmetic Library* (GMP) to avoid rounding errors. Below we give more technical details.

System Description

The prototype tool takes as input a textual description of the system, specified by the following grammar:

```
system-decl ::= { process-decl }* { conflict-decl }*
process-decl ::= process-id { step-decl }
step-decl ::= [ l , u ]
conflict-decl ::= shared := type { step-id , { step-id }* }
type ::= STRICT
       | FIFO
step-id ::= process-id.step-num
```

Conflicts are defined through *shared* variables by specifying all steps $P_j^{\mathbf{i}}$ that use the same resource. An example of such description is depicted in Listing-4.1 where processes $P^{\mathbf{1}}$ and $P^{\mathbf{2}}$ execute their respective second step on the same resource.

```
P1 { [2,8] [3,7] [4,8] }
P2 { [4,10] [4,12] [2,5] }

shared := STRICT { P1.s2 , P2.s2 }
```

Listing 4.1: Example of description

Constraints Generation

Constraints are generated in two steps. First we generate those implied by independent steps, those where *starting* and *ending* do not influence steps from other processes. Constraints for conflicting steps are handled in a separated step and imply *cases splitting*. Depending on scheduling policy, the termination of steps before the conflict influences execution order on the shared resource and also the start time of successors steps.

As an example consider 2 process $P^{\mathbf{1}}$ and $P^{\mathbf{2}}$ having 3 steps and executing their respective second step $P_2^{\mathbf{1}}$ and $P_2^{\mathbf{2}}$ on the same resource. For a FIFO policy there are 4 cases as shown in Table-4.1 and for a strict policy with $P^{\mathbf{1}} > P^{\mathbf{2}}$ there are 2 cases:

- $t_2^{\mathbf{1}} < t_1^{\mathbf{2}}$ implies $P_2^{\mathbf{2}}$ starts after $P_2^{\mathbf{1}}$ (real conflict)
- $t_2^{\mathbf{1}} > t_1^{\mathbf{2}}$ implies $P_2^{\mathbf{2}}$ starts after $P_1^{\mathbf{2}}$ (false conflict)

Depending on the number of conflicts several cases needs to be computed. The number of cases for a single *FIFO scheduler* implying s steps is bounded by $(2^{(s-1)} \cdot s!)$. This bound for a *STRICT scheduler* implying s steps is 2^s. The total number of cases for n processes with i resources sharing s_i steps in FIFO and j resources sharing s_j steps in strict policy is then bounded by:

$$n \cdot \prod_i (2^{(s_i-1)} \cdot s_i!) \cdot \prod_j 2^{(s_j)}$$

Note that during generation, constraints are also encoded as zones, enabling elimination of unfeasible cases, by emptiness check.

Finally for computing expected time we need to know the variable corresponding to the last step responsible for the global execution time. This splits all previous cases according to the identity of the last process, by adding constraints $\forall i, t_n^{\mathbf{i}} < t_n^{\mathbf{last}}$. Expected time is then computed according to variable $t_n^{\mathbf{last}}$ in each case.

```
// order:
// [1,4,2,5,3,6]
case1
{
Dt1 := [2,8]
2<t1<8
Dt4 := [4,10]
4<t4<10
Dt2 := [3,7]
3<t2-t1<7
t1<t2
//False conflict
Dt5 := [4,12]
4<t5-t4<12
t4<t5
t2<t4
t4<t5
Dt3 := [4,8]
4<t3-t2<8
t2<t3
Dt6:=[2,5]
2<t6-t5<5
t5<t6
t3<t6
expT := t6
}
```

Listing 4.2: Case 1

```
// order:
//[1,4,2,5,6,3]
case2
{
Dt1 := [2,8]
2<t1<8
Dt4 := [4,10]
4<t4<10
Dt2 := [3,7]
3<t2-t1<7
t1<t2
//False conflict
Dt5:=[4,12]
4<t5-t4<12
t4<t5
t2<t4
t4<t5
Dt3 := [4,8]
4<t3-t2<8
t2<t3
Dt6 := [2,5]
2<t6-t5<5
t5<t6
t6<t3
expT := t3
}
```

Listing 4.3: Case 2

```
// order:
// [1,4,2,5,3,6]
case3
{
Dt1 := [2,8]
2<t1<8
Dt4 := [4,10]
4<t4<10
Dt2 := [3,7]
3<t2-t1<7
t1<t2
//Real conflict
Dt5 := [4,12]
4<t5-t2<12
t2<t5
t4<t2
t4<t5
Dt3 := [4,8]
4<t3-t2<8
t2<t3
Dt6 := [2,5]
2<t6-t5<5
t5<t6
t3<t6
expT := t6
}
```

Listing 4.4: Case 3

```
// order:
//[1,4,2,5,6,3]
case4
{
Dt1 := [2,8]
2<t1<8
Dt4 := [4,10]
4<t4<10
Dt2 := [3,7]
3<t2-t1<7
t1<t2
//Real conflict
Dt5 := [4,12]
4<t5-t2<12
t2<t5
t4<t2
t4<t5
Dt3 := [4,8]
4<t3-t2<8
t2<t3
Dt6 := [2,5]
2<t6-t5<5
t5<t6
t6<t3
expT := t3
}
```

Listing 4.5: Case 4

For the example depicted in Listing-4.1, the constraints are split into 4 cases corresponding to the several possible orders of execution, shown in Listing 4.2, 4.3, 4.4, 4.5. Note that, here, each step of the processes are associated with a variable t_i defined in a lexicographic order i.e process $P^{\mathbf{1}}\{t_1, t_2, t_3\}$ and process $P^{\mathbf{2}}\{t_4, t_5, t_6\}$. Variable Dt_i defines the domain of each steps and variable $expT$ correspond to the last step leading to the global execution time, so expected time will be computed according to variable $expT$.

Computation

For each case, resulting from constraint generation we compute the volume of the corresponding polyhedron defined as a zone. This volume is divided by the domain of each variable to get the probability of each case. The associated expected time value is computed according to the variable which defines the execution of the last step in the global behavior. Results for the example of Listing-4.1 is shown in Table-4.2.

Case	Probability	Expected time
1	0.143017	20.2595
2	0.000501644	15.654
3	0.844586	21.9855
4	0.0118956	17.9216
global	1	21.6867

Table 4.2: Expected time for example of Listing-4.1

At implementation, the symbolic volume SV is represented as a list of pairs $\langle Z_i, E_i \rangle$ where Z_i is a zone and E_i is a polynomial on t variables. Recall φ_C is the constraint describing the whole time-stamp space, $Z_i \subseteq C_i$ with φ_{C_i} is the constraint of each case.

Initially $SV = \langle Z_i, 1 \rangle$ and then, iteratively, we eliminate all variables t , that is we compute the integral of E_i with respect to t. At each iteration, SV grows depending on the number of splits implied by the integration calculation. We explain in the following how this computation works.

Variable Elimination

The variable elimination computation is shown in Algorithm-2. The function *isDegenerate()* checks if the zone is not empty and if all bounds $l \leq x_i - x_j \leq u$ are thick that is $l \neq u$. The function *compact()* merges nodes with the same zone to avoid splitting explosion.

In general integration takes place in $\mathbb{R}^{nk}$ and its complexity depends on the following factors. First, the number of scenarios (orders of resource utilizations and their combinations) determines the number of zones whose volume we might need to compute, in case they are not detected beforehand to be empty. Then the order of variables elimination influences the number of splits during integration as shown in Fig-4.5. The naive order is to eliminate variable in a lexicographic order (Order 1) from t_1 to t_n but this gives bad results. To counter that, different heuristics have been implemented. One of them consists in computing the minimal constraints system [129] for each zone and then the least constrained variable is eliminated (Order 2). Topological ordering is another one and consists in eliminating variables according to the reverse order implied by the global execution. For the case depicted in Listing-4.2 the order of elimination using this heuristic is:

$$t_6 \prec t_3 \prec t_5 \prec t_2 \prec t_4 \prec t_1$$

In a similar heuristic, for each zone and each variable t we compute I_t, the projection of the zone on t. Then we define a partial order relation between these intervals such that $I_t < I_{t'}$ if the upper

Algorithm 2 Variable elimination

```
function ELIMINATE(X,L)
//L is a list of nodes, X is a the variable to eliminate //Here, elimination means computation of the integral with respect to X
    Res ← $\emptyset$                                        ▷ The resulting node list
    for all $N = \langle Z, E\rangle \in L$ do               ▷ Z and E are zone and polynomial of node N
        for $i = 0 \to N_Z$ do                               ▷ $N_Z$ is the dimension of $Z$
            if $i \neq x$ then
                for $j = 0 \to N_Z$ do
                    $Z_c$ ← new ZONE                         ▷ $Z_c$ encode bounds constraints
// Split and enforce these bounds as maximal, resp minimal
                    for $k = 0 \to N_Z$ do
                        if $k \neq x \wedge k \neq i$ then
                            $Z_c \wedge \{X_k - X_i \leq Z[k][x] - Z[i][x]\}$
                        end if
                    end for
                    for $k = 0 \to N_Z$ do
                        if $k \neq x \wedge k \neq j$ then
                            $Z_c \wedge \{X_j - X_k \leq Z[x][k] - Z[x][j]\}$
                        end if
                    end for
// Intersect bounds constraints with current zone
                    $Z_{split} \leftarrow Z \wedge Z_c$
                    if $\neg$ isDegenerate($Z_{split}$) then  ▷ check if current zone is not degenerated
// Create a new node with $Z_{split}$ and integrate current polynomial $E$
                        Res ← Res $\cup\langle Z_{split}, \int_{X_i-Z[i][x]}^{X_j-Z[x][j]}$ E d$X\rangle$
                    end if
                end for
            end if
        end for
    end for
    compact(Res)
end function
```

bound of I_t is smaller than the lower bound of $I_{t'}$. Then we construct a compatible linear order and integrate backwards (Order 3).

The chosen order determines the number of case splits but also the form of the integration domains and the polynomials obtained during integration. We experienced orders of integration that generate more splits but take less overall computation time. However, experimental result show that lexicographic order behaves the worst. None of the other heuristics is universally better than the other. For the moment we have no systematic explanation for these variations.

We show in Tables 4.3 and 4.4 the difference on the number of splits during integration according to the chosen elimination ordering. Order 1,2 and 3 stand for the ordering heuristics presented above. Each row corresponds to a non-empty case. Column *Splits* indicates the number of splits that occurred during integration (i.e the maximum size of the list SV) and column *Int* indicates the number of integrations that have been performed (at least as many as the number of variables incremented by the number of splits). Column *Proba* and *ExpectedTime* give the probability and expected time for each case. The row *Time* shows the execution time for computing all cases. We see that the number of splits is extremely sensitive to the elimination order.

```
P1 { [34,41] [36,42] [35,42] [38,48] [32,40] [35,44] }
P2 { [40,49] [34,37] [38,46] [31,36] [40,44] [35,41] }

shared := FIFO { P1.s2 , P2.s4 }
shared := FIFO { P1.s4 , P2.s3 }
```

	Order 1		**Order2**		**Order3**			
Case	**Splits**	**Int**	**Splits**	**Int**	**Splits**	**Int**	**Proba**	**ExpectedTime**
1	11125	22408	70	175	80	190	0.76160	242.13086
2	7215	15948	76	190	82	192	0.15962	237.73817
3	4730	9672	463	811	442	788	0.07084	238.54240
4	4050	8687	475	837	434	774	0.00794	233.20148
Global	27120	56715	1084	2013	1038	1944	1	241.105
Time	2m5.003s		0m12.381s		0m10.196s			

Table 4.3: Example 1

```
P1{[36,40][33,41][39,42][37,45][40,45]}
P2{[38,46][37,42][31,35][31,41][37,46]}
P3{[31,41][32,39][40,45][40,43][31,35]}

shared:=FIFO{P1.s4,P2.s4}
shared:=FIFO{P1.s2,P3.s3}
shared:=FIFO{P2.s2,P3.s2}
```

	Order 1		**Order2**		**Order3**			
Case	**Splits**	**Int**	**Splits**	**Int**	**Splits**	**Int**	**Proba**	**ExpectedTime**
1	7579	16550	1214	2298	362	731	0.00260	234.50898
2	8144	18244	1450	2653	378	760	0.00436	233.28736
3	3944	7203	42	108	31	82	0.75366	234.54334
4	726	1788	28	76	21	59	0.00000	227.77534
5	6476	12343	115	249	128	279	0.18376	231.99327
6	5236	11386	106	233	97	220	0.00633	228.96075
7	19881	33919	1605	2978	355	732	0.02428	233.50729
8	timeout 10min		1716	3133	369	756	0.02501	232.10406
Global	51986	43785	6276	5135	1741	1632	1	233.948
Time	timeout 10min		1m34.070s		0m31.207s			

Table 4.4: Example 2

Experiments

We started conducting some more systematic experiments to assess the feasibility bounds of this approach. For each value of n from 1 to 5 and for each value of k from 1 to 40, we choose a number of conflicts (between 0 and 3) and a number of participants in each conflict (2 or 3). Each choice in this space defines a problem type for which we draw 10 concrete problems by randomly choosing the identity of the conflicting steps as well as step duration intervals of the form $[c-d, c+d]$ with c drawn uniformly in $[40, 50]$ and d in $[0, c/20]$. Then we try to compute expected termination times for a FIFO scheduler with a timeout of 3 minutes per problem on a computer with a Pentium processor at 2.0GHz and 2 GB of memory.

The experiments with $n = 1$ compute the volume of one zone, the time-stamp space. Applying the reverse order integration we can compute up to dimension 63 in 0.4 seconds (currently this is a limitation of our DBM library). This performance is due to the fact that the elimination ordering is optimal in the sense that no split is created during computation and, in that case, the number of integration steps is minimal and equals the number of variables.

As already mentioned we tested several orders of integration that generate more or less splits. Since there is a lot of exploration and fine tuning ahead, this is definitely not the last word on the topic. To give an idea, we mention some problem types for which we managed to compute for all the test cases. These include $(n, k) = (2, 12)$ with 2 conflicts, $(3, 6)$ and $(4, 6)$ with 3 binary conflicts or 2 ternary conflicts and $(5, 4)$ with 2 binary conflicts. The results are listed in Table-4.5. Column $<$ *3 min* indicates the percentage of examples which have been computed in less than 3 minutes.

4.6 Conclusions

We have presented a computational technique to evaluate schedulers in a non-Markovian setting. To the best of our knowledge no similar computational results have been reported. The analysis was based on splitting the space of valuations of the random variables and computing volumes. This technique relies heavily on computing volumes and integrals over zones. Finding good heuristics for this activity is a challenging research problem. We mention some future work for a longer term:

1. To analyze larger systems one needs to develop algorithms that do not explore all classes of qualitative behaviors but restrict the exploration to a high-volume small subset of those, whenever such exists.

2. Another major challenge is to extend this framework to *cyclic* systems, define the appropriate performance measures and study their steady-state behavior.

3. Finally, it would be interesting to compare the analytic method developed here with statistical approaches based on random simulation. It is intriguing to see how many simulation runs are needed to approximate our results with a good confidence.

Process	Steps/Process	Conflict	Steps/Conflict	Computation Time			< 3 min
				min	mean	max	
1	63	0	0	0.404	0.408	0.412	100.00%
2	10	0	0	1.992	12.372	18.301	100.00%
2	15	0	0	110.471	153.084	178.868	100.00%
2	6	2	2	0.004	0.434	2.748	100.00%
2	8	1	2	0.008	10.848	51.459	100.00%
2	10	1	2	0.012	7.611	17.313	100.00%
2	10	2	2	0.012	16.308	63.164	100.00%
2	12	1	2	0.080	45.890	110.343	100.00%
3	2	0	0	0.004	0.011	0.024	100.00%
3	3	0	0	0.008	0.117	0.332	100.00%
3	4	0	0	0.004	1.188	3.912	100.00%
3	5	0	0	0.032	16.959	59.336	100.00%
3	6	0	0	1.400	51.745	152.454	90.00%
3	7	0	0	2.004	35.927	183.407	70.00%
3	3	1	2	0.004	0.005	0.008	100.00%
3	3	1	3	0.004	0.008	0.016	100.00%
3	5	1	2	0.008	3.431	15.697	100.00%
3	5	1	3	0.008	0.077	0.524	100.00%
3	6	1	2	0.008	15.868	89.006	100.00%
3	6	2	2	0.012	1.051	5.936	70.00%
3	6	2	3	5.976	5.976	5.976	90.00%
3	6	3	2	0.012	7.562	41.059	90.00%
3	8	1	2	0.040	0.573	1.564	30.00%
3	8	2	2				0.00%
3	8	2	3	3.784	37.986	62.284	50.00%
4	4	0	0	1.488	27.823	116.159	100.00%
4	5	0	0	0.148	58.457	128.340	70.00%
4	3	1	2	0.004	0.006	0.008	100.00%
4	3	2	2	0.056	0.228	0.444	100.00%
4	5	1	2	0.016	53.420	205.249	80.00%
4	5	2	2	0.036	0.040	0.044	40.00%
4	5	2	3	0.044	26.483	169.223	100.00%
4	6	1	2	0.088	60.349	181.943	70.00%
4	6	2	2	0.024	16.078	96.242	60.00%
4	6	2	3	0.104	4.558	17.477	80.00%
4	6	3	2	0.368	29.363	148.921	60.00%
5	2	0	0	0.012	0.211	0.824	100.00%
5	3	0	0	0.008	7.952	46.395	100.00%
5	4	0	0	0.012	38.167	183.523	100.00%
5	5	0	0	1.344	76.169	151.653	40.00%
5	3	2	2	0.032	0.416	1.136	100.00%
5	4	1	2	0.012	22.474	132.820	80.00%
5	4	2	2	0.016	15.618	99.534	80.00%
5	4	3	2	0.016	18.035	157.926	90.00%

Table 4.5: Experiments results

Chapter 5

Duration Probabilistic Automata: Synthesis

We now move to the *synthesis* of optimal schedulers that minimize the expected termination time. We develop techniques for value/policy iteration (dynamic programming) that compute a value function and optimal action for any point in the extended state space (including clock values). To this end we define a *stochastic time-to-go* function which assigns to any state of the schedule (global state of the automaton and the values of active clocks) the density of the time to total termination starting from this state under the optimal strategy. These functions are piecewise-continuous and we show how they can be computed backwards from the final state. This will require several definitions.

5.1 Preliminary Definitions

Definition 5.1.1 (Bounded Support Time Density)**.** *A time density is a function $\psi : \mathbb{R}_+ \rightarrow \mathbb{R}_+$ satisfying*

$$\int_0^\infty \psi[t]dt = 1.$$

A time density is of bounded support when $\psi(t) \neq 0$ iff $t \in I = [a, b]$. A partial time density satisfies the weaker condition: $\int \psi[t]dt < 1$. A time density is uniform if $\psi[t] = 1/(b-a)$ inside its support $[a, b]$.

We will use non-standard notation for *distributions*:

$$\psi[\leq t] = \int_0^t \psi[t']dt' \quad \psi[> t] = 1 - \psi[\leq t]$$

with $\psi[\leq t]$ indicating the probability of a duration which is at most t. We use $\mathbf{c}$ to denote the "deterministic" (Dirac) density which gives the constant c with probability 1, which would be written in density terms as

$$\psi[t] = \begin{cases} \infty & \text{if } t = c \\ 0 & \text{otherwise} \end{cases}$$

The *expected value* of a time density ψ is $\mathbb{E}(\psi) = \int \psi[t] \cdot tdt$.

We will use such densities to specify durations of tasks (process steps) as well as the remaining time to termination given some state of the system. To this end we need the following *operators* on densities:

1. *Convolution*, to characterize the duration of two or more tasks composed sequentially;

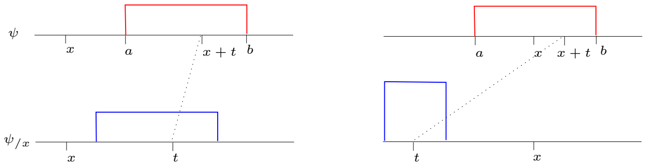


Figure 5.1: The shift operator $\psi_{/x}$. left: $x < a$; right: $a < x < b$. The intuition is that the passage of time shifts the rectangle for ψ backward by x until $\psi[x + t]$ coincides with $\psi_{/x}[t]$. When $x > 0$ the rectangle reaches the zero (causality) wall and the (fixed) volume is pushed upwards.

2. *Shift*, to reflect the change in the remaining time *given* that the process has *already* spent some amount of time x.

Definition 5.1.2 (Convolution and Shift)**.** *Let ψ , ψ_1 and ψ_2 be uniform densities supported by $I = [a, b]$, $I_1 = [a_1, b_1]$ and $I_2 = [a_2, b_2]$, respectively.*

- *The convolution $\psi_1 * \psi_2$ is a density ψ' supported by $I' = I_1 \oplus I_2 = [a_1 + a_2, b_1 + b_2]$ defined as*

$$\psi'[t] = \int_0^t \psi_1[t']\psi_2[t - t']dt'$$

- *The residual density (shift) of ψ relative to a real number $0 \leq x < b$ is $\psi' = \psi_{/x}$ such that*

$$\psi'[t] = \psi[x + t] \cdot \gamma_{a,b}(x)$$

where

$$\gamma_{a,b}(x) = \begin{cases} 1 & \textit{if } 0 < x < a \\ \frac{b-a}{b-x} & \textit{if } a < x < b \end{cases}$$

Note that when $x < a$, $\psi_{/x}$ is a simple shift of ψ. When $x > a$ we already know that the actual duration is at least x (restricted to the sub-interval $[x, b]$) and hence we need to normalize, as illustrated in Figure 5.1.[1] Note also that $(\psi * \mathbf{c})[t] = (\mathbf{c} * \psi)[t] = \psi[t - c]$ and that $\mathbf{0}$ is the identity element for convolution. We can write $\psi' = \psi_{/x}$ more explicitly as

$$\psi'[t] = \begin{cases} 0 & \text{when } x + t < a \\ \frac{1}{b-a} & \text{when } x < a, a < x + t < b \\ \frac{1}{b-x} & \text{when } a < x, x + t < b \\ 0 & \text{when } b < x + t \end{cases} \tag{5.1}$$

One can verify that the shift satisfies:

1. $\psi_{/x}[t - x] = \gamma_{a,b}(x) \cdot \psi[t]$.

2. $(\psi_{/x})_{/y} = \psi_{/(x+y)}$.

A subset of a hyper-rectangle is called a *zone* if it can be expressed as a *conjunction* of *orthogonal* and *difference* constraints, namely constraints of the form $x_i \leq c$, $x_i - x_{i'} \leq c$, etc.

[1] The definition can be extended to any bounded-support density with the normalization factor being $\int_a^b \psi[t]dt / \int_x^b \psi[t]dt$.

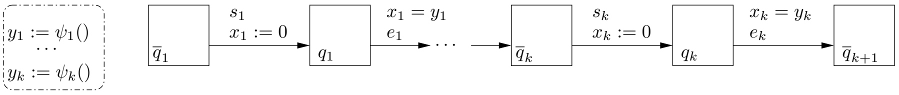


Figure 5.2: A simple DPA.

5.2 Processes in Isolation

The processes in our model are inspired by the jobs in the job-shop problem. Each process consists of an ordered sequence of steps such that a step can start executing only after its predecessor has terminated.[2] For simplicity of notation we assume all processes to have the same number k of steps and let $K = \{1, \ldots, k\}$.

Definition 5.2.1 (Process). *A sequential stochastic process is a pair $P = (\mathcal{I}, \Psi)$ where $\mathcal{I} = \{I_j\}_{j \in K}$ is a sequence of duration intervals and $\Psi = \{\psi_j\}_{j \in K}$ is a matching sequence of densities with ψ_j being the uniform density over $I_j = [a_j, b_j]$, indicating the duration of step j.*

Probabilistically speaking, step durations can be viewed as a finite sequence of *independent* uniform random variables $\{y_j\}_{j \in K}$ that we denote as points $y = (y_1, \ldots, y_k)$ ranging over a *duration space* $D = I_1 \times \cdots \times I_k \subseteq \mathbb{R}^k$ with density $\psi(y_1, \ldots, y_k) = \psi_1(y_1) \cdots \psi_k(y_k)$. A state-based representation of a process is given by simple DPA.

Definition 5.2.2 (SDPA). *A simple duration probabilistic automaton (SDPA) of k steps is a tuple $\mathcal{A} = (\Sigma, Q, \{x\}, Y, \Delta)$ where $\Sigma = \Sigma_s \uplus \Sigma_e$ is the alphabet of* start *and* end *actions with $\Sigma_s = \{s_1, \ldots, s_k\}$ and $\Sigma_e = \{e_1, \ldots, e_k\}$. The state space is an ordered set $Q = \{\overline{q}_1, q_1, \overline{q}_2, \ldots, q_k, \overline{q}_{k+1}\}$ with $\overline{q}_j$ states considered* idle *and q_j states are* active*, x is a clock variable and $Y = \{y^1, \ldots, y^k\}$ is a set of auxiliary random variables. The transition relation Δ consists of tuples of the form (q, g, r, q') with q and q' being the source and target of the transition, g is a guard, a precondition (external or internal) for the transition and r is an action (internal or external) accompanying the transition. The transitions are of two types:*

1. Start *transitions: for every idle state $\overline{q}_j$, $j < k + 1$ there is one transition of the form $(\overline{q}_j, s_j, \{x\}, q_j)$. The transition, triggered by a scheduler command s_j, activates clock x is resets it to zero;*

2. End *transitions: for every active state q_j, there is a transition, conditioned by the clock value, of the form $(q_j, x = y_j, e_j, \overline{q}_{j+1})$. This transition renders clock x inactive and outputs an e_j event.*

The operational interpretation is the following (see Fig. 5.2). First, for each step j we draw a duration y_j according to ψ_j. Upon a scheduler command s_j the automaton moves from a waiting state $\overline{q}_j$ to active state q_j in which clock x advances with derivative 1. The *end* transition is taken when $x = y_j$, that is, y_j time after the corresponding *start* transition. An extended state (configuration) of the automaton a pair (q, x) consisting of a discrete state and a clock value[3] which represents the time elapsed since the last *start* transition. The generalized state-space of the SPDA is thus

$$S = \{(\overline{q}_j, \perp) : j \leq k + 1\} \cup \{(q_j, x) : j \leq k \wedge x \leq b_j\}$$

where $\perp$ indicates the inactivity of the clock in waiting/idle states.

[2] See [2] for a straightforward generalization to partial-order precedence constraints.

[3] To avoid additional notations we use x both for the clock variable and its value.

Note the difference between transition labels s_j and e_j: the *start* transitions are *controllable* and are issued by the scheduler that we want to optimally synthesize while the *end* transitions represent the *uncontrolled* external (to the scheduler) environment which is assumed to be uniformly distributed. Without a scheduler, the SDPA is *under-determined* and can issue a *start* transition any time. The derivation of an optimal scheduler is done via the computation of a *time-to-go* function that we first illustrate on the degenerate case of one process in isolation, where each state has only one successor and any waiting between steps unnecessarily increases the time to termination.

Definition 5.2.3 (Local Stochastic Time to Go)**.** *The local stochastic time-to-go function associates with every state (q, x) a time density $\mu(q, x)$ with $\mu(q, x)[t]$ indicating the probability to terminate within t time* given *that we start from (q, x) and apply the optimal strategy.*

This function admits the following inductive definition:

$$\mu(\overline{q}_{k+1}, \bot) = \mathbf{0} \tag{5.2}$$

$$\mu(\overline{q}_j, \bot) = \mu(q_j, 0) \tag{5.3}$$

$$\mu(q_j, x)[t] = \int_0^t \psi_j[x + t'] \cdot \gamma_{a,b}(x) \cdot \mu(\overline{q}_{j+1}, 0)[t - t']dt' \tag{5.4}$$

Line (5.2) indicates the final state while (5.3) comes from the fact that in the absence of conflicts the optimal scheduler need not wait and should start each step immediately when enabled. Equation (5.4) computes the probability for termination at t based on the probabilities of terminating the current step in some t' and of the remaining time-to-go being $t - t'$. It can be be summarized in a functional language as

$$\mu(q_j, x) = \psi_{j/x} * \mu(\overline{q}_{j+1}, \bot) = \psi_{j/x} * \mu(q_{j+1}, 0) \tag{5.5}$$

The successive application of (5.5) yields, not surprisingly, $\mu(q_1, 0) = \psi_1 * \cdots * \psi_k$.

Definition 5.2.4 (Local Expected Time to Go)**.** *The expected time-to-go function is $V : Q \times X \to \mathbb{R}_+$ defined as*

$$V(q, x) = \int \mu(q, x)[t] \cdot tdt = \mathbb{E}(\mu(q, x)).$$

This measure satisfies $V(q_j, x) = \mathbb{E}(\psi_{j/x}) + V(q_{j+1}, 0)$ where the first term is the expected termination time of step j starting from x. For the initial state this yields

$$V(q_1, 0) = \mathbb{E}(\psi_1 * \cdots * \psi_k) = \mathbb{E}(\psi_1) + \cdots + \mathbb{E}(\psi_k) = \sum_{j=1}^{k}(a_j + b_j)/2.$$

5.3 Conflicts and Schedulers

We now extend the model to express n processes, indexed by $N = \{1..n\}$, that may run in parallel except for steps which are mutually conflicting due to the use of the same resource.

Definition 5.3.1 (Process System)**.** *A process system is a triple $(\mathcal{P}, M, h)$ where*

$$\mathcal{P} = P^{\mathbf{1}} || \cdots || P^{\mathbf{n}} = \{(\mathcal{I}^{\mathbf{i}}, \Psi^{\mathbf{i}})\}_{i \in N}$$

is a set of processes, M is a set of resources, and $h : N \times K \to M$ is a function which assigns to each step the resource it uses.

We use notations $P^{\mathbf{i}}_j$ to refer to step j of process i and $\psi^{\mathbf{i}}_j$ and $I^{\mathbf{i}}_j = [a^{\mathbf{i}}_j, b^{\mathbf{i}}_j]$ for the respective densities and their support intervals. Likewise we denote the corresponding controllable and uncontrollable actions by $s^{\mathbf{i}}_j$ and $e^{\mathbf{i}}_j$, respectively. Without loss of generality we assume there is one instance of each resource type, hence two steps $P^{\mathbf{i}}_j$ and $P^{\mathbf{i}'}_{j'}$ such that $h(i,j) = h(i',j')$ are in *conflict* and cannot execute simultaneously. Each process is modeled as an SPDA $\mathcal{A}^{\mathbf{i}} = (\Sigma^{\mathbf{i}}, Q^{\mathbf{i}}, \{x^{\mathbf{i}}\}, Y^{\mathbf{i}}, \Delta^{\mathbf{i}})$ and the global system is obtained as a product of those restricted to conflict-free states. We write global states as $q = (q^{\mathbf{1}}, \dots, q^{\mathbf{n}})$ and exclude states where for some i and i', $q^{\mathbf{i}} = q^{\mathbf{i}}_j$, $q^{\mathbf{i}'} = q^{\mathbf{i}'}_{j'}$ and steps $P^{\mathbf{i}}_j$ and $P^{\mathbf{i}'}_{j'}$ are conflicting. We say that action $s^{\mathbf{i}}_j$ (respectively, $e^{\mathbf{i}}_j$) is enabled in q if $q^{\mathbf{i}} = \overline{q}^{\mathbf{i}}_j$ (resp. $q^{\mathbf{i}} = q^{\mathbf{i}}_j$). Since only one transition per process is possible in a global state, we will sometime drop the j-index and refer to those as $s^{\mathbf{i}}$ and $e^{\mathbf{i}}$.

Definition 5.3.2 (Duration Probabilistic Automata). *A duration probabilistic automaton (DPA) is a composition $\mathcal{A} = \mathcal{A}^{\mathbf{1}} \circ \cdots \circ \mathcal{A}^{\mathbf{n}} = (\Sigma, Q, X, Y, \Delta)$ of n SDPA with the action alphabet being $\Sigma = \bigcup_i \Sigma^{\mathbf{i}}$. The discrete state space is $Q \subseteq Q^{\mathbf{1}} \times \cdots Q^{\mathbf{n}}$ (with forbidden states excluded). The set of clocks is $X = \{x^{\mathbf{1}}, \dots, x^{\mathbf{n}}\}$, the extended state-space is $S \subseteq S^{\mathbf{1}} \times \cdots S^{\mathbf{n}}$ and the auxiliary variables are $Y = \bigcup_i Y^{\mathbf{i}}$ ranging over the joint duration space $D = D^{\mathbf{1}} \times \cdots \times D^{\mathbf{n}}$. The transition relation Δ is built using interleaving, that is, a transition (q, g, r, q') from $q = (q^{\mathbf{1}}, \dots, q^{\mathbf{i}}, \dots, q^{\mathbf{n}})$ to $q' = (q^{\mathbf{1}}, \dots, q'^{\mathbf{i}}, \dots, q^{\mathbf{n}})$ exist in Δ if a transition from $(q^{\mathbf{i}}, g, r, , q'^{\mathbf{i}})$ exists in $\Delta^{\mathbf{i}}$, provided that q' is not forbidden.*

The DPA thus defined (Fig. 4.6-a) is not probabilistically correct as it admits non-determinism of a non probabilistic nature: in a given state the automaton may choose between several *start* transitions or decide to wait for an *end* transition (the termination of an active step). A *scheduler* selects *one* action in any state and then the only non-determinism that remains is due to the probabilistic task durations. A discussion on different types of schedulers can be found in [119].

Definition 5.3.3 (Scheduler). *A scheduler for a DPA $\mathcal{A}$ is a function $\Omega : S \to \Sigma_s \cup \{\mathbf{w}\}$ such that for every $s \in \Sigma_s$, $\Omega(q, x) = s$ only if s is enabled in q and $\Omega(q, x) = \mathbf{w}$ (wait) only if q admits at least one active component.*

Composing the scheduler with the DPA (see Fig. 4.6-b) renders it input-deterministic in the sense that any point $y \in D$ induces a unique[4] run of the automaton composed of an alternation of discrete transitions and periods of time elapse.

Definition 5.3.4 (Steps and Runs). *The steps of a controlled DPA $\mathcal{A} \circ \Omega$, induced by a point $y \in D$ are of the following types:*

- *Start steps:* $(q, x) \xrightarrow{s^{\mathbf{i}}_j} (q', x')$ *iff* $q^{\mathbf{i}} = \overline{q}^{\mathbf{i}}_j$ *and* $\Omega(q, x) = s^{\mathbf{i}}_j$*;*
- *End steps:* $(q, x) \xrightarrow{e^{\mathbf{i}}_j} (q', x')$ *iff* $q^{\mathbf{i}} = q^{\mathbf{i}}_j$ *and* $x^{\mathbf{i}} = y^{\mathbf{i}}_j$*;*
- *Time steps:* $(q, x) \xrightarrow{t} (q, x + t)$ *iff* $\forall i \; (q^{\mathbf{i}} = q^{\mathbf{i}}_j \Rightarrow x^{\mathbf{i}} + t < y^{\mathbf{i}}_j)$.

The run associated with y is a sequence of steps starting at $(\overline{q}^{\mathbf{1}}_1, \dots, \overline{q}^{\mathbf{n}}_1)$ and ending in $(\overline{q}^{\mathbf{1}}_{k+1}, \dots, \overline{q}^{\mathbf{n}}_{k+1})$.

The duration of a run is the sum of the time steps and it coincides with the termination time of the last process, known as the *makespan* in the OR jargon. In a state q where $P^{\mathbf{i}}$ is active, its *t-i-successor*, denoted by $\sigma^i(t, q, x)$, is the state (q', x') reached after a time step of duration t followed by an $e^{\mathbf{i}}$ transition.

[4] We define a priority order among the $s^{\mathbf{i}}$-actions so that in the (measure zero) situation where two actions are taken simultaneously we impose the order to guarantee a unique run and avoid the artifacts of the interleaving semantics.

5.4 Expected Time Optimal Schedulers

In [119] we developed a method to compute the expected termination time under a *given* scheduler by computing volumes in the duration space. We now show how to optimally synthesize such schedulers from an uncontrolled DPA description. To this end we will have to extend the formulation of stochastic time-to-go from a single process (5.4) to multiple processes (5.8).

We define a partial order relation on global states based on the order on the local states with $q \preceq q'$ if for every i, $q^{\mathbf{i}} \preceq q^{\mathbf{i}'}$. We lift this relation to extended states letting $(q, x) \preceq (q', x')$ if $q \preceq q'$ or $q = q' \wedge x \leq x'$. The *forward cone* of a state q is the set of all states q' such that $q \prec q'$. The immediate successors of a state q are the states reachable from it by one transition. A *partial scheduler* is a scheduler defined only on a subset of Q. To optimize the action of the scheduler in a state we need compare the effect of the action on the time-to-go.

Definition 5.4.1 (Local Stochastic Time-to-Go). *Let $\mathcal{A}$ be a DPA with a partial strategy whose domain includes the forward cone of a state q. With every i, x and every $s \in \Sigma_s \cup \{\mathbf{w}\}$ enabled in q, the time density $\mu^{\mathbf{i}}(q, x, s) : \mathbb{R}_+ \to [0, 1]$ characterizes the stochastic time-to-go for process $P^{\mathbf{i}}$ if the controller issues action s at state (q, x) and continues from there according to the partial strategy.*

Note that for any successor q' of q the optimal action has already been chosen and we denote its associated time-to-go by $\mu(q, x)$. Once $\mu^{\mathbf{i}}(q, x, s)$ has been computed for every i, the following measures, all associated with action s, can be derived from it.

Definition 5.4.2 (Global Stochastic Time-to-Go). *With every state (q, x) and action s enabled in it, we define*

- *The stochastic time-to-go for total termination (makespan):*

$$\mu(q, x, s) = \max\{\mu^{\mathbf{1}}(q, x, s), \ldots, \mu^{\mathbf{n}}(q, x, s)\}$$

- *The expected total termination time:*

$$V(q, x, s) = \int t \cdot \mu(q, x, s)[t]dt$$

The computation of μ for a state q, based on the stochastic time-to-go of its successors, is the major contribution of this chapter. The hard part is the computation of the time-to-go associated with *waiting* in a state where several processes are active. In this situation (known as a *race*) the automaton may leave q via different transitions and μ should be computed based on the probabilities of these transitions (and their timing) and the cost-to-go from the respective successor states.

With each state (q, x) we associate a family $\{\rho^{\mathbf{i}}(q, x)\}_{i \in N}$ of partial time densities with the intended meaning that $\rho^{\mathbf{i}}(q, x)[t]$ is (the density of) the probability that the *first* process to terminate its current step is $P^{\mathbf{i}}$ *and* that this occurs within t time. This definition is relative, of course, to the fact that the time elapsed since the initiation of each and every active step is captured by the respective clock value in x.[5]

[5] It is worth noting that the dependence on the time already elapsed is in contrast with the *memoryless exponential* distribution where this time does not matter for the future. For those distributions the time-to-go is associated only with the discrete state and is much easier to compute, see [2] for the derivation of optimal schedulers for DPA with exponential distribution.

Definition 5.4.3 (Race Winner)**.** *Let q be a state where n processes are active, each in a step admitting a time density $\psi^{\mathbf{i}}$. With every clock valuation $x = (x^{\mathbf{1}}, \ldots, x^{\mathbf{n}}) \leq (b^{\mathbf{1}}, \ldots, b^{\mathbf{n}})$ and every i we associate the partial density:*

$$\rho^{\mathbf{i}}(q,x)[t] = {\psi^{\mathbf{i}}}_{/x^{\mathbf{i}}}[t] \cdot \prod_{i' \neq i} {\psi^{\mathbf{i}'}}_{/x^{\mathbf{i}'}}[\geq t]$$

Definition 5.4.4 (Computing Stochastic Time-to-go)**.** *For every i, the function $\mu^{\mathbf{i}}$ is defined inductively as*

$$\mu^{\mathbf{i}}((\ldots \overline{q}^{\mathbf{i}}_{k+1} \ldots), x) = \mathbf{0} \tag{5.6}$$

$$\mu^{\mathbf{i}}(q, x, s^{\mathbf{i}'}) = \mu^{\mathbf{i}}(\sigma^{\mathbf{i}'}(0, q, x))) \tag{5.7}$$

$$\mu^{\mathbf{i}}(q, x, \mathbf{w})[t] = \sum_{i'=1}^{n} \int_{0}^{t} \rho^{\mathbf{i}'}(q,x)[t'] \cdot \mu^{\mathbf{i}}(\sigma^{\mathbf{i}'}(t', q, x))[t - t']dt' \tag{5.8}$$

For any global state where $P^{\mathbf{i}}$ is in its final state, $\mu^{\mathbf{i}}$ is zero (5.6). Each enabled *start* action $s^{\mathbf{i}'}$ leads immediately to the successor state and the cost-to-go is inherited from there (5.7). For waiting we make a convolution between the probability of $P^{\mathbf{i}'}$ winning the race and the value of $\mu^{\mathbf{i}}$ in the post-transition state and sum up over all the active processes (5.8). The basic iterative step in computing the value function and strategy is summarized in Algorithm 3. A *dynamic programming* algorithm starting from the final state and applying the above procedure will produce the expected-time optimal strategy for this scheduling problem. Since we are dealing with *acyclic* systems, the question of convergence to a fixed point is not raised at all. The only challenge is to show that the defined operators are computable.

Algorithm 3 Value Iteration

```
Input: A global state $q$ such that $\Omega(q', x)$ and $\mu^{\mathbf{i}}(q', x)$ have been computed for each of its successors $q'$ and every $i$
Output: $\Omega(q, x)$, and $\mu^{\mathbf{i}}(q, x)$

    % COMPUTE:
for all $s \in \Sigma_s \cup \{\mathbf{w}\}$ do
    for $i = 1 \to n$ do
        compute $\mu^{\mathbf{i}}(q, x, s)$ according to (5.7-5.8)
    end for
    compute $\mu(q, x, s)$                          ▷ max of random variables
    compute $V(q, x, s)$                            ▷ expected makespan
end for
    % OPTIMIZE:
for all $x \in Z_q$ do
    $V(q, x) = \min_s(V(q, x, s))$
    $s_* = \arg\min_s V(q, x, s)$
    $\Omega(q, x) = s_*$
end for
    % UPDATE:
for $i = 1 \to n$ do
    $\mu^{\mathbf{i}}(q, x) = \mu^{\mathbf{i}}(q, x, s_*)$
end for
```

If we look at the algorithm more carefully we see that it splits into three parts, the third being merely book-keeping. In the first we essentially *compute* the outcome of *waiting* by race analysis and of *starting*. As we shall see, starting from a specific class of time densities, this part can be done in a symbolic/analytic way, resulting in *closed-form expressions* over x and t for the values of $\mu^{\mathbf{i}}$, μ and V associated with each action. The second part involved *optimization*: to characterize the extended states according to the action that optimizes V in them. For this part we have not characterized the class of partitions of the clock space obtained but proved a monotonicity property (non-lazy schedulers) that will facilitate the task of tracing the boundary. Since this work is not fully completed, it is delegated to Appendix A.

5.5 Computational Aspects

We now turn to the properties of the previously defined functions and the way to compute them. For every q, $\mu^{\mathbf{i}}(q, x)$ is in fact an infinite family of functions parameterized by the clock valuations inside the rectangle Z_q of clock valuations which are possible as state q.

Definition 5.5.1 (Zone-Polynomial Time Densities)**.** *A function $\mu : Z \to (\mathbb{R}_+ \to [0, 1])$, where Z is a rectangular clock space, is zone polynomial if it can be written as*

$$\mu(x^{\mathbf{1}}, \ldots, x^{\mathbf{n}})[t] = \begin{cases} f_1(x^{\mathbf{1}}, \ldots, x^{\mathbf{n}})[t]) & \text{if} \quad Z_1(x^{\mathbf{1}}, \ldots, x^{\mathbf{n}}) \quad \text{and} \quad l_1 \leq t \leq u_1 \\ f_2(x^{\mathbf{1}}, \ldots, x^{\mathbf{n}})[t]) & \text{if} \quad Z_2(x^{\mathbf{1}}, \ldots, x^{\mathbf{n}}) \quad \text{and} \quad l_2 \leq t \leq u_2 \\ \ldots & \\ f_L(x^{\mathbf{1}}, \ldots, x^{\mathbf{n}})[t]) & \text{if} \quad Z_L(x^{\mathbf{1}}, \ldots, x^{\mathbf{n}}) \quad \text{and} \quad l_N \leq t \leq u_L \end{cases}$$

where

- *For every r, $Z_r(x^{\mathbf{1}}, ..., x^{\mathbf{n}})$ is a zone included in the rectangle Z, which moreover satisfies either $Z_r \subseteq [x^{\mathbf{i}} \leq a^{\mathbf{i}}]$ or $Z_r \subseteq [a^{\mathbf{i}} \leq x^{\mathbf{i}}]$, for every $i = 1..n$.*
- *For every r, the bounds l_r, u_r of the t interval are either nonnegative integers c or terms of the form $c - x^{\mathbf{i}}$, with $i = 1, n$, $c \in \mathbb{Z}^+$. Moreover, the interval $[l_r, u_r]$ must be consistent with the underlying zone, that is, $Z_r \subseteq [l_r, u_r]$.*
- *For every r, $f_r(x^{\mathbf{1}}, ..., x^{\mathbf{n}})[t] = \sum\limits_k \frac{P_k(x^{\mathbf{1}}, ..., x^{\mathbf{n}})}{Q_r(x^{\mathbf{1}}, ..., x^{\mathbf{n}})} t^k$ where P_k are arbitrary polynomials and Q_r is a characteristic polynomial associated with zone Z_r defined as $\prod\limits_i (b^{\mathbf{i}} - \max\{x^{\mathbf{i}}, a^{\mathbf{i}}\})$.*

  Note that for each zone, the max *is attained uniformly as either $a^{\mathbf{i}}$ or $x^{\mathbf{i}}$.*

Theorem 2 (Closure of Zone-Polynomial Densities)**.** *Zone-polynomial time densities are closed under 5.7 and 5.8).*

Sketch of Proof Operation 5.7 is a simple substitution. Closure under summation is also evident - you just need to refine the partitions associated with the summed functions and make them compatible and then apply the operation in each partition block. The only intricate part concerns the quasi-convolution part of (5.8). The function $\mu^{\mathbf{i}}(\sigma^{\mathbf{i}'}(t', q, x))[t - t']$ is not a zone polynomial time density. Due to the time progress by t' enforced by the substitution $\sigma^{\mathbf{i}'}$ it might happen that polynomials of the form $(b_i - (x^{\mathbf{i}} - t'))$ appear in the denominators. But, in all feasible cases, they will be simplified through multiplication by $\rho^{\mathbf{i}'}(q, x)[t']$ which contains the same polynomials as factors (within ${\psi^{\mathbf{i}}}_{/x^{\mathbf{i}}}[\geq t']$, see Def. 5.4.3). Hence, integration of t' is always trivially completed as t' occurs only on numerator polynomials and/or powers of the form t'^k and $(t - t')^k$. Moreover, after integration, the remaining constraints on t and x can also be rewritten to match the required form of the zone-polynomial time densities. ▟

We realized a prototype implementation of operator (5.8) that we detail in the next section and apply to a simple example.

5.6 Implementation

In the following we denote by F a zone polynomial function $\mu(x^{\mathbf{1}}, \ldots, x^{\mathbf{n}})[t]$ (Definition 5.5.1). We provide a library for the manipulation of such function. At implementation F is represented as a list of *nodes* $\{N_1, \ldots, N_k\}$ and

$$F = \bigcup_i N_i \tag{5.9}$$

A node $N_i = \langle Z_i, L_i \rangle$ is defined by:

- Z_i a zone
- L_i a list of pairs $\langle C^i_j, F^i_j \rangle$ where:
  - C^i_j is a constraints on variable t
  - E^i_j is an expression defined as a polynomial fraction $f_r(x^{\mathbf{1}}, ..., x^{\mathbf{n}})[t]$ as in Definition 5.5.1

$$F = \left\{ \begin{array}{|c|c|c|} \hline Z & C & E \\ \hline & l^1_1 < t < u^1_1 & E^1_1 \\ Z_1 & \ldots & \ldots \\ & l^1_{k^1} < t < u^1_{k^1} & E^1_{k^1} \\ \hline \ldots & \ldots & \ldots \\ \hline & l^n_1 < t < u^n_1 & E^1_1 \\ Z_n & \ldots & \ldots \\ & l^n_{k^n} < t < u^n_{k^n} & E^n_{k^n} \\ \hline \end{array} \right. \tag{5.10}$$

As an example $\psi_{/x}[t]$ defined by (5.1) is encoded as:

$$\psi_{/x}[t] = \left\{ \begin{array}{|c|c|c|} \hline Z & C & E \\ \hline & 0 < t < a - x & 0 \\ 0 < x < a & a - x < t < b - x & \frac{1}{b-a} \\ & b - x < t & 0 \\ \hline a < x < b & 0 < t < b - x & \frac{1}{b-x} \\ & b - x < t & 0 \\ \hline \end{array} \right. \tag{5.11}$$

Manipulation of zones are performed using the DBM library of IF [49]. We extend the polynomial library developed for volume computation in the previous chapter for the manipulation of polynomial fractions. To compute $\mu(q, x, s)$ several operations on nodes are necessary. We present in the sequel how they have been implemented. Note that, by construction we ensure that F is in a *disjoint form*. That is, $\forall N_i, N_j \in F, Z_i \cap Z_j = \emptyset \wedge \forall C^i_j, C^i_k \in L_i, C^i_j \cap C^i_k = \emptyset$

Disjoint nodes

Let us first explain how nodes are kept disjoint during all computations. It is a necessary operation for further computations, especially to facilitate integration of zone polynomial function according to the t variables. This operation is performed by:
$\forall i \neq j$, if $Z_i \wedge Z_j \neq \emptyset$ remove N_i, N_j and add new nodes:

- $\{Z_i \wedge Z_j, L_i + L_j\}$
- $\{Z_i - Z_j, L_i\}$
- $\{Z_j - Z_i, L_j\}$

The subtraction of two zones leads to several zones implying creation of new nodes. It is computed as follows:

$$Z_1 - Z_2 = \bigcup_{i,j} (Z_1 \cap \neg Z_2[i][j])$$

The number of resulting nodes is bounded by n^2 where n is the dimension of the zone. More efficient algorithm based on minimal constraint graph can be used to minimize the number of splits, readers can refer to [71] for more details.

The operation $L_i + L_j$ is done by adding each pair $\langle C, E \rangle$ from the two lists together. This addition results in 3 new pairs defines as follows:

$$\langle C_1, E_1 \rangle + \langle C_2, E_2 \rangle = \begin{cases} \langle C_1 \cap C_2, E_1 + E_2 \rangle \\ \langle C_1 \cap \neg C_2, E_1 \rangle \\ \langle \neg C_1 \cap C_2, E_2 \rangle \end{cases} \tag{5.12}$$

Inside the operation (5.12), constraints on t are defined as conjunction of bounds, i.e with $C_i = l_i < t < u_i$ and $C_i = l_j < t < u_j$:

$$C_i \cap C_j = \max(l_i, l_j) < t < \min(u_i, u_j)$$

Note that the list L_i associated with each disjoint zone must also be disjoint, that is:

$$\forall \langle C_i^k, E_i^k \rangle, \langle C_j^k, E_j^k \rangle \in L_k,\ C_i \cap C_j = \emptyset$$

For that we need to compute *unique* bounds l and u on variable t. This is done by creating a new case for each possible pair of bounds as shown in Algorithm-4.

Algorithm 4 CANONIZE ($\langle C, E \rangle$)

```
$Res = \emptyset$                                          ▷ The resulting node list
for all $l_i \in C$ do
    for all $u_j \in C$ do
//set $l_i$ and $u_j$ as lower and upper bound
        $C_{tmp} \leftarrow \{l_i < t < u_j\}$
        $Z_{tmp} \leftarrow$ new ZONE
        for all $l_k \neq l_i$ do
// $l_i > l_k \Rightarrow x^{i'} - a^{i'} > x^{k'} - a^{k'} \Rightarrow x^{k'} - x^{i'} < a^{k'} - a^{i'}$
            $Z_{tmp} \leftarrow Z_{tmp} \cap \{l_i > l_k\}$
        end for
        for all $u_k \neq u_j$ do
// $u_j < u_k \Rightarrow x^{i'} - a^{i'} < x^{k'} - a^{k'} \Rightarrow x^{i'} - x^{k'} < a^{i'} - a^{k'}$
            $Z_{tmp} \leftarrow Z_{tmp} \cap \{u_j < u_k\}$
        end for
        $Z_{tmp} \leftarrow Z_{tmp} \cap \{l_i < u_j\}$          ▷ check if new bounds are correct
        if ¬ isDegenerate($Z_{tmp}$) then
            Res $\leftarrow Res \cup \langle Z_{tmp}, \{\langle C_{tmp}, E \rangle\} \rangle$
        end if
    end for
end for
return Res
```

Product

The multiplication of two zone polynomial functions F_1 and F_2 is done by multiplying all their nodes in pairs. The result of $N_i \cdot N_j$ is a node $N_k = \langle Z_i \cap Z_j, L_i \cdot L_j \rangle$ and the computation is shown in Algorithm-5.

Algorithm 5 MULTIPLY (N_1, N_2)

```
$N_r = \langle Z_r, L_r \rangle$                                    ▷ The resulting node
$Z_r = Z_1 \cap Z_2$
$L_r = \emptyset$
if $\neg$ isDegenerate($Z_r$) then
    for all $\langle E_i, C_i \rangle \in L_1$ do
        for all $\langle E_j, C_j \rangle \in L_2$ do
            if $C_i \cap C_j \neq \emptyset$ then
                $L_r \leftarrow L_r \cup \langle E_i \cdot E_j, C_i \cap C_j \rangle$
            end if
        end for
    end for
end if
return $N_r$
```

Addition

The addition of two two zone polynomial functions F_1 and F_2 is performed in the same way as for multiplication, i.e by adding all their nodes in pairs as shown in Algorithm 6. The results of the addition of two nodes is:

$$N_i + N_j = \begin{cases} Z_i \cap Z_j & L_i + L_j \\ Z_i \cap \neg Z_j & L_i \\ \neg Z_i \cap Z_j & L_j \end{cases}$$

Note that the zone subtraction did not appear in the multiplication operation because the resulting nodes are empty like $\langle Z_i \cap \neg Z_j, L_i \cdot 0 = 0 \rangle$.

Integration

Integration is performed on zone polynomial functions that are in disjoint form as explained previously. Constraints on variable t are disjoint and are sorted in increasing order. Note that they are defined continuously between lowest and uppermost bounds i.e $u_i = l_{i-1}$. For all nodes integration is performed on each E according to bounds on t. In the following $F[e]$ denotes substitution of variable t by expression e. Consider one node $\langle Z, L \rangle$ of a probability density function. We compute its distribution $\int_{t'<t} \psi_{/x}[t']\mathrm{d}t'$ by computing iteratively integrals from 0 to t according to the bounds in the ordered list

$$L = \{\langle a_1 < t < a_2, E_1 \rangle, \ldots, \langle a_{n-1} < t < a_n, E_n \rangle\}$$

which gives a node $\langle Z, L' \rangle$ where

$$L' = \left\{ \left\langle a_1 < t < a_2, E'_1 = \int_{a_1}^{t} E_1[t']\mathrm{d}t' \right\rangle, \ldots, \left\langle a_{n-1} < t < a_n, E'_{n-1} + \int_{a_{n-1}}^{t} E_n[t']\mathrm{d}t' \right\rangle \right\}$$

Algorithm 6 ADD (N_1, N_2)

```
Res ← $\emptyset$                                              ▷ The resulting node list
if $Z_1 \cap Z_2 \neq \emptyset$ then
    $N_{tmp} \leftarrow$ new $\langle Z_1 \cap Z_2, L_{tmp}\rangle$          ▷ $L_{tmp}$ list of resulting pairs $\langle C, E\rangle$
    for all $\langle E_i, C_i\rangle \in L_1$ do
        for all $\langle E_j, C_j\rangle \in L_2$ do
            $L_{tmp} \leftarrow L_{tmp} \cup \langle C_i \cap C_j, E_1 + E_2\rangle$
            $L_{tmp} \leftarrow L_{tmp} \cup \langle C_i \cap \neg C_j, E_1\rangle$
            $L_{tmp} \leftarrow L_{tmp} \cup \langle \neg C_i \cap C_j, E_2\rangle$
        end for
    end for
    Res ← Res $\cup N_{tmp}$
end if
for all $Z \in Z_1 - Z_2$ do
    Res ← Res $\cup \langle Z, L_1\rangle$
end for
for all $Z \in Z_2 - Z_1$ do
    Res ← Res $\cup \langle Z, L_2\rangle$
end for
return Res
```

As an example recall (5.11) for which we can get the probability distribution (5.14). In the same way we can compute $\int_{t<t'} \psi_{/x}[t']\mathrm{d}t'$ as shown in (5.15).

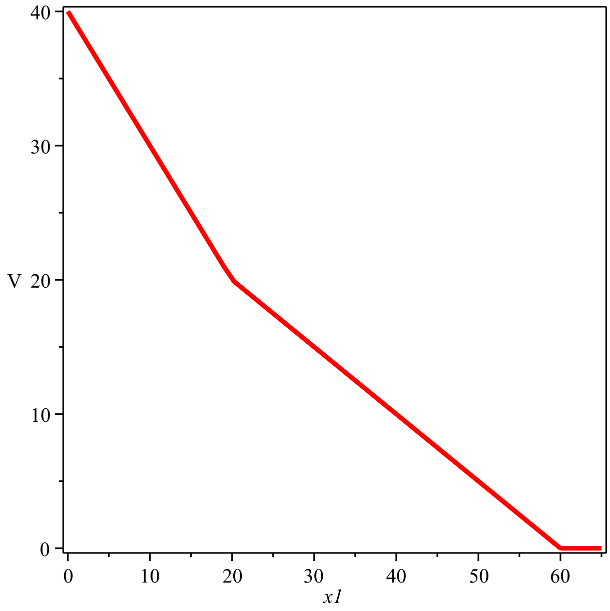
$$\int \psi_{/x}[t]\mathrm{d}t = \left\{ \begin{array}{|c|c|c|} \hline Z & C & E \\ \hline 0 < x < a & \emptyset & 1 \\ \hline a < x < b & \emptyset & 1 \\ \hline \end{array} \right. \tag{5.13}$$

$$\int_0^t \psi_{/x}[t']\mathrm{d}t' = \left\{ \begin{array}{|c|c|c|} \hline Z & C & E \\ \hline & 0 < t < a - x & 0 \\ 0 < x < a & a - x < t < b - x & \frac{t-a+x}{b-a} \\ & b - x < t & 1 \\ \hline a < x < b & 0 < t < b - x & \frac{t}{b-x} \\ & b - x < t & 1 \\ \hline \end{array} \right. \tag{5.14}$$

$$\int_t^\infty \psi_{/x}[t']\mathrm{d}t' = \left\{ \begin{array}{|c|c|c|} \hline Z & C & E \\ \hline & 0 < t < a - x & 1 \\ 0 < x < a & a - x < t < b - x & \frac{b-x-t}{b-a} \\ & b - x < t & 0 \\ \hline a < x < b & 0 < t < b - x & \frac{b-x-t}{b-x} \\ & b - x < t & 0 \\ \hline \end{array} \right. \tag{5.15}$$

5.7 An Example

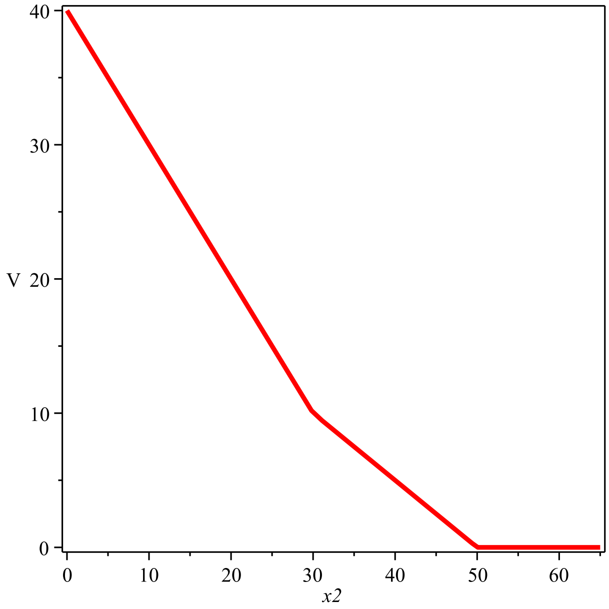
We illustrate the above computation on a simple example consisting of two one-step processes $P^{\mathbf{1}}$ and $P^{\mathbf{2}}$ with respective duration intervals $I_1^{\mathbf{1}} = [20, 60]$ and $I_1^{\mathbf{2}} = [30, 50]$. We start the backward value iteration from the final state $V(q_2^{\mathbf{1}}, q_2^{\mathbf{2}}, \bot, \bot) = \mathbf{0}$. In each of the two predecessor states $(q_1^{\mathbf{1}}, q_2^{\mathbf{2}})$ and $(q_2^{\mathbf{1}}, q_1^{\mathbf{2}})$ only one clock is active. The output of our tool and the plots of the value function are depicted in Figure 5.3 and 5.4.



```
N1:
 Z:  x1=[0,20]
 E:  40-x1

N2:
 Z:  x1=[20,60]
 E:  (1800-60*x1+1/2*x1^2)  /  (60-x1)
```

Figure 5.3: The value function $V(q_1^{\mathbf{1}}, q_2^{\mathbf{2}}, x^{\mathbf{1}}, \bot)$



```
N1:
 Z:  x2=[0,30]
 E:  40-x2

N2:
 Z:  x2=[30,50]
 E:  (1250-50*x2+1/2*x2^2)  /  (50-x2)
```

Figure 5.4: The value function $V(q_2^{\mathbf{1}}, q_1^{\mathbf{2}}, \bot, x^{\mathbf{2}})$

The more involved and interesting case is in state $(q_1^{\mathbf{1}}, q_1^{\mathbf{2}})$ where we have a race and a lot of case splitting according to the relation between clocks. Listing 5.1 shows the output of our tool which yields (5.16) as plotted in Fig 5.5.

We can compute the expected time-to-go from the initial state

$$V(q_1^{\mathbf{1}}, q_1^{\mathbf{2}}, 0, 0) = \frac{545}{12} = 45.4167$$

and compare it with the clock-free technique developed in the previous chapter. The results are indeed identical.

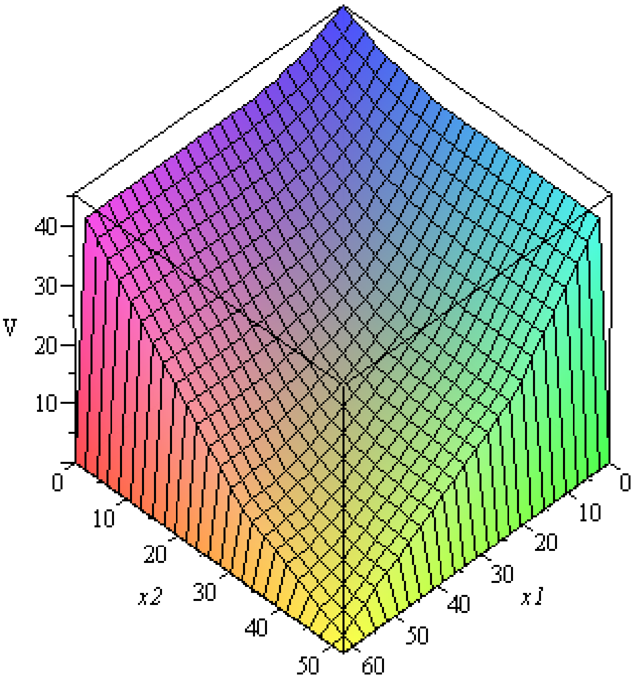


Figure 5.5: The value function $V(q_1^{\mathbf{1}}, q_1^{\mathbf{2}}, x^{\mathbf{1}}, x^{\mathbf{2}})$

5.8 Concluding Remarks

We have built a framework to define and synthesize optimal schedulers in a non-Markovian setting. To the best of our knowledge no similar computational results have been reported. The synthesis algorithm is based on backward value iteration that computes the stochastic time-to-go function over the extended state space.

$$
\begin{cases}
\frac{365}{8} - \frac{7}{16}x_1 - \frac{3}{80}x_1x_2 + \frac{1}{1600}x_1x_2^2 + \frac{3}{160}x_1^2 - \frac{1}{1600}x_1^2x_2 + \frac{1}{4800}x_1^3 - \frac{9}{16}x_2 + \frac{3}{160}x_2^2 - \frac{1}{4800}x_2^3 & \textbf{if } \begin{array}{l} 10 \le x_1 \le 20 \\ 20 \le x_2 \le 30 \\ 10 \le x_2 - x_1 \le 20 \end{array} \\
\frac{365}{8} - \frac{7}{16}x_1 - \frac{3}{80}x_1x_2 + \frac{1}{1600}x_1x_2^2 + \frac{3}{160}x_1^2 - \frac{1}{1600}x_1^2x_2 + \frac{1}{4800}x_1^3 - \frac{9}{16}x_2 + \frac{3}{160}x_2^2 - \frac{1}{4800}x_2^3 & \textbf{if } \begin{array}{l} 0 \le x_1 \le 10 \\ 10 \le x_2 \le 30 \\ 10 \le x_2 - x_1 \le 30 \end{array} \\
\frac{365}{8} - \frac{9}{16}x_1 - \frac{3}{80}x_1x_2 - \frac{1}{1600}x_1x_2^2 + \frac{3}{160}x_1^2 + \frac{1}{1600}x_1^2x_2 - \frac{1}{4800}x_1^3 - \frac{7}{16}x_2 + \frac{3}{160}x_2^2 + \frac{1}{4800}x_2^3 & \textbf{if } \begin{array}{l} 10 \le x_1 \le 20 \\ 0 \le x_2 \le 10 \\ -20 \le x_2 - x_1 \le -10 \end{array} \\
\frac{545}{12} - \frac{1}{2}x_1 - \frac{1}{40}x_1x_2 + \frac{1}{80}x_1^2 - \frac{1}{2}x_2 + \frac{1}{80}x_2^2 & \textbf{if } \begin{array}{l} 0 \le x_1 \le 10 \\ 0 \le x_2 \le 10 \\ -10 \le x_2 - x_1 \le 10 \end{array} \\
\frac{545}{12} - \frac{1}{2}x_1 - \frac{1}{40}x_1x_2 + \frac{1}{80}x_1^2 - \frac{1}{2}x_2 + \frac{1}{80}x_2^2 & \textbf{if } \begin{array}{l} 10 \le x_1 \le 20 \\ 0 \le x_2 \le 10 \\ -10 \le x_2 - x_1 \le 0 \end{array} \\
\frac{545}{12} - \frac{1}{2}x_1 - \frac{1}{40}x_1x_2 + \frac{1}{80}x_1^2 - \frac{1}{2}x_2 + \frac{1}{80}x_2^2 & \textbf{if } \begin{array}{l} 10 \le x_1 \le 20 \\ 10 \le x_2 \le 30 \\ -10 \le x_2 - x_1 \le 10 \end{array} \\
\frac{545}{12} - \frac{1}{2}x_1 - \frac{1}{40}x_1x_2 + \frac{1}{80}x_1^2 - \frac{1}{2}x_2 + \frac{1}{80}x_2^2 & \textbf{if } \begin{array}{l} 0 \le x_1 \le 10 \\ 10 \le x_2 \le 20 \\ 0 \le x_2 - x_1 \le 10 \end{array} \\
\frac{2000+x_1x_2-50x_1-40x_2}{50-x_2} & \textbf{if } \begin{array}{l} 10 \le x_1 \le 20 \\ 40 \le x_2 \le 50 \\ 30 \le x_2 - x_1 \le 40 \end{array} \\
\frac{2000+x_1x_2-50x_1-40x_2}{50-x_2} & \textbf{if } \begin{array}{l} 0 \le x_1 \le 10 \\ 40 \le x_2 \le 50 \\ 30 \le x_2 - x_1 \le 50 \end{array} \\
\frac{2000+x_1x_2-50x_1-40x_2}{50-x_2} & \textbf{if } \begin{array}{l} 0 \le x_1 \le 10 \\ 30 \le x_2 \le 40 \\ 30 \le x_2 - x_1 \le 40 \end{array} \\
\frac{\frac{4225}{2}-\frac{155}{4}x_1+\frac{1}{4}x_1x_2+\frac{1}{80}x_1x_2^2+\frac{3}{8}x_1^2-\frac{1}{80}x_1^2x_2+\frac{1}{240}x_1^3-\frac{205}{4}x_2+\frac{3}{8}x_2^2-\frac{1}{240}x_2^3}{50-x_2} & \textbf{if } \begin{array}{l} 10 \le x_1 \le 20 \\ 30 \le x_2 \le 40 \\ 10 \le x_2 - x_1 \le 30 \end{array} \\
\frac{\frac{4225}{2}-\frac{155}{4}x_1+\frac{1}{4}x_1x_2+\frac{1}{80}x_1x_2^2+\frac{3}{8}x_1^2-\frac{1}{80}x_1^2x_2+\frac{1}{240}x_1^3-\frac{205}{4}x_2+\frac{3}{8}x_2^2-\frac{1}{240}x_2^3}{50-x_2} & \textbf{if } \begin{array}{l} 10 \le x_1 \le 20 \\ 40 \le x_2 \le 50 \\ 20 \le x_2 - x_1 \le 30 \end{array} \\
\frac{\frac{4225}{2}-\frac{155}{4}x_1+\frac{1}{4}x_1x_2+\frac{1}{80}x_1x_2^2+\frac{3}{8}x_1^2-\frac{1}{80}x_1^2x_2+\frac{1}{240}x_1^3-\frac{205}{4}x_2+\frac{3}{8}x_2^2-\frac{1}{240}x_2^3}{50-x_2} & \textbf{if } \begin{array}{l} 0 \le x_1 \le 10 \\ 30 \le x_2 \le 40 \\ 20 \le x_2 - x_1 \le 30 \end{array} \\
\frac{2625-\frac{125}{2}x_1-\frac{1}{2}x_1x_2-\frac{1}{40}x_1x_2^2+\frac{3}{4}x_1^2+\frac{1}{40}x_1^2x_2-\frac{1}{120}x_1^3-\frac{75}{2}x_2+\frac{3}{4}x_2^2+\frac{1}{120}x_2^3}{60-x_1} & \textbf{if } \begin{array}{l} 20 \le x_1 \le 30 \\ 0 \le x_2 \le 20 \\ -30 \le x_2 - x_1 \le -10 \end{array} \\
\frac{2625-\frac{125}{2}x_1-\frac{1}{2}x_1x_2-\frac{1}{40}x_1x_2^2+\frac{3}{4}x_1^2+\frac{1}{40}x_1^2x_2-\frac{1}{120}x_1^3-\frac{75}{2}x_2+\frac{3}{4}x_2^2+\frac{1}{120}x_2^3}{60-x_1} & \textbf{if } \begin{array}{l} 30 \le x_1 \le 60 \\ 0 \le x_2 \le 30 \\ -30 \le x_2 - x_1 \le -10 \end{array} \\
\frac{\frac{7850}{3}-60x_1+\frac{1}{2}x_1^2-40x_2+\frac{1}{2}x_2^2}{60-x_1} & \textbf{if } \begin{array}{l} 20 \le x_1 \le 30 \\ 20 \le x_2 \le 30 \\ -10 \le x_2 - x_1 \le 10 \end{array} \\
\frac{\frac{7850}{3}-60x_1+\frac{1}{2}x_1^2-40x_2+\frac{1}{2}x_2^2}{60-x_1} & \textbf{if } \begin{array}{l} 30 \le x_1 \le 40 \\ 20 \le x_2 \le 30 \\ -10 \le x_2 - x_1 \le 0 \end{array} \\
\frac{\frac{7850}{3}-60x_1+\frac{1}{2}x_1^2-40x_2+\frac{1}{2}x_2^2}{60-x_1} & \textbf{if } \begin{array}{l} 20 \le x_1 \le 30 \\ 10 \le x_2 \le 20 \\ -10 \le x_2 - x_1 \le 0 \end{array} \\
\frac{2400-40x_1+x_1x_2-60x_2}{60-x_1} & \textbf{if } \begin{array}{l} 40 \le x_1 \le 60 \\ 0 \le x_2 \le 30 \\ -60 \le x_2 - x_1 \le -30 \end{array} \\
\frac{2400-40x_1+x_1x_2-60x_2}{60-x_1} & \textbf{if } \begin{array}{l} 30 \le x_1 \le 40 \\ 0 \le x_2 \le 10 \\ -40 \le x_2 - x_1 \le -30 \end{array} \\
\frac{111000-3050x_1+50x_1x_2-\frac{1}{2}x_1x_2^2+30x_1^2-\frac{1}{6}x_1^3-3000x_2+30x_2^2}{3000-50x_1+x_1x_2-60x_2} & \textbf{if } \begin{array}{l} 40 \le x_1 \le 60 \\ 30 \le x_2 \le 50 \\ -30 \le x_2 - x_1 \le -10 \end{array} \\
\frac{\frac{332500}{3}+60x_1x_2-3000x_1-\frac{1}{2}x_1^2x_2+25x_1^2-3050x_2+25x_2^2-\frac{1}{6}x_2^3}{3000-50x_1+x_1x_2-60x_2} & \textbf{if } \begin{array}{l} 30 \le x_1 \le 40 \\ 30 \le x_2 \le 50 \\ -10 \le x_2 - x_1 \le 20 \end{array} \\
\frac{\frac{332500}{3}+60x_1x_2-3000x_1-\frac{1}{2}x_1^2x_2+25x_1^2-3050x_2+25x_2^2-\frac{1}{6}x_2^3}{3000-50x_1+x_1x_2-60x_2} & \textbf{if } \begin{array}{l} 20 \le x_1 \le 30 \\ 30 \le x_2 \le 50 \\ 0 \le x_2 - x_1 \le 30 \end{array} \\
\frac{\frac{332500}{3}+60x_1x_2-3000x_1-\frac{1}{2}x_1^2x_2+25x_1^2-3050x_2+25x_2^2-\frac{1}{6}x_2^3}{3000-50x_1+x_1x_2-60x_2} & \textbf{if } \begin{array}{l} 40 \le x_1 \le 60 \\ 30 \le x_2 \le 50 \\ -10 \le x_2 - x_1 \le 10 \end{array}
\end{cases}
\tag{5.16}
$$

(5.16) The value function $V(q_1^{\mathbf{1}}, q_1^{\mathbf{2}}, x^{\mathbf{1}}, x^{\mathbf{2}})$.
Since we have polynomials in x we use the x_i rather than the $x^{\mathbf{i}}$ notation for clocks.

```
N1:
 Z: x1=[10,20] x2=[20,30] x2-x1=[10,20]
 E:(365/8 - 7/16*x1 - 3/80*x1*x2+1/1600*x1*x2^2+3/160*x1^2 - 1/1600*x1^2*x2+1/4800*x1^3 - 9/16*x2+3/160*x2^2 - 1/4800*x2^3)
N2:
 Z: x1=[0,10] x2=[10,30] x2-x1=[10,30]
 E:(365/8 - 7/16*x1 - 3/80*x1*x2+1/1600*x1*x2^2+3/160*x1^2 - 1/1600*x1^2*x2+1/4800*x1^3 - 9/16*x2+3/160*x2^2 - 1/4800*x2^3)
N3:
 Z: x1=[10,20] x2=[0,10] x2-x1=[-20,-10]
 E:(365/8 - 9/16*x1 - 3/80*x1*x2 - 1/1600*x1*x2^2+3/160*x1^2+1/1600*x1^2*x2 - 1/4800*x1^3 - 7/16*x2+3/160*x2^2+1/4800*x2^3)
N4:
 Z: x1=[0,10] x2=[0,10] x2-x1=[-10,10]
 E:(545/12 - 1/2*x1 - 1/40*x1*x2+1/80*x1^2 - 1/2*x2+1/80*x2^2)
N5:
 Z: x1=[10,20] x2=[0,10] x2-x1=[-10,0]
 E:(545/12 - 1/2*x1 - 1/40*x1*x2+1/80*x1^2 - 1/2*x2+1/80*x2^2)
N6:
 Z: x1=[10,20] x2=[10,30] x2-x1=[-10,10]
 E:(545/12 - 1/2*x1 - 1/40*x1*x2+1/80*x1^2 - 1/2*x2+1/80*x2^2)
N7:
 Z: x1=[0,10] x2=[10,20] x2-x1=[0,10]
 E:(545/12 - 1/2*x1 - 1/40*x1*x2+1/80*x1^2 - 1/2*x2+1/80*x2^2)
N8:
 Z: x1=[10,20] x2=[40,50] x2-x1=[30,40]
 E:(2000+x1*x2-50*x1-40*x2) / (50-x2)
N9:
 Z: x1=[0,10] x2=[40,50] x2-x1=[30,50]
 E:(2000+x1*x2-50*x1-40*x2) / (50-x2)
N10:
 Z: x1=[0,10] x2=[30,40] x2-x1=[30,40]
 E:(2000+x1*x2-50*x1-40*x2) / (50-x2)
N11:
 Z: x1=[10,20] x2=[30,40] x2-x1=[10,30]
 E:(4225/2 - 155/4*x1+1/4*x1*x2+1/80*x1*x2^2+3/8*x1^2 - 1/80*x1^2*x2+1/240*x1^3 - 205/4*x2+3/8*x2^2 - 1/240*x2^3)/(50 - x2)
N12:
 Z: x1=[10,20] x2=[40,50] x2-x1=[20,30]
 E:(4225/2 - 155/4*x1+1/4*x1*x2+1/80*x1*x2^2+3/8*x1^2 - 1/80*x1^2*x2+1/240*x1^3 - 205/4*x2+3/8*x2^2 - 1/240*x2^3)/(50 - x2)
N13:
 Z: x1=[0,10] x2=[30,40] x2-x1=[20,30]
 E:(4225/2 - 155/4*x1+1/4*x1*x2+1/80*x1*x2^2+3/8*x1^2 - 1/80*x1^2*x2+1/240*x1^3 - 205/4*x2+3/8*x2^2 - 1/240*x2^3)/(50 - x2)
N14:
 Z: x1=[20,30] x2=[0,20] x2-x1=[-30,-10]
 E:(2625 - 125/2*x1 - 1/2*x1*x2 - 1/40*x1*x2^2+3/4*x1^2+1/40*x1^2*x2 - 1/120*x1^3 - 75/2*x2+3/4*x2^2+1/120*x2^3)/(60 - x1)
N15:
 Z: x1=[30,60] x2=[0,30] x2-x1=[-30,-10]
 E:(2625 - 125/2*x1 - 1/2*x1*x2 - 1/40*x1*x2^2+3/4*x1^2+1/40*x1^2*x2 - 1/120*x1^3 - 75/2*x2+3/4*x2^2+1/120*x2^3)/(60 - x1)
N16:
 Z: x1=[20,30] x2=[20,30] x2-x1=[-10,10]
 E:(7850/3 - 60*x1+1/2*x1^2-40*x2+1/2*x2^2)/(60 - x1)
N17:
 Z: x1=[30,40] x2=[20,30] x2-x1=[-10,0]
 E:(7850/3 - 60*x1+1/2*x1^2-40*x2+1/2*x2^2)/(60 - x1)
N18:
 Z: x1=[20,30] x2=[10,20] x2-x1=[-10,0]
 E:(7850/3 - 60*x1+1/2*x1^2-40*x2+1/2*x2^2)/(60 - x1)
N19:
 Z: x1=[40,60] x2=[0,30] x2-x1=[-60,-30]
 E:(2400 - 40*x1+x1*x2-60*x2)/(60 - x1)
N20:
 Z: x1=[30,40] x2=[0,10] x2-x1=[-40,-30]
 E:(2400 - 40*x1+x1*x2-60*x2)/(60 - x1)
N21:
 Z: x1=[40,60] x2=[30,50] x2-x1=[-30,-10]
 E:(111000 - 3050*x1+50*x1*x2 - 1/2*x1*x2^2+30*x1^2 - 1/6*x1^3 - 3000*x2+30*x2^2)/(3000 - 50*x1+x1*x2-60*x2)
N22:
 Z: x1=[30,40] x2=[30,50] x2-x1=[-10,20]
 E:(332500/3+60*x1*x2-3000*x1 - 1/2*x1^2*x2+25*x1^2-3050*x2+25*x2^2 - 1/6*x2^3)/(3000 - 50*x1+x1*x2-60*x2)
N23:
 Z: x1=[20,30] x2=[30,50] x2-x1=[0,30]
 E:(332500/3+60*x1*x2-3000*x1 - 1/2*x1^2*x2+25*x1^2-3050*x2+25*x2^2 - 1/6*x2^3)/(3000 - 50*x1+x1*x2-60*x2)
N24:
 Z: x1=[40,60] x2=[30,50] x2-x1=[-10,10]
 E:(332500/3+60*x1*x2-3000*x1 - 1/2*x1^2*x2+25*x1^2-3050*x2+25*x2^2 - 1/6*x2^3)/(3000 - 50*x1+x1*x2-60*x2)
```

Listing 5.1: The value function $V(q_1^{\mathbf{1}}, q_1^{\mathbf{2}}, x^{\mathbf{1}}, x^{\mathbf{2}})$ as produced by our tool

Chapter 6

Modeling embedded systems with timed automata

6.1 Preliminaries

Our guiding modeling principle is to abstract away as much as possible from low-level details such as the application code itself or hardware protocols and compensate for the lack of precise information by increasing the uncertainty margins and taking this uncertainty seriously in the analysis. There will be several types of under-determination in the durations of tasks and data transfers or their arrival rates. These can be due to various origins: tentative ignorance in early development stages, true data-dependent variability in the algorithms or unmodeled variability in the architecture workload and physical conditions.

Our goal is to provide HW/SW designers with a tool for rapid design space exploration, that is, given an application and architecture description with a specific deployment scheme, provide performance evaluation at early stage in the design flow. For this we provide a high level language for model description, on the top of timed automata models, which is the input for simulation and formal verification (Fig-6.1).

Timed automata [6] have been invented to model delays and execution times in quantitative way. We choose this formalism as a basis of our modeling framework. More precisely we use IF toolset [48] where the timed model is that of *timed automata with urgency* [44]. In this chapter, we present our modeling framework, while the syntax of our *description language* as well as the techniques used for performance evaluation will be discussed in Chapter 7.

Evaluation can be done on different metrics such as platform cost or application latency in order to compare several deployment strategies. Such criteria can be conflicting in the sense that improving one of them implies worsening the other. Initially, designers have several alternatives both in application implementation and in the choice of the target platform. Providing a way for rapid evaluation of different options at early design stages, can help them to identify weakness sooner. That's why we believe that a clear distinction has to be made between applications and target platforms, enabling a quick comparison between various *combinations* of implementations to evaluate trade-offs.

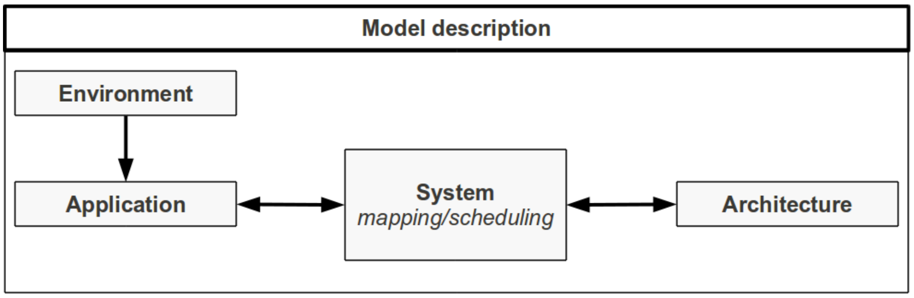


Figure 6.1: Model description

As depicted in Fig-6.1 we divide the model description into 4 parts:

- **Applications** are modeled at the task level, i.e a piece of code is modeled as a timed process. The basic unit is a *job* decomposed into tasks. A task is characterized by quantitative estimations of its duration and the amount of data it exchanges with other tasks. Such a description is compatible in spirit with numerous data-flow and component-based framework [31, 82, 179, 183, 22] advocated for writing such applications.
- **Environment** models the dynamics of task arrival according to some logical and timing constraints.
- **Architecture** is an abstraction of MPSoC including high-level performance related features such as processor speeds, bandwidth and latency of communication mechanisms, static and dynamic power consumption of architecture elements, etc.
- **System** specifies the deployment policy for the application on the platform, namely mapping, scheduling and buffer sizing.

6.2 Application Model

Application reflects software components and their logical interactions without considering target platforms. They are described by task-data graphs (Fig-6.2) which are a simple generalization of the common task-graph model [64]. The basis component is a *task* that is an atomic computational entity. Application behavior is determined by the relationship between these tasks, defining control and data flows. Control flow is represented by *precedence links* meaning that a task cannot start before some other tasks terminate. On the other hand, data flow specifies that data has to be communicated to a computational entity. It can be defined in two ways. In the first, a task produces an amount of data which is consumed by another task. This is achieved by adding on precedence link a quantity of data that has to be transferred between a task and its successors. However, communications are not always the results of an explicit producer/consumer scheme. A computation may need a chunk of some specific data structure for executing, like for example in image processing where images are encoded by some object and several operations work on pieces of it. The initial object is placed in some memory component, depending of the target architecture, not necessarily known when the application is modeled. For this reason, we introduce the concept of *abstract data* and *communication task* to model the fact that some task needs a quantity of data to execute, meaning that data will be transferred between the memory component on which data is initially placed and the local memory of the processing element on which the task has been mapped. Depending on the mapping of the tasks onto the architecture and the data transfer mechanism used, e.g. DMA (direct memory access) or inter-process communication, these transfers are

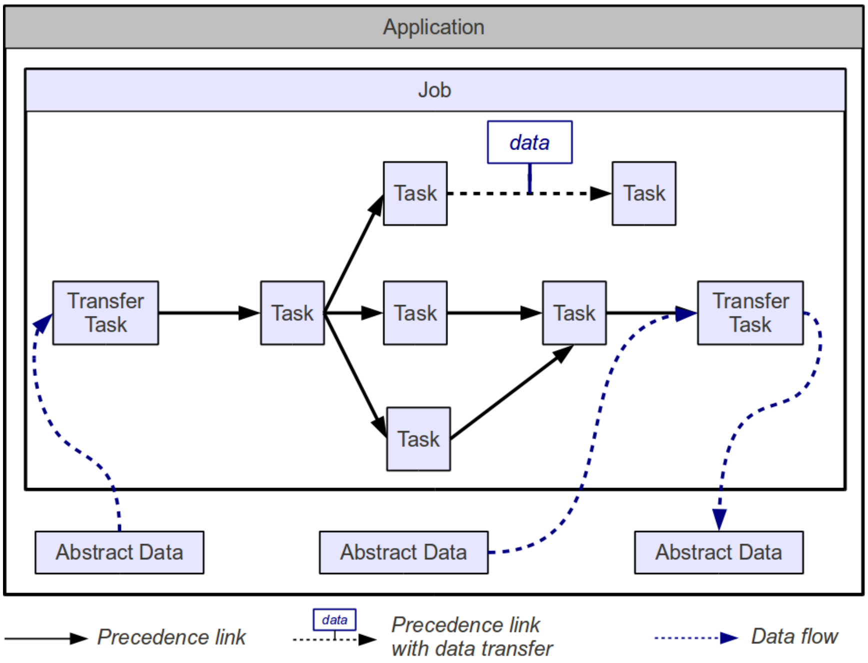


Figure 6.2: Application model

transformed into special communication tasks. The whole task-data graph is called a job type and it is the basic unit of work whose instances arrive to be executed.

6.2.1 Task Model

A task is an atomic computational entity, similar to actors in SDF [134], processing kernels or filters in streaming models of computation [183]. We distinguish computational tasks, reflecting the execution of some piece of code, and transfer tasks, expressing data communications. Task models are defined independently from target platform, that is, explicit deployment is specified by the *system* component, and implies resource immobilization and a translation of tasks attributes into duration.

A computational task is characterized by an amount of work measured by instructions or cycles. Once a task is scheduled to execute on a processor with a given frequency, its amount of work is translated into duration. At this level of abstraction it may be difficult to estimate precisely this amount. To compensate for this, we define it with bounded uncertainty. The termination of a task may be a pre-condition to the initiation of other tasks, that is, a task cannot start before some other tasks terminate and this is modeled by *precedence links*. Precedence is viewed as a control dependency but is also a common way to model data dependency between computational entities and can be decorated with some amount of data when tasks need to communicated a non-negligible quantity of data.

Definition 6.2.1. ***(Task)*** *A Task τ is defined by an amount of work $w_\tau = [l_\tau, u_\tau]$ with $l_\tau \leq u_\tau$ measured by instructions or cycles. When a task τ is executed on a processing element working at frequency f, its execution time is in the interval $[\frac{l_\tau}{f}, \frac{u_\tau}{f}]$*

A generic timed automaton for a task is shown in Fig-6.3. Symbols ? and ! denote respectively input and output signals. For every task, its automaton stays in state *Wait Pred* until all its predecessors terminate (incoming signal *pred*). Termination of predecessors includes control

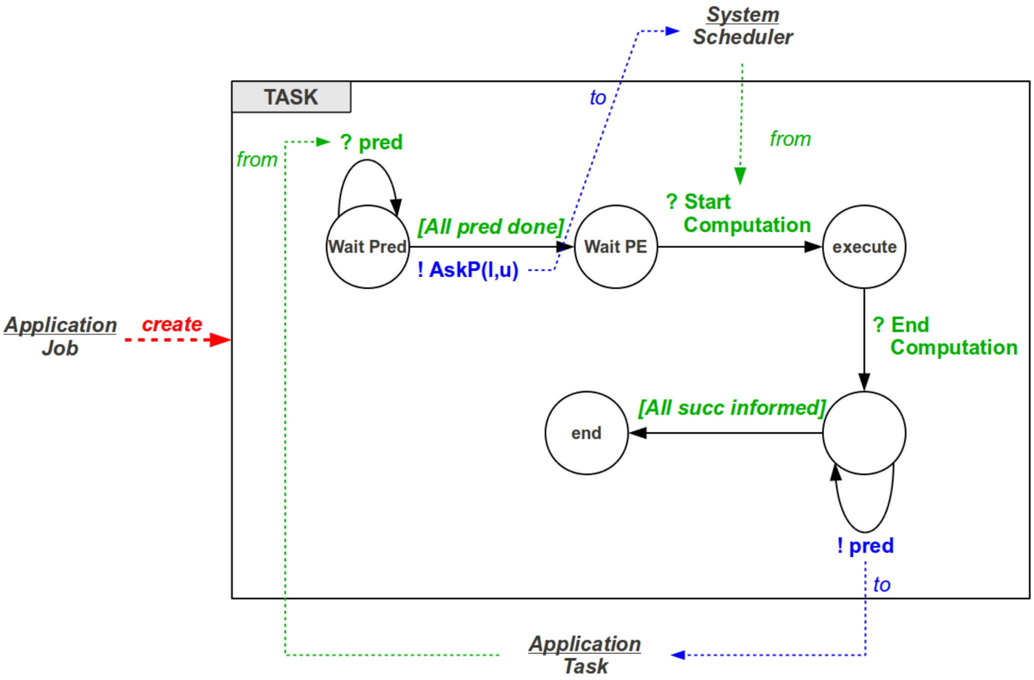


Figure 6.3: Timed automaton for a task

dependency of other tasks and data dependency involving communication latency. Once all precedence constraints are satisfied, the task requests a PE for an interval of workload (signal *askP(l,u)*). When the system component maps it to a PE (signal *Start Computation*) it moves to the *execute* state. Termination time (signal *End Computation*) depends on the task workload and the PE characteristics. Notice that execution are non-preemptive, that is, once a task is mapped on a processing element, the processor is released only after the entire task terminates. Signal *pred* is a way to inform all successors of task termination.

6.2.2 Data Model

As explained in [112] data is an important aspect of all models, and going from one representation of data to another representation often marks a distinctive step in the design flow. Idealized data types, such as real or integer numbers, or tokens, are useful for investigating the principal properties of an algorithm or a functionality not blurred by the maneuvers necessary to deal with concrete, implementation-oriented data types. *Symbols* are used to further abstract from the detailed properties of data which in reality may contain hundreds of thousands of bits arranged according to some specific structure.

We abstract data as simple data blocks which can be accessed by read/write operations when tasks are transferring some amount of data. We do not focus on any structural aspect of these data blocks. Here the use of *abstract data* is a way to deal with data placement, that is on which memory component data resides and to which memory component it has to be transferred. This *mapping* will be defined in the *system* component and defining *abstract data* is done on the *application* side *independently* from a specific architecture.

Data transfer is modeled by *communication tasks* which simply specify that some amount of data has to be communicated between two application components. Concrete communication realization depends on the mapping and the platform communication model e.g. DMA (direct memory access) or inter-process communication.

Definition 6.2.2. ***(Communication task)*** *A communication task is defined by a triple $\tau^c = (src, dst, \delta)$ where $\delta = [l_{\tau^c}, u_{\tau^c}]$ defines a quantity of data measured in bytes to be communicated from a software component src to another software component dst.*

Because some treatments, like image processing, can produce a variable amount of data, we allow bounded uncertainty on the data quantity to be communicated. Another characteristic for *communication tasks* is precedence as for other tasks, it models the fact that data are produced or consumed by some computational entity.

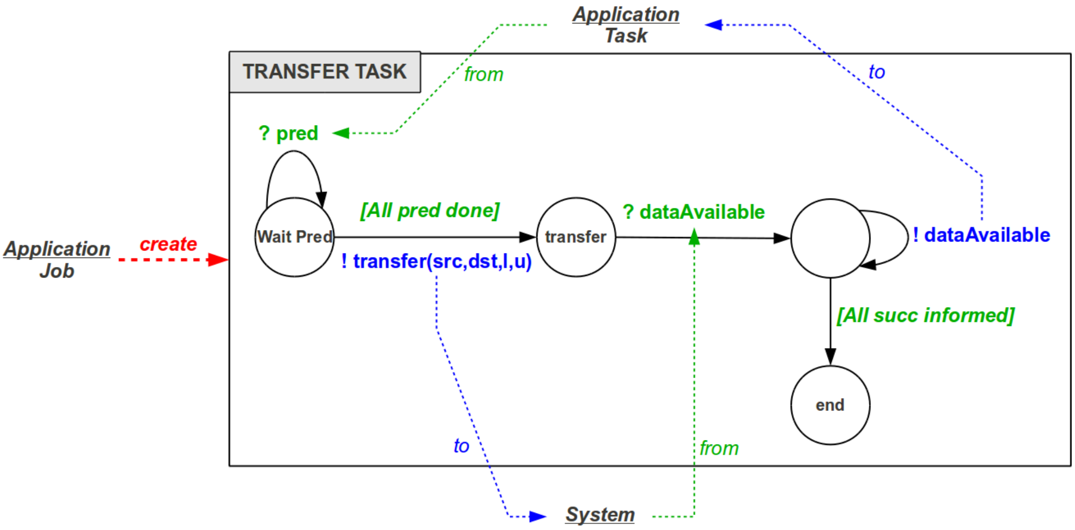


Figure 6.4: Timed automaton for a communication task

The timed automaton for a *communication task* (Fig-6.4) stays in state *Wait Pred* until all its predecessors have finished. The triggering of a signal *transfer* to the system initiate the transfer of an amount of data ($[l, u]$) from src to dst. Depending on the mapping to the target platform, this will imply more or less latency and immobilization of different components: memory, buses, DMA.

Once data has been written on target memory, the system component reports this fact by sending a signal *dataAvailable* which denotes the end of the transfer and signal *dataAvailable* is used to inform all successors.

The reason for using *communication tasks* is to make the model more general than the classical task graph model where data transfer are specified on precedence links. With this model, data can be communicated, not only between two computational entities, but from or to some abstract data whose placement is defined afterwards depending on target architectures.

To illustrate this consider the simple example shown in Fig-(6.5). Given this simple algorithm we would like to evaluate different implementation alternatives. At the application level, objects manipulation are abstracted using *abstract data components*, omitting data placement on target platform. The performance of the application on specific hardware architecture will then depend on data mapping, i.e placement of A and B on memory more or less distant from the processing element on which computation is done will give more or less communication latency. Evaluating different mapping strategies can then be done easily without changing neither application nor architecture models.

As we said previously, *communication tasks* are used to model communications between *software components*, involving data transfer between two processing entities. We can distinguish different cases:

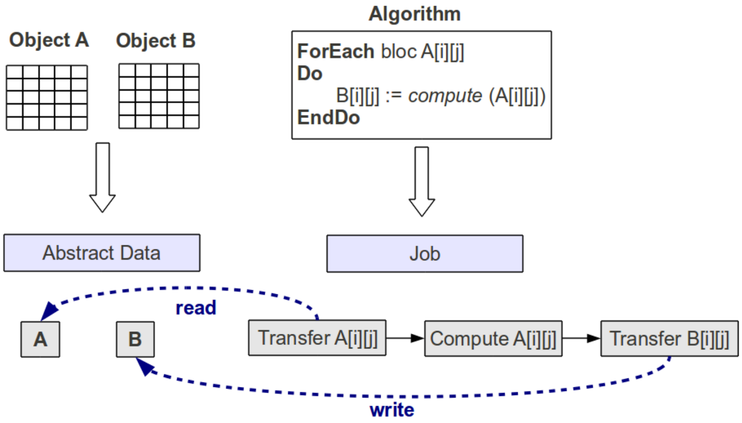


Figure 6.5: Example of communication task usage

- transfer(τ_1, τ_2, δ) for transferring δ bytes produced by task τ_1 to task τ_2 i.e from local memory of processor on which τ_1 has been mapped to local memory of processor on which τ_2 has been mapped.

- transfer(d, τ, l, u) for reading pieces of an abstract data d, used by τ. Transfer is done from memory component where d has been mapped to local memory of processor on which τ has been mapped.

- transfer(τ, d, l, u) for writing pieces of an abstract data d produced by τ. Transfer is done from local memory of processor on which τ has been mapped to memory component where d has been mapped.

The use of this kind of communication model is useful for describing dataflow concepts such as pipelining or data prefetching. This will be discussed when we present DMA communication in the architectural side.

6.2.3 Job Model

Components defined above are structured in a particular object named *Job* which is the basic unit of work whose instances arrive to be executed.

Definition 6.2.3. *(Job) A Job is directed acyclic graph $J = (T, C, E)$ where*

- *$T = \{\tau_1, ...\tau_n\}$ is a set of computation tasks*

- *$C = \{\tau_1^c, ...\tau_m^c\}$ is a set of communication tasks*

- *E is a set of ordered pairs of tasks (precedence link), defining a strict partial order precedence relation on $T \cup C$*

Jobs instantiation depends on *environment* components called *generators* that define their arrival scheme. A job and its underlying elements are created dynamically according to generators triggering.

Jobs have been extended to allow *loops* modeling, that is, jobs that execute a finite number of iterations sequentially. In this case, instantiation of a new iteration is done by the job itself at the end of its own execution.

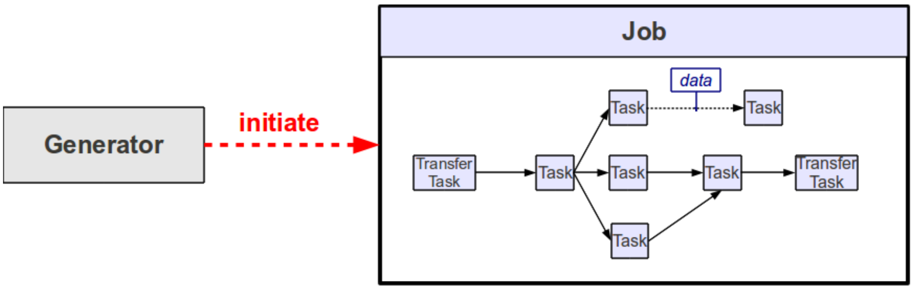


Figure 6.6: Job instantiation

6.3 Environment Model

One aspect of scheduling which is treated differently along communities is the dynamic aspect: in classical real-time scheduling new task instances arrive *periodically* or quasi-periodically but these are traditionally simple tasks without precedence constraints. In contrast, job-shop and task-graph problems typically do not handle the dynamic "reactive" aspect, that is, a *stream* of job instances that arrive one after the other, for example, a sequence of encoded image frames or web queries. This aspect is extremely important, first because it represents the real nature of these applications and, secondly, it favors solutions based on *pipelining* which is the concurrent execution of tasks that belong to *different* job *instances* (see some definitions and theoretical investigations in [76]). One approach to treat this recurrence aspect is to use *cyclic* task-graphs admitting a loop from the last to the first task. While this might be suitable for modeling loops in programs where the termination of one instance enables the execution of the next one, it is not at all natural for jobs arriving from the *outside*, often independently of their processing by the system. To this end we use the concept of an *input generator*, a process that generates a timed sequence of job instances subject to some logical and timing constraints. The simplest generator is the deterministic periodic generator which produces an instance of a job repeatedly in regular time intervals. Strictly periodic generators are sometimes idealization of more time-noisy processes and we allow additional types of non-deterministic generators listed below. The environment model is not restricted to a single generator, but one can define several different generators. In all following definitions, O denotes an offset i.e the time at which the first event is generated.

Single Shot Generator

This is the simplest generator which generates only one job instance after some non-deterministic *offset* O, defined as a time interval. The automaton for this generator is depicted on Fig-6.7. A job instance is created in the time interval $[l, u]$.

Periodic Generator

A periodic generator generates an event e every P units of time. The sequence $\{t_k\}_{k=1...n}$ of time stamps for the events satisfies $\forall k \in \mathbb{N}, t_k = O + k \cdot P$

Periodic Generator with Uncertainty

For this generator the time between two events ranges in the interval $[P, P + J]$, that is, the time stamps t_k satisfy $\forall k \in \mathbb{N}, t_k + P \leq t_{k+1} \leq t_k + P + J$ where $t_0 = O$. This type of generator ensures that two events are separated by at least P and by at most $P + J$ time units.

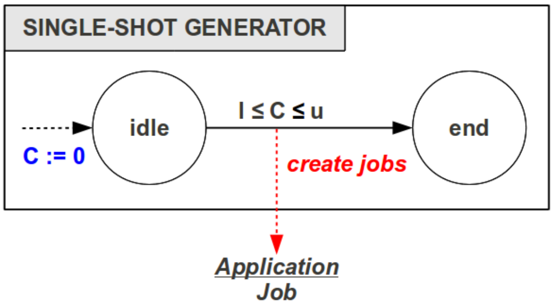


Figure 6.7: Timed automaton for a single shot generator

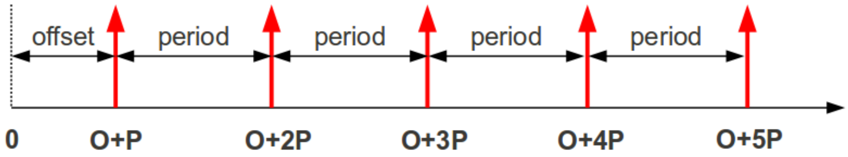


Figure 6.8: Periodic generator timeline

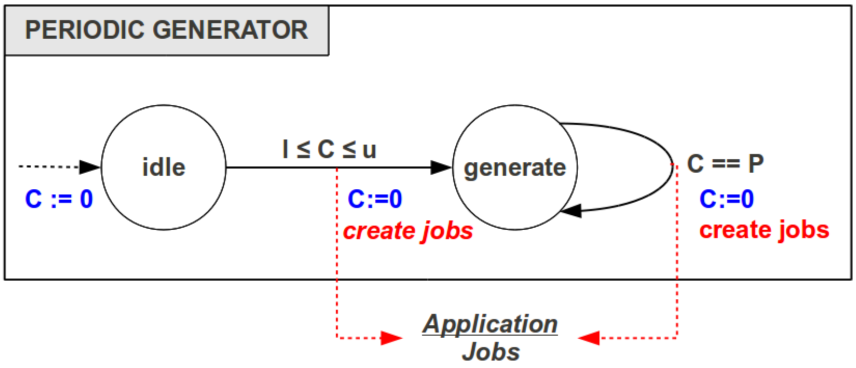


Figure 6.9: Timed automaton for a periodic generator

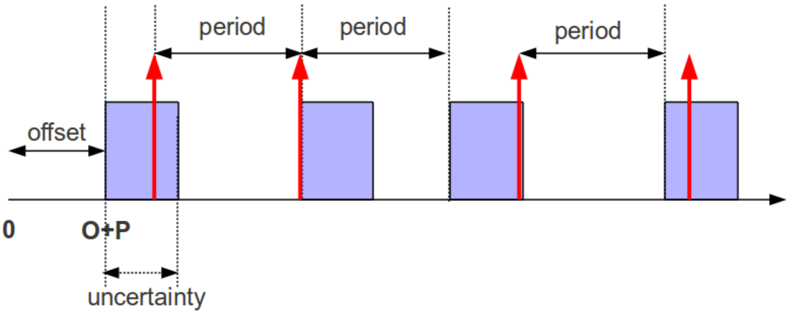


Figure 6.10: Periodic generator with uncertainty

The timed automaton (Fig-6.11) for this generator uses one clock. The first event is generated after offset in interval $[O, O + J]$ and then after each event the clock is reset to zero and the next event is generated when $C \in [P, P + J]$

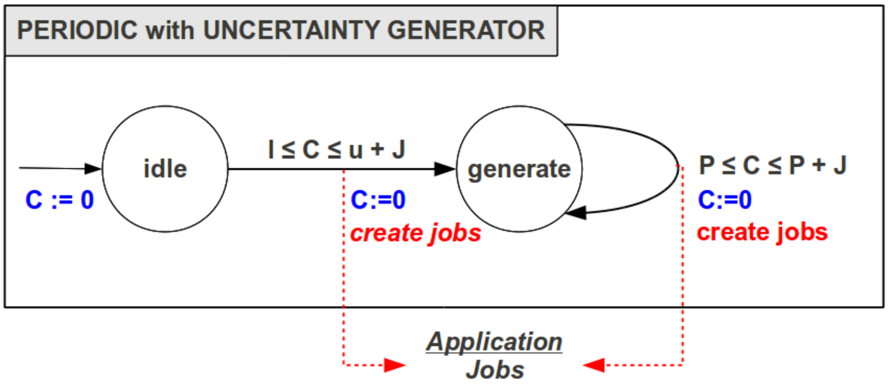


Figure 6.11: Timed automaton for a periodic generator with uncertainty

Periodic Generator with Jitter

The time stamps of the events produces by this generator satisfy, $\forall k \in \mathbb{N}, O + k \cdot P \leq t_k \leq O + k \cdot P + J$

Notice that we allow jitter to be greater than period in order to model *bursts*, a finite number of events generated, practically, at the same time.

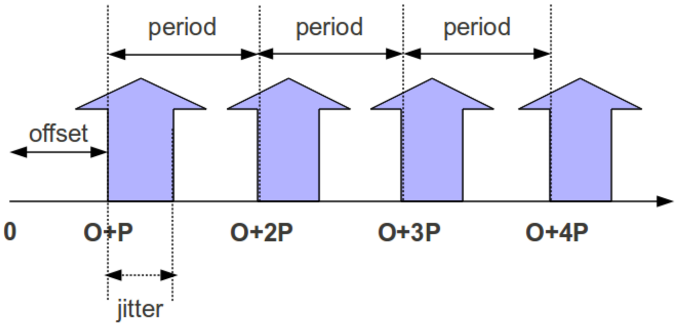


Figure 6.12: Periodic generator with jitter smaller than period

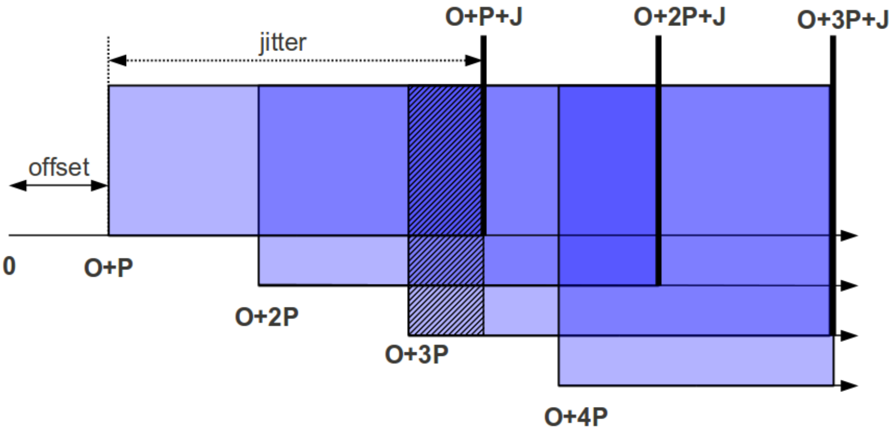


Figure 6.13: Periodic with jitter greater than period

The timed automaton for this generator (Fig-6.14) uses a counter N to enable jitter J to be greater than the period P. Each time, clock C reaches the time period, N is incremented and the next event must be generated in the interval $[0, J - P \cdot N]$. If $J \leq P$ then events are generated in $[0, J]$ as shown in Fig-6.12. Otherwise, if $J > P$ many events could be generated in a short period. In Fig-6.13 we can see an example where $J = 2.5 \times P$, the shaded area shows a time frame, intersection of 3 "triggering intervals", where 3 events can be generated simultaneously.

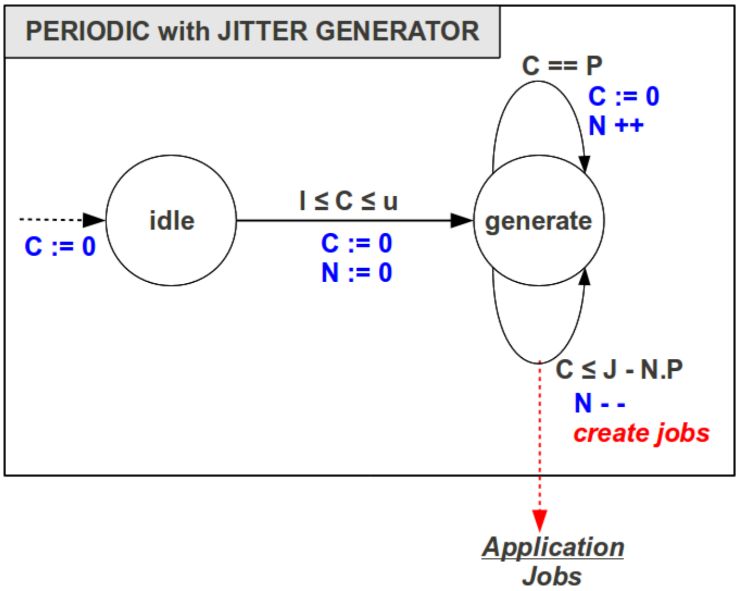


Figure 6.14: Timed automaton for periodic generator with jitter

Bounded Variability Generator

Let N_I be the number of events in interval I. For this generator in every interval of time length Δ the number of events e is at most n, $\forall t, N_{[t,t+\Delta]} \leq n$. Fig-6.15 shows an example of timeline for a bounded variability generator which generates at most 2 events in every time interval of length Δ.

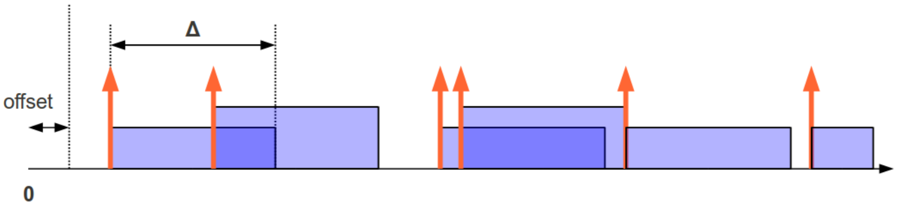


Figure 6.15: A bounded variability generator

The associated timed automaton (Fig-6.16) uses a vector of clocks of size MAX. Variable MAX denotes the maximum number of events that can be generated in any interval of length Δ. Variable *first* (resp. *last*) is used to memorize the index of the first (resp. last) clock measuring time since the generation of the oldest event in the time frame of length Δ. Variable N counts the number of events in the current time frame. Each time the first clock reaches Δ, N is decremented and the index *first* is incremented to the next clock. An event can be generated at any time when

$N < M$ and the time at which it has been generated is tracked by a new clock indexed by variable *last*. When the number of events is equal to M we wait until *first* time frame ended before enabling new event to be generated.

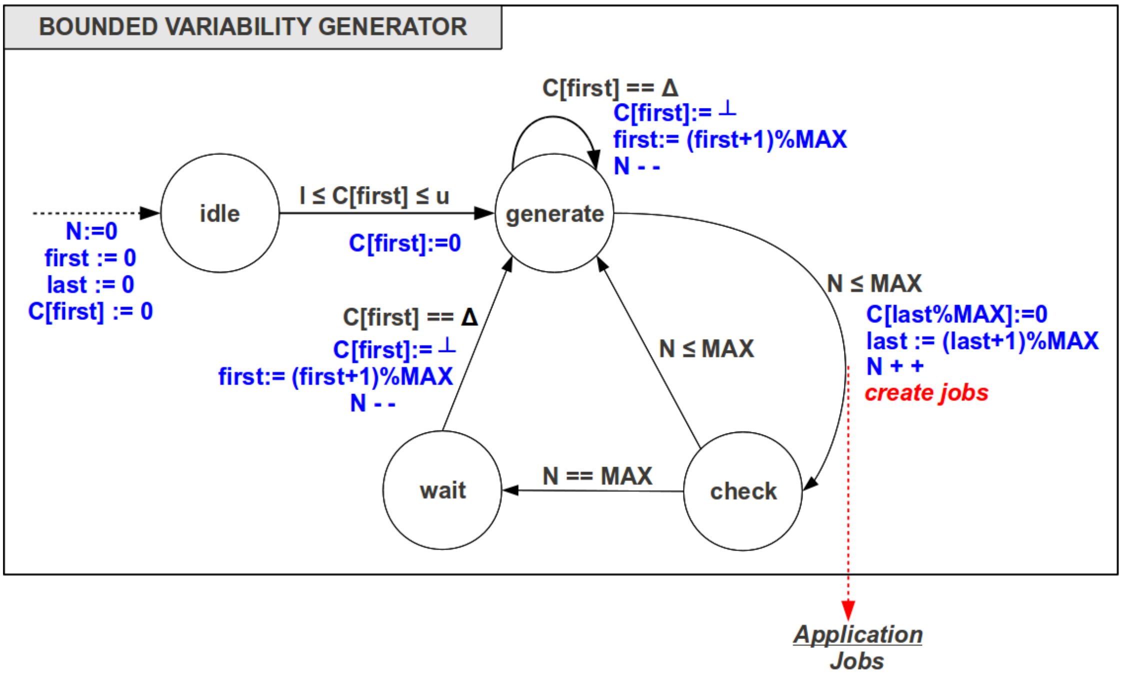


Figure 6.16: Timed automaton for bounded variability generator

Bi-Bounded Variability Generator

For every interval of length Δ the number of events e is at least m and at most n, $\forall t, m \leq N_{[t,t+\Delta]} \leq n$. Fig-6.17 shows an example of timeline for a bi-bounded variability generator which generate at least 2 and at most 3 events in any time interval of length Δ (shown as blue rectangles).

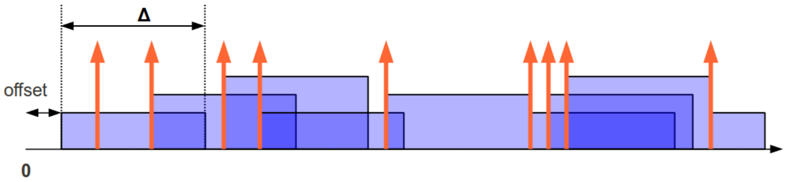


Figure 6.17: A Bi-bounded variability generator

The timed automaton (Fig-6.18) works in the same way as that of bounded variability for the generation of maximal number of event MAX. The difference comes from the state *checkMin* which is reached each time the current time frame ends, and we remain in this state until the minimal number of event MIN have been generated.

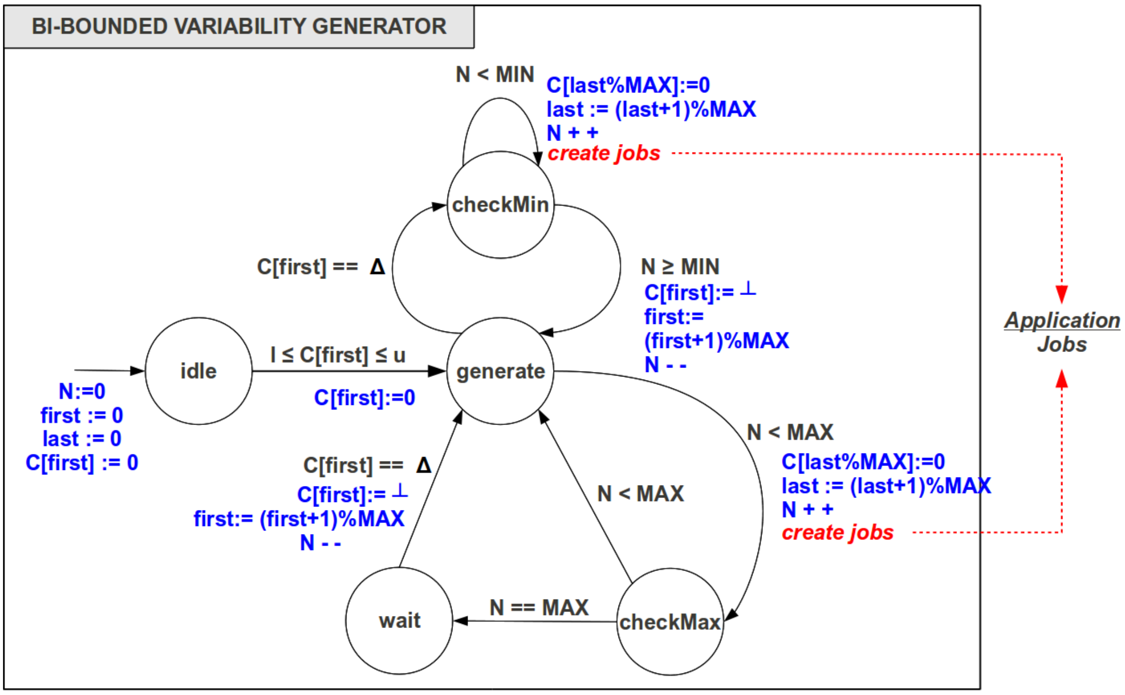


Figure 6.18: Bi-bounded variability generator timed automaton

6.3.1 Generators Characteristics

We summarize here some metrics about the different generators previously presented. Symbol Δ represents a reference interval length and is associated with the period P in case of periodic-kind generators. $Min(a)/\Delta$ (resp. $Max(a)/\Delta$) shows the minimal (resp. maximal) number of events that can be generated *in any* interval of length Δ. Finally inter-arrival Min (resp. Max) indicates the minimal (resp. maximal) inter-arrival duration between two consecutive events. Notice that we assume that interval defined by Δ is left-closed and right-open. For example for periodic generator, considering a closed-interval, $Max(e)/\Delta$ will be 2 instead of 1.

Table 6.1: Characteristics of the different generators

				Inter-arrival	
Generator	Min(e)/Δ	Max(e)/Δ	Δ	Min	Max
Periodic	1	1	P	P	P
Periodic with jitter	0	$\lceil \frac{J}{P} \rceil$	P	$max(0, P - J)$	$P + J$
Periodic uncertainty	0	1	P	P	$P + J$
Bounded variability	0	n	Δ	0 (Δ if $n = 1$)	∞
Bi-bounded variability	m	n	Δ	0	Δ

6.4 Architecture Model

The architecture model defines the platform topology that is processing elements and memory connected through some communication infrastructure. Modeling these components is generally done with low level specification using HDL (VHDL, Verilog) or combined with software specification at system level (e.g SystemC). For rapid evaluation of different platforms or applications we argue that it is necessary to keep applications and platforms as *independent* as possible. We will present now, how we specify MPSoC architecture at a high level using timed automata models. Examples of different complete platforms will be shown in chapter 8, while here we focus on models for each of the components.

6.4.1 Processing Elements

Processors can be of any kind (general purpose, controller, DSP, ASIC . . .) and are differentiated by their specific properties. The simplest model for a processor is defined with a frequency/speed which sets the number of cycles/instructions per time unit.

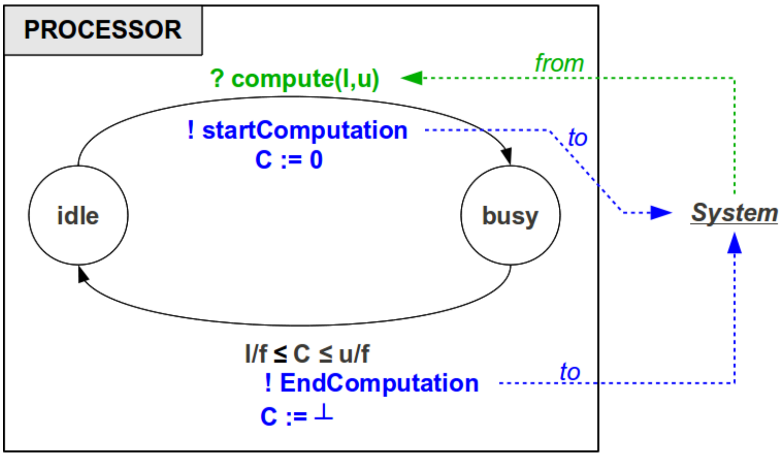


Figure 6.19: processor timed automaton

The timed automaton (Fig-6.19) for a simple processor model consists of two states, it stays in state *idle* until it receive *compute(l,u)* from a task component through the system component then it goes to state *busy* for a duration depending on the required amount of work ($[l, u]$ defined as cycle or instruction number) and its frequency f (i.e $[\frac{l}{f}, \frac{u}{f}]$).

Power consumption is, in some situations, no less important than execution time. We distinguish *energy* and *power* as power is energy consumption per time unit. The high level power consumption characteristics of processing elements and other components can be identified [198, 27] by:

- **Voltage drops**: The dynamic power consumption is proportional to the square of the power supply voltage (V^2). Therefore, by reducing the power supply voltage to the lowest level that provides the required performance, we can significantly reduce power consumption
- **Toggling**: A circuit uses most of its power when it is changing its output value. By reducing the speed at which the circuit operates, we can reduce its power consumption (although not the total energy required for the operation, since the result is available later).
- **Leakage**: Even when a circuit is not active, some charge leaks out of the circuit nodes. Completely disconnecting the power supply eliminates power consumption, but it takes a significant amount of time to reconnect it.

As an example Fig-6.20 shows the power state machine [27] of StrongARM SA-1100 [111] which provides three power modes, *run* mode is normal operation and has highest power consumption, *idle* mode saves power by stopping the processor clock and *sleep* mode shuts off most of the activity.

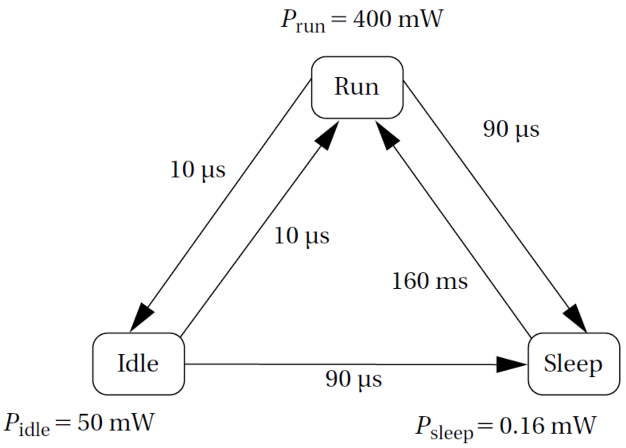

$P_{\text{run}} = 400$ mW
Run
10 µs
90 µs
10 µs
160 ms
Idle
90 µs
Sleep
$P_{\text{idle}} = 50$ mW
$P_{\text{sleep}} = 0.16$ mW

Figure 6.20: Power state machine of StrongARM SA-1100 *from [27]*

Note that adding power consumption to the model we move beyond timed automata to the model of linearly-priced timed automata [24, 128] but since we are applying discrete event simulation we do not have to worry about it. In fact, discrete event simulation can be applied also to richer models of hybrid automata whose reachability problem is undecidable. Power consumption estimation will be explained in Chapter-7.

In the same spirit we define a model for handling Dynamic Frequency Scaling (DFS). This is a widely used technique aimed at adjusting computational power to application needs. It is often associated with Dynamic Voltage Scaling (DVS) enabling significant power reductions when computing demand is low. We define the model of this kind of processor with a fixed number of frequency level. For each of them one has to define corresponding power consumption information and latency for switching between different modes.

6.4.2 Memory

Memory can be of different types, here we restrict ourself to non banked memory (i.e reading and writing are blocking operation). *Writing* and *Reading* are separated to allow different parameters definition depending on each operation, for example, consumption can be different for reading or writing on a memory or it could be faster to read than write or conversely.

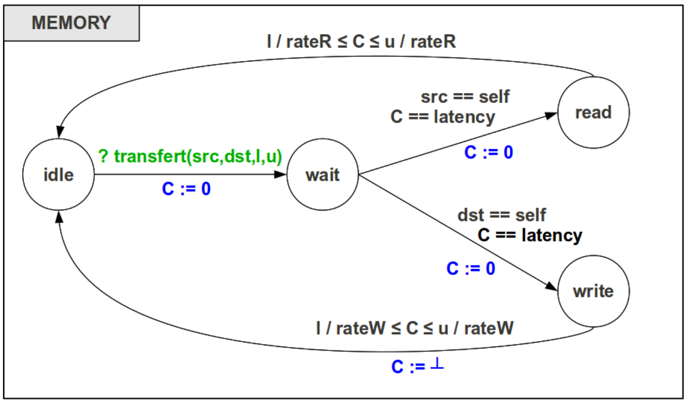


Figure 6.21: Timed automaton for a memory component

A memory component is defined with different latency parameters, an initial latency which occurs each time the component is accessed, and a rate (one for reading and one for writing) which defines the number of *data units* treated per time unit. Associated with different states of the automaton (Fig-6.21) we define consumption parameters.

One memory component can be linked to a processing element in order to model local memory or to a bus component to model shared/off-chip memory as shown in Fig-6.22.

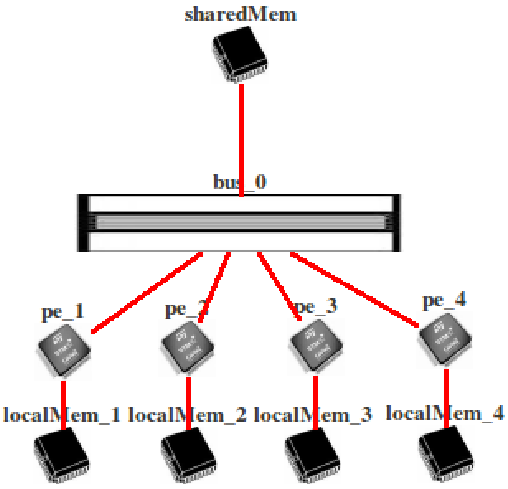


Figure 6.22: Memory architecture example

6.4.3 Communication

Communication is modeled through message passing. Each architectural component is modeled with some kind of *network controller* containing static routing table. *Communication messages* are injected in the architecture model by the system component, according to some application transfer task, on a specific architectural component. For example the routing table of a processing element contains informations about the path to its local memory or to a bus. All routing tables are computed statically from the whole architecture model which is simply a graph.

6.4.3.1 Bus-based Communication

The base component for modeling communication is the *bus*. It consists of a component which links many other components together (memory, processors). It has a finite number of communication channels allowing multiple communications do be done simultaneously.

The bus model stores *communication messages* in a FIFO queue and dispatches them on link of the target component through *communication channels*. Transferring a message has some latency depending on the message size.

6.4.3.2 NOC Communication

The topology of a network on chip specifies the structure in which routers connect IPs together. There are many different structures adopted for the NOC domains such as tree [98] butterfly-tree mesh [159] torus [127] folded torus [69] variations of the ring in octagon [117] and spidergon [66]

We focus here on spidergon scheme of the XSTREAM architecture [32]. In the spidergon topology nodes are connected by unidirectional links. Let the number of nodes be an even number

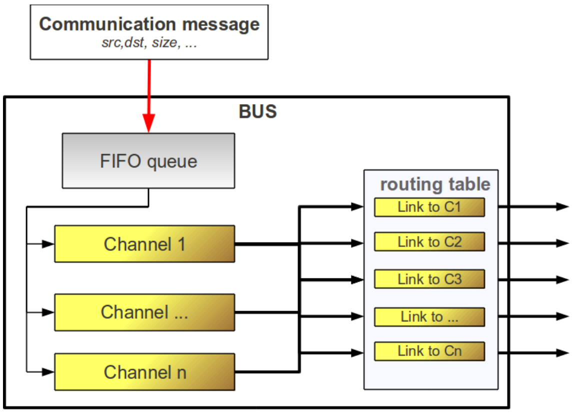
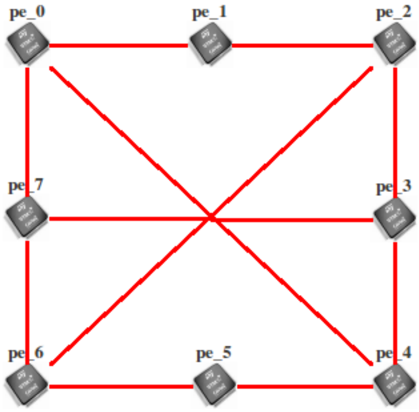


Figure 6.23: Bus model

$N = 2 \times n$. Each node, i.e processing element, in the network pe_i for $0 \leq i \leq N$ is directly connected to node pe_j for $j = (i+1)\ mod\ N$, $j = (i-1)\ mod\ N$ and $j = (i + \frac{N}{2})\ mod\ N$. An example with 8 processing elements is shown in Fig-6.24

Figure 6.24: Spidergon topology with 8 processing elements



A Spidergon topology of N nodes has $\frac{3}{2} \times N$ link and a diameter of $\lceil \frac{N}{4} \rceil$ hops. Each node has its own routing table defined statically and defines the shortest path to a target node. When a *communication message* arrives in a node, the router has to choose between 3 possible outcomes: the clockwise right link l_r, the clockwise left link l_l and the *across* link l_a. As an example the routing table for processing element pe_0 from Fig-6.24 can be defined as shown in Table-6.2

Table 6.2: Routing table for processing element pe_0

Link	Target PEs
l_r	$\{pe_1, pe_2\}$
l_l	$\{pe_7, pe_6\}$
l_a	$\{pe_3, pe_4, pe_5\}$

Communication latency is modeled with link bandwidth parameter and can be of different speeds. The *energy consumption* model is defined by associating a static power value to state *idle* (there is no communication) and a dynamic power value to state *busy* when the link is communicating. Communication among links is blocking, that is, when a transmission starts on a link, the next communication has to wait until the link is released for starting, and *communication messages* are scheduled in FIFO order on the same link in a non-preemptive manner.

6.4.3.3 DMA-based Communication

DMA are dedicated processing elements that can transfer large amounts of data between memory locations without processor intervention. They can be used in a double buffering scheme shown in Fig-6.25, to improve performance.

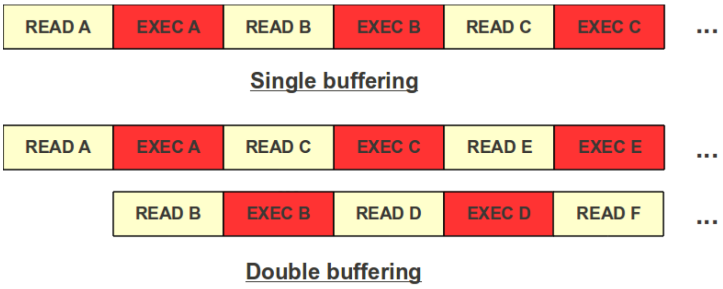


Figure 6.25: Data transfer: single versus double buffering

Typically a DMA engine is used to transfer data between the local memory of processing elements and off-chip memory. We do not restrict ourself to a particular architecture, one can use DMA with PEs having each their own local memory with a global DMA (Fig-6.26) or with DMA dedicated to each PE (Fig-6.27) or with PEs sharing a common local memory (Fig-6.28).

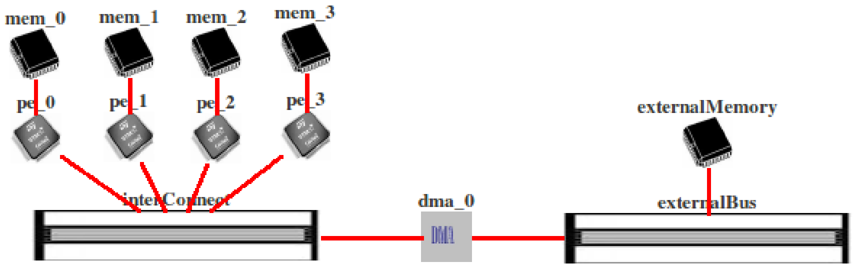


Figure 6.26: DMA architecture with local memory for each PE

DMA is defined with an initialization latency each time a new transmission request is done and some dynamic latency depending on the size of the data to be transmitted. Energy consumption is defined as for other components with static and dynamic power value.

6.4.4 On Different Abstraction Levels

The granularity at which models are defined plays an important role in the simulation and verification steps. Given the opportunity to use different levels of abstraction even at high level of specification can be very useful. One way to use architecture model at highest level of abstraction

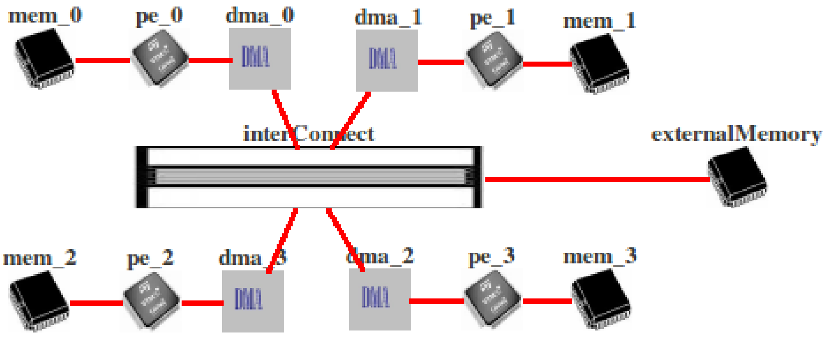


Figure 6.27: Architecture with separate DMA for each PE

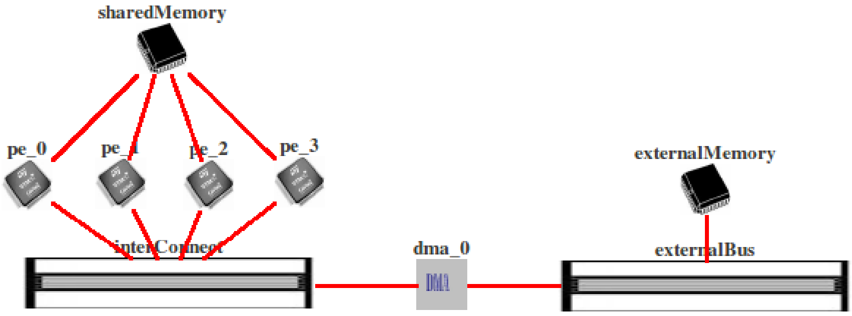


Figure 6.28: DMA architecture with a shared memory for all PEs

could be to entirely abstract communication architecture and is commonly the case in analysis framework ([190, 158, 193, 168]) or in scheduling problems [64, 175, 139]

If the interface (i.e signal transmission) remains the same, then it would be simple to replace a certain bus or NoC model by another by merely replacing the corresponding timed automata. Because each component is defined by its own automaton communicating through *signals* interface, it would not affect the behavior for other hardware components at all. This allows easy investigation of the impact of any different network scheme that can be modeled as a timed automaton.

The simplest model could consist of modeling only processing elements and a bus component which defines the entire communication scheme. A communication task can be *scheduled* on this bus according to some policy and latency is proportional to the amount of data. With this abstraction, memory components are ignored and R/W operations are included in the latency model of the bus. In the same spirit, abstracting network while modeling the use of a DMA can be achieved by defining the transfer time with a more detailed function as

$$T(d) = I + \alpha(p) \times d \tag{6.1}$$

which is inspired from [168] where $T(d)$ represents the time needed for transferring d amount of data through the network (including memory reading and writing, DMA cost), I is some initialization latency (which can correspond to DMA or Bus initialization), $\alpha(p)$ is some transferring cost parametrized by the number of processors sharing the interconnect.

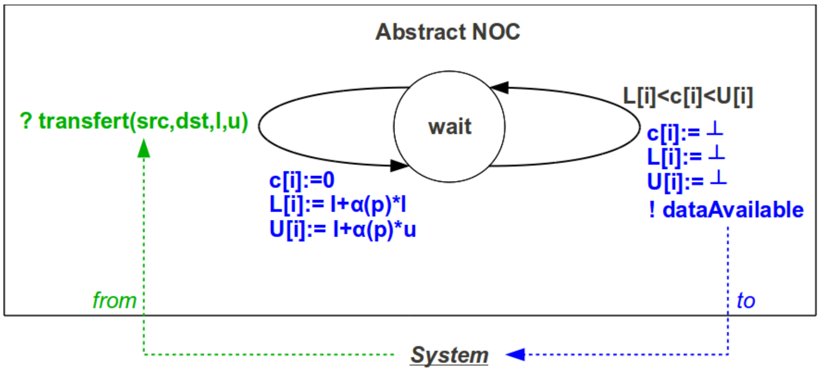


Figure 6.29: Abstract NoC timed automaton

The timed automaton for this *abstract NoC* is depicted on Fig-6.29. Each time a transfer is requested by signal *transfer* a clock $c[i]$ is initialized and the latency is computed according to (6.1). When transfer time for a specific clock elapses the system component is informed by sending the signal *dataAvailable*.

In the same way we can provide more detailed models, especially for data transmission among bus components. Bus accesses are non-preemptive and multiple requests are treated sequentially, while in reality it is done in a round-robin manner, that smoothens the latency of pending transfers. This can be achieved by cutting data transfers into multiple smaller ones, and scheduling them on buses according to a round-robin scheme.

One advantage of our modeling framework is that defining different platform models can be done easily by defining a new timed automaton simulating the desired behavior according to signal arrival. We will discuss how one can extend already existing models in Chapter 7.

6.5 System Model

The aim of the system component is to characterize the *deployment* of an application on an architecture. This deployment is specified through mapping and scheduling definitions. We do not attempt here to find an optimal one but to provide a way to evaluate different *models* easily, that is comparing different scheduling strategies, evaluating an application on different architectures or several applications on a given architecture. For this reason, application and architecture are modeled independently and the link between them is done through this system component.

Scheduling consists in assigning a starting time for each task on a specific architectural component. Optimal schedulers can be sometimes computed, on simplified models, under deterministic behaviors. One might be interested in evaluating them while admitting uncertainty. A schedule can be computed according to different criteria such the overall termination time, the overall energy consumption, the overall communication time, architecture cost, etc. Considering opposing criteria there is no unique optimal solutions but rather a set of incomparable trade-offs.

One has to define several system components depending both on application and architecture (Fig-6.30). *Abstract data* need to be mapped initially on concrete memory components, computation tasks need to be scheduled with some scheme on processing elements and transfer tasks have to be scheduled on the network (maybe initialized by some component like processor or memory).

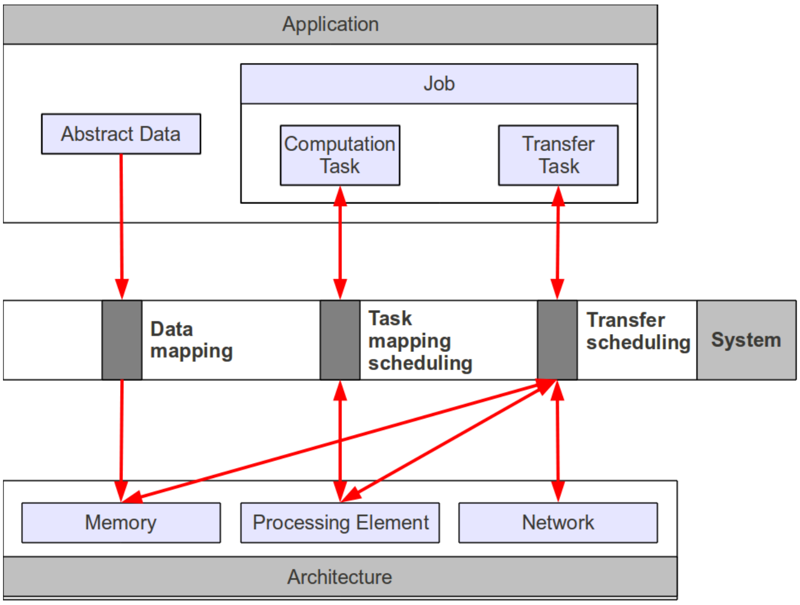


Figure 6.30: System model

6.5.1 Computation Task Scheduling

Task scheduling can be done according to different schemes which can be local to one processing element or global to all of them. We will present in the following those that we have implemented. Adding new scheduling policies can be done easily by writing the scheduler as a timed automaton. Notice that for all following schedulers we assume that the latency of the scheduling operation itself is negligible.

FIFO Scheduling

Tasks are pushed into processing elements according to their arrival order. One can choose a global scheduling with one global FIFO queue, where tasks are mapped to the first available processor (Fig-6.31). In that case, task deployment is done dynamically during execution and task instances could execute on different processing elements.

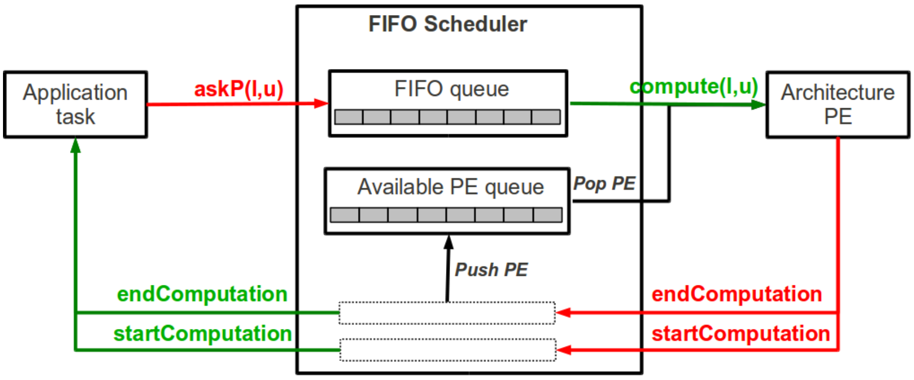


Figure 6.31: Global FIFO scheduler

Another way to use FIFO scheduling is to define a fixed mapping, i.e each task is assigned a specific processing element and each instance will execute on it. For that we define one FIFO queue for each processing element and tasks are pushed in queues according to a mapping table (Fig-6.32).

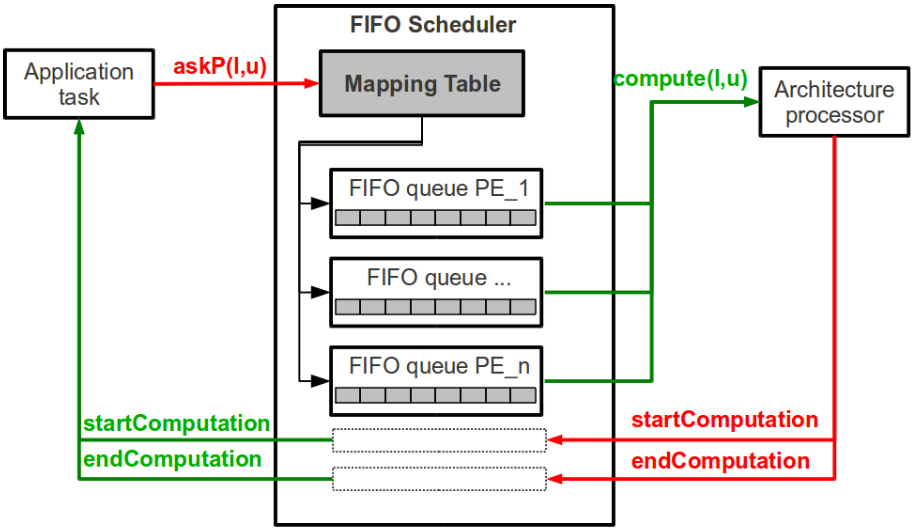


Figure 6.32: Local FIFO scheduler

Fixed Priority Scheduling

Fixed priority scheduling is an extension of the FIFO policy. Each computation task is associated with one PE and a fixed priority, several tasks can have the same priority so that they will be executed according to FIFO scheduling. FPS is implemented by using a priority table, i.e. there are a finite number of priority queues and each one is popped according to FIFO policy when all queues of higher priority are empty. Each time a computation task wants to acquire the PE it is put in its corresponding priority queue. As for FIFO schedulers, one can define FPS globally or locally.

Strict Priority Scheduling

It is a way to model *static scheduling* where the schedule is defined deterministically, i.e the task ordering is fixed and known before execution. Static schedule are given as a list of triples (τ_i, t_i, p_i) saying that task τ_i will start at time t_i on processing element p_i. That can be interpreted as priority assignments on tasks that ensure the same execution order.

Each computation task is associated with one PE and a strict priority and each task mapped on the same PE has a different priority. For each PE all associated tasks will be executed according to a strict order. Each time a computation task wants to acquire the PE it is put in a waiting list and has to wait until all higher priority tasks have been executed.

Frequency Scaling Scheduling

Each computation task is associated with one PE , a fixed/strict priority, a starting frequency/speed to which the PE will switch before the beginning of the computation. Optionally an ending frequency/speed can be specified enabling switching at the end of the computation. This can be used, for example, for halting a PE after some computation.

6.6 Evaluation

The aim of this modeling framework is to provide design space exploration for performance evaluation. This is achieved by composing all automaton (application, environment, architecture and system) to yield a global timed automaton which captures the semantics (all behaviors) of the system and this is the object of our various methods of analysis and simulation. For that we need a way to specify performance metrics that have to be measured. We focus here on property checking, that is how to ensure that some property is not violated. Quantitative evaluation will be done by statistical analysis on simulation traces and will be discussed in chapter 7.

Observers

For property verification with reachability analysis we use the concept of *observers* defined in the IF toolset that we have presented in chapter-3. Observers express in an operational way safety properties of a system by characterizing its acceptable executions sequences.

An example of property we are interested in is response time. We want to ensure that a job or a task always meets its deadline. For that we define an observer that monitors job/task behavior i.e it create a clock when the task becomes active and leads to an *error* state if the task does not finish until some specified deadline. The observer for this property is depicted in Fig-6.33. In case of violation we can retrieve a trace of the execution leading to this state. We present trace generation in more detail in chapter 7.

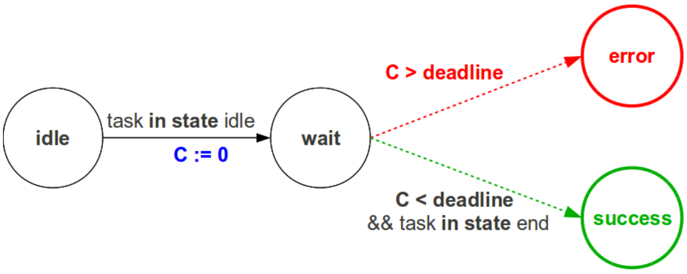


Figure 6.33: Observer for checking the deadline of a task

In this chapter we have shown examples of the basic components used in our modeling framework. They include computation and data transfer tasks encapsulated into jobs on the application side, input generators that model job arrivals, architecture components such as processors and busses as well as deployment (mapping and scheduling) policies. All those components are translated into a unified semantic models of timed automata whose composition provides the model for the various performance evaluation methods. As is also described in Chapter 7, one can add new models, or refine existing ones to be more detailed, at relatively small investment. The effect of model refinement on the feasibility of analysis depends and the method used and is, of course, more significant for methods based on verification (reachability computation) than for simulation-based techniques.

Chapter 7

Realization: The DESPEX Tool

In the context of this thesis we developed tool for performance evaluation at early design stage, called DESPEX (DEsign SPace EXplorer). According to [81], a formal model of a design should consist of the following components:

- A functional specification of the system;
- A set of properties that the design must satisfy;
- A set of performance indices that evaluate the quality of the design (cost, reliability, speed, size);
- A set of constraints on performance indices.

We will discuss in this chapter how the above components are implemented and how one can use the tool for rapid design space exploration.

7.1 General architecture

Evaluating performances of an application on a given platform starts with the specification of the whole system. We provide a description language dedicated to this purpose. Users have two alternatives to describe an abstract MPSoC model, by using either *textual* or *graphical* modeling language. Once the model is defined, automatic translation to timed automata is performed by generating IF code. The IF code is then compiled using the IF toolset and provides an executable model on which one can perform reachability analysis or simulation. The result of such analysis is a timed trace on which several processing can be done such as graphical visualization or statistical analysis. Simulation results can be saved by automatic report generation in various formats. We will discuss in the following how the different steps, shown on fig-7.1 are done.

We provide a command-line tool chain and a graphical user interface. The tool consists mainly of 25K lines of C++ code, developed with a concern for easy scalability, using established software design concepts.

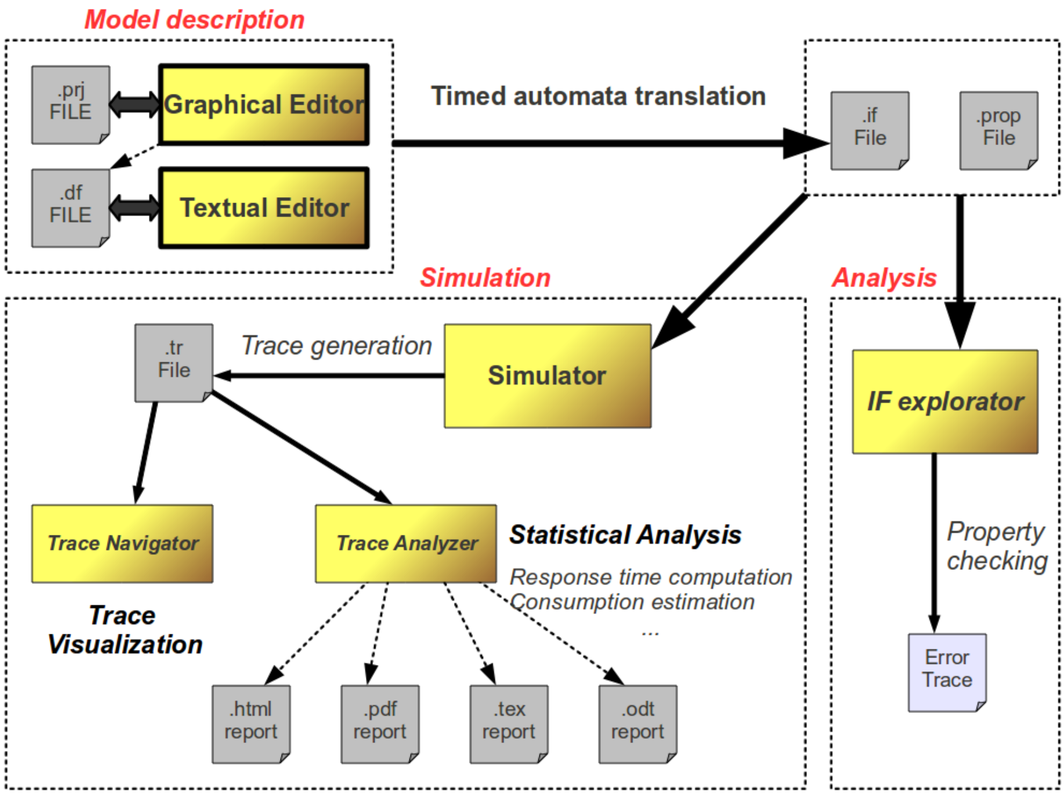
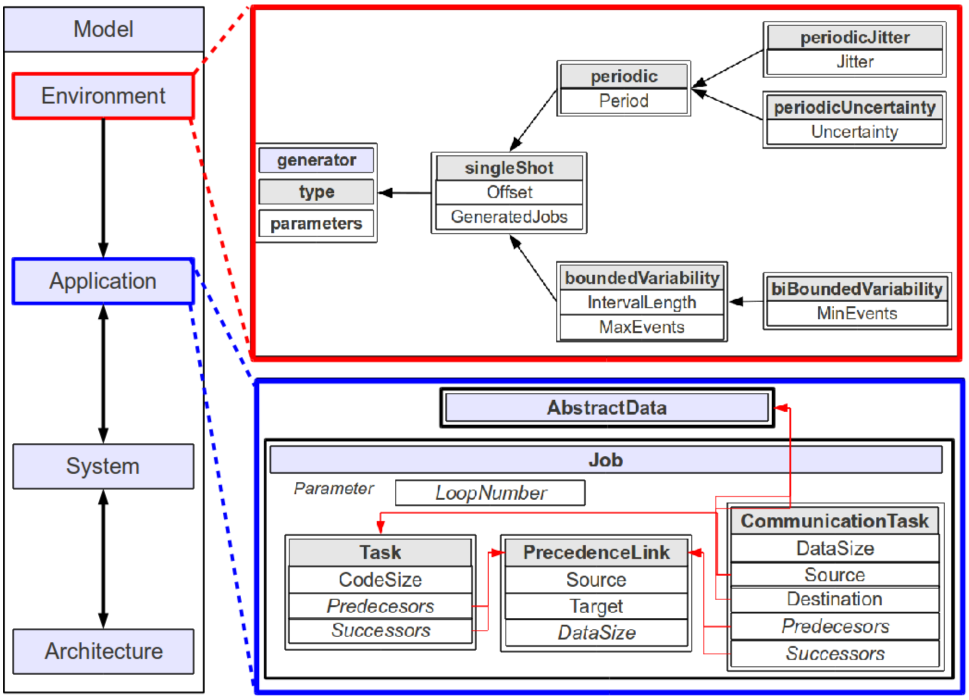


Figure 7.1: Tool architecture

7.2 Model description

Performance analysis is based on models which are generally defined using some specification language. We developed a *textual language* for modeling MPSoCs and applications at a high level of abstraction. An aim of this language is to remain as generic as possible while being easily expandable. As shown in chapter 6, our model is structured in four parts: application, environment, architecture and system. Each of their components is specified by a set of high level characteristics influencing timed automaton behavior. This can be viewed as hierarchical components defined with *typed* parameters. Fig-7.2 shows this hierarchy for application and environment components.

Figure 7.2: Component characteristics



- An application is specified by *jobs* where:
  - Tasks are characterized by an amount of work measured by instructions or cycles.
  - Communication tasks are characterized by a quantity of data to be transferred from a source to a destination
  - Precedence links models precedence constraints over entities and can be labeled with a quantity of data to specify data flow.
- The environment is specified using *input generators* characterized by particular parameters that define triggering scheme (period, jitter . . .) .
- The architecture is specified by a graph structure connecting the following components with specific parameters:
  - Processor: frequency, power consumption values.
  - Memory: R/W rate, access latency, power consumption values.
  - Bus: bandwith, access latency, power consumption values.
  - DMA: access latency, power consumption values.
- The system is defined by:
  - A list of local *schedulers* depending on the number of processors within the platform, or by a global scheduler. Each scheduler is typed according to its policy and is characterized by a mapping table, which associates tasks with processors.
  - a list of *data mappers* that associates *abstract data* with memory elements.

Concretely, the implementation of this model relies on what we call *abstract model*, allowing genericity and easy scalability of our modeling framework. The *abstract model* provides independently from the factual model, basic operations for parsing a textual description, edition through a graphical user interface (GUI) or exploration. We present, in the sequel, its structure more in detail.

The *abstract model* defines generic hierarchical objects on which basic operations can be applied. Basically, it is structured as a generic hierarchical graph pattern, on which textual description is built. We implement an *abstract component* as an object with generic parameters and possibly containing other components based on *composite design pattern*. Fig-7.3 shows the class hierarchy.

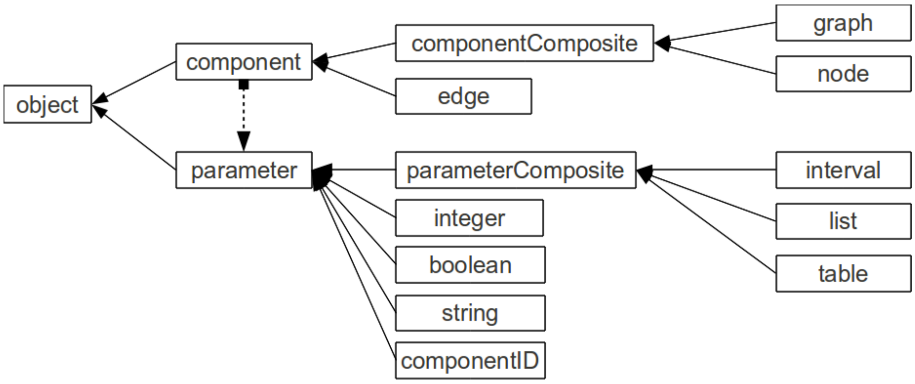


Figure 7.3: Class hierarchy

An abstract component is specified with a type and an identifier, implemented as string parameters, and may contain other components. Fig-7.4 shows the specific grammar associated to these abstract components, enabling description with a textual language.

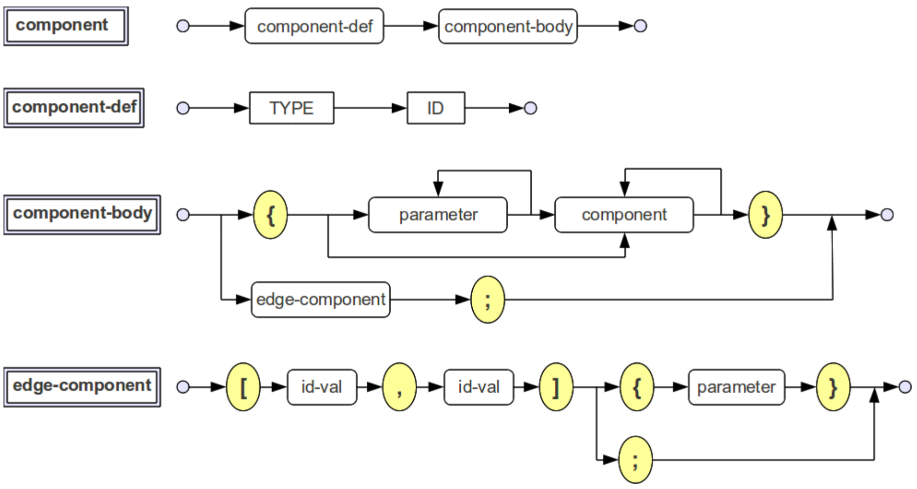


Figure 7.4: Abstract components grammar

Parameters are implemented in the same way, as simple types like integers, booleans or strings or as compound ones like lists or tables. For textual description, a grammar (Fig-7.5) is also associated with *parameters objects*.

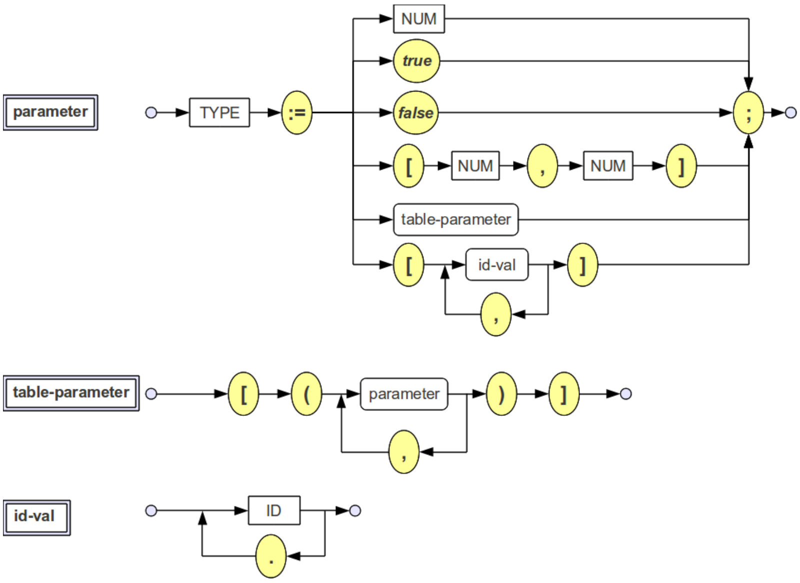


Figure 7.5: Abstract parameters grammar

This *abstract model* is the center of our software architecture, depicted on Fig-7.6. The *concrete model* which defines concrete instantiations is derived from it and most operations are done on this abstract object. This makes it possible to extend the modeling framework with low implementation effort.

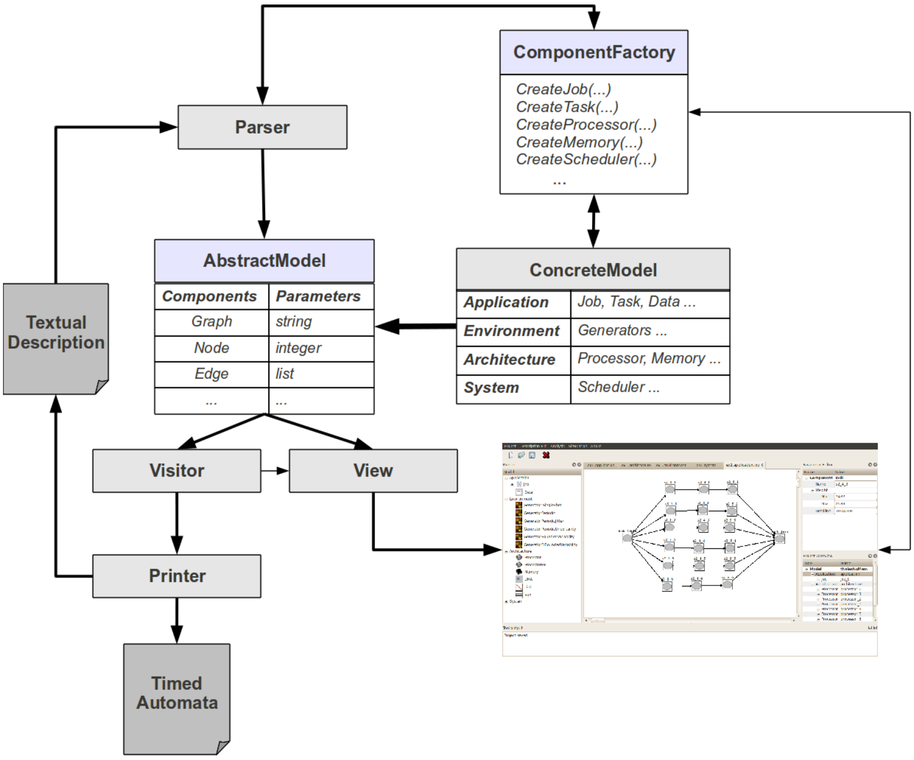
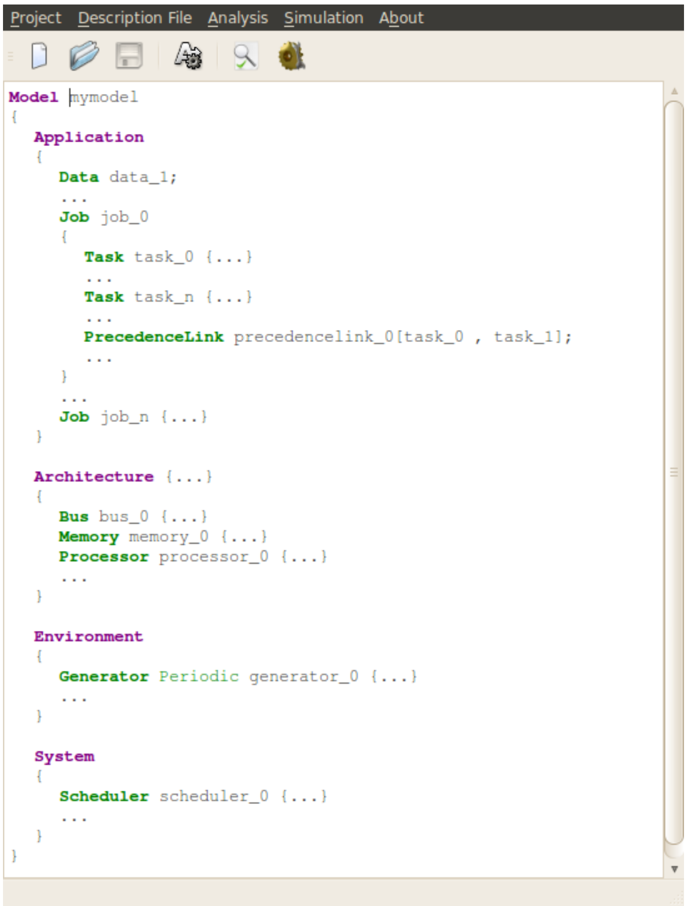


Figure 7.6: Model implementation

Basically, the implementation is based on Model-View-Controller (MVC) design pattern. Any textual description that follows the grammar defined above, can be read by the *parser*. Therefor, object assignment is managed by a *factory component*. Model exploration uses a *visitor design pattern*, that gives the ability to add new operations to existing structures without modifying those structures. Visitors are used for different processing like timed automata generation, textual printing of the model or for graphical viewing.

Graphical language intends to help users describe models in a graphical style, like Statecharts [104] or UML [167]. We provide both a textual language and a graphical language for model specification. An overview of the graphical interface editors is shown in Fig 7.7 and Fig 7.8. The graphical interface, written using Qt [41], works directly with the abstract model, that is, each abstract component is associated with a different graphical view which allows model edition. The reason for working on the abstract model is the same as for the textual language, it gives more genericity, that is, the GUI is not only dedicated to a specific language, but language extension does not imply much implementation work.

Graphical views depend on component or parameter type. A composite component is defined through a *scene editor* in which one can add some other components by pushing *graphic items*. Components parameters are viewed and editable through dedicated items depending on their type and are collected in a *treeview*. As for the textual language parser, a factory component is responsible for graphical objects creation.



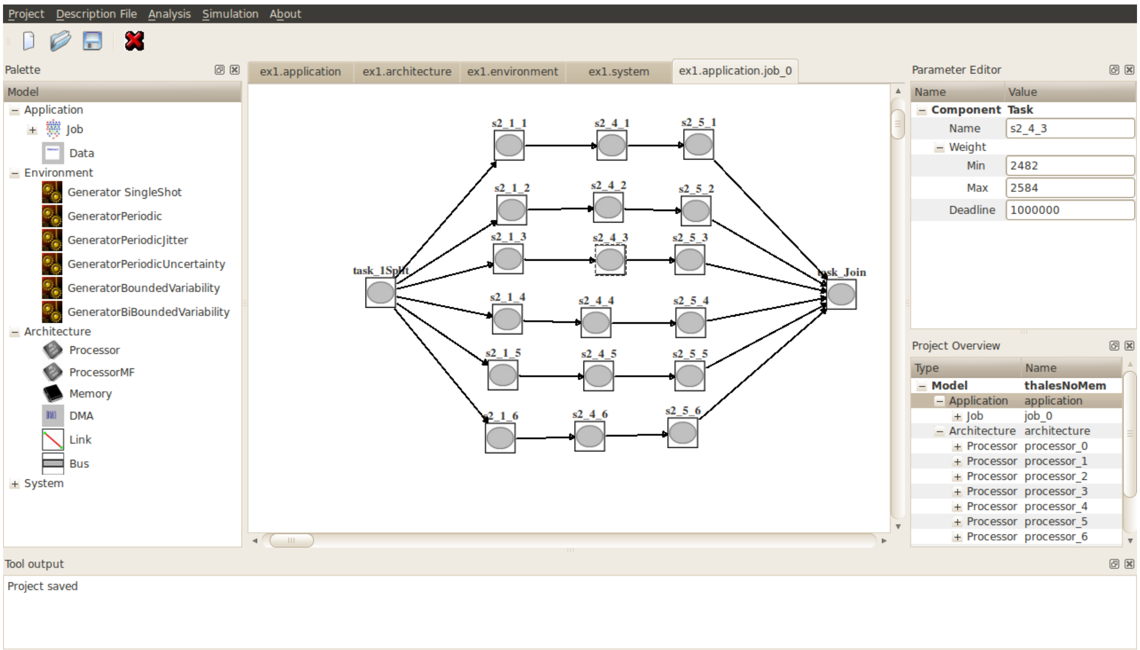
```
Model mymodel
{
  Application
  {
    Data data_1;
    ...
    Job job_0
    {
      Task task_0 {...}
      ...
      Task task_n {...}
      ...
      PrecedenceLink precedencelink_0[task_0 , task_1];
      ...
    }
    ...
    Job job_n {...}
  }

  Architecture {...}
  {
    Bus bus_0 {...}
    Memory memory_0 {...}
    Processor processor_0 {...}
    ...
  }

  Environment
  {
    Generator Periodic generator_0 {...}
    ...
  }

  System
  {
    Scheduler scheduler_0 {...}
    ...
  }
}
```

Figure 7.7: Textual editor.



Name	Value
Component	Task
Name	s2_4_3
Weight	
Min	2482
Max	2584
Deadline	1000000

Type	Name
Model	thalesNoMem
Application	application
Job	job_0
Architecture	architecture
Processor	processor_0
Processor	processor_1
Processor	processor_2
Processor	processor_3
Processor	processor_4
Processor	processor_5
Processor	processor_6

Project saved

Figure 7.8: Graphical editor.

Property Definition

The evaluation of a model is done by estimating performance indices. By specifying constraints on those indices, one can perform reachability analysis to check whether the model satisfies a set of properties. As explained previously, these properties are encoded as *timed automata observers* and can also be used during stochastic simulation. In that case, they are assimilated to monitoring processes, close to the work done in [155] using a temporal logic front-end. Timing behavior can be described by the time interval between a specified pair of events. A property on such interval is then defined by associating a *deadline* constraint. On the other hand, statistical estimation on these intervals, is provided by stochastic simulation and trace analysis.

We provide an XML-based language for properties specification, as constraints on time intervals between *start* and *end* events, in timed automata models. An example of definition for the response time of a job is shown in Listing-7.1 and the language grammar is shown on Fig-7.9

```
<property name=”ResponseTimejob” deadline=750>
  <start process=”{job}0”>
    <state name=”init” flag=”IN”></state>
  </start>
  <end process=”{job}0”>
    <event name=”kill” ></event>
  </end>
</property>
```

Listing 7.1: Property definition example

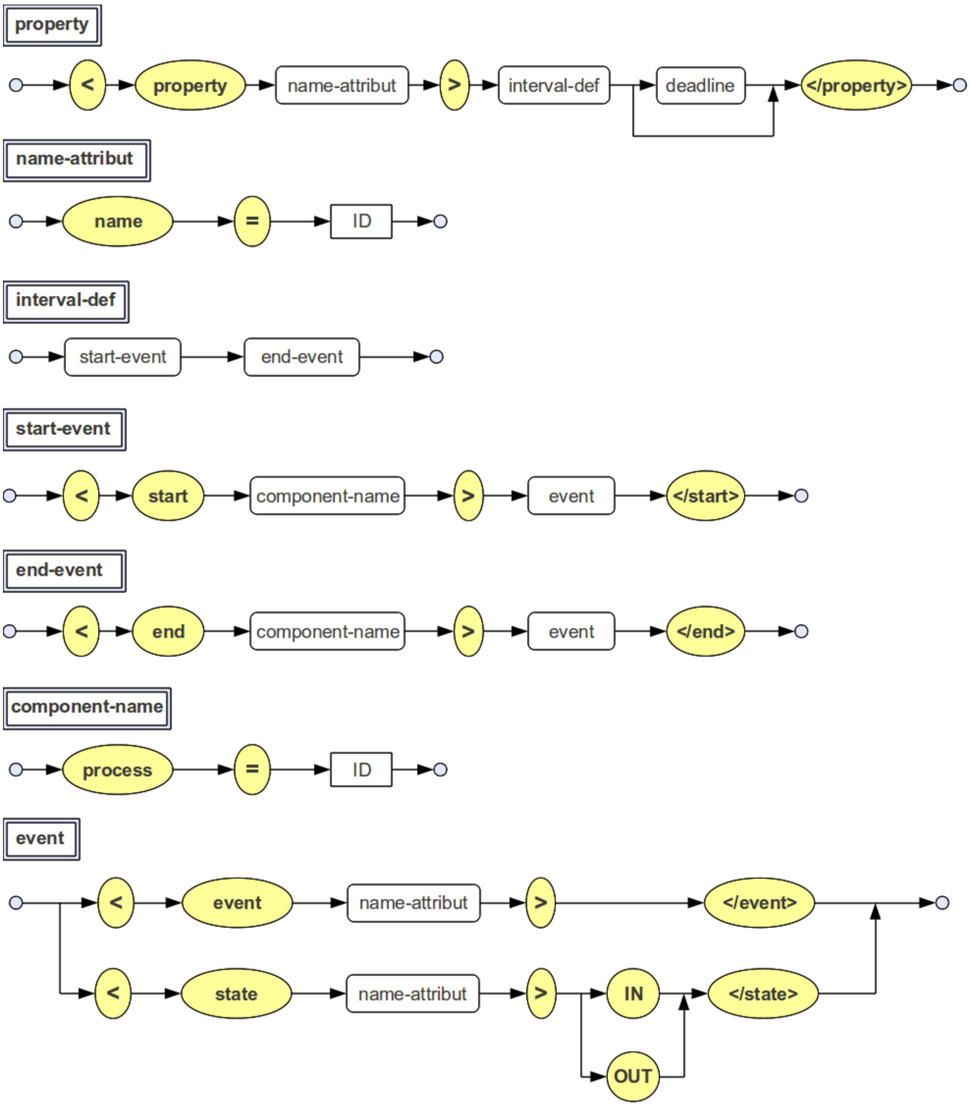


Figure 7.9: Properties language grammar

In the same way, one can define power consumption values for hardware components, by associating particular measure with states as shown in Listing-7.2 and Fig-7.10.

```
//consumption parameters for a memory component
<consumption process=”locMem”>
  <state name=”idle”> 61 </state>
  <state name=”wait”> 61 </state>
  <state name=”read”> 91 </state>
  <state name=”write”> 91 </state>
</consumption>
//consumption parameters for a processor component
<consumption process=”pe0”>
<state name=”idle”> 72 </state>
<state name=”busy”> 362 </state>
</consumption>
```

Listing 7.2: Power consumption definition example

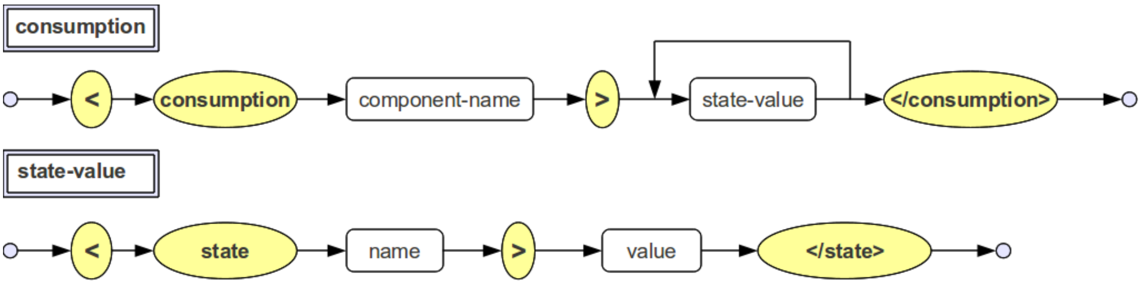


Figure 7.10: Power consumption specification grammar

Model Extension

The model can be easily extended with minimal programming effort. Adding new parameters is the simplest operation, consisting simply in adding them in a concrete object and will automatically be propagated in graphical views and in the textual language without any reprogramming. Adding new components consists in defining a new object deriving from any abstract component and some coding is required in the various *component factory*. Finally, a timed automaton for the new component has to be provided in the IF printer.

7.3 Translation to Timed Automata

Once specification is done, we provide automatic translation to timed automata by generating IF code. Each component is associated with a timed automaton communicating via signals. One can see the overall architecture as a library of timed automata and extending or adding new component in the model implies defining its corresponding automaton as an IF process properly managing input and output signals.

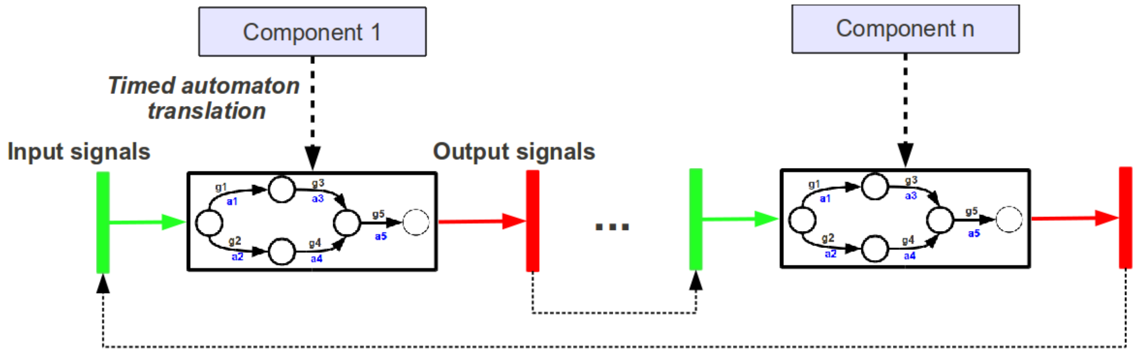


Figure 7.11: Timed automata translation

Properties encoded as *timed automata observers* are generated at the same time. The whole *timed automata library* is then compiled into an executable model provided by the IF toolset (Fig-7.12). This allows reachability analysis or stochastic simulation.

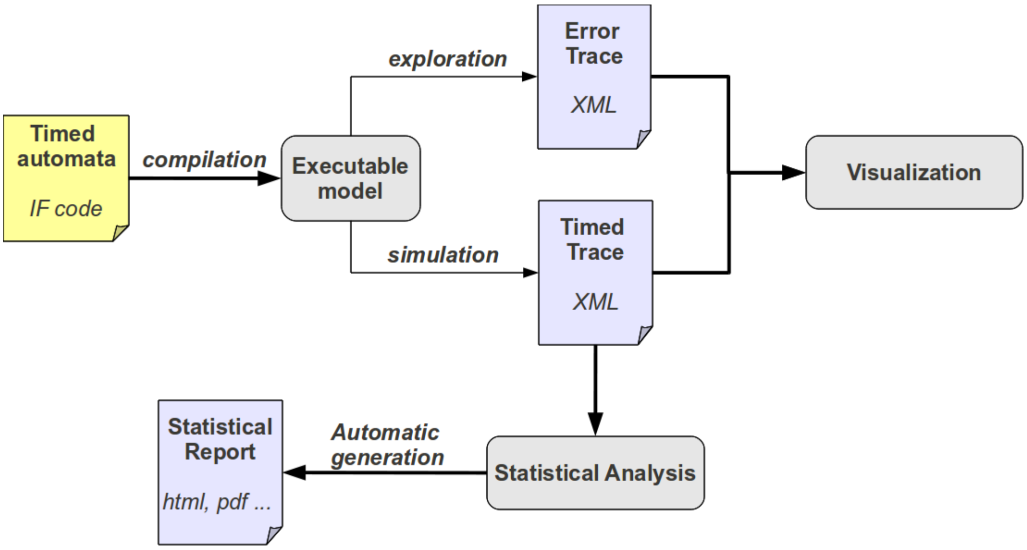


Figure 7.12: Simulation flow

Command Line Usage

Parsing and generation of an executable simulator is done by invoking the following command:

```
DFtoSimulator -i <DFfile> -o <simulator> [option]

 -i <DFfile>:     model specification in the description language
 -o <simulator>:  specify name of generated executable simulator

 [option]:
  -a :          if specified generate simulator for reachability analysis
                otherwise simulator is compiled for stochastic simulation
  -p <file> :   properties file input
```

This generates an executable *simulator.x*, if option *-a* is not specified, *simulator.x* is compiled with all needed library for stochastic simulation including random number generation utility.

7.3.1 Reachability Analysis

The goal of reachability analysis is to check whether some properties are satisfied or not. For a given model and a property we construct an *observer*, an automaton that monitors the global behavior and reports an error on property violation. The verification process consists of exploring the state space of the product system. In case of property violation, one get an *error trace* which can be visualized with our tool to give designers a better understanding on components behaviors that leads to an undesirable state.

In the same spirit as *observers* we use *observer variables* to store some specific values, like worst case response time of a job, during the exploration process. These variables are simply defined as static global variables and are not part of the state variables, i.e they do not influence system dynamics. For example, to get the worst case response time of a job, we can use an observer (fig-7.13). Variable WRT is an *observer variable*, each time the job ends we store the maximal response time value. In the same manner we can retrieve best cases values.

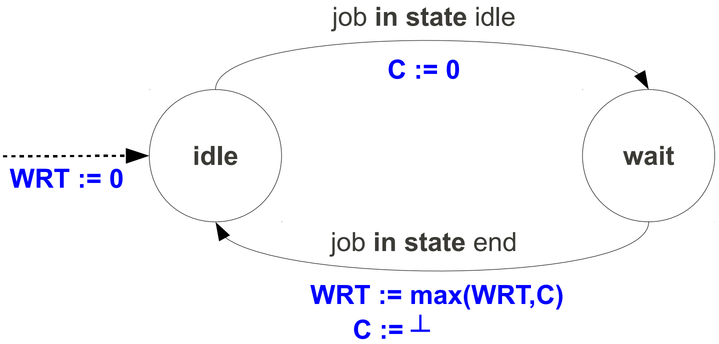


Figure 7.13: Observer for retrieving worst response time of a job

Command Line Usage

Running reachability analysis is done by:

```
simulator.x [traversal] [options] [-q file] [-t file]
  -q file      : save all explored state in file
  -t file      : save all explored transition in file

  [traversal] must be one of:
    -bfs       : exhasutive breadth first search
    -dfs       : exhaustive depth first search
    -inter     : interactive exploration

  [options] can be any of:
    -se        : stop on error states (bfs,dfs traversal only)
    -ce        : cut on error states (bfs,dfs traversal only)
    -te        : trace paths to error states (dfs traversal only)
    -po        : partial order reduction
```

7.3.2 Stochastic Simulation

Depending on the size of the model it may be difficult or even impossible to perform reachability analysis due to state space explosion. We provide stochastic simulation as an alternative on semantically equivalent models. Timed automata can be used to perform discrete event simulation by using a randomized reachability exploration. One point which is not defined in timed automata formalism is how timing uncertainty are distributed. The stochastic model can be viewed as a refinement of the TA model. It has the same set of behaviors but with a probability measure defined on the space of behaviors which is non-zero exactly on the feasible ones.

From Timed Automata to Duration Probabilistic Automata

We restrict ourself to bounded uncertainty and we explain in this section how we integrate probabilistic information into timed automata model. The resulting model of duration probabilistic automata (DPA) is investigated theoretically in Chapter 4 and 5 while here we focus on using it for simulation based statistical analysis.

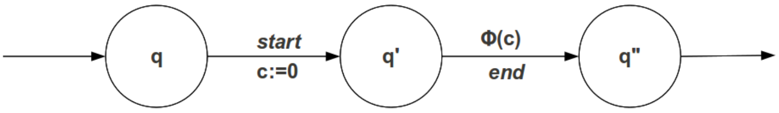


Figure 7.14: Standard description

The idea, here, is to define a random variable for each uncertainty interval, and to replace every guard which is in the form of interval by drawing from the random variable. To illustrate this, consider the automaton of Fig-7.14. Clock c, which is set to zero upon the *start* transition, measures the time elapsed since the activation. This *start* transition is instantaneous and can be initiated by some external supervisor. The timing of leaving state q' is based on the clock value and the temporal guard $\phi(c)$, which is simply the condition $c \in [l, u]$. In the stochastic model, we associate a probability density with the duration of such step, which is technically expressed as the distribution over the values of clock c when we leave state q'. We use a slightly modified (but equivalent) version of the basic automaton, as shown at Fig-7.15. Rather than having the *start* transition deterministic and delegating the non-determinism to the *end* transition, we use an auxiliary variable y which is assigned non-deterministically upon *start* and which should be equal to c upon *end*. In the set-theoretic setting this means an assignment $y \in [l, u]$ while for stochastic model this means drawing a value for y according to distribution ϕ, which we denote by $y := \phi()$.

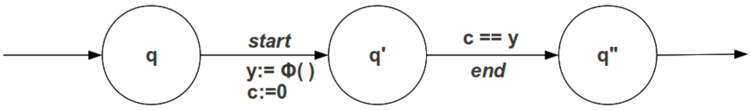


Figure 7.15: Non determinism is decoupled through random variable y

Models remain the same as those used in reachability analysis except for timing transitions that are translated into stochastic ones. Random variables associated with intervals are uniformly distributed in our model. This choice was made because it is the simplest one, and at the level of abstraction on which we work, it may be difficult for designers to estimate precise characteristics. Based on past experiences it seems quite easy to give lower and upper bounds, but without further information on how it may be distributed between them, uniform distribution could be considered as a good assumption. In the case where more information is available one can define other bounded-support distributions. For example, finding precise values for number of cycles in a task can be a difficult work at early design stage. One can estimate lower and upper bounds or if it is possible one can discretize the distribution by splitting interval into smaller one as depicted on Fig-7.16.

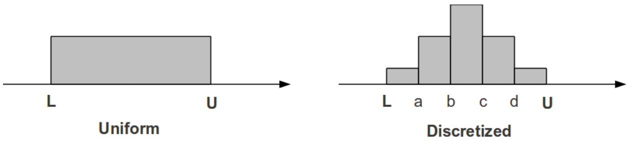


Figure 7.16: Uniform versus discretized distribution

Concretely, we extend the IF toolset with random number generators that performs correctly according to given probability distribution. Since we want to be able to compare, by statistical simulation the performance of different deployments of the same application with the same distributions on duration, we should be careful to ensure experimental repeatability and sample the random variables in a way which is independent of the order of execution of different tasks. Because of the use of a random number generator, this will not happen if we draw the duration just before we start the task. Instead, we draw from all the duration random variables at initialization time, and repeat the same values when we compare deployments.

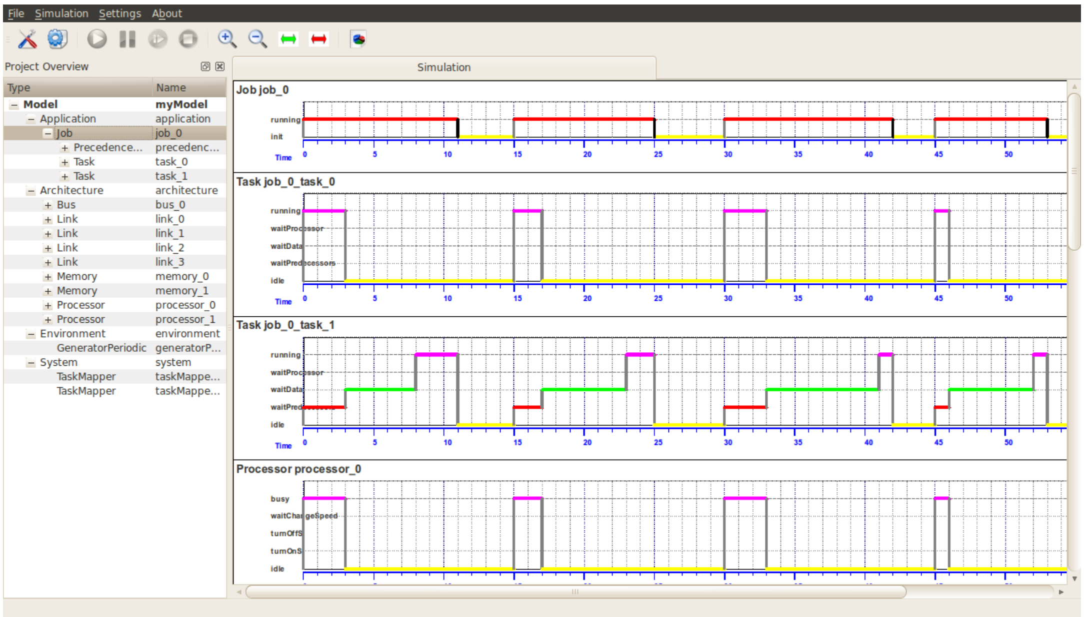
Command Line Usage

Running a stochastic simulation is done by:

```
simulator.x -simDuration <num> -savesim <traceFile>
 -simDuration <num>:   simulation length where <num> is in tick unit
 -savesim <traceFile>: generated trace file
```

Graphical Interface Usage

Simulation and reachability analysis can be done through the GUI. One can do interactive or step by step simulation and visualize timed traces on the fly or offline for error traces resulting from reachability analysis. Fig-7.17 gives an overview of the tool capabilities.



Type	Name
− **Model**	**myModel**
− Application	application
− Job	job_0
+ Precedence...	precedenc...
+ Task	task_0
+ Task	task_1
− Architecture	architecture
+ Bus	bus_0
+ Link	link_0
+ Link	link_1
+ Link	link_2
+ Link	link_3
+ Memory	memory_0
+ Memory	memory_1
+ Processor	processor_0
+ Processor	processor_1
− Environment	environment
GeneratorPeriodic	generatorP...
− System	system
TaskMapper	taskMappe...
TaskMapper	taskMappe...

Figure 7.17: GUI for simulation and reachability analysis

7.4 Trace Analysis

Timed traces result from stochastic simulation or from property violation during reachability analysis. To give designers a better understanding on components behaviors that leads to an undesirable state after reachability analysis, we provide trace visualization with a graphical viewer called *TraceNavigator*. An example of trace view is shown on Fig-7.18.

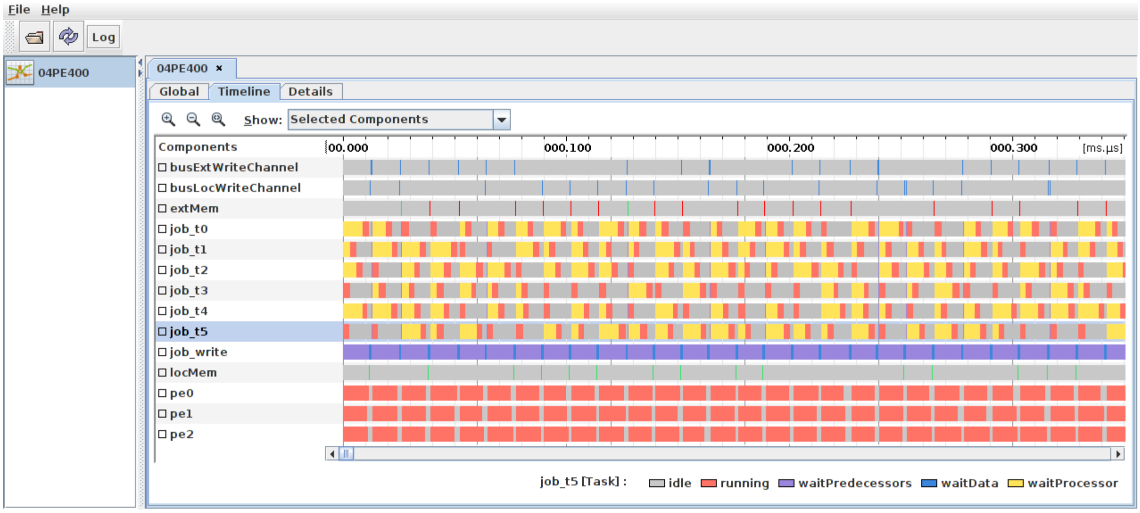


Figure 7.18: Simulation trace visualization with TraceNavigator

Stochastic simulation produces timed traces in XML format, on which statistical evaluation is done off-line using *TraceAnalyzer*. It takes as input a *properties file* and a simulation trace. The trace is then parsed and annotated with the list of values for each property. Statistics are then calculated and graphics are generated by using *scipy* [114] and *matplotlib* [109] Python library. Statistical reports are automatically generated as rst [95] files allowing easy formatting in various ways (html, pdf, odt . . .).

Timing Evaluation

To properly design an embedded system it is important to understand the effect that scheduling has on system performance. Scheduling analysis calculates worst-case and best case task response times, i.e the time between task activation and task completion. Worst case response time can then be compared against deadlines. We have seen that computing worst case response time or checking deadlines can be achieved with reachability analysis. In addition to the best-case, worst-case bounds, it can also be very helpful to know how response time is distributed between bounds. This is achieved by statistical analysis of traces from simulation. Typical statistical measures are:

- State distribution for components (ex: processor or bus utilization)
- Response time distribution, mean, maximum and minimum values.
- Latency (bus access, memory R/W . . .)

These measures are done according to *properties* specification, defined as timed intervals. For each interval, we get general statistical information like maximal, minimum and mean values together with probability distribution as shown on Fig-7.19.

Power Consumption Estimation

Recall that power information transforms the timed automaton into a linearly-priced timed automaton [24, 128] whose analysis is implemented in UPPAAL [23] but not in IF. For this reason (and also due to scalability) we avoid reachability computation and focus on simulation. Energy consumption estimation is done via trace analysis. Each state of an architecture component is associated with a power value. So we can derive from a simulation trace the energy consumption

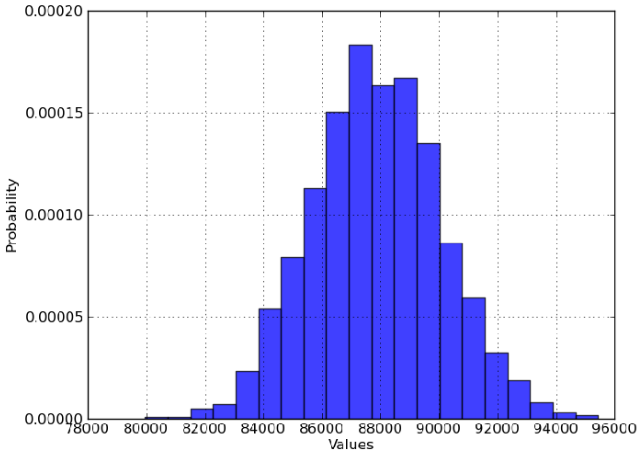


Figure 7.19: Example of response time distribution

for each component. Let δ_{s_i} be the time duration between entering and leaving state s_i and P_{s_i} be the power value associated with state s_i. The energy consumption for a component C_j is then

$$E_{C_j} = \sum_{s_i} \delta_{s_i} \cdot P_{s_i}$$

and the *mean power* for component C_j is

$$P_{C_j} = \frac{E_{C_j}}{\delta_{sim}}$$

with δ_{sim} is the simulation duration.
The energy consumption E_S and mean power P_S for the whole system are given by:

$$E_S = \sum_{C_j} E_{C_j}$$

$$P_S = \sum_{C_j} P_{C_j}$$

Complementary we can estimate maximal power as

$$max_t(\sum_{s_k} P_{s_k})$$

with s_k all enabled state at time t.

As an example Fig-7.20 shows energy consumption proportion between hardware components and power of the whole system during time on Fig-7.21.

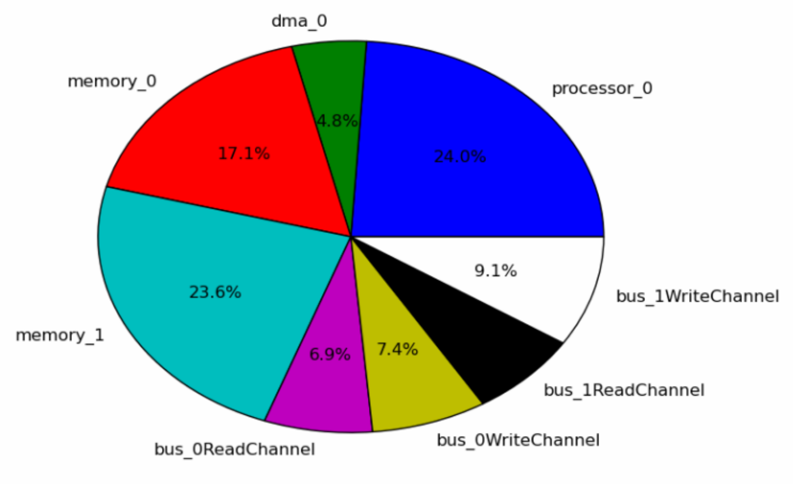


Figure 7.20: Energy proportion

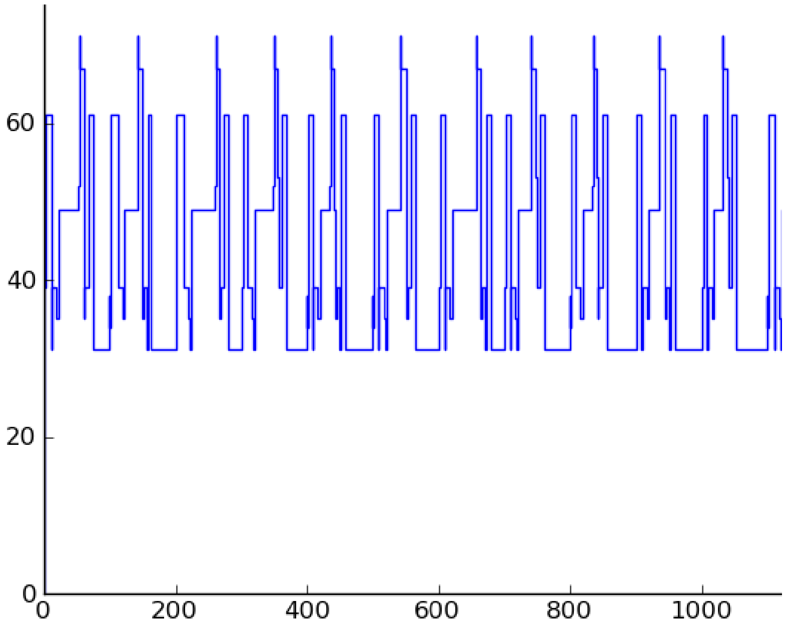


Figure 7.21: Power

Command Line Usage

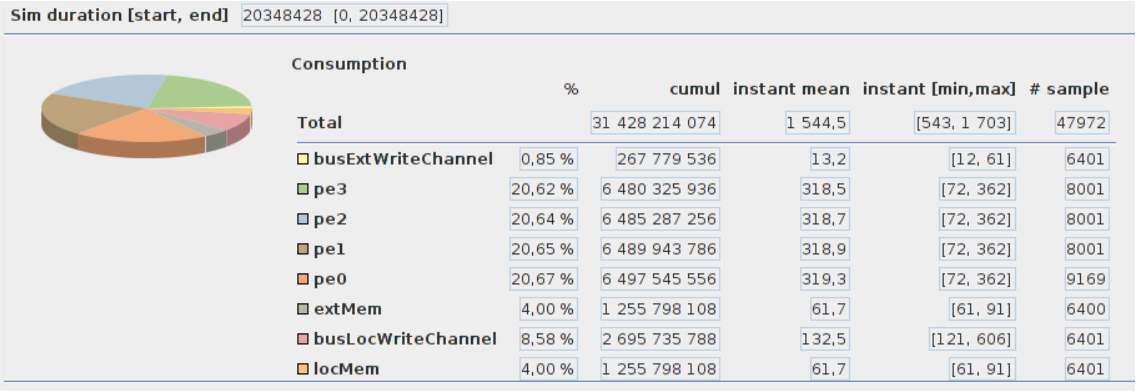
```
TraceAnalyzer -i <traceFile> -p <propertyFile> -o <report> [options]

 -i <traceFile>:   simulation trace file
 -p <propertyFile>:  property definition file
 -o <report>:   output report name

Options:
 -f [html|tex|pdf|odt]  : specify output format
```

Statistical Evaluation with TraceNavigator

The first functionality of *TraceNavigator* is to provide trace visualization as shown on Fig-7.18. In addition, it gives a view of quantitative estimation for a given set of metrics like power consumption of different hardware components (Fig-7.22), response time of software components (Fig-7.24) or states distribution (Fig-7.23).

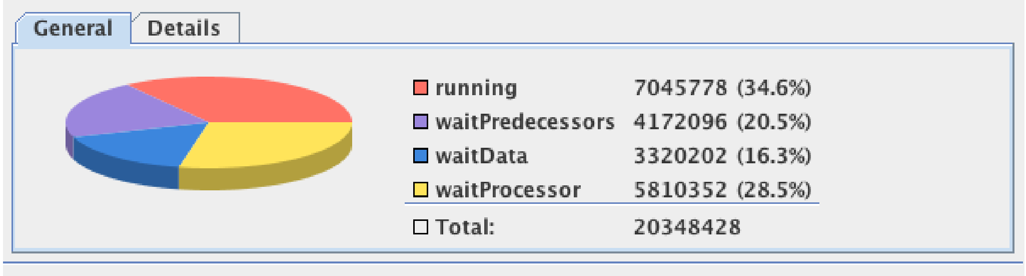


Sim duration [start, end] 20348428 [0, 20348428]

Consumption

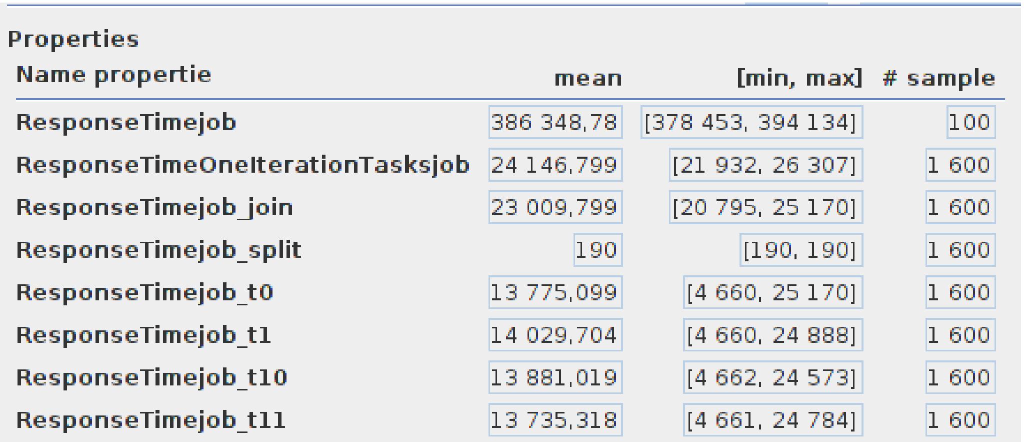
	%	cumul	instant mean	instant [min,max]	# sample
Total		31 428 214 074	1 544,5	[543, 1 703]	47972
🟨 busExtWriteChannel	0,85 %	267 779 536	13,2	[12, 61]	6401
🟩 pe3	20,62 %	6 480 325 936	318,5	[72, 362]	8001
□ pe2	20,64 %	6 485 287 256	318,7	[72, 362]	8001
■ pe1	20,65 %	6 489 943 786	318,9	[72, 362]	8001
🟧 pe0	20,67 %	6 497 545 556	319,3	[72, 362]	9169
■ extMem	4,00 %	1 255 798 108	61,7	[61, 91]	6400
🟥 busLocWriteChannel	8,58 %	2 695 735 788	132,5	[121, 606]	6401
🟧 locMem	4,00 %	1 255 798 108	61,7	[61, 91]	6401

Figure 7.22: Consumption overview with TraceNavigator



🟥 running	7045778 (34.6%)
🟪 waitPredecessors	4172096 (20.5%)
🟦 waitData	3320202 (16.3%)
🟨 waitProcessor	5810352 (28.5%)
□ Total:	20348428

Figure 7.23: States distribution for a task component



Properties

Name propertie	mean	[min, max]	# sample
ResponseTimejob	386 348,78	[378 453, 394 134]	100
ResponseTimeOneIterationTasksjob	24 146,799	[21 932, 26 307]	1 600
ResponseTimejob_join	23 009,799	[20 795, 25 170]	1 600
ResponseTimejob_split	190	[190, 190]	1 600
ResponseTimejob_t0	13 775,099	[4 660, 25 170]	1 600
ResponseTimejob_t1	14 029,704	[4 660, 24 888]	1 600
ResponseTimejob_t10	13 881,019	[4 662, 24 573]	1 600
ResponseTimejob_t11	13 735,318	[4 661, 24 784]	1 600

Figure 7.24: Response time overview

Chapter 8

Case Studies

In this chapter we demonstrate the application of the tool to several examples. We show first on a synthetic example how temporal uncertainty influences performances analysis. We then present two case studies: an evaluation of various alternatives for an image processing application and the deployment of a radio sensing application on an embedded platform.

8.1 Reachability vs. Corner-Case Simulation

This section will demonstrate on a simple example that an analysis based exclusively on worst case execution times might not catch the worst case behavior.

8.1.1 Model Description

The application consists of the task graph of Figure-8.1 and the architecture consists of two processors of different types P_1 and P_2 as shown in Figure-8.2. The tasks are partitioned into two types with A tasks running on P_1 and B tasks on P_2. The lower and upper-bound on the task durations are shown in Table-8.1 (for simplicity we normalize processors speeds to 1). We assume one instance of this job produced by a single-shot generator at time 0. We assume a FIFO scheduler for each processor which executes, *without preemption*, the appropriate tasks in the order they become enabled.

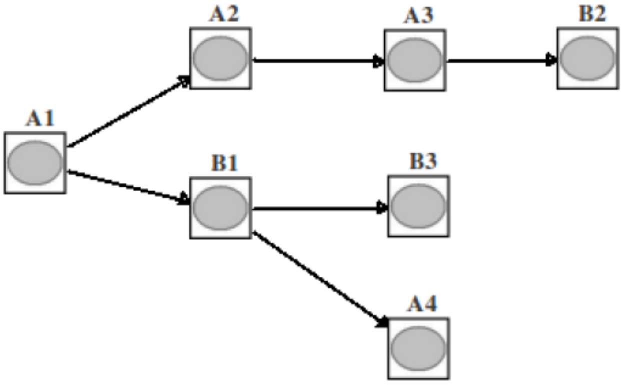


Figure 8.1: Task graph example

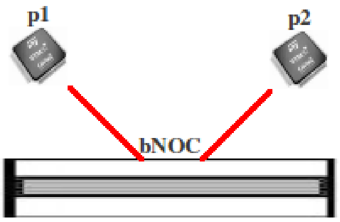


Figure 8.2: Simplified Heterogeneous Architecture

Task	Workload min	Workload max
A1	2	3
A2	3	4
A3	4	5
A4	1	7
B1	2	6
B2	3	4
B3	4	6

Table 8.1: Task workload

8.1.2 Analysis

For that example, the property to verify is that the response time of the task-graph does not exceed 20. We first do analysis based on deterministic values (lower and upper) and then show some results using non-deterministic workloads for computation tasks.

8.1.2.1 Worst Case Analysis

We take the application model with a deterministic workload for each task. It is *common* to take the worst case execution time for each task in Table-8.1, in order to get a worst scenario for the whole execution.

The analysis gives a response time of 19 time units which is less than the deadline (20). The execution trace is depicted in Fig-8.3. Colors are associated with component states, executing in red, waiting for predecessors in purple and waiting for the processor in yellow. Based on this result, one might conclude naively that the timing requirement is satisfied because the worst case response time is under the deadline.

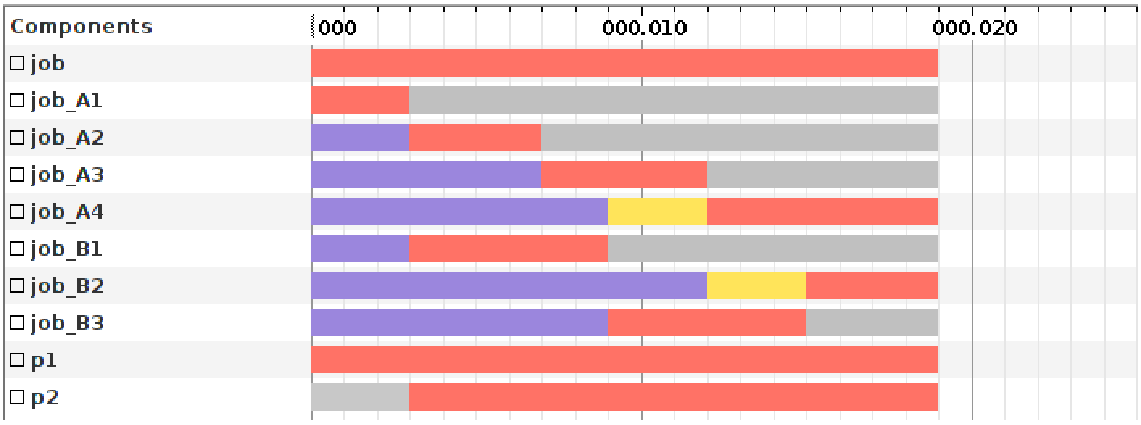


Figure 8.3: Execution trace for worst-case duration

8.1.2.2 Best Case Analysis

Likewise we could take the best case for each task from table-8.1 in order to get an estimation of the best scenario. The analysis gives a response time for the task graph of 13 time units. The execution trace is shown on Fig-8.4. Based on worst case and best case analysis one might therefore conclude that response time will be in [13, 19] and so the timing requirement is satisfied.

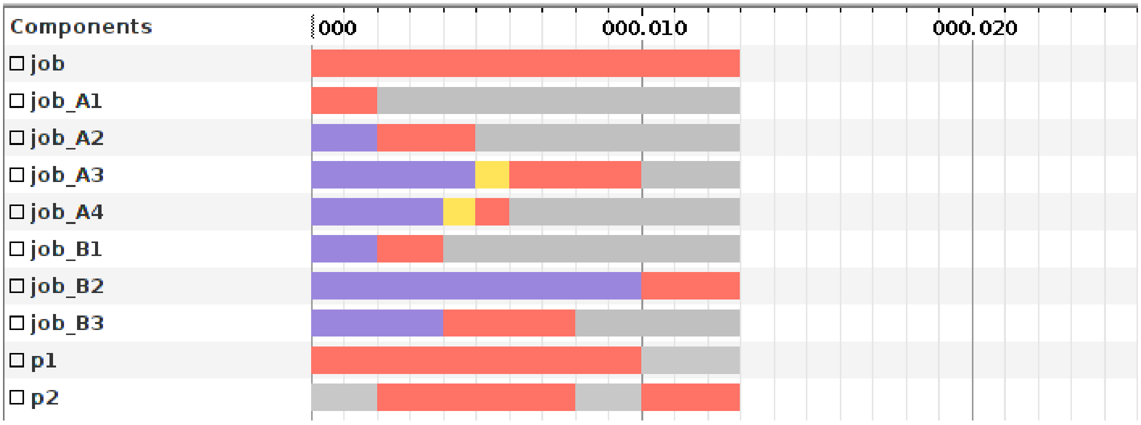


Figure 8.4: Execution trace for best-case duration

8.1.2.3 Reachability Analysis with Uncertainty

Now we take the original model for the application i.e tasks workload are defined as the *intervals* in Table-8.1 and do reachability analysis on the underlying timed automata model. The analysis detects scenarios with response time of 23 (Fig-8.5) and 12 (Fig-8.6).

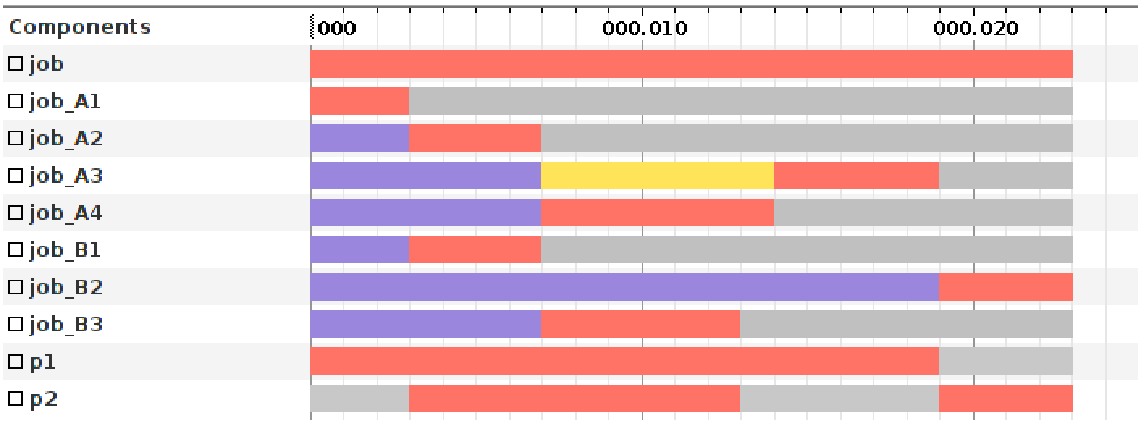


Figure 8.5: The execution trace which gives the worst response time

We see that lower and upper bounds on task durations do not allow us to infer the correct bounds on the response time for this particular example. If this application is executed in a safety critical environment, with *hard real time constraints*, conclusion of the analysis is that the deadline can be violated. However, in case of soft real time constraints, QoS (Quality of Service) evaluation in a quantitative way is more appropriate.

8.1.2.4 Quantitative Estimation

In addition to bounds on execution time, it can also be very helpful to know how the response time is distributed. To get more quantitative informations we use stochastic simulation in order to answer questions like: what is the mean response time? What is the probability that the response time is greater than 20 time units?

For the purpose of this quantitative estimation we make the hypothesis that the execution times are uniformly distributed in the intervals. We draw 10000 duration vectors randomly and perform

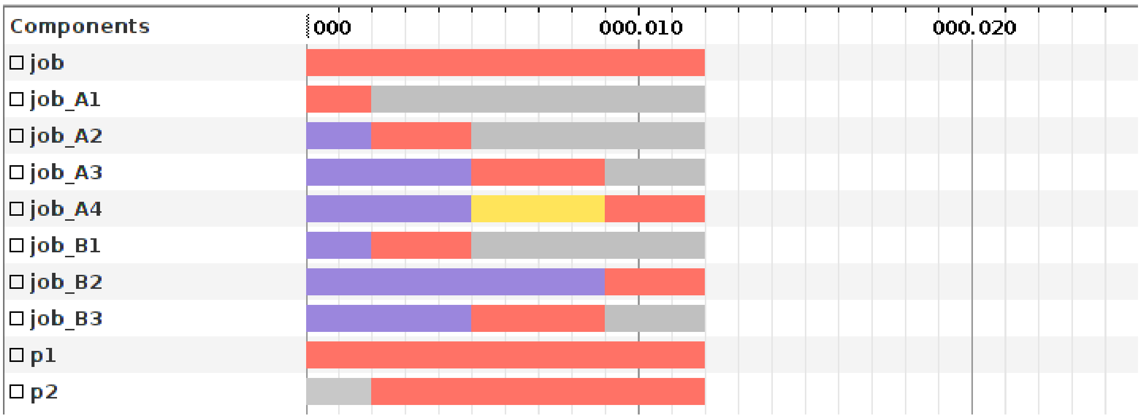


Figure 8.6: The execution trace which gives the best response time

simulation with each value. The results are summarized in Table-8.2 and Fig-8.7.

Min value	12.97
Max value	22.17
Mean value	16.82

Table 8.2: Statistics

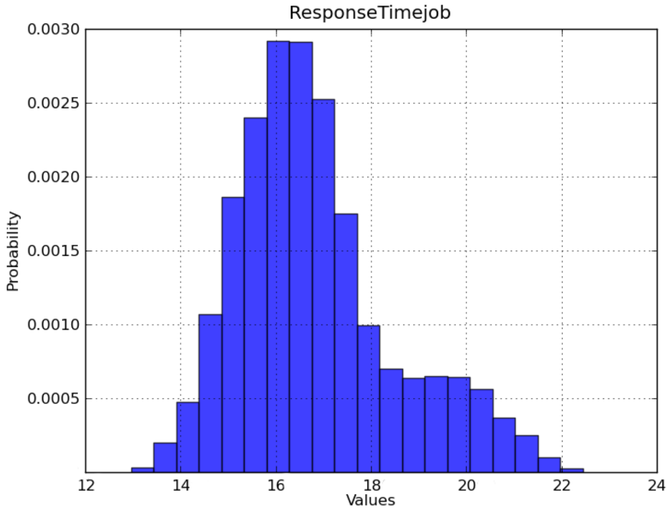


Figure 8.7: Distribution

8.1.3 Summary

With this simple example we have shown that analysis based on local extremal values (lower and upper) might give incorrect bounds on the global response time. On the other hand timed automata reachability analysis gives us the correct bounds, but no quantitative information about the distribution of values between them. In the case where timing constraints are not critical such pessimistic estimation can lead to over-designed and under-utilized systems, thus resulting in unnecessarily increased costs. Moreover, stochastic simulation does not catch tight bounds but gives more useful quantitative information about average performance.

8.2 Video Processing on P2012

Platform 2012 (P2012, [177]) is an ongoing project of ST Microelectronics (the largest European semi-conductor manufacturer) and CEA-LETI to develop a multi-core architecture to serve as an accelerator (computation fabric) for high throughput computational tasks (video processing, radio sensing, image analysis) for embedded (smart phones) and other (TV set top boxes) devices. P2012 is viewed as an alternative to GPUs as a replacement of dedicated hardware currently used for these functions. The flexibility and productivity gains of software are supposed to compensate for a tolerable degradation in performance compared to hardware. However, writing parallel software is not a trivial matter and deploying it efficiently on the multi-core platform (mapping, memory allocation, scheduling of computations and data transfers) is a hard combinatorial optimization problem with a significant variations in performance over its feasible solutions.

STMicrolectronics has defined a benchmark based on typical applications in the context of P2012. It is an augmented reality application called FAST (Features from Accelerated Segment Test). This application was developed and parallelized on P2012 virtual architecture and a sequential version (set as reference and easier to analyze) was given to us. In this section we demonstrate the applicability of our tool in exploring and comparing different deployment solutions for this application.

We will use two variants of the application and of the P2012 architecture to demonstrate the functionality of our tool. All these experiments should be taken with a grain of salt concerning their realism since the development of P2012 and its applications was in a stage where models were very approximate. The main purpose of the exploration is to illustrate the types of analysis provided by our framework.

8.2.1 Model description

The P2012 Platform

Merely, P2012 is a many-core computing fabric based on multiple clusters. Clusters feature up to 16 processors sharing multi-banked memories, a DMA engine and are connected via a high-performance network-on-chip. In the context of this case study we restrict our models to one such cluster.

The model of P2012 architecture used in this section is shown on Fig-8.8. It consists of 16 processing elements sharing a local memory and connected through a bus which models the actual network-on-chip. The entire *cluster* is connected to an external memory via another bus and a DMA engine is used to communicate data between local and off-chip memories.

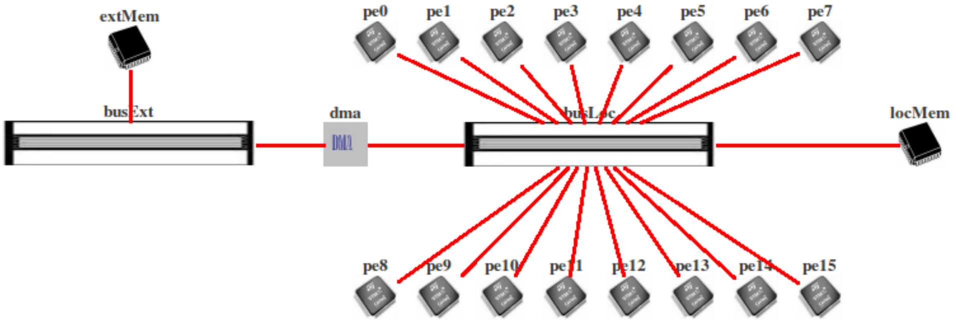


Figure 8.8: A model of a 16-processor instance of P2012

We associate with each component a list of synthetic parameters for power consumption shown in Table-8.3. The realism of these parameters is limited due to the difficulty to get real values from a virtual platform.

Component	Frequency	State	Consumption
Processor	200Hz	idle	42
		busy	212
	400Hz	idle	72
		busy	362
	600Hz	idle	121
		busy	606
Memory		idle	61
		busy	91
LogInterconnect		idle	12
		busy	61
BusExt		idle	121
		busy	606

Table 8.3: Power consumption values for HW components

In the context of this case study we explore different alternatives by playing with the number of available processors and their frequency parameters to have a picture of their influence on performance.

FAST Algorithm Presentation

The FAST algorithm, developped by Edward Rosten et Tom Drummond [166], is a corner detection method, used to extract feature points and later used to track and map objects in many computer vision tasks. It mainly consists in computing the detection (function *circular detection*) on a chunk of an image. From an architectural point of view, the image resides initially in the off-chip memory and has to be brought to local memory and dispatched to the processors for execution.

The whole image does not fit into the local memory inducing several alternatives for its splitting and transfer to local memories. For our case study, we consider two different implementations one based on bands and one based on blocks.

Band Treatment

This approach consists in dividing the image into bands and bringing each band to local memory, then available processors work on different parts of this band. The number of parts will depend on the number of processors available in the target platform. Fig-8.9 shows this splitting with 8 available processors. After computation is completed for the entire band, a synchronization is done between all working processors and the result for the whole band is stored on off-chip memory.

We model this algorithm with our tool as a task graph (Fig-8.10). The entire image is modeled as an *abstract data* and nodes *read* and *write* are *communication tasks* for accessing the data. Task *split* and *join* model the synchronization process. This task graph represents the computation of one band. Notice that the task graph is cyclic, that is it executes a *finite* number of iteration, corresponding to the number of bands, for the computation of the entire image. In this case study the size of a band is kept fixed.

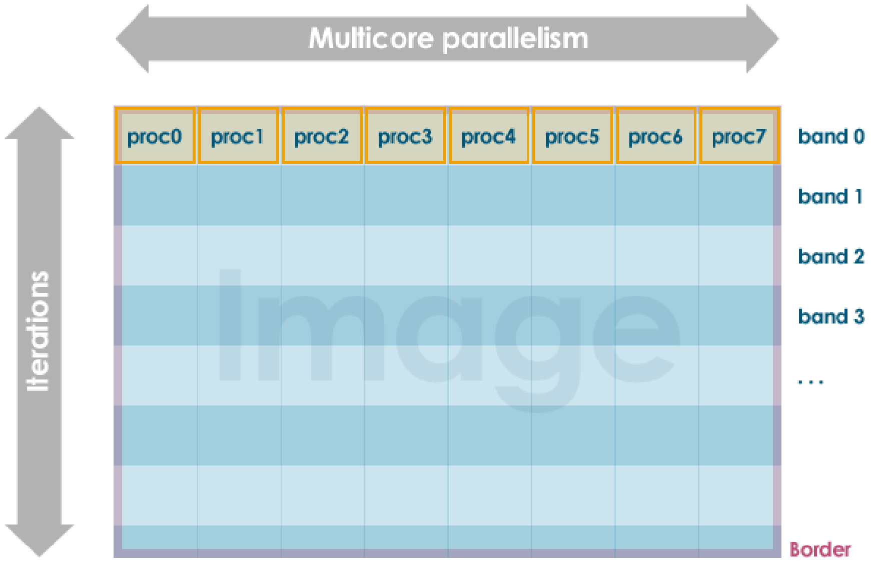


Figure 8.9: Bands splitting and dispatching on PEs

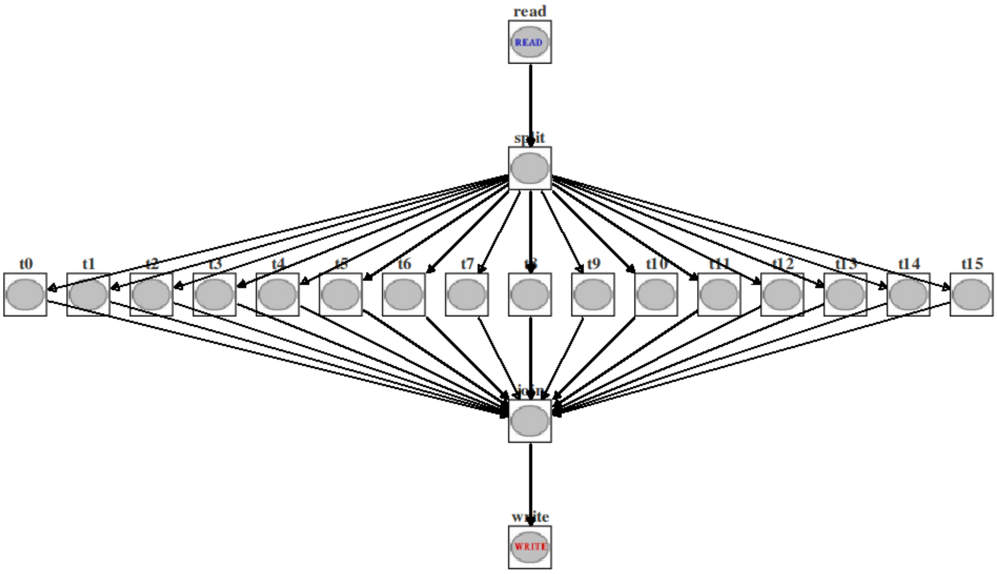


Figure 8.10: One centralized read, split and merge

Blocks Treatment

The second version consists in dividing the image into blocks directly and each read/write operation is done independently for each available processor. An abstract model of this second version can be represented by one task for reading data, one for computation (*circldetect* task) and one task for writing data back. This model, depicted in Fig-8.11, consisting in several *jobs*, each dedicated to the computation of each block. Notice that as for the first model, each job does several computations sequentially, that is the start of an iteration is triggered by the end of the previous one until the whole image has been processed.

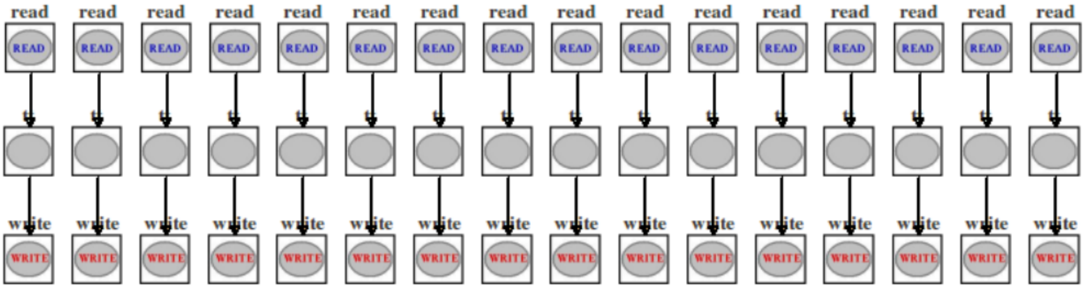


Figure 8.11: 16 independent reads and writes

Parameters Estimation

In order to perform analysis, one has to specify parameters such as computational workload for tasks and quantity of transferred data. Profiling of the sequential implementation showed us that the main computation is done by the *circular detection* function and all other functions have negligible workload. This is why our model contains only one computational task. Based on the sequential implementation we were able to extract workloads for the computational tasks, i.e, minimum and maximum number of cycles needed for the treatment of one pixel. Based on those values, we compute estimations for the execution of one block of pixels depending on its size. In our model, those execution vary up to 20% around the mean value. Note that the size of the output is 6 times larger than the size of the input.

8.2.2 Analysis

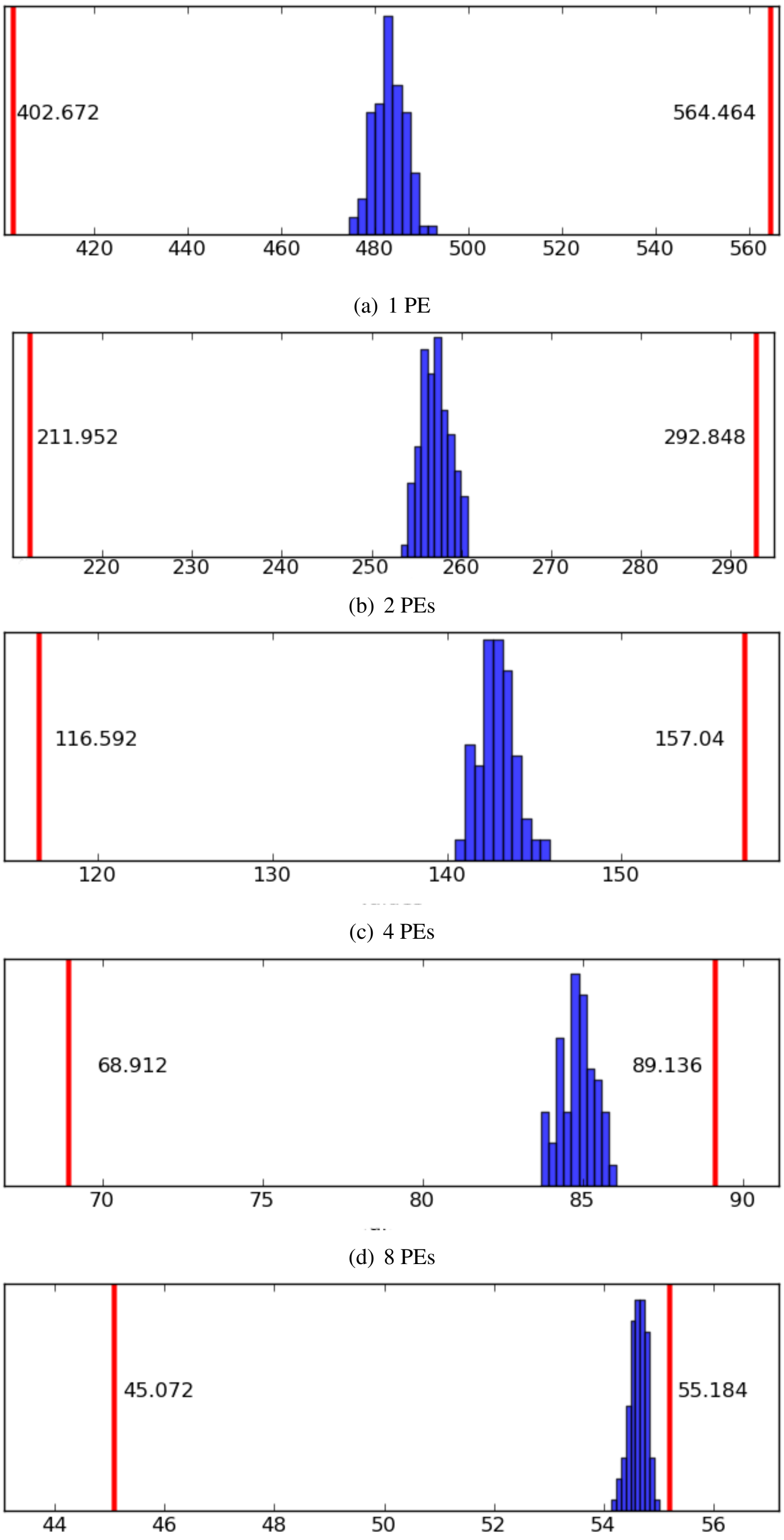
8.2.2.1 Worst Case vs Statistics

Consider the first implementation depicted by the task-graph of Fig-8.10 which represents the treatment of a horizontal band (16 blocks) of the image. All the blocks are fetched by a single read command and the data is split onto 16 tasks whose output is merged and written back to the offchip memory. Execution times for processing a single block admit roughly up 20 % deviation from their average.

We first run a timed automata (TA) based analysis of the execution of this job on architecture instances with various numbers of processors to obtain the respective lower- and upper-bounds on execution times. Then we apply statistical analysis, based on 100 random simulation runs. Fig-8.12 shows a histogram of these runs for different number of processors. Note that when there is one processor per task, the average is close to the worst-case (for that configuration) because the total termination time is defined as the max of individual task termination times. On the other hand, when the number of processors is smaller and some tasks are executed sequentially, the convolution effect renders the distribution more normal-like. Analysis and simulation results are summarized in Table-8.4.

8.2.2.2 Reading Granularity

We make a comparison between the previously mentioned strategies for fetching the data based on bands and blocks. We assume here, an image made of 256 blocks and a band consisting of 16 blocks for the first strategy with associated task graph is depicted in Fig-8.10. The second one is an alternative specification where each block is read separately (Fig-8.11). The whole job for 256 blocks is represented by sequential concatenation of 16 copies of the basic task-graphs.



(a) 1 PE

(b) 2 PEs

(c) 4 PEs

(d) 8 PEs

(e) 16 PEs

Figure 8.12: The distribution of total termination times using 1, 2, 4, 8 and 16 processors working at 600Hz. The red vertical lines indicate the lower- and upper-bounds. Note the change of scale.

		TA Analysis		Simulation		
Frequency	PE nb	Min	Max	Min	Mean	Max
200	1	1165.552	1651.184	1377.174	1405.946	1426.697
200	2	593.392	836.208	715.049	728.614	741.454
200	4	307.312	428.720	378.453	386.348	394.134
200	8	164.272	224.976	216.509	206.045	212.247
200	16	92.752	123.104	119.776	121.214	122.258
400	1	593.392	836.080	702.889	713.974	725.086
400	2	307.312	428.656	368.718	374.989	383.316
400	4	164.272	224.944	200.303	203.442	206.939
400	8	92.752	123.088	113.498	116.687	118.317
400	16	56.992	72.160	70.304	71.232	71.690
600	1	402.672	564.464	474.497	483.100	493.192
600	2	211.952	292.848	253.268	257.086	260.640
600	4	116.592	157.040	140.468	142.829	145.902
600	8	68.912	89.136	83.690	84.835	86.063
600	16	45.072	55.184	54.137	54.611	54.998

Table 8.4: Results of TA Analysis versus Simulation

Fig-8.13 shows the speedup obtained by the second, more flexible policy, as the number of processors grows. Note that the speed-up in the average-case is much more significant. Fetching data by bands requires a synchronization for writing back the results. This synchronization implies a latency coming from the slower computation. On the other hand block treatment is more flexible, once a computation terminates the transfer of the next block can start immediately and communication becomes more fluid.

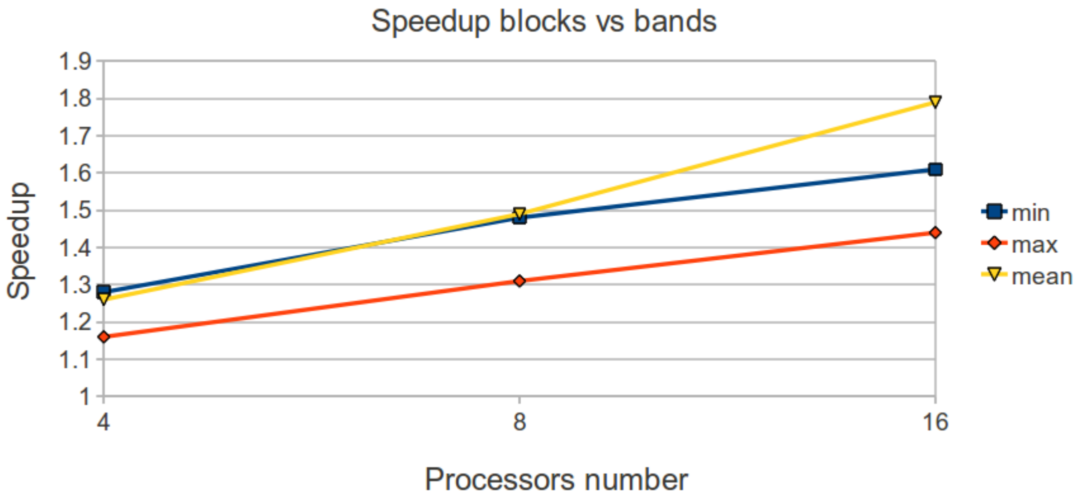
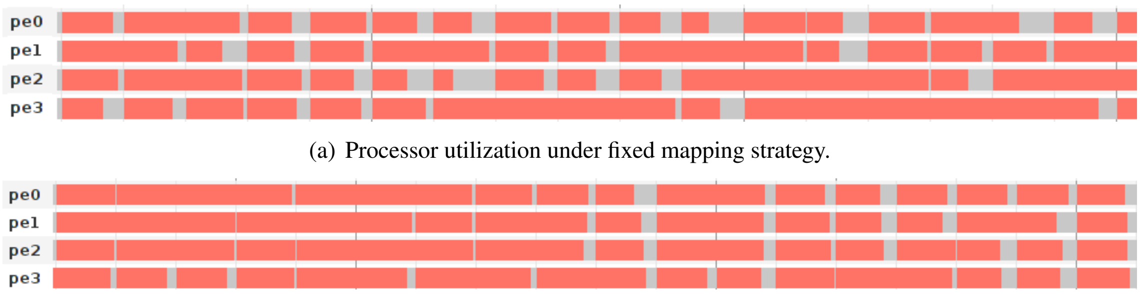


Figure 8.13: The speed-up obtained by reading single blocks compared to reading 16-block bands

8.2.2.3 Fixed vs Flexible Mapping

Next we move to a situation where there is a very large variability in the execution times of the tasks, namely $[150, 2100]$, and compare a fixed mapping with a local FIFO scheduler for each PE against a flexible mapping by a global scheduler on an instance of P2012 with 4 processors. We take the task graph of Fig-8.11 and use a periodic event generators with jitter. Using 4 processors, each PE is assigned 4 tasks (exactly for the fixed mapping policy and approximately for the flexible policy) and hence the worst-case execution time for a job instance is around 8400. For arrival periods which are smaller than the worst-case execution time, a worst-case analysis naturally shows the possibility of an unbounded accumulated backlog and, hence, unbounded latency. We perform simulations with arrival periods 7000, 6000, 5000, and 4500. Not surprisingly, the global strategy yields a much better average performance and its advantage increases with the arrival rate. Decreasing the period to 4000 (below the average execution time) leads to frequent overflows.



(a) Processor utilization under fixed mapping strategy.

(b) Processor utilization under flexible mapping strategy.

Figure 8.14: Processor utilization

The processor occupancy is smoothened when using the block strategy as illustrated in Fig 8.14. The two strategies behave similarly when the arrival period is long enough but when it becomes shorter, the global mapping strategy, due to its flexibility, shows a relative advantage as depicted in Fig 8.15.

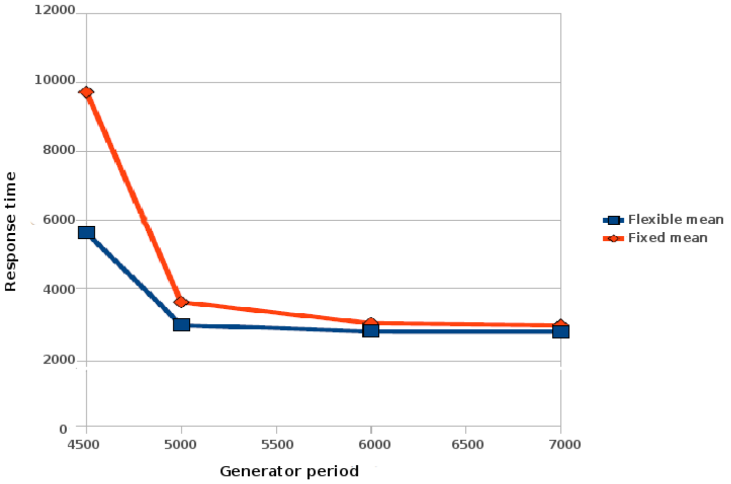


Figure 8.15: Comparing the average performance of the fixed and flexible mapping strategies as a function of the arrival period.

8.2.3 Power Consumption

In the last experiment we compare different configurations of P2012 for the trade-offs between response time and power consumption that they provide. We consider again a job consisting of a concatenation of 16 copies of the task graph of Fig-8.10 and execute it on instances of the architecture with 1, 2, 4, 8 and 16 active processors, all running in either 200, 400 or 600 MHz. For each configuration we run 100 simulations and compute the average response-time and consumption. Fig-8.16 shows the trade-offs obtained. Such plots are extremely useful for detecting regions where power consumption can be significantly reduced with a modest performance degradation which still meets the system requirements. For example several configurations (16PE-600Hz, 16PE-400Hz, 8PE-600Hz) give a response time around 100 ms with consumptions ranging over $[400, 640]$ mW. Opting for configurations such as 16PE-200Hz, 8PE-400Hz or 4PE-600Hz reduces power consumption, on average, by 2 while the response time is very slightly degraded.

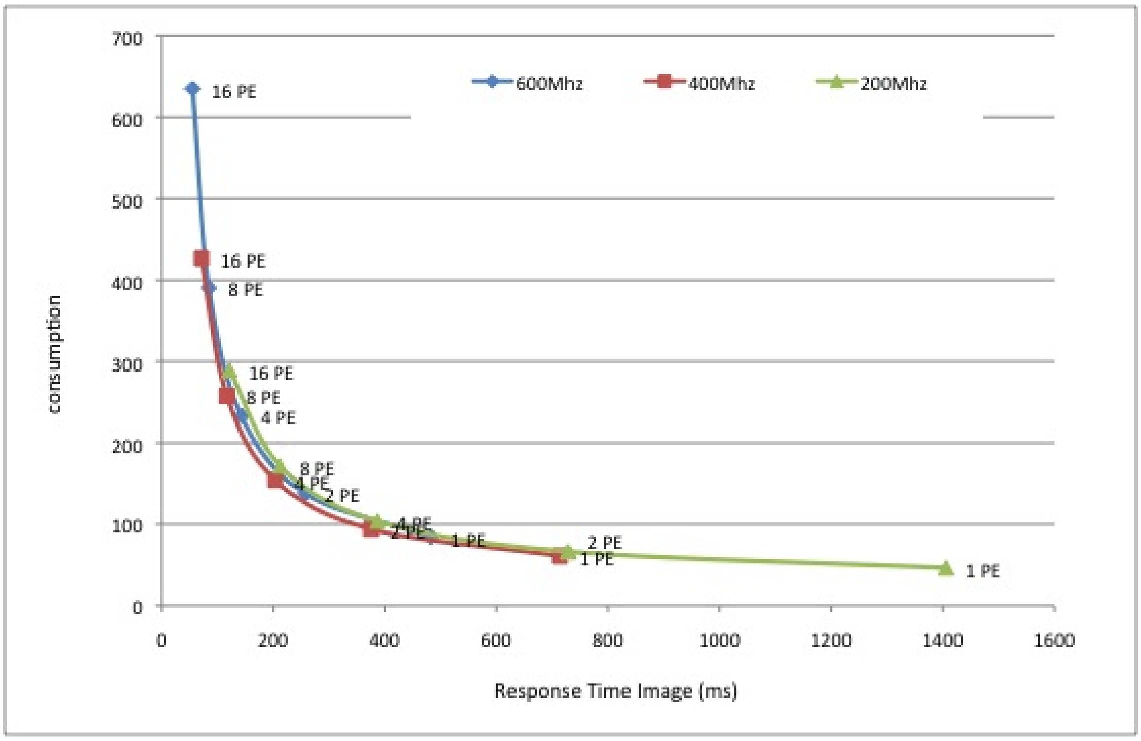


Figure 8.16: Power-performance trade-offs obtained on different configurations (number of processors and frequencies)

8.2.4 Summary

We have demonstrated how DESPEX can be used to solve very realistic problems in design-space exploration, quantify the performance differences between different design choices and represent available cost/performance trade-offs.

8.3 A Radio Sensing Application

Finally we demonstrate how the tool was used for evaluating a radio sensing application developed by Thales. The goal of this case study was to check the feasibility of porting the application, currently running sequentially on a powerful desktop, to an embedded multi-processor platform.

This case study demonstrates more the power of the methodology supported by our tool framework than performance itself. Porting a sequential implementation onto an embedded system involves several difficulties. The ability to perform rapid design space exploration at early stages is a key to identify difficult issues as soon as possible. Depending on the availability of simulators at different level of accuracy, models are then refined and DESPEX fits well into the classic Y-chart based design space exploration [122] presented in Fig-2.4 of Chapter-2.

8.3.1 Model Description

The application was evaluated on the P2012 platform [177] described previously and the xSTream platform [32]. As shown in Fig-8.17, the xSTream architecture is defined as a computing nodes fabric connected to a *system bus*. Each node admits a processing element (xPE) and has its own local memory (LM) and communicates through a high-performance network on chip (xSTNoc). Note that the processing elements of the two architectures belong to the same ISS family.

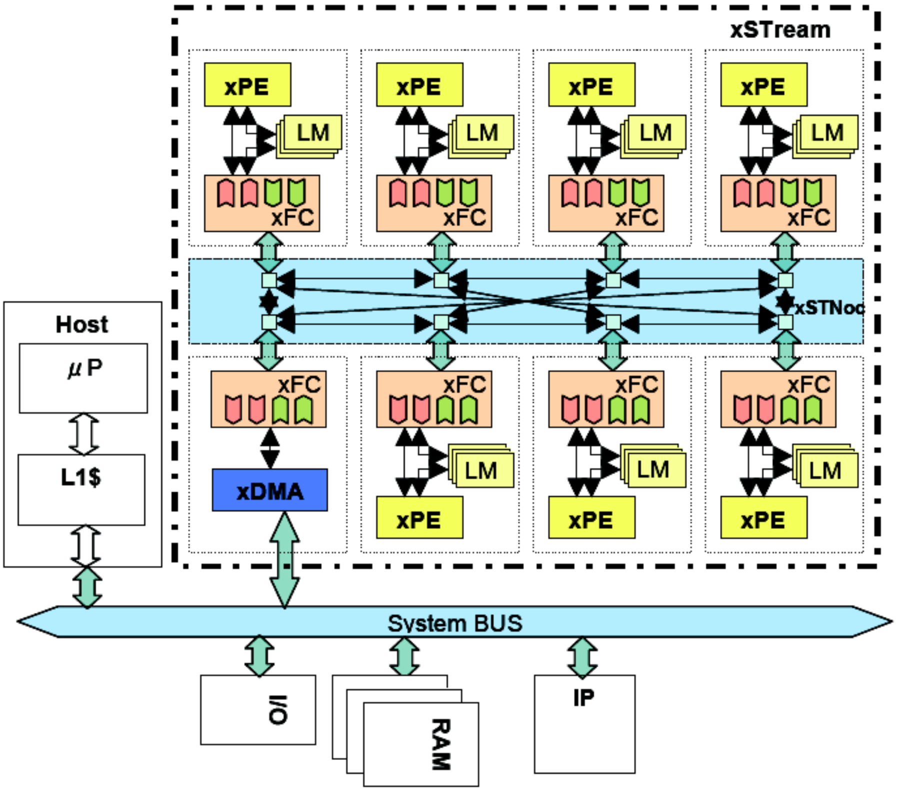


Figure 8.17: The xSTream architecture from [32]

The abstract model of the xSTream architecture is depicted on Fig-8.18. It consists in 8 processing elements connected to local memory components. We model the network on chip (xSTNoc) as a bus component. As for the P2012 model, synthetic parameters are associated with each component.

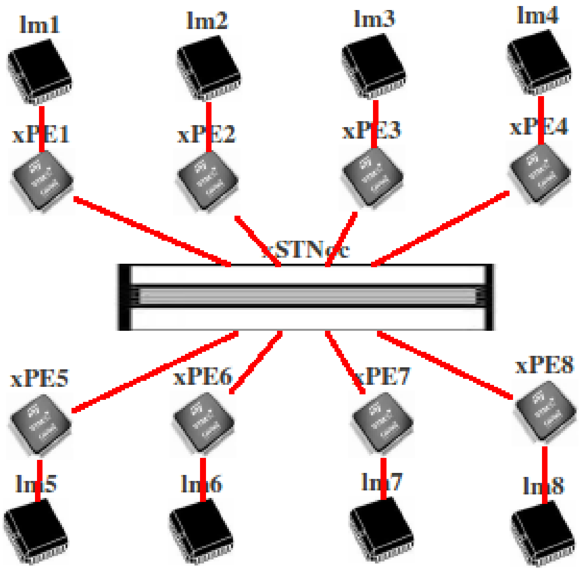


Figure 8.18: The xSTream model

Radio Sensing Application

Sensing is one function of cognitive radio. A cognitive radio is a transceiver which automatically detects available channels in the wireless spectrum and accordingly changes its transmission or reception parameters so that more wireless communications may run concurrently in a given spectrum band. According to [4] the main functions of cognitive radio are:

- Sensing: detecting unused spectrum and sharing the spectrum without harmful interference with other users;
- Management: capturing the best available spectrum to meet user communication requirements;
- Mobility: maintaining seamless communication requirements during the transition to a better spectrum;
- Sharing: Providing the fair spectrum scheduling method among coexisting users.

We focus on sensing which consists in two steps: extraction of the spectrum (GSM in this example) and then extraction of the different channels inside the spectrum (200 in GSM).

For this example we consider one step of the sensing function, the so called characterization that consists in several operations such as transposition, quantification and filtering on chunks of the signal to be processed. Splitting of the signal is done according to 200 different channels. A task-graph model of this application is shown on Fig-8.19. Boxes represent the concatenation of several computations into task for the model used in our tool which is shown on Fig-8.20.

Parameters Estimation

Parameters have been estimated using an Instruction Set Simulator (ISS) provided in the context of the ATHOLE project. That is, workload and amount of produced data are obtained for each function after instrumentation and execution of the concrete C code . Note that the results obtained for the PE of xSTream can be extrapolated to the PE of P2012 due to similarity.

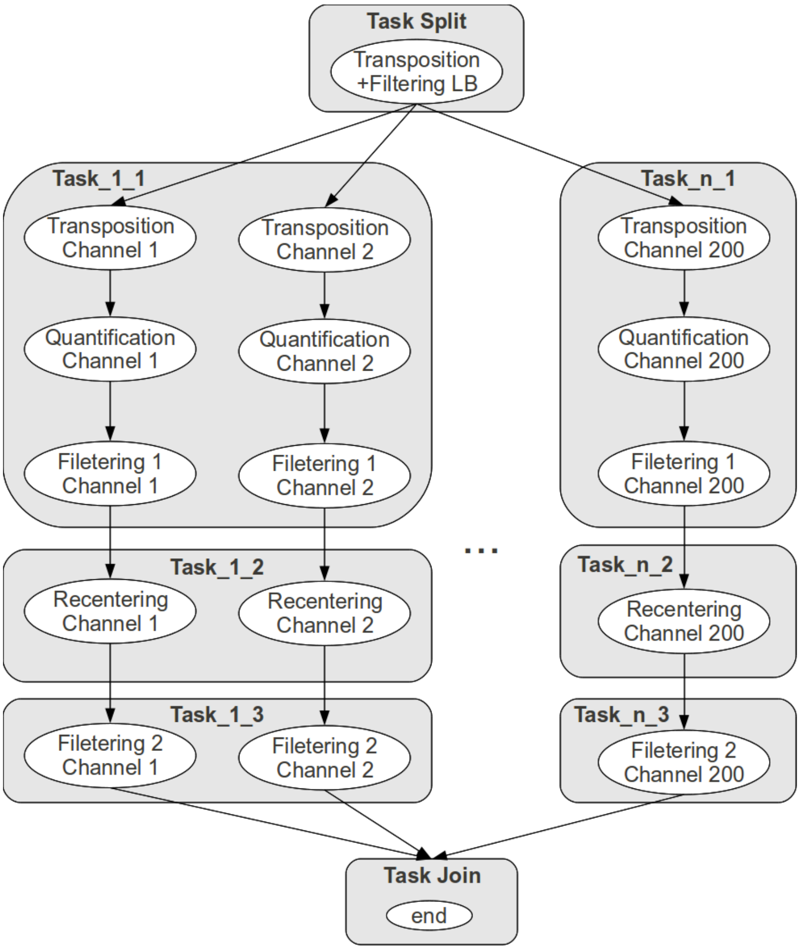


Figure 8.19: Sensing Application

8.3.2 Performance evaluation

Parallelization

Programming multi-processors embedded systems such as xSTream and P2012 implies the use of appropriated programming model. The sequential implementation needs therefore to be modified for exploiting parallelism provided by such architectures. The use of DESPEX, requires to transform the sequential implementation into a task-graph, by decomposing the initial implementation into *tasks* that can possibly run in parallel and identify control and data flow.

This work shows that the sequential implementation is easily parallelizable and MPSoCs can, in principle, be a solution to execute this kind of application. However, the modeling and analysis effort has revealed that the amount of data transfer is huge and memory capability of embedded architectures can be a bottleneck. Actually, the application is highly parallelizable because the same computation is done on different chunks of data, nevertheless the workload required depends on the size of the blocks, currently limited by local memory capacity.

Evaluation

According to the analysis performed with DESPEX, the application in its current form cannot be ported to MPSoCs such as xSTream and P2012 while providing a real-time response. One conclusion from this experience is that porting to embedded devices requires re-design of the

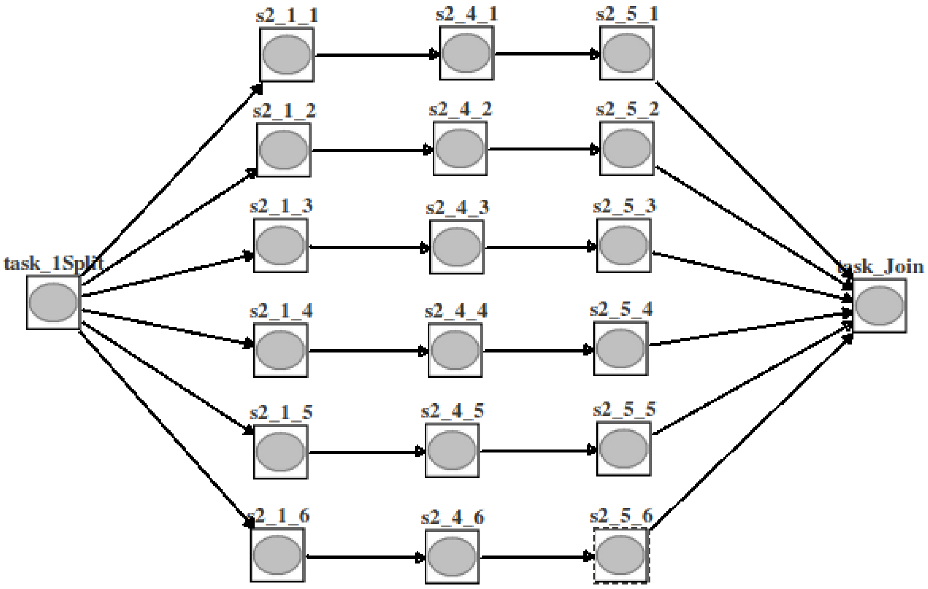


Figure 8.20: Sensing Application Model

algorithms, focusing on more modest usage of scarce resources such as memory. Nevertheless, evaluation of different mapping strategies resulted in a 5% improvement in the execution time for a fragment of the application working on 40 channels.

Methodologically, this case study demonstrates the advantage of separating as much as possible the models of the architecture and the platform. We have profiled the tasks on the processor of xSTream and could use the numbers, slightly calibrated, to estimate their execution time on P2012 (featuring processors of the same ISS family) and evaluate the overall execution time on the new architecture without much effort.

8.3.3 Summary

With this example, we have shown how our tool can fit into classical design flow for rapid design space exploration. The modeling and re-design effort needed for using our tool, may serve for detecting bottlenecks (such as memory data transfer) at early design stage. Such re-design is, anyway, an unavoidable step in rewriting the application for parallel execution. Once parameters have been extracted correctly, models can be used for evaluating several configuration in short time without having to wait the availability of accurate system-wide simulators.

Chapter 9

Conclusions and Future Work

This thesis constitutes a contribution to the important problem of designing embedded multi-core architectures and efficiently deploying application programs on these platforms. We have provided a tool-supported methodology for high-level modeling of the major components that participate in the execution: hardware architecture, application program, external environment and mapping/scheduling policies. These models abstract away from many low-level details of the program and the architecture and provide for rapid simulation and performance evaluation, as a complement to the more detailed and accurate models used in the development process. These models are then translated into timed automata and are analyzed by complementary techniques that include standard timed automata reachability, Monte Carlo simulation and analytic calculation of performance indices. The modeling framework and analysis techniques have been implemented into an extensive and extendible toolset, DESPEX, the *design-space explorer*, featuring many important functionalities. The tool, which served as a major deliverable in an academic-industrial collaborative project, has been applied to several interesting case studies.

From a theoretical standpoint, we provided novel ideas and results concerning analysis and scheduler synthesis for continuous-time stochastic processes where task durations are distributed uniformly over a bounded interval. Compared to exponential "memoryless" distribution often treated in the literature, for our distributions the clock values which represent the time elapsed since the beginning of each active step, influence the temporal probability of future events. Hence, analyzing such systems required several conceptual innovations including the delegation of timing uncertainty into the duration space to facilitate volume computation and the definition of density function families parameterized by clocks values. Realizing the proposed algorithms required an implementation of a specialized integration package.

As for the future, we foresee numerous research directions in all dimensions of this work: theory, implementation, methodology and application. We list a few of them below.

- The tool itself, like all tools developed within a project and a thesis, will require a major re-design and implementation in the future before it could be given at the hands of real users.[1] Such modifications include the improvement of the translation of components into timed automata, accelerating the simulation (currently performed on top of IF which is not optimized for such simulations), compacting the generated traces which are currently too verbose and improving the user interface.

- From a methodological point of view, a more systematic way to populate the high-level models with performance numbers should be developed. It should be based on a combination of profiling the available components by low-level simulation and interaction with the user

[1] Although the case study described in Section 8.3 was conducted by Thales working directly with the tool.

in the style of decision-support systems. In general, a tighter integration with the software development flow (which is still under construction for multi-core systems) is needed.

- On the application side, our ambition is to see DESPEX adapted and applied more extensively to P2012 in the framework of a new forthcoming project. This will involve more refined modeling of the specific components of the architecture as well as a tighter integration with the software development process for this platform. It should be noted that the application and architecture concepts developed here are not restricted to the scale of MPSoCs, and the problematics of finding the right balance between computations and data transfers can be found in other scales such as distributed or scientific computing.

- On the theory side, the work on optimal scheduler synthesis is still incomplete. The exact form of the boundaries in the decision space between the domains of the different actions should be characterized and an approximation algorithm with guaranteed bounded error should be implemented. Synthesis itself (in the sense of scheduler/controller synthesis) is an open unsolved problem in terms of complexity, and some of the ideas developed in this part can be used to find a more compositional way to achieve it. Finally, a more thorough comparison of the computational trade-offs between analytic and statistical methods will help in assessing their relative advantages and shortcomings.

Appendix A

DPA: Optimizing the Value Function (Work in Progress)

A.1 Non-Lazy Schedulers

To complete the implementation we need to handle the *optimization* part of the iteration, computing $V(q,x) = \min_s(V(q,x,s))$ for every x. In other words we should trace the boundary in the clock space between subsets where waiting or starting are preferred. To facilitate the task we first prove an important property of optimal strategies which simplifies the partition of the clock space according to the optimal action. This "non-laziness" property has been formulated in the deterministic setting in [2] and it simply captures the intuition that waiting and refraining from taking a resource is *not* useful *unless* some other process benefits from the resource during the waiting period.[1] The proof in the deterministic setting is based on taking a schedule that admits laziness and transforming it iteratively into a schedule with less laziness (but no inferior performance) until a non-lazy schedule is obtained. The proof in the non-deterministic context is more involved.

Definition A.1.1 (Laziness). *A scheduling policy Ω is lazy in (q,x) if $\Omega(q,x+t) = s^{\mathbf{i}}$ for some i and $\Omega(q,x+t') = \mathbf{w}$ for every $0 \le t' < t$. A schedule is non-lazy if no such (q,x) exists.*

Theorem 3 (Non Lazy Optimal Schedulers). *The optimal value V can be obtained by a non-lazy strategy.*

To prove the theorem we first need the following lemma.

Lemma 1 (Value of Progress). *Let q be a state and let x and x' be two clock valuations which are identical except for $x'^{\mathbf{i}} = x^{\mathbf{i}} + \delta$. Then the value of (q,x') is at least as good as the value of (q,x), that is, $V(q,x') \le V(q,x)$.*

Sketch of Proof: We prove it by induction on the number of transitions remaining between q and the final state.
Base case:*Base case*: When there is only one *end* transition pending and one active clock working toward a duration characterized by a time density ψ, the value is defined as

$$V(q,x) = \int_{\max(x,a)}^{b} \psi_{/x}(y)(y-x)dy = \begin{cases} (a+b)/2 - x & \text{when } x \le a \\ (b-x)/2 & \text{when } x \ge a \end{cases}$$

and the derivative dV/dx is negative (-1 and then $-1/2$). Hence $V(q,x') < V(q,x)$.
Inductive case: suppose the lemma holds for all states beyond q. We will show that each action

[1] Or if some information gathered during the period has increased the expected time-to-go associated with waiting, which is impossible in our setting, see discussion.

taken in (q, x') can lead to a state at least as advanced as the state reached via the same action from (q, x). For an immediate *start* transition this is evident. Suppose we take action $\mathbf{w}$ at (q, x') and (q, x). Then there are three possibilities: if the race winner at (q, x') is some $P^{\mathbf{i}'}$, $i' \neq i$, taking an $e^{\mathbf{i}'}$ transition to q', then the same $P^{\mathbf{i}'}$ will win the race from (q, x) within the same time, landing in q' with value of $x^{\mathbf{i}}$ less advanced by δ and the inductive hypothesis holds, see Figure A.1-(a) and the corresponding runs ξ_x and $\xi_{x'}$, depicted below while omitting the discrete state and the other clocks:

$$\xi_{x'} : \ (x^{\mathbf{i}} + \delta, x^{\mathbf{i}'}) \xrightarrow{t} (x^{\mathbf{i}} + \delta + t, x^{\mathbf{i}'} + t) \xrightarrow{e^{\mathbf{i}'}} (x^{\mathbf{i}} + \delta + t, \bot)$$

$$\xi_x : \ (x^{\mathbf{i}}, x^{\mathbf{i}'}) \xrightarrow{t} (x^{\mathbf{i}} + +t, x^{\mathbf{i}'} + t) \xrightarrow{e^{\mathbf{i}'}} (x^{\mathbf{i}} + t, \bot).$$

Suppose, on the contrary, that $P^{\mathbf{i}}$, the process whose clock is more advanced in x' than in x, wins the race from (q, x') within t time. If $P^{\mathbf{i}}$ is also the winner from (q, x), the $e^{\mathbf{i}}$ transition will be taken by ξ_x after $t + \delta$ time. By waiting at q' for δ time, run $\xi_{x'}$ can reach the same state as ξ_x, see Figure A.1-(b) and the corresponding runs:

$$\xi_{x'} : \ (x^{\mathbf{i}} + \delta, x^{\mathbf{i}'}) \xrightarrow{t} (x^{\mathbf{i}} + \delta + t, x^{\mathbf{i}'} + t) \xrightarrow{e^{\mathbf{i}}} (\bot, x^{\mathbf{i}'} + t) \xrightarrow{\delta} (\bot, x^{\mathbf{i}'} + t + \delta)$$

$$\xi_x : \ (x^{\mathbf{i}}, x^{\mathbf{i}'}) \xrightarrow{t+\delta} (x^{\mathbf{i}} + \delta + t, x^{\mathbf{i}'} + \delta + t) \xrightarrow{e^{\mathbf{i}}} (\bot, x^{\mathbf{i}'} + t + \delta).$$

Finally, suppose the race winner from (q, x) is $P^{\mathbf{i}'}$ within some $t + \delta'$ time, $\delta' < \delta$, leading to state q''. This splits into two sub-cases. If at q'' the run ξ_x does not start a new step, then, by waiting at q' for $\delta - \delta'$ time, $\xi_{x'}$ can reach within $t + \delta'$ the same extended state that ξ_x has reach in more time $t + \delta$:

$$\xi_{x'} : \ (x^{\mathbf{i}} + \delta, x^{\mathbf{i}'}) \xrightarrow{t} \xrightarrow{e^{\mathbf{i}}} (\bot, x^{\mathbf{i}'} + t) \xrightarrow{\delta'} (\bot, x^{\mathbf{i}'} + t + \delta') \xrightarrow{e^{\mathbf{i}'}} (\bot, \bot)$$

$$\xi_x : \ (x^{\mathbf{i}}, x^{\mathbf{i}'}) \xrightarrow{t+\delta'} \xrightarrow{e^{\mathbf{i}'}} (x^{\mathbf{i}} + \delta' + t, \bot) \xrightarrow{\delta-\delta'} (x^{\mathbf{i}} + \delta + t, \bot) \xrightarrow{e^{\mathbf{i}}} (\bot, \bot).$$

If, on the other hand, during the interval of duration $\delta - \delta'$ in which ξ_x catches up with the progress of $\xi_{x'}$ in $P^{\mathbf{i}}$, ξ_x makes an $s^{\mathbf{i}'}$ transition to start a next step of $P^{\mathbf{i}'}$, so can $\xi_{x'}$ and they will reach the same state:

$$\xi_{x'} : \ (x^{\mathbf{i}} + \delta, x^{\mathbf{i}'}) \xrightarrow{t} \xrightarrow{e^{\mathbf{i}}} (\bot, x^{\mathbf{i}'} + t) \xrightarrow{\delta'} (\bot, x^{\mathbf{i}'} + t + \delta') \xrightarrow{e^{\mathbf{i}'}} \xrightarrow{s^{\mathbf{i}'}} (\bot, 0) \xrightarrow{\delta-\delta'} (\bot, \delta - \delta')$$

$$\xi_x : \ (x^{\mathbf{i}}, x^{\mathbf{i}'}) \xrightarrow{t+\delta'} \xrightarrow{e^{\mathbf{i}'}} \xrightarrow{s^{\mathbf{i}'}} (x^{\mathbf{i}} + \delta' + t, 0) \xrightarrow{\delta-\delta'} (x^{\mathbf{i}} + \delta + t, \delta - \delta') \overset{e^{\mathbf{i}}(\bot,\delta-\delta').}{\longrightarrow}$$

These two cases are illustrated in Figure A.1-(c).

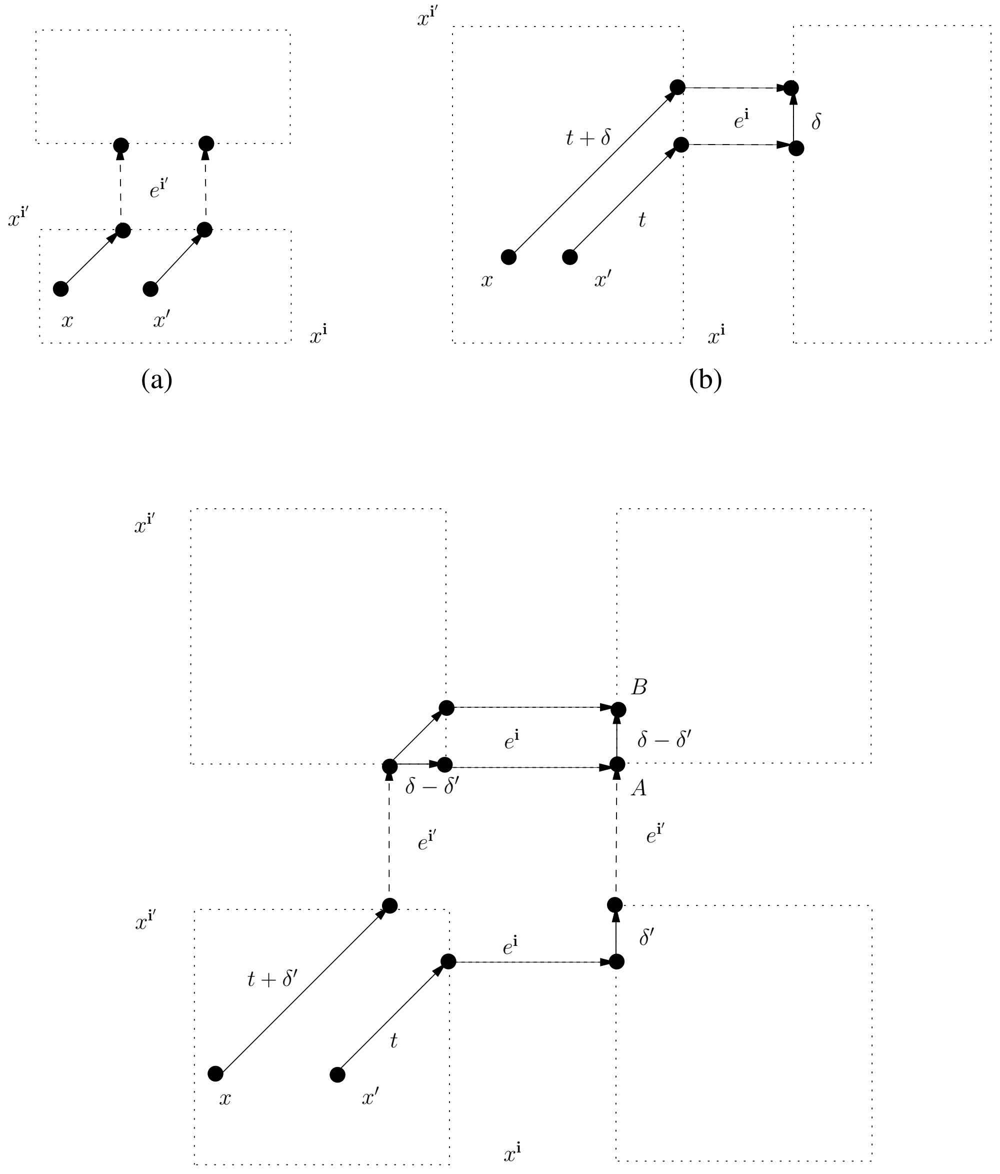
The bottom line of this case analysis is that for every action taken in (q, x) and (q, x') and for every y, there exist durations t_y and t'_y, such that $0 \leq t'_y \leq t_y$, a state p_y such that $q \prec p_y$ and clock valuations z_y and z'_y such that $z_y \leq z'_y$ in the value of one clock (as in the premise of the lemma) and

$$(q, x) \xrightarrow{t_y} (p_y, z_y) \quad \text{and} \quad (q, x') \xrightarrow{t'_y} (p_y, z'_y).$$

Since the cost to go from (q, x) can be written as

$$V(q, x) = \int \psi(y) \cdot (t_y + V(p_y, z_y))dy$$

and since for every y, $t'_y \leq t_y$ and $V(p_y, z'_y) \leq V(p_y, z_y)$ by the inductive hypotheses, we have $V(q, x') \leq V(q, x)$. ▟



(a)

(b)

(c)

Figure A.1: Proof of lemma: (a) $P^{\mathrm{i}'}$ wins from both x and x'; (b) P^{i} wins from both. (c) P^{i} wins from x' and $P^{\mathrm{i}'}$ wins from x. Here either $\xi_{x'}$ reaches point A in less time than does ξ_x, or both reach point B at the same time.

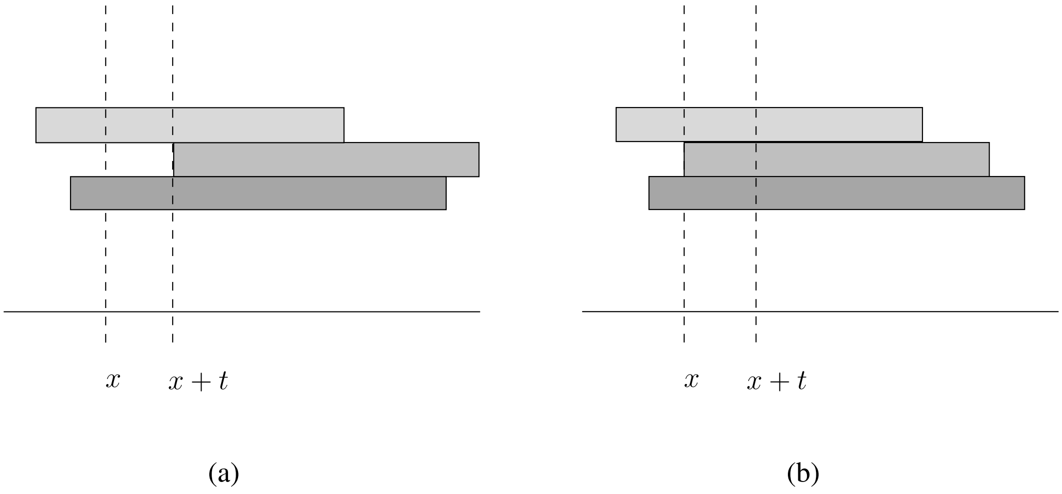


Figure A.2: Proof of theorem.

Note that although the lemma can be interpreted as saying that always a more advanced state has a better value, this is true in general only for progress that does not increase the possibility of blocking due to resource conflicts. Advancing the clock of an already-active step, or starting a step on a resource that has no other future users are such forms of progress while starting a step on a resource that has other users in its horizon is not. Otherwise, FIFO schedulers were always optimal.

Proof of Theorem 3: Imagine a strategy Ω which is lazy at (q, x) and takes its earliest *start* at $(q, x + t)$, as illustrated in Figure A.2-(a). A alternative strategy that would start at x, can be after t time in a state where one clock is more advanced (Figure A.2-(b)) and hence satisfying the condition of Lemma 1. ▟

Let us repeat that the theorem does *not* say that a FIFO scheduler which makes a *start* transition as soon as possible is optimal. It might be useful to delay starting until one of the competing tasks has used the resource. Only delays that are *shorter* than that are considered lazy.

To illustrate the limited scope of the lemma and theorem consider a task with whose duration characterized by a *discrete* probability with probability p for a and $1 - p$ for b. In this case, the value function associated with waiting is

$$V(x) = \begin{cases} p(a-x) + (1-p)(b-x) & \text{when } x < a \\ 0(a-x) + 1(b-x) & \text{when } x > a \end{cases}$$

Here at $x = a$ there is a *jump* in V from $(1 - p)(b - a)$ to $(b - a)$ which is, intuitively, due to the accumulation of information: after a time, the non-occurrence of an *end* event tells us that the duration is certainly b. Such a situation contradicts the lemma, because for $x < a < x'$ we may have $V(x') > V(a)$. This jump in the expected time-to-go for waiting can also justify laziness: when $x < a$ waiting can be better then starting, but after $x = a$ the relation between these two values may change.

A.2 Upward Closed Strategies and Rectangular Approximations

In the sequel we present the implications of non-laziness on the structure of the optimal scheduler. To simplify notations, but without loss of generality, we assume throughout this part that each resource $m \in M$ is used by exactly one step of each of the processes. Although formally the

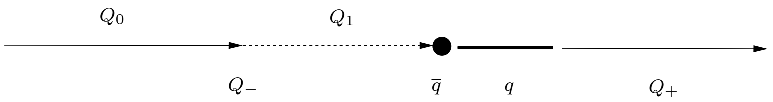


Figure A.3: The timeline of a process partitioned with respect to a resource used in state q and the duration of a conflicting step of another process.

strategy is defined over Q, it can be decomposed into nk *partial* strategies, each representing a *start/wait* decision associated with one step of one process. The strategy can be written as

$$\Omega = \bigcup_{m \in M} \bigcup_{i \in N} \Omega^{\mathbf{i}}_m$$

where $\Omega^{\mathbf{i}}_m$ corresponds to the step where $P^{\mathbf{i}}$ uses resource m, say step j. In this case $\Omega^{\mathbf{i}}_m$ has the form

$$\Omega^{\mathbf{i}}_m : Q^{\mathbf{1}} \times \cdots \{\overline{q}^{\mathbf{i}}_j\} \cdots \times Q^{\mathbf{n}} \to \{s^{\mathbf{i}}_j, \mathbf{w}\}.$$

We call the domain of $\Omega^{\mathbf{i}}_m$ the *decision surface* associated with $s^{\mathbf{i}}_j$. In general, the state space of a process P which uses a resource m in state q can be partitioned as

$$Q = Q_- \cup \{\overline{q}\} \cup \{q\} \cup Q_+,$$

namely the states *before* the step in question, the *waiting* state (where decisions are to be made), the *active* state (where other processes cannot use the resource) and the states after that, see Figure A.3.

When the scheduler has to decide whether to let *another* process P' take resource m, the only reason for not doing so could be the possibility of blocking P in the future if P is in some pre-conflict state in Q_-. However, not all pre-conflict states represent a real conflict with P'. If the upper bound on the duration of the step of P' is b, and P is in a state in Q_- whose minimal distance to $\overline{q}$ is larger than b, P' will release the resource before P becomes enabled and no blocking will occur. This justifies a finer refinement of the pre-conflict states, $Q_- = Q_0 \cup Q_1$, with Q_0 consisting of states of P where we need not worry about its being blocked by P' (see Figure A.3). Note that Q_0 is a static *under-approximation* of the set of states having this property under a given scheduling strategy because it is based on a minimal distance, assuming P progresses toward $\overline{q}$ without waiting. We stress again the fact that the boundary between Q_0 and Q_1 is specific to each P'.

We illustrate the implications of arithmetics, mutual exclusion and non-laziness on the partition of the state space induced the optimal strategy on a system of of 3 processes where we analyze the decision surface $Q^{\mathbf{2}} \times Q^{\mathbf{3}}$ for the sub-scheduler

$$\Omega^{\mathbf{1}}_m : \{\overline{q}^{\mathbf{1}}\} \times Q^{\mathbf{2}} \times Q^{\mathbf{3}} \to \{s, \mathbf{w}\}$$

for process $P^{\mathbf{1}}$. First we partition the timelines of the competing processes $P^{\mathbf{2}}$ and $P^{\mathbf{3}}$ with respect to the duration of the relevant step of $P^{\mathbf{1}}$. One can observe that:

- If the other processes are in $(Q^{\mathbf{2}} \times \{q^{\mathbf{3}}\}) \cup (\{q^{\mathbf{2}}\} \times Q^{\mathbf{3}})$, that is, one of them uses the resource, then only action $\mathbf{w}$ is possible;
- If the other processes are in $(Q^{\mathbf{2}}_0 \cup Q^{\mathbf{2}}_+) \times (Q^{\mathbf{3}}_0 \cup Q^{\mathbf{3}}_+)$ then action s is chosen because there will be no blocking;
- The strategy needs really to be computed over $(Q^{\mathbf{2}}_1 \times (Q^{\mathbf{3}} - \{q^{\mathbf{3}}\})) \cup ((Q^{\mathbf{2}} - \{q^{\mathbf{2}}\})) \times Q^{\mathbf{3}}_1)$ and there we can enjoy the monotonicity property.

The optimal partition of the decision surface is illustrated in Figure A.4-(a) with the grey areas indicating the subsets where where we need to compute. In each of this areas, the set $\Omega^{-1}(\mathbf{w})$ is upward closed with respect to $\preceq$. Any upward closed set can be written as a union of rectangular cones, but this union can be infinite, for example for a half-space which is not axes parallel and certainly for sets with a curved boundary. In the absence of some algebraic miracle that will turn $\Omega^{-1}(\mathbf{w})$ to be a finite union of zones, the best we can hope for is an *approximation* scheme.

In [138], a multi-dimensional binary search algorithm was presented for approximating the boundary between an upward-closed set and its downward-closed complement. We show below how it can be applied to approximate the boundary between $\Omega^{-1}(\mathbf{w})$ and $\Omega^{-1}(s)$. Suppose we fix a point (q, x) in the decision surface and compute its value under s and under $\mathbf{w}$. If $V(q, x, \mathbf{w}) < V(q, x, s)$ the optimal strategy satisfies not only $\Omega(q, x) = \mathbf{w}$ but also $\Omega(q', x') = \mathbf{w}$ for any (q', x') in the *upward rectangular cone* above (q, x), that is, any state satisfying $(q, x) \preceq (q', x')$ with no resource utilization between q and q'. Likewise if $V(q, x, \mathbf{w}) > V(q, x, s)$ the optimal strategy is s for all points in the downward cone of (q, x).

By continuing in this manner, evaluating V on various states, we reach a situation in which the decision surface is partitioned into at most 3 parts: Ω_s consisting of states where the optimal decision is known to be s, $\Omega_{\mathbf{w}}$ where the optimal decision is known to be $\mathbf{w}$, and $\Omega_?$ which is the rest of the decision surface where we do not know yet, see Fig. A.4-(b). The technique of [138] is geared toward reducing the distance between the boundaries of Ω_s and $\Omega_{\mathbf{w}}$ and hence reducing the size of $\Omega_?$. For any desired distance ε, the algorithm will converge after a *finite* number of evaluations of V.

Based on this partial knowledge we can derive an *approximately optimal* strategy, by selecting an arbitrary decision for all states in $\Omega_?$. In the sequel we consider *eager* approximations that make s in all the states in $\Omega_?$. How far is such a strategy from the optimal one? The following lemma bounds the derivative of V with respect to any of the clocks.

Lemma 2 (Derivative of V). *Let V be the value function associated with a problem then for every (q, x) and for every i*

$$\frac{\partial V}{\partial x^{\mathrm{i}}}(q, x) \geq -1$$

Sketch of Proof: Consider first the local value function of a process in isolation. In any state q, the domain of the clock is split into two parts. When $x < a$, the derivative is naturally -1. When $x > a$ the rate of progress is slower (because progress is combined with accumulation of optimism-contradicting information) and the magnitude of the derivative is smaller.[2] When running together, the progress of each process is bounded by its progress in isolation or by the progress of another process that blocks it. The progress of the expected termination of the last process (makespan) is bounded by the progress of each of the processes. ▟

Lemma 3 (Approximation). *Let x, x' be two points such that $x < x'$, $d(x, x') < \varepsilon$, $\Omega(q, x) = s$ and $\Omega(q, x') = \mathbf{w}$ for the optimal strategy Ω. Consider an eager approximation Ω' making s in all states $(q, x'') \in \Omega_?$ such that $x \leq x'' < x'$. Then for every such x'' we have $|\Omega(q, x'') - \Omega'(q, x'')| \leq \varepsilon$.*

Proof: According to what we know about the optimal strategy we have

$$V(q, x', \mathbf{w}) < V(q, x', s) < V(q, x, s) < V(q, x, \mathbf{w})$$

and according to Lemma 2, $V(q, x, \mathbf{w}) - V(q, x', \mathbf{w}) \leq \varepsilon$. For any x'', $V(q, x'', s)$ and $V(q, x'', \mathbf{w})$ are inside the interval and the distance between them is bounded by ε. ▟

[2] The derivative of V represents the progress toward the average duration, minus the growth of the average itself when $x > a$.

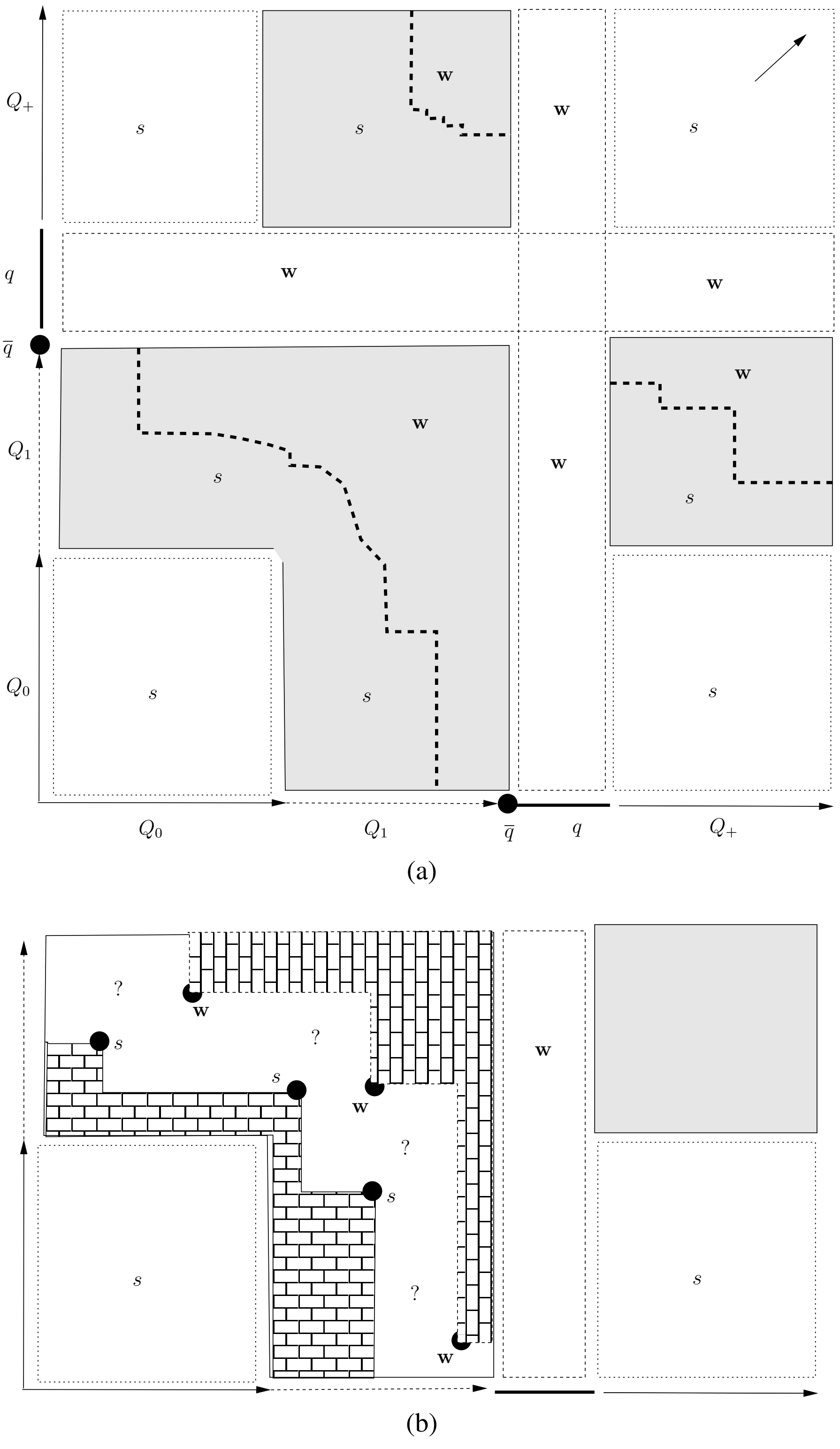


Figure A.4: (a) The decision surface for $P^{\mathbf{2}} \times P^{\mathbf{3}}$ for the decision of $P^{\mathbf{1}}$. Only in the grey area the decision is not self-evident and there the domain of $\mathbf{w}$ is upward closed; (b) A query-based rectangular approximation of the optimal strategy. In the blank area the optimal strategy is not known but approximated.

This result gives a basis for an approximation procedure, however there will be, at least theoretically, a *wrapping effect*, because the approximation of the value at q will be used for computing the value of its predecessors, and a path that makes k decisions, in all of which the boundary is approximated with accuracy ε relative to its successor, can in principle, deviate from the optimal time-to-go by $k\varepsilon$. But this is very theoretical and we expect real-life to behave much better.

Unfortunately, this part of the work has not been fully completed at the time of writing and we had to postpone the complete implementation and experimental evaluation to the future.

Bibliography

[1] The International Technology Roadmap for Semiconductors (ITRS), System Drivers, 2009, http://www.itrs.net/.

[2] Yasmina Abdeddaïm, Eugène Asarin, and Oded Maler. Scheduling with timed automata. *Theoretical Computer Science*, 354(2):272–300, 2006.

[3] B. Ackland, A. Anesko, D. Brinthaupt, S.J. Daubert, A. Kalavade, J. Knobloch, E. Micca, M. Moturi, C. J. Nicol, J.H. O'Neill, J. H. O'neill, J. Othmer, E. Säckinger, K. J. Singh, J. Sweet, C. J. Terman, and J. Williams. A single-chip 1.6 billion 16-b mac/s multiprocessor dsp, 2000.

[4] Ian F. Akyildiz, Won-Yeol Lee, Mehmet C. Vuran, and Shantidev Mohanty. Next generation/dynamic spectrum access/cognitive radio wireless networks: A survey. *COMPUTER NETWORKS JOURNAL (ELSEVIER*, 50:2127–2159, 2006.

[5] R. Alur, C. Courcoubetis, and D.L. Dill. Model-checking for probabilistic real-time systems. In *ICALP*, pages 115–126, 1991.

[6] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[7] Rajeev Alur and Mikhail Bernadsky. Bounded model checking for GSMP models of stochastic real-time systems. In *In Proc. of HSCC'06, LNCS 3927*, pages 19–33. Springer, 2006.

[8] Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In Mike Paterson, editor, *ICALP*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer, 1990.

[9] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. *J. ACM*, 41(1):181–204, 1994.

[10] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times: A tool for schedulability analysis and code generation of real-time systems. In Kim Guldstrand Larsen and Peter Niebert, editors, *FORMATS*, volume 2791 of *Lecture Notes in Computer Science*, pages 60–72. Springer, 2003.

[11] ARM. Amba specification v2.0, 1999.

[12] ARM. Multi-layer ahb overview, 2001.

[13] E. Asarin, P. Caspi, and O. Maler. Timed regular expressions. *J. ACM*, 49(2):172–206, 2002.

[14] E. Asarin and O. Maler. As soon as possible: Time optimal control for timed automata. In Frits W. Vaandrager and Jan H. van Schuppen, editors, *HSCC*, volume 1569 of *Lecture Notes in Computer Science*, pages 19–30. Springer, 1999.

[15] Eugene Asarin, Oded Maler, and Amir Pnueli. Reachability analysis of dynamical systems having piecewise-constant derivatives. *Theor. Comput. Sci.*, 138(1):35–65, 1995.

[16] Eugene Asarin, Oded Maler, Amir Pnueli, and Joseph Sifakis. Controller synthesis for timed automata. In *Proc. IFAC Symposium on System Structure and Control*, pages 469–474, 1998.

[17] Rabie Ben Atitallah, Smail Niar, Samy Meftali, and Jean-Luc Dekeyser. An mpsoc performance estimation framework using transaction level modeling. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '07, pages 525–533, Washington, DC, USA, 2007. IEEE Computer Society.

[18] Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, and Joost-Pieter Katoen. Model-checking algorithms for continuous-time markov chains. *IEEE Trans. Software Eng.*, 29(6):524–541, 2003.

[19] 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 Academic Publishers, 1997.

[20] Twan Basten, Emiel Van Benthum, Marc Geilen, Martijn Hendriks, Fred Houben, Georgeta Igna, Frans Reckers, Sebastian De Smet, Lou Somers, Egbert Teeselink, Nikola Trčka, Frits Vaandrager, Jacques Verriet, Marc Voorhoeve, and Yang Yang. Model-driven design-space exploration for embedded systems: the octopus toolset. In *Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation - Volume Part I*, ISoLA'10, pages 90–105, Berlin, Heidelberg, 2010. Springer-Verlag.

[21] Ananda Basu, Saddek Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous component-based system design using the BIP framework. *IEEE Software*, 28(3):41–48, 2011.

[22] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in bip. In *SEFM*, pages 3–12. IEEE Computer Society, 2006.

[23] Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim G. Larsen, Paul Pettersson, and Judi Romijn. Efficient guiding towards cost-optimality in uppaal. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001, pages 174–188, London, UK, UK, 2001. Springer-Verlag.

[24] Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim Guldstrand Larsen, Paul Pettersson, Judi Romijn, and Frits W. Vaandrager. Minimum-cost reachability for priced timed automata. In *Proceedings of the 4th International Workshop on Hybrid Systems: Computation and Control*, HSCC '01, pages 147–161, London, UK, UK, 2001. Springer-Verlag.

[25] G. Beltrame, D. Sciuto, and C. Silvano. Multi-accuracy power and performance transaction-level modeling. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 26(10):1830–1842, October 2007.

[26] R. Ben-Salah, M. Bozga, and O. Maler. On interleaving in timed automata. In *CONCUR*, pages 465–476, 2006.

[27] Luca Benini, Alessandro Bogliolo, and Giovanni De Micheli. A survey of design techniques for system-level dynamic power management. *IEEE TRANSACTIONS ON VLSI SYSTEMS*, 8(3):299–316, 2000.

[28] Luca Benini and Giovanni De Micheli. Networks on chips: A new soc paradigm. *Computer*, 35(1):70–78, January 2002.

[29] Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis. D-finder: A tool for compositional deadlock detection and verification. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 614–619. Springer, 2009.

[30] A. Benveniste and P. Le Guernic. Hybrid Dynamical Systems Theory and the
SIGNAL
Language. *IEEE Transactions on Automatic Control*, 35(5):535–546, May 1990.

[31] Albert Benveniste, Paul Caspi, Paul Le Guernic, and Nicolas Halbwachs. Data-flow synchronous languages. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *REX School/Symposium*, volume 803 of *Lecture Notes in Computer Science*, pages 1–45. Springer, 1993.

[32] Jean-José Berenguer, Nicolas Coste, Iker De Poy Alonso, Giuseppe Desoli, Etienne Lantreibecq, Julien Legriel, and Gilbert Richard. xstream : Architecture multi-coeur sur puce pour des applications multimédia. http://vasy.inria.fr/multival/documents/xstream-2009.pdf, 2009.

[33] Reinaldo A. Bergamaschi and William R. Lee. Designing systems-on-chip using cores. In *In the 37th Design Automation Conference*, pages 420–425, 2000.

[34] Jan A. Bergstra and Jan Willem Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.

[35] Mikhail Bernadsky and Rajeev Alur. Symbolic analysis for GSMP models with one stateful clock. In Alberto Bemporad, Antonio Bicchi, and Giorgio C. Buttazzo, editors, *HSCC*, volume 4416 of *Lecture Notes in Computer Science*, pages 90–103. Springer, 2007.

[36] Gerard Berry, Georges Gonthier, Ard Berry Georges Gonthier, and Place Sophie Laltte. The esterel synchronous programming language: Design, semantics, implementation, 1992.

[37] Bernard Berthomieu and Michel Diaz. Modeling and verification of time dependent systems using time petri nets. *IEEE Trans. Software Eng.*, 17(3):259–273, 1991.

[38] Dirk Beyer, Claus Lewerentz, and Andreas Noack. Rabbit: A tool for BDD-based verification of real-time systems. In *In: Computer-Aided Verification (CAV 2003), Volume 2725 of Lecture Notes in Computer Science, Springer-Verlag*, pages 122–125. Springer-Verlag, 2003.

[39] R.H. Bishop. *LabVIEW 8 Student Edition*. National instruments. Pearson Prentice Hall, 2007.

[40] Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of network-on-chip. *ACM Comput. Surv.*, 38(1), June 2006.

[41] Jasmin Blanchette and Mark Summerfield. *C++ GUI Programming with Qt 4 (2nd Edition) (Prentice Hall Open Source Software Development Series)*. Prentice Hall, 2 edition, February 2008.

[42] Roderick Bloem, Krishnendu Chatterjee, Thomas A. Henzinger, and Barbara Jobstmann. Better quality in synthesis through quantitative objectives. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 140–156. Springer, 2009.

[43] Borzoo Bonakdarpour, Marius Bozga, Mohamad Jaber, Jean Quilbeuf, and Joseph Sifakis. A framework for automated distributed implementation of component-based models. *Distributed Computing*, Mar 2012.

[44] Sébastien Bornot, Joseph Sifakis, and Stavros Tripakis. Modeling urgency in timed systems. In Willem P. de Roever, Hans Langmaack, and Amir Pnueli, editors, *COMPOS*, volume 1536 of *Lecture Notes in Computer Science*, pages 103–129. Springer, 1997.

[45] P. Bouyer. *From Qualitative to Quantitative Analysis of Timed Systems*. Mémoire d'habilitation, Université Paris 7, 2009.

[46] Patricia Bouyer, Uli Fahrenberg, Kim G. Larsen, and Nicolas Markey. Quantitative analysis of real-time systems using priced timed automata. *Commun. ACM*, 54(9):78–87, 2011.

[47] M. Bozga and Y. Lakhnech. IF-2.0: Common Language Operational Semantics. Technical report, Verimag, 2002.

[48] Marius Bozga, Susanne Graf, and Laurent Mounier. If-2.0: A validation environment for component-based real-time systems. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 343–348. Springer, July 2002.

[49] Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis. The if toolset. In *SFM*, pages 237–267, 2004.

[50] Marius Bozga, Mohamad Jaber, and Joseph Sifakis. Source-to-source architecture transformation for performance optimization in BIP. In *IEEE Fourth International Symposium on Industrial Embedded Systems - SIES 2009, Ecole Polytechnique Federale de Lausanne, Switzerland, July 8 - 10, 2009*, pages 152–160. IEEE, 2009.

[51] E. Brinksma, H. Hermanns, and J.-P. Katoen, editors. *Lectures on Formal Methods and Performance Analysis*, volume 2090 of *LNCS*. Springer, 2001.

[52] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, June 1984.

[53] Joseph T. Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogenous systems. *Int. Journal in Computer Simulation*, 4(2):0–, 1994.

[54] Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2005.

[55] L. Carnevali, L. Grassi, and E. Vicario. State-density functions over DBM domains in the analysis of non-Markovian models. *IEEE Trans. Software Eng.*, 35(2):178–194, 2009.

[56] Christos G. Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems*. Springer Publishing Company, Incorporated, 2nd edition, 2010.

[57] Pavol Cerný and Thomas A. Henzinger. From boolean to quantitative synthesis. In Samarjit Chakraborty, Ahmed Jerraya, Sanjoy K. Baruah, and Sebastian Fischmeister, editors, *EMSOFT*, pages 149–154. ACM, 2011.

[58] Zhou Chaochen. Duration calculus, a logical approach to real-time systems. In Armando Martin Haeberer, editor, *AMAST*, volume 1548 of *Lecture Notes in Computer Science*, pages 1–7. Springer, 1998.

[59] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[60] Don Cherepacha and David Lewis. Dp-fpga: An fpga architecture optimized for datapaths. *VLSI Design*, 4(4):329–343, 1996.

[61] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and Alberto L. Sangiovanni-Vincentelli. A formal specification model for hardware/software codesign. Technical Report UCB/ERL M93/48, EECS Department, University of California, Berkeley, 1993.

[62] Edmund M. Clarke, Alexandre Donzé, and Axel Legay. On simulation-based probabilistic model checking of mixed-analog circuits. *Formal Methods in System Design*, 36(2):97–113, 2010.

[63] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. Mit Press, 1999.

[64] E.G. Coffman. *Computer and Job-shop Scheduling Theory*. Wiley, 1976.

[65] Multicore Association Communications. The multicore association communications api.

[66] M. Coppola et al. Spidergon: a novel on-chip communication network. In *Proceedings of International Symposium on System-on-Chip*, 2004.

[67] T.T. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to algorithms*. MIT Press, 1990.

[68] David Culler, J. P. Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/-Software Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, 1 edition, August 1998.

[69] William J. Dally and Charles L. Seitz. The torus routing chip. *Distributed Computing*, 1(4):187–196, 1986.

[70] Pedro R. D'Argenio, Holger Hermanns, Joost-Pieter Katoen, and Ric Klaren. Modest - a modelling and description language for stochastic timed systems. In *Proceedings of the Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification*, PAPM-PROBMIV '01, pages 87–104, London, UK, UK, 2001. Springer-Verlag.

[71] Alexandre David, John Håkansson, Kim G. Larsen, and Paul Pettersson. Minimal dbm substraction. In Paul Pettersson and Wang Yi, editors, *Nordic Workshop on Programming Theory*, number 2004-041 in IT Technical Report of Uppsala University, pages 17–20, October 2004.

[72] Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikucionis, Danny Bogsted Poulsen, Jonas van Vliet, and Zheng Wang. Statistical model checking for networks of priced timed automata. In Uli Fahrenberg and Stavros Tripakis, editors, *FORMATS*, volume 6919 of *Lecture Notes in Computer Science*, pages 80–96. Springer, 2011.

[73] Conrado Daws, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. The tool KRONOS. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Hybrid Systems*, volume 1066 of *Lecture Notes in Computer Science*, pages 208–219. Springer, 1995.

[74] Conrado Daws and Sergio Yovine. Reducing the number of clock variables of timed automata. In *IEEE Real-Time Systems Symposium*, pages 73–81. IEEE Computer Society, 1996.

[75] E. A. de Kock, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, K. A. Vissers, and G. Essink. Yapi: application modeling for signal processing systems. In *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, pages 402–405, New York, NY, USA, 2000. ACM.

[76] Aldric Degorre and Oded Maler. On scheduling policies for streams of structured jobs. In Franck Cassez and Claude Jard, editors, *FORMATS*, volume 5215 of *Lecture Notes in Computer Science*, pages 141–154. Springer, 2008.

[77] Volker Diekert. *The Book of Traces*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1995.

[78] D.L. Dill. Timing assumptions and verification of finite-state concurrent systems. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 1989.

[79] J. L. Doob. *Stochastic Processes (Wiley Classics Library)*. Wiley-Interscience, January 1990.

[80] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. In *PROCEEDINGS OF THE IEEE*, pages 366–390. IEEE, 1999.

[81] Stephen Edwards, Luciano Lavagno, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. Readings in hardware/software co-design. In Giovanni De Micheli, Rolf Ernst, and Wayne Wolf, editors, *Readings in Hardware/Software Co-Design*, chapter Design of embedded systems: formal models, validation, and synthesis, pages 86–107. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[82] Stephen A. Edwards and Edward A. Lee. The semantics and execution of a synchronous block-diagram language. *Sci. Comput. Program.*, 48(1):21–42, 2003.

[83] Noel Eisley, Vassos Soteriou, and Li-Shiuan Peh. High-level power analysis for multi-core chips. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '06, pages 389–400, New York, NY, USA, 2006. ACM.

[84] Johan Eker, Jorn Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.

[85] Haytham Elmiligi, Ahmed A. Morgan, M. Watheq El-Kharashi, and Fayez Gebali. Power optimization for application-specific networks-on-chips: A topology-based approach. *Microprocess. Microsyst.*, 33(5-6):343–355, August 2009.

[86] J. Esparza and K. Heljanko. *Unfoldings – A Partial-Order Approach to Model Checking*. Springer, 2008.

[87] Peter H. Feiler, David P. Gluch, and John J. Hudak. The architecture analysis & design language (AADL): An introduction. Technical Report CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, 2006.

[88] Masahiro Fujita and Hiroshi Nakamura. The standard specc language. In *Proceedings of the 14th international symposium on Systems synthesis*, ISSS '01, pages 81–86, New York, NY, USA, 2001. ACM.

[89] Daniel D. Gajski, Samar Abdi, Andreas Gerstlauer, and Gunar Schirner. *Embedded System Design: Modeling, Synthesis and Verification*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[90] D.D. Gajski. *Specification and design of embedded systems*. PTR Prentice Hall, 1994.

[91] R. German. Non-markovian analysis. In Ed Brinksma, Holger Hermanns, and Joost-Pieter Katoen, editors, *European Educational Forum: School on Formal Methods and Performance Analysis*, volume 2090 of *Lecture Notes in Computer Science*, pages 156–182. Springer, 2000.

[92] Frank Ghenassia. *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[93] P.W. Glynn. A GSMP formalism for discrete event systems. *Proceedings of the IEEE*, 77(1):14–23, 1989.

[94] Luís Gomes and João Paulo Barros. Models of computation for embedded systems. In *The Industrial Information Technology Handbook*, pages 1–17. CRC Press, Inc., 2005.

[95] David Goodger. An introduction to restructuredtext, 2012.

[96] Ian Grout. *Digital Systems Design with FPGAs and CPLDs*. Newnes, Newton, MA, USA, 2008.

[97] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. Fixed-priority multiprocessor scheduling with liu and layland's utilization bound. In *Proceedings of the 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '10, pages 165–174, Washington, DC, USA, 2010. IEEE Computer Society.

[98] Pierre Guerrier and Alain Greiner. A generic architecture for on-chip packet-switched interconnections. In *DATE*, pages 250–256. IEEE Computer Society, 2000.

[99] Soonhoi Ha. Model-based programming environment of embedded software for mpsoc. In *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, ASP-DAC '07, pages 330–335, Washington, DC, USA, 2007. IEEE Computer Society.

[100] Soonhoi Ha, Sungchan Kim, Choonseung Lee, Youngmin Yi, Seongnam Kwon, and Young-Pyo Joo. Peace: A hardware-software codesign environment for multimedia embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 12(3):24:1–24:25, May 2008.

[101] Soonhoi Ha, Choonseung Lee, Youngmin Yi, Seongnam Kwon, and Young-Pyo Joo. Hardware-software codesign of multimedia embedded systems: the peace. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '06, pages 207–214, Washington, DC, USA, 2006. IEEE Computer Society.

[102] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

[103] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*. McGraw-Hill, 1998.

[104] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[105] Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst. System level performance analysis - the symta/s approach. In *IEE Proceedings Computers and Digital Techniques*, 2005.

[106] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Inf. Comput.*, 111(2):193–244, 1994.

[107] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[108] Jingcao Hu and Radu Marculescu. Energy-aware mapping for tile-based noc architectures under performance constraints. In *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, ASP-DAC '03, pages 233–239, New York, NY, USA, 2003. ACM.

[109] John D. Hunter. Matplotlib: A 2D Graphics Environment. *Computing in Science and Engineering*, 9(3):90–95, May 2007.

[110] IDT. Idt peripheral bus (ipbus). intermodule connection technology enables broad range of system-level integration, 2002.

[111] Intel. Intel strongarm sa-1100 microprocessor technical reference manual. Technical report, Intel, March 1999.

[112] Axel Jantsch. *Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[113] Ahmed A. Jerraya, Aimen Bouchhima, and Frédéric Pétrot. Programming models and hw-sw interfaces abstraction for multi-processor soc. In *Proceedings of the 43rd annual Design Automation Conference*, DAC '06, pages 280–285, New York, NY, USA, 2006. ACM.

[114] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–.

[115] M. Jurdzinski, D. Peled, and H. Qu. Calculating probabilities of real-time test cases. In *FATES*, pages 134–151, 2005.

[116] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY, 1974.

[117] Faraydon Karim, Anh Nguyen, and Sujit Dey. An interconnect architecture for networking systems on chips. *IEEE Micro*, 22(5):36–45, 2002.

[118] D. Kartson, G. Balbo, S. Donatelli, G. Franceschinis, and G. Conte. *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons, 1994.

[119] Jean-Francois Kempf, Marius Bozga, and Oded Maler. Performance evaluation of schedulers in a probabilistic setting. In Uli Fahrenberg and Stavros Tripakis, editors, *FORMATS*, volume 6919 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2011.

[120] Yonit Kesten, Amir Pnueli, Joseph Sifakis, and Sergio Yovine. Integration graphs: A class of decidable hybrid systems. In Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 179–208. Springer, 1992.

[121] Kurt Keutzer, Sharad Malik, Richard Newton, Jan Rabaey, and Alberto Sangiovanni-Vincentelli. System Level Design: Orthogonolization of Concerns and Platform-Based Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12), December 2000.

[122] Bart Kienhuis, Ed F. Deprettere, Pieter van der Wolf, and Kees A. Vissers. A methodology to design programmable embedded systems - the y-chart approach. In *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation - SAMOS*, pages 18–37, London, UK, UK, 2002. Springer-Verlag.

[123] Dohyung Kim and Soonhoi Ha. Static analysis and automatic code synthesis of flexible fsm model. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, ASP-DAC '05, pages 161–165, New York, NY, USA, 2005. ACM.

[124] Somayyeh Koohi, Mohammad Mirza-Aghatabar, Shaahin Hessabi, and Masoud Pedram. High-level modeling approach for analyzing the effects of traffic models on power and throughput in mesh-based nocs. In *Proceedings of the 21st International Conference on VLSI Design*, VLSID '08, pages 415–420, Washington, DC, USA, 2008. IEEE Computer Society.

[125] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, 1997.

[126] Ari Kulmala, Erno Salminen, and Timo D. Hämäläinen. Distributed bus arbitration algorithm comparison on fpga-based mpeg-4 multiprocessor system on chip. *IET Computers & Digital Techniques*, 2(4):314–325, 2008.

[127] Shashi Kumar, Axel Jantsch, Mikael Millberg, Johnny Öberg, Juha-Pekka Soininen, Martti Forsell, Kari Tiensyrjä, and Ahmed Hemani. A network on chip architecture and design methodology. In *ISVLSI*, pages 117–124, 2002.

[128] Kim Guldstrand Larsen, Gerd Behrmann, Ed Brinksma, Ansgar Fehnker, Thomas Hune, Paul Pettersson, and Judi Romijn. As cheap as possible: Efficient cost-optimal reachability for priced timed automata. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 493–505. Springer, 2001.

[129] Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Efficient verification of real-time systems: compact data structure and state-space reduction. In *IEEE Real-Time Systems Symposium*, pages 14–24. IEEE Computer Society, 1997.

[130] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1-2):134–152, 1997.

[131] Luciano Lavagno, Alberto Sangiovanni-Vincentelli, and Ellen Sentovich. Models of computation for embedded system design. In *in System-Level Synthesis*, pages 45–102. Kluwer Academic Publishers, 1998.

[132] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*. Springer-Verlag, Berlin, Heidelberg, 2001.

[133] Edward A. Lee. Finite state machines and modal models in ptolemy ii. Technical Report UCB/EECS-2009-151, EECS Department, University of California, Berkeley, Nov 2009.

[134] Edward A. Lee and David G. Messerschmitt. Synchronous data flow: Describing signal processing algorithm for parallel computation. In *COMPCON*, pages 310–315, 1987.

[135] Edward A. Lee and Haiyang Zheng. Operational semantics of hybrid systems. In *Hybrid Systems: Computation and Control (HSCC), volume LNCS 3414*, pages 25–53. Springer-Verlag, 2005.

[136] Jaejin Lee, Jungwon Kim, Sangmin Seo, Seungkyun Kim, Jungho Park, Honggyu Kim, Thanh Tuan Dao, Yongjin Cho, Sung Jong Seo, Seung Hak Lee, Seung Mo Cho, Hyo Jung Song, Sang-Bum Suh, and Jong-Deok Choi. An opencl framework for heterogeneous multicores with local memory. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 193–204, New York, NY, USA, 2010. ACM.

[137] Seung Eun Lee and Nader Bagherzadeh. A high level power model for network-on-chip (noc) router. *Comput. Electr. Eng.*, 35(6):837–845, November 2009.

[138] Julien Legriel, Colas Le Guernic, Scott Cotton, and Oded Maler. Approximating the pareto front of multi-criteria optimization problems. In Rupak Majumdar Javier Esparza, editor, *TACAS*, volume 6015 of *LNCS*, pages 69–83. Springer, 2010.

[139] Julien Legriel and Oded Maler. Meeting deadlines cheaply. In *Proceedings of the 2011 23rd Euromicro Conference on Real-Time Systems*, ECRTS '11, pages 185–194, Washington, DC, USA, 2011. IEEE Computer Society.

[140] Rainer Leupers and Olivier Temam. *Processor and System-on-Chip Simulation*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[141] C.L. Liu and James Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment, 1973.

[142] Jane W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[143] D. Lugiez, P. Niebert, and S. Zennou. A partial order semantics approach to the clock explosion problem of timed automata. *Theoretical Computer Science*, 345:27–59, 2005.

[144] O. Maler. On optimal and reasonable control in the presence of adversaries. *Annual Reviews in Control*, 31(1):1–15, 2007.

[145] O. Maler, K.G. Larsen, and B.H. Krogh. On zone-based analysis of duration probabilistic automata. In Yu-Fang Chen and Ahmed Rezine, editors, *INFINITY*, volume 39 of *EPTCS*, pages 33–46, 2010.

[146] Oded Maler. On under-determined dynamical systems. In Samarjit Chakraborty, Ahmed Jerraya, Sanjoy K. Baruah, and Sebastian Fischmeister, editors, *EMSOFT*, pages 89–96. ACM, 2011.

[147] César A. M. Marcon, Ney Laert Vilar Calazans, Fernando Gehm Moraes, Altamiro Amadeu Susin, Igor M. Reis, and Fabiano Hessel. Exploring noc mapping strategies: An energy and timing aware technique. In *DATE*, pages 502–507, 2005.

[148] Peter Marwedel. *Embedded system design*. Kluwer, 2003.

[149] The Mathworks. *MATLAB Simulink Student Version 2012a*. Pearson Academic Computing, 2012.

[150] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[151] Milica Mitic and Mile Stojcev. An overview of on-chip buses. *Facta universitatis - series: Electronics and Energetics*, 19(3):405–428, 2006.

[152] S. Mohanty, V. K. Prasanna, S. Neema, and J. Davis. Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems: software and compilers for embedded systems*, LCTES/SCOPES '02, pages 18–27, New York, NY, USA, 2002. ACM.

[153] Edward F. Moore. Gedanken-experiments on sequential machines. In Claude Shannon and John McCarthy, editors, *Automata Studies*, pages 129–153. Princeton University Press, Princeton, NJ, 1956.

[154] MPI Forum. MPI: A Message-Passing Interface Standard. Version 2.2, September 4th 2009.

[155] Dejan Nickovic and Oded Maler. Amt: A property-based monitoring tool for analog systems. In *FORMATS*, pages 304–319, 2007.

[156] Xavier Nicollin and Joseph Sifakis. The algebra of timed processes, atp: Theory and application. *Inf. Comput.*, 114(1):131–178, 1994.

[157] NXP. Nexperia pnx15xx/952x series data book, December 2007.

[158] Ümit Y. Ogras, Paul Bogdan, and Radu Marculescu. An analytical approach for network-on-chip performance analysis. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 29(12):2001–2013, 2010.

[159] Partha Pratim Pande, Cristian Grecu, André Ivanov, and Res Saleh. Design of a switch for network on chip applications. In *ISCAS (5)*, pages 217–220, 2003.

[160] L. C. Paulson. The foundation of a generic theorem prover. *J. Autom. Reason.*, 5(3):363–397, September 1989.

[161] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für instrumentelle Mathematik, Bonn, 1962.

[162] Andy D. Pimentel, Cagkan Erbas, and Simon Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. Computers*, 55(2):99–112, 2006.

[163] Andy D. Pimentel, Mark Thompson, Simon Polstra, and Cagkan Erbas. Calibration of abstract performance models for system-level design space exploration. *J. Signal Process. Syst.*, 50(2):99–114, February 2008.

[164] Anne Vinter Ratzer, Lisa Wells, Henry Michael Lassen, Mads Laursen, Jacob Frank Qvortrup, Martin Stig Stissing, Michael Westergaard, Soren Christensen, and Kurt Jensen. Cpn tools for editing, simulating, and analysing coloured petri nets. In *Proceedings of the 24th international conference on Applications and theory of Petri nets*, ICATPN'03, pages 450–462, Berlin, Heidelberg, 2003. Springer-Verlag.

[165] George M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theor. Comput. Sci.*, 58:249–261, 1988.

[166] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. In Ales Leonardis, Horst Bischof, and Axel Pinz, editors, *ECCV (1)*, volume 3951 of *Lecture Notes in Computer Science*, pages 430–443. Springer, 2006.

[167] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual*. Addison-Wesley Professional, 2nd edition, 2010.

[168] Selma Saidi, Pranav Tendulkar, Thierry Lepley, and Oded Maler. Optimizing explicit data transfers for data parallel applications on the cell architecture. *ACM Trans. Archit. Code Optim.*, 8(4):37:1–37:20, January 2012.

[169] Roberto Segala. A compositional trace-based semantics for probabilistic automata. In *CONCUR*, pages 234–248, 1995.

[170] Roberto Segala. *Modeling and verification of randomized distributed real-time systems*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, 1995. Not available from Univ. Microfilms Int.

[171] Roberto Segala and Nancy A. Lynch. Probabilistic simulations for probabilistic processes. *Nord. J. Comput.*, 2(2):250–273, 1995.

[172] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vencentelli. Addressing the system-on-a-chip interconnect woes through communication-based design. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 667–672, New York, NY, USA, 2001. ACM.

[173] Marco Sgroi, Luciano Lavagno, and Alberto Sangiovanni-Vincentelli. Formal models for embedded system design. *IEEE Des. Test*, 17(2):14–27, April 2000.

[174] Joseph Sifakis and Sergio Yovine. Compositional specification of timed systems (extended abstract). In Claude Puech and Rüdiger Reischuk, editors, *STACS*, volume 1046 of *Lecture Notes in Computer Science*, pages 347–359. Springer, 1996.

[175] Oliver Sinnen. *Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2007.

[176] IEEE Computer Society. *IEEE Standards Intepretations: IEEE Std 1076-1987, IEEE Standard VHDL Language Reference Manual*. IEEE Standards Office, New York, NY, USA, 1992.

[177] STMicroelectonics and CEA. Platform 2012: A many-core programmable accelerator for ultra-efficient embedded computing in nanometer technology. Technical report, STMicro-Electronics CEA, 2010.

[178] STMicroelectronics. Nomadik - open multimedia platform for next generation mobile devices. Technical report, STMicroElectronics, 2004.

[179] Sander Stuijk, Marc Geilen, and Twan Basten. SDF^3: SDF for free. In *ACSD*, pages 276–278. IEEE Computer Society, 2006.

[180] Stuart Sutherland, Simon Davidmann, and Peter Flake. *SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling*. Springer Publishing Company, Incorporated, 2nd edition, 2010.

[181] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *in ISCAS*, pages 101–104, 2000.

[182] Lothar Thiele and Ernesto Wandeler. Performance analysis of distributed embedded systems, 2005.

[183] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In R. Nigel Horspool, editor, *CC*, volume 2304 of *Lecture Notes in Computer Science*, pages 179–196. Springer, 2002.

[184] Donald E. Thomas and Philip R. Moorby. *The VERILOG Hardware Description Language*. Kluwer Academic Publishers, Norwell, MA, USA, 3rd edition, 1996.

[185] TI. Ti omap.

[186] Ken Tindell and John Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocess. Microprogram.*, 40(2-3):117–134, April 1994.

[187] Frank Vahid and Tony D. Givargis. *Embedded System Design: A Unified Hardware/Software Introduction*. Wiley, international student edition edition, October 2001.

[188] Pieter van der Wolf, Erwin de Kock, Tomas Henriksson, Wido Kruijtzer, and Gerben Essink. Design and programming of embedded multiprocessors: an interface-centric approach. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '04, pages 206–217, New York, NY, USA, 2004. ACM.

[189] Enrico Vicario, Luigi Sassoli, and Laura Carnevali. Using stochastic state classes in quantitative evaluation of dense-time reactive systems. *IEEE Trans. Software Eng.*, 35(5):703–719, 2009.

[190] Jelte Peter Vink, Kees van Berkel, and Pieter van der Wolf. Performance analysis of soc architectures based on latency-rate servers. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '08, pages 200–205, New York, NY, USA, 2008. ACM.

[191] Ernesto Wandeler, Lothar Thiele, Marcel Verhoef, and Paul Lieverse. System architecture evaluation using modular performance analysis: a case study. *STTT*, 8(6):649–667, 2006.

[192] Farn Wang. Efficient verification of timed automata with BDD-like data structures. *STTT*, 6(1):77–97, 2004.

[193] J. Wang, Y. Li, and Q. Peng. A Novel Analytical Model for Network-on-Chip using Semi-Markov Process. *Advances in Electrical and Computer Engineering*, 11(1):111–118, 2011.

[194] Tim Weilkiens. *Systems Engineering with SysML/UML: Modeling, Analysis, Design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[195] Pierre Wodey, Geoffrey Camarroque, Richard Hersemeule, and Jean-Philippe Cousin. Lotos code generation for model checking of stbus based soc: the stbus interconnect. In *Proceedings of the First ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, MEMOCODE '03, pages 204–, Washington, DC, USA, 2003. IEEE Computer Society.

[196] W. Wolf, A.A. Jerraya, and G. Martin. Multiprocessor system-on-chip (mpsoc) technology. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(10):1701–1713, Oct. 2008.

[197] Wayne Wolf. The future of multiprocessor systems-on-chips. In Sharad Malik, Limor Fix, and Andrew B. Kahng, editors, *DAC*, pages 681–685. ACM, 2004.

[198] Wayne Wolf. *Computers as Components, Second Edition: Principles of Embedded Computing System Design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2008.

[199] Jinwen Xi and Peixin Zhong. A transaction-level noc simulation platform with architecture-level dynamic and leakage energy models. In *Proceedings of the 16th ACM Great Lakes symposium on VLSI*, GLSVLSI '06, pages 341–344, New York, NY, USA, 2006. ACM.

[200] Wang Yi. Ccs + time = an interleaving model for real time systems. In Javier Leach Albert, Burkhard Monien, and Mario Rodríguez-Artalejo, editors, *ICALP*, volume 510 of *Lecture Notes in Computer Science*, pages 217–228. Springer, 1991.

[201] Håkan L. S. Younes and Reid G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 223–235. Springer, 2002.

[202] S. Yovine. Kronos: A verification tool for real-time systems. *STTT*, 1(1-2):123–133, 1997.

[203] Sergio Yovine. *Methodes et outils pour la verification symbolique de systemes temporises*. PhD thesis, Institut National Polytechnique de Grenoble, France, May 1993.

[204] J. Zhao. Partial order path technique for checking parallel timed automata. In *FTRTFT*, pages 417–432, 2002.