Johanna Tuominen

Formal Power Analysis of Systems-on-Chip

Turku Centre for Computer Science

TUCS Dissertations
No 121, November 2009

Formal Power Analysis of Systems-on-Chip

Johanna Tuominen

To be presented, with the permission of the Faculty of Mathematics and Natural Sciences of the University of Turku, for public criticism in Auditorium Lambda on December 7, 2009, at 12 noon.

University of Turku
Department of Information Technology
University of Turku, FI-20014 Turku, Finland

2009

Supervisors

Adjunct professor Juha Plosila
Department of Information Technology
University of Turku
FI-20014, Turku
Finland

D.Sc. (Tech.) Tomi Westerlund
Department of Information Technology
University of Turku
FI-20014, Turku
Finland

Reviewers

Associate Professor Paul Pop
Department of Informatics and Mathematical Modeling
Technical University of Denmark
DK-2800 Kongens Lyngby
Denmark

Senior Researcher, Ph.D. Luigia Petre
Department of Information Technologies
Åbo Akademi University
Joukahaisenkatu 3-5 A, FI-20520 Turku
Finland

Opponent

Associate Professor Koen Claessen
Department of Computer Science and Engineering
Chalmers University of Technology
41296 Gothenburg
Sweden

ISBN 978-952-12-2356-3 (PRINT)
ISBN 978-952-12-2357-0 (PDF)
ISSN 1239-1883

Abstract

The design methods and languages targeted to modern System-on-Chip designs are facing tremendous pressure of the ever-increasing complexity, power, and speed requirements. To estimate any of these three metrics, there is a trade-off between accuracy and abstraction level of detail in which a system under design is analyzed. The more detailed the description, the more accurate the simulation will be, but, on the other hand, the more time consuming it will be. Moreover, a designer wants to make decisions as early as possible in the design flow to avoid costly design backtracking.

To answer the challenges posed upon System-on-chip designs, this thesis introduces a formal, power aware framework, its development methods, and methods to constraint and analyze power consumption of the system under design. This thesis discusses on power analysis of synchronous and asynchronous systems not forgetting the communication aspects of these systems. The presented framework is built upon the Timed Action System formalism, which offer an environment to analyze and constraint the functional and temporal behavior of the system at high abstraction level. Furthermore, due to the complexity of System-on-Chip designs, the possibility to abstract unnecessary implementation details at higher abstraction levels is an essential part of the introduced design framework. With the encapsulation and abstraction techniques incorporated with the procedure based communication allows a designer to use the presented power aware framework in modeling these large scale systems. The introduced techniques also enable one to subdivide the development of communication and computation into own tasks. This property is taken into account in the power analysis part as well. Furthermore, the presented framework is developed in a way that it can be used throughout the design project. In other words, a designer is able to model and analyze systems from an abstract specification down to an implementable specification.

Acknowledgements

It gives me great pleasure to conclude many years of work by expressing my gratitude to the individuals and institutions that have influenced the research presented in this thesis. First of all, I would like to thank my supervisors Juha Plosila and Tomi Westerlund for their expert guidance, support, and insight during my doctoral studies. I wish to extend special thanks to professor Kaisa Sere for all her encouragement and support at early stages of my studies.

I also wish to thank associate professor Paul Pop from Technical University of Denmark and Ph.D. Luigia Petre from Åbo Akademi for their detailed reviews of this thesis and for providing constructive comments and suggestions for improvement. Their valuable comments and recommendations help me to improve my thesis.

I gratefully acknowledge the financial support that made my Ph.D. thesis possible. I wish to thank the Turku Centre for Computer Science (TUCS), Department of Information Technology (University of Turku), and Department of Information Technologies (Åbo Akademi) for financing my studies. In addition, I wish to express my sincerest gratitude to Ulla Tuominen Foundation and the Turku University Foundation for their financial support through their scholarship and travel grant.

I wish to express my gratitude towards colleagues, office workers, and technical staff in the Department of Information Technology (University of Turku) and at the TUCS for providing various technical and administrative support and enjoyable coffee breaks during all these years. Especially I would like to thank the colleagues from Computer Systems laboratory for creating excellent and inspiring working environment. I wish also thank Tomi Westerlund, Tero Säntti, and Juha Plosila for co-authoring papers with me.

Finally I wish to thank all my friends and my closest relatives to show me that there is life outside the academic environment. I especially wish to

thank my parents for their wholehearted support and encouragement during my doctoral studies and throughout my life. I also wish to extend the special thanks to all the wonderful people in Hayashi and Three Beers are thanked for enjoyable discussions of science, life, and everything in between them during these years.

In conclusion and most of all, I thank Tero Säntti for his companionship for better or for worse during these years. Your endless love and support gave me the strength to make this work an actual thesis.

Turku, October 2009

Johanna Tuominen

Contents

1 Introduction **1**
1.1 Objectives . . . 3
1.2 List of Publications . . . 5
1.3 Overview of the Thesis . . . 6
1.4 Related Work . . . 6

2 Action Systems **13**
2.1 Actions . . . 13
2.2 Semantics of Actions . . . 15
2.3 Non-Atomic Operation of Actions . . . 17
2.4 Properties of Actions . . . 18
2.5 Action Systems . . . 19
2.5.1 Parallel action systems . . . 21
2.5.2 Hierarchical action systems . . . 21
2.6 Procedures . . . 23
2.6.1 Procedure based communication . . . 24
2.7 Refinement . . . 25
2.7.1 Data refinement . . . 26
2.7.2 Refinement of action systems . . . 26
2.7.3 Development of action systems . . . 28
2.8 Chapter Summary . . . 29

3 Timed Action Systems **31**
3.1 Timed Actions . . . 31
3.1.1 Interpretation of system behavior . . . 32
3.1.2 Components of timed action . . . 32
3.1.3 Timed action state predicates . . . 35
3.1.4 Weakest precondition of a timed action . . . 35
3.2 Non-atomic Operation of Timed Actions . . . 36
3.3 Synchronous Operation . . . 36
3.4 Timed Action System . . . 39
3.4.1 Computation model . . . 41

3.4.2 Timed action semantics 41
3.4.3 Computation path 43
3.5 Delay 45
3.5.1 Delay predicates 46
3.5.2 Delay calculation rules 46
3.5.3 Procedure delay 47
3.5.4 Procedure based communication delay 47
3.5.5 Computation path delay 49
3.6 Chapter Summary 50

4 Power Estimation 51
4.1 Area Complexity of an Action 52
4.1.1 Area complexity of variables 54
4.1.2 Area complexity of non-deterministic assignment . . . 54
4.1.3 Area complexity of a guarded action 58
4.1.4 Calculation rules for area complexity 58
4.1.5 Tuning the area complexity model 60
4.2 Average Power of Timed Actions 63
4.2.1 Energy 63
4.2.2 Dynamic Power Consumption 65
4.2.3 Static power consumption 66
4.2.4 Power consumption of timed action 67
4.3 Power Modeling at System Level 67
4.3.1 Area complexity of Action Systems 68
4.3.2 Average power of Timed Action Systems 68
4.3.3 Power dissipation in computation paths 73
4.4 Power Consumption of Killed Timed Actions 76
4.5 Instantaneous Power Dissipation 82
4.6 Discussion on Models 84
4.7 Chapter Summary 86

5 Specification of System Constraints 87
5.1 Deadline 87
5.2 Area Complexity 88
5.3 Power Consumption 88
5.3.1 Timed action 88
5.3.2 Timed action system 89
5.4 Chapter Summary 90

6 Development of Systems 91
6.1 System Requirements 91
6.2 Refinement 92
6.2.1 Refinement of Timed Action Systems 93

6.2.2 Power aware refinement 95
6.2.3 Refinement of constraints 100
6.2.4 Decomposition 111
6.3 Chapter Summary 113

7 Analyzing System Models 115
7.1 Synchronous Systems 115
7.1.1 Power analysis of synchronous timed actions 116
7.1.2 Synchronous timed action systems 121
7.2 Asynchronous Systems 126
7.3 Communication Networks 131
7.3.1 Modeling communication networks 132
7.3.2 Ring based communication network 140
7.4 Chapter Summary 144

8 Experiment 145
8.1 Background 145
8.2 The Formal Model of the Co-Processor 146
8.2.1 Preliminaries 146
8.2.2 Co-processor system 148
8.3 Analysis 158
8.3.1 Analyzing memory components 159
8.3.2 Arithmetic logic unit 163
8.3.3 Communication structures 165
8.3.4 System level power modeling 170
8.4 Multicore Processor Approach 175
8.5 Multicore Framework 176
8.5.1 Threads 176
8.5.2 Forming the multiple co-processor model 176
8.6 On Analyzing Power Consumption 178
8.7 Chapter Summary 181

9 Conclusion 183
9.1 Future Work 186

List of Figures

1.1 Interaction between a system $\mathcal{A}$ and its environment $\mathcal{E}nv$ . . 4

2.1 Direct communication between action systems 24

3.1 The computation model of Timed Action Systems 42
3.2 The computation paths of the composition $A; ((B;C) \parallel (D;E))$ 45
3.3 Direct communication between timed action systems 48

4.1 Illustration of a non-deterministic assignment $x := x'.Q$ . . . 54
4.2 Tree model construction 55
4.3 Example area complexity modeling 57
4.4 Example area complexity modeling 57
4.5 Illustration of a guarded non-deterministic assignment 59
4.6 Dynamic power of a timed action 66
4.7 Example set of timed actions $\mathcal{A}_T$ 69
4.8 Execution of timed action A during observation period T . . 70
4.9 Illustration of the selected observation period 72
4.10 Parallel behavior of timed actions. 76
4.11 Timed action state predicates 77
4.12 Timed action state predicates kill operation 78
4.13 Example composition with kill 81
4.14 Illustration of time segments 83
4.15 Instantaneous power estimation per time segments 84

6.1 System / environment interaction 92
6.2 Observation period for the system $\mathcal{S}'$ 103
6.3 Closed timed action system $\mathcal{A}$ 103

7.1 An atomic synchronous composition 117
7.2 Asynchronous signaling protocols 127
7.3 Asynchronous operation of systems $\mathcal{A}rb$ and $\mathcal{B}us$ 129
7.4 Communication through point-to-point network $\mathcal{N}$ 133
7.5 Buffered point-to-point network 134
7.6 General communication network 137

7.7 Ring based communication 140

8.1 Illustration of the initial system construct $\mathcal{S}$ 147
8.2 Example execution sequence 148
8.3 Block diagram of the system $\mathcal{C}o - Proc$ 149
8.4 Subsystem $\mathcal{M}em$ and its interface towards other system blocks 151
8.5 $\mathcal{C}omm$ and its interface towards other system blocks 153
8.6 Block diagram of the subsystem $\mathcal{E}xec$ 155
8.7 Example threading schemes 177
8.8 Illustration of the multicore co-processor scene 178

List of Tables

1.1 Power consumption growth vs. technology decrease 2

4.1 Tuning the complexity factor. 63
4.2 Example operations and their complexity factor. 63
4.3 Results (area complexity). 85

7.1 Clock distribution performance metrics 116

8.1 Operations performed in $\mathcal{A}LU$. 159
8.2 Area Complexities of the system $\mathcal{C}o-proc$. 171
8.3 Area Complexities of $\mathcal{M}em$. 172
8.4 Area Complexities of the Memory Units. 173
8.5 Area Complexities of $\mathcal{C}omm$. 173
8.6 Area Complexities of $\mathcal{E}xec$. 173
8.7 Area Complexities $\mathcal{A}LU$. 174

List of Abbreviations

ALU Arithmetic Logic Unit

BDD Binary Decision Diagram

BLIF Berkeley Logic Interchange Format

CMOS Complementary Metal Oxide Semiconductor

CPU Central Processing Unit

FPGA Field Programmable Gate Array

HW Hardware

IC Integrated Circuit

IP Intellectual Property

IRQ Interrupt Request

ITRS International Technology Roadmap for Semiconductors

JIT Just In Time compilation

JVM Java Virtual Machine

MVSIS Multivalued Logic Synthesis

NoC Network-on-Chip

OBDD Ordered Binary Decision Diagram

PDA Personal Digital Assistant

RAM Random Access Memory

RTL Register Transfer Level

SW Software

SoC Systems-on-Chip

VHSIC Very High Speed Integrated Circuit

VHDL Very High Speed Integrated Circuit (VHSIC) Hardware Description Language

VIS Verification Interacting with Synthesis

VLSI Very Large Scale Integration

Chapter 1

Introduction

The design of complex chips has undergone a series of revolutions during the last two decades. In the 1980's a language based design approach and synthesis was introduced. In the 1990's there was the adoption of design reuse and Intellectual Property (IP) as a mainstream design practice. These technological advances made possible to integrate entire system into a single chip, hence the term System-on-Chip. After the millennium, the concept of deep submicron technology, from 130 nm down, made possible to integrate more and more gates into a single chip. This development lead to the situation where several systems were integrated into a single chip. This Systems-on-Chip (SoC) approach, however, poses a new set of design problems. That is, it is possible to implement tens of millions of gates on a reasonably small die leading to a power density and total power dissipation that is at the limits of what packaging, cooling, and other infrastructure can support.

The total power consumption of SoC consist of dynamic power consumption and static power consumption. The former is the power consumed when the device is active, that is, when signals are changing values. The latter is the power consumed when the device is powered up but no signals are changing values. In Complementary Metal Oxide Semiconductor (CMOS) devices, the primary source of the dynamic power consumption is the switching power, that is, the power required to charge and discharge the output capacitance of the gate. The static power consumption, however, is caused by the leakage current [49], which is the combination of the subthreshold leakage (a weak inversion current across the device), and the gate leakage (a tunneling current through the gate oxide insulation). As technology has shrunk to 90 nm and below, the leakage current is increasing dramatically to the point where, in 65 nm designs, the leakage current is nearly as large as the dynamic current [47]. The above described design challenges have a significant effect on how chips are designed. The power density of the highest performance chips has grown to the point where it is no longer pos-

Table 1.1: Power consumption growth vs. technology decrease

Node	90 nm	65 nm	45 nm
Dynamic Power per cm^2	$1x$	$1.4x$	$2x$
Static Power per cm^2	$1x$	$2.5x$	$6.5x$
Total Power per cm^2	$1x$	$2x$	$4x$

sible to increase the clock speed as the technology shrinks. As a result, multi-processor chips are designed instead of single-chip ultra high speed processor cores. The relationship between the growth in power density and the decrease in feature size is shown in Table 1.1 according to the International Technology Roadmap for Semiconductors (ITRS) [6]. In addition, for battery-powered devices, which comprise one of the fastest growing electronic market, the leakage is a major problem. According to ITRS the battery life for these devices peaked in 2004. After that, the battery life has been decreased as features have been added faster than the power (per feature) has been reduced.

Until recently, power has been the second order concern in chip design, following the first order issues such as timing, area, and cost. Today, for most SoC designs, a power budget is one of the most important design goals of the project [47]. That is, exceeding the power budget can be fatal to a project, whether it means changing a cheap plastic package with an expensive ceramic one, causing an unacceptable poor reliability due to an excessive power density, or failing to meet the required battery life.

Performance analysis of various system properties can be carried out at different abstraction levels during the design process. In general, there is a trade-off between accuracy on one hand and design time and cost on the other hand. Typically, lower level estimation tools offer greater estimation accuracy, but their use to explore architectural trade-offs tends to be time consuming. Higher level tools allow faster and less costly iteration cycles at the price of lower accuracy. High abstraction level functionality specification for both Hardware (HW) and Software (SW) is one of the design challenges to increase the design productivity in technologies above 32 nm according to the predictions made by ITRS [6]. However, technologies below 32 nm lists the complete formal verification of designs as their design challenge. Furthermore, the accuracy requirements set for high-level performance estimates (area, power, time) are assumed to be at least 70 % from the measured value by the end of 2010.

In general, the development of a SoC design starts from a high-level specification. A functional specification defines how the system should operate according to its inputs and a temporal behavior defines the timing requirements that must be satisfied by the system components. Physical properties (such as area and power) can be estimated based on both the functional behavior and the temporal behavior. The development phases must ensure that the end product reflects the abstract specification. The correctness of an implementation is traditionally verified using simulation based methods where unintended mismatches between the models cause new design cycles. This approach is error prone as well as time consuming. To overcome these tedious design cycles a promising alternative is to use formal methods with a stepwise refinement method. They provide means with which a high-level system specification can be transformed to an implementable model. To answer the design challenges presented above, formal methods provide an environment to specify, design, and verify systems with the benefits of rigorous mathematical basis.

1.1 Objectives

As the power consumption becomes one of the fundamental features of recent SoC designs, it is imperative to have reliable power analysis frameworks. The objective of this thesis is to define a power analysis framework based on the existing *the Timed Action Systems formalism* [86], that includes techniques to constrain and analyze the system. The developed framework the challenges described in the previous section, namely to provide:

- A framework to estimate a system's physical characteristics (area, power) at a high abstraction level with different system construct such as synchronous and asynchronous systems.
- Correctness preserving development of functional and physical properties.
- A framework to analyze power consumption within different communication constructs.

The Timed Action Systems formalism is an extension of the Action Systems formalism [20],that has been already successfully applied to the development of both synchronous [76] and asynchronous [67] SoC designs. Action Systems is a state-based formalism initially proposed by Back and Kurki-Suonio [17] and based on an extended version of a guarded command language introduced by Dijkstra [40]. It was first tailored to specification and correctness preserving development of reactive systems, but it has been thereafter successfully applied to SoC designs.

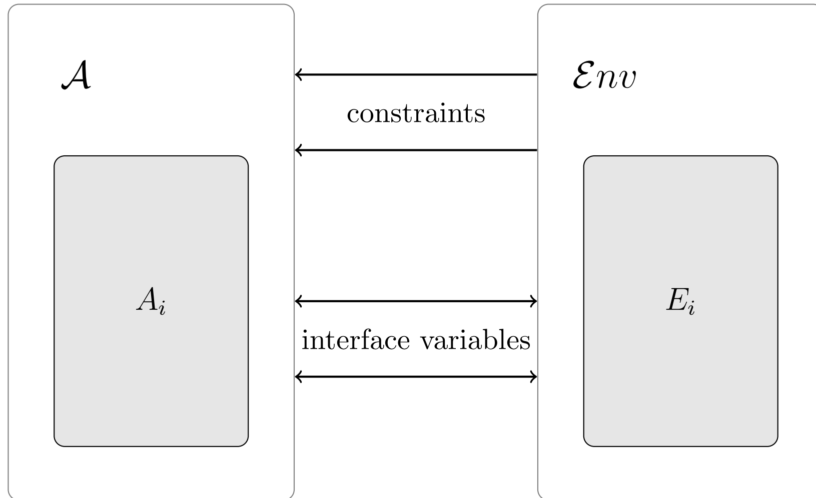


Figure 1.1: Interaction between a system $\mathcal{A}$ and its environment $\mathcal{E}nv$

Timed Action Systems provides an environment to reason about both functional and timing properties of the abstract system model. At high abstraction level, one is able to estimate the area and power dissipation of the system model by exploiting the properties of the formalism. The key components in the power estimation framework are area and time, because at this abstraction level there is no information of the technology or technology platform on which the design will be physically implemented. As mentioned above, the timing model is adopted from Timed Action Systems whereas the rules to estimate area are given in this thesis. Combining the area information and the timing information one is able to reason whether the requirements set for power dissipation holds or not. The requirements against which the results are verified are expressed in terms of constraints. Naturally, timing and area properties can be verified as well. In other words, constraints are used to steer the development of the system towards an implementation whose functional, timing, and physical characteristics fulfil the requirements given by the specification.

The correctness of both logical and temporal characteristics of the system under design are verified within the refinement calculus framework [22] and its time aware extension [86]. In this thesis, the time aware refinement calculus is further extended to ensure the physical properties of the system. That is, this thesis presents a refinement methodology with which the functional properties of the abstract system specification can be transformed using the standard refinement calculus framework towards a more concrete system preserving its timing and power characteristics.

In terms of power analysis, different system constructs have properties that demand specific discussion on how their power estimate is evaluated, for example, clocking in synchronous circuits. Regardless of which specific properties a system has the modeling environment is the same. The modeling environment describes the way the system and its environment interact. This is illustrated in Fig. 1.1, where a system $\mathcal{A}$ interacts with its environment $\mathcal{E}nv$. Furthermore, the actions A_i of the system A and E_i of the system E, operate in a *non-final* manner meaning that there is always at least one enabled action either in the system $\mathcal{A}$ or in its environment $\mathcal{E}nv$. The environment reads from the output variables of the system and writes onto the input variables of the system. Furthermore, the environment sets constraints for the system.

Communication in modern digital systems often forms a bottleneck in terms of time, area, and power. Therefore, separate models and analyze need to be applied for different communication structures. Communication inside an action system is assumed to be part of the area of the action system whereas communication between Action Systems requires longer interconnects. That is, separate timing and power analysis frameworks are required.

1.2 List of Publications

The work discussed in this thesis is based on and extended from the publications listed below:

1. Johanna Tuominen, Tomi Westerlund and Juha Plosila. Feasibility Report on Formal Area Complexity Estimation. In *TUCS Technical Report 907*, Turku Centre for Computer Science, Finland, August 2008.

2. Johanna Tuominen, Tomi Westerlund and Juha Plosila. Power Aware System Refinement. In *Electronic Notes in Theoretical Computer Science (ENTCS)*, volume 201C, pages 223-253, 2008.

3. Johanna Tuominen, Tomi Westerlund and Juha Plosila. Formal Power Analysis of On-Chip Communication. In *Brasilian Symposium on Formal Methods (SBMF)*, pages 87-102, 2007.

4. Johanna Tuominen, Tero Säntti and Juha Plosila. Towards a Formal Power Estimation Framework for Hardware Systems. In *International Symposium on System-On-Chip*, pages 96-99, 2005.

5. Johanna Tuominen and Juha Plosila. Formal Specification of an Asynchronous Viterbi Decoder. In *23rd IEEE Norchip Conference*, pages 214-217, 2005.

1.3 Overview of the Thesis

The rest of the thesis is organized as follows. Chapter 2 describes the Action Systems formalism, which is the formal basis of the time and power aware modeling framework. In Chapter 3 the time aware extension of Action Systems, Timed Action Systems is defined upon which the power modeling framework is constructed. The power modeling framework for Timed Action Systems is presented in Chapter 4. The constraints are presented in Chapter 5 and the refinement methodology for Timed Action Systems is described and extended to cover the power modeling aspects in Chapter 6. Chapter 7 applies the power modeling framework for different system models such as synchronous and asynchronous systems, and, furthermore, communication networks are introduced. The case study, described in Chapter 8, illustrates the usage of the formal power estimation with its modeling and development methods. Finally, Chapter 9 concludes the thesis and presents future directions for the formal power estimation framework.

1.4 Related Work

Recent years have shown, based on the active research carried out in the field, that there is a need for a rigorous development framework that operates at higher abstraction levels than the traditional approaches. That is, there is a need to evaluate the performance (time, area and power) of the system above Register Transfer Level (RTL) allowing us to detect performance related bugs earlier. The target application fields among the presented formalism varies from software systems to hardware systems and to embedded systems.

Time-aware Action Systems

Modeling passage of time in Action Systems is not a novel idea. That is, in addition to Timed Action Systems, there exists at least the following time-embedding extensions of Action Systems [86]: *Hybrid Action Systems* [73, 74] and *Continuous Action Systems* [18]. In [86], Timed Action Systems was compared with the above two time-embedding formalisms. The target environment of Timed Action Systems is digital SoC, whereas the target environment of Hybrid Action Systems and Continuous Action System is a hybrid system. The hybrid system is (typically) composed of digital programs that interact with an analog environment. From this it follows that the major problem to adopt Hybrid Action Systems (1) or Continuous Action Systems (2) was the definition of time: (1) The lapse of time in Hybrid Systems is captured with differential equations: equations whose first derivative is continuous. These equations do not directly refer to time, that is, the progress of time is implicit. (2) The value of time is updated by an

action in Continuous Action Systems. However discrete changes are performed in *jumps*, which are instantaneous and can be observed in a point of time. In other words, a state change does not consume time. For instance, the operations performed in these jumps are executed by computers whose operation time with respect to the physical object or phenomenon is negligible. That is, in hybrid system, the time consumed by control mechanism, a computer (SoC design) is negligible with respect to the time consumed by physical objects. Thus, owing to the importance of the time consumed in the operations within the SoC designs, these formalisms are not in their best in modeling these systems.

Timed formalisms and SoC designs

To model Very Large Scale Integration (VLSI) systems several synchronous formalism exist such as *Signal* [24], *Lustre* [32] and *Esterel* [26]. All of these approaches rely on the *synchronous hypothesis* in which computations and behaviors are divided into a discrete sequence of steps with deterministic concurrency. Signal is applied to modeling and validating globally asynchronous design in synchronous networks [60] and Esterel is extended to multiple clock domains in [27] and [72] allowing one to model both multiclocked and asynchronous systems, and furthermore, to capture asynchronous behavior within a synchronous framework. These extensions enable one to use the formalisms for the same application area as the Action Systems. However, the research presented in this study is targeted to formal power modeling framework where time is a significant measure. Therefore, one should consider the timing analysis capabilities as well. The timing analysis in these synchronous languages is more restrictive than the timing analysis in the timed spiced extension of Action Systems, Timed Action Systems, because they rely on the *perfect synchrony hypothesis* that defines that the outputs are produced synchronously with the inputs. Furthermore, the rigorous system development is supported only in Signal. It supports system refinement via semantics-preserving transformations [81], but its mathematical basis seems to be less rigorous than the Refinement Calculus Paradigm [22] defined for Action Systems.

Esterel Studio [4] is a tool set targeted to design SoC systems. It uses a formal description language and a verification environment to produce RTL system descriptions. These RTL descriptions can be exported at least in Verilog, VHSIC Hardware Description Language (VHDL), and SystemC. Furthermore, the generated system descriptions have shown to be equally good or in many cases better than hard coded ones [4]. The tool does not directly offer any area evaluation methods but its verification methods can be used to make the design as efficient as possible. For instance, a power manager design [41] framework relates to Esterel Studio. The system specifi-

cations are written in terms of hierarchical concurrent state machines where the formal verification makes it possible to check critical properties and preserve behavior when beautifying the specification. That is, the Esterel verification environment is used to define a more efficient power management system for a SoC. The power management device optimizes dynamic and static power reduction by dynamically distributing and controlling clock, reset, and power distribution for various SoC parts. This approach, however, is targeted to control power consumption by designing a specific component using the most effective approach available. In this thesis, the presented approach in terms of area and power consumption is more general. That is, one can estimate performance related metrics to all components that are valid for the formalism. Furthermore, the presented model is more flexible since it is not restricted to synchronous systems.

Area

In an early work [79] of Shannon the area complexity of Boolean function was studied (switch count). In this paper Shannon proved that the asymptotic complexity of Boolean functions is exponential in the number of inputs (m), and that for large m, almost every Boolean function is exponentially complex. Muller demonstrated the same result for Boolean functions implemented using logic gates (gate-count measure) in [61]. As one tries to apply that model to realistic VLSI circuits, it quickly breaks down due to the exponential dependence on the number of inputs. Over the years several other researchers have reported results related to the area complexity of Boolean functions, for instance, the relationship between area complexity and entropy ($\mathcal{H}$) is reported, for instance, in [48], [66], [37] and [35]. Cheng and Agrawal [35] used the entropy measure for the complexity of multi-output Boolean functions. Nemani et. al. [65] proposed a method for predicting the area of a single output Boolean function given only its functional specification and no structural information. This area complexity model is based on the *area cube complexity* and the results were compared with the SIS high-level synthesis tool nowadays known as Multivalued Logic Synthesis (MVSIS) [7]. The model was reasonably accurate compared with the results given by the SIS tool.

Another approach to model the area complexity of a Boolean function is to use a graphical model. A Boolean function can be represented as a directed acyclic graph, where the size of a function can be evaluated by calculating the number of nodes needed to present the function. These graphs are often referred to as *Binary Decision Diagrams (BDDs)*, and described, for instance, in [12, 29, 30]. BDD represent a Boolean function as a directed acyclic graph with each vertex labeled by a Boolean variable. In an *Ordered Binary Decision Diagram (OBDD)*, the vertex label occurs in the

same order along all directed paths. This presentation has many desirable algorithmic properties. For instance, it has proved to work well as a data structure for symbolically representing and manipulating Boolean functions [29]. Furthermore, for a given variable ordering, the smallest OBBD for a particular Boolean function is unique.

Several tools and packages exist to automate the BDD manipulation. For instance, packages such as CUDD [3] and BuDDy [2] offer functions to manipulate BDDs via a C++ interface. However, benchmark circuits such as the ISCAS and ACM/SIGDA benchmark sets do not support these tools. University of Berkeley has research groups [7, 10] on high level synthesis and verification, offering their own tool sets for BDD manipulation. Furthermore, decision diagrams are often related to verification tools. For instance, *Esterel* verification environment (*Xeve*) [5] uses BDDs to symbolically describe input events to analyze state machines.

The above mentioned approaches are targeted to RTL analysis, i.e., a RTL description that includes complete information on the memory elements but only functional (Boolean) information on combinatorial logic. In this thesis, the presented area evaluation methods are only partly applicable because the area modeling approach targeted to Action Systems operates at a higher abstraction level. The higher abstraction level, in this context, states that the system descriptions are written using Boolean type of expressions but the exact functionality (Boolean function) is unknown or inaccurate. Thus, new evaluation methods are required.

Power consumption

A number of power estimation techniques have been proposed for gate-level power estimation in VLSI designs (see [63] for survey). However, by the time the design has been specified down to the gate level, it may be too late or too expensive to go back and fix high-power problems. Therefore, power estimation techniques that can estimate the power at high abstraction levels are required to reduce costly redesign steps. In response to this need several high-level power estimation techniques have been proposed (see [51] for a survey). Two style of techniques have been proposed, which are referred to as top-down and bottom-up. In top-down techniques [55, 64] a combinatorial circuit is specified as a Boolean function with no information on circuit structure such as the number of gates. Top-down methods are useful when one is designing a logic block that has not been previously designed, that is, its internal structure is unknown. In contrast, the bottom up methods, for instance [25, 44, 71], are useful when one is reusing a previously designed logic block. In this case, a *power macro model* is developed for this block. This model can be used in power estimation for the block without performing a more expensive gate-level simulation on it. Most of the power estimation

methods mentioned are targeted to estimate average power consumption over a long time period. However, in many applications the average power may not be enough. Indeed, it is often important to know the instantaneous power dissipation as a time waveform, which one may refer to as a *transient power* waveform. For instance, transient power is needed in the analysis of the power and ground bus networks for finding the IR-drop problems. Gupta et. al presented a RTL macro modeling technique for estimating the energy dissipated and peak current drawn in a logic circuit for every input vector pair [45]. This model concentrated particularly on modeling instantaneous power dissipation.

At the RTL, the primitives are functional blocks such as adders, multipliers, controllers, register files and memory units. The difficulty in estimating power at this level stems from the fact that the gate, circuit, and layout level details of the design may not have been specified [51]. Furthermore, a floorplan may not be available, making analysis of interconnect and clock distribution networks difficult. One way to estimate the power consumption of a particular RTL description is use the fundamental that describes the physical capacitance and activity of a design. Complexity based models relies on the fact that the complexity of a chip architecture can be described roughly in terms of "gate equivalents". Basically, the gate equivalent count of a design specifies an approximate number of reference gates that would be required to implement a particular function. The power consumption for each functional block can be estimated by multiplying the approximate number of gate equivalents by the average power consumed by each gate. An example of this technique is given by the Chip Estimation Technique [62]. One disadvantage of this technique is that the energy estimate is based on an energy consumption of a single reference gate and therefore does not take into account different circuit styles or layout techniques. In [54], Liu and Svensson improved the above model by applying customized estimation techniques to the different design entities such as logic, memory, interconnect and clock. The advantage of these complexity based solutions is that they require very little information. On the other hand, they do not provide a model for circuit activity or they provide a fixed, user determined activity factor. Another approach is to use activity based methods where the basic idea is to relate the power that a functional block consumes to the amount of computational work it performs. In [64], the power estimation methodology consists of running a RTL simulation of the design to measure the input and output entropies of the functional blocks then to translate these measurements into prediction on average power.

The power analysis techniques mentioned above are divided into two categories: top-down and bottom-up techniques where the latter is based on generating a macro model of the circuit under analysis. The macro model generation requires some knowledge of the circuit under analysis to generate

the model, and therefore it cannot be used to describe the power modeling framework presented in this thesis. This is due to the fact that the presented modeling framework falls into top down category because the purpose is to model power consumption from abstract design steps to implementable system description. The difference to the presented top-down methodologies is that in this study an abstract level refers to descriptions above RTL. Furthermore, all of the models presented above discussed on dynamic power consumption whereas the model presented in this study covers both static and dynamic power consumption.

Chapter 2

Action Systems

This chapter describes the Action Systems formalism [17, 20], that is the formal basis for the work presented in this thesis. Action Systems is used for modeling concurrent systems and developing them towards implementation in a correctness preserving manner. It is based on an extended version of Dijkstra's guarded command languages [40] and defined using *weakest precondition* predicate transformers. The basic building block of the formalism is an *action*, a *guarded command* in Dijkstra's notation. A collection of actions operating iteratively on various variables is called an *action systems*. The development of action systems is done in a stepwise manner within the refinement calculus framework [15, 22] using correctness preserving transformations.

2.1 Actions

An *action* A is defined (for example) by:

$A ::= \ abort$	(*abortion, non-termination*)
$\mid skip$	(*empty statement*)
$\mid \{p\}$	(*assert statement*)
$\mid [p]$	(*assumption statement*)
$\mid x := x'.Q$	(*non-deterministic assignment*)
$\mid x := e$	(*assignment*)
$\mid p \rightarrow A$	(*guarded action*)
$\mid A_1 \ [] \ A_2$	(*non-deterministic choice*)
$\mid A_1; A_2$	(*sequential composition*)
$\mid \lvert[\mathbf{var}\ x := x0; A]\rvert$	(*block with local variables*)
$\mid \mathbf{do}\ A\ \mathbf{od}$	(*iterative composition*)

where A and A_i, $i = 1, 2$, are actions; x is a variable; e is an expression; and p and Q are predicates (Boolean expression). Notice that any action A can be written in the form $true \rightarrow A$, and thus each of the above actions can be considered a guarded command. If an action does not establish any postcondition it behaves as an *abort* statement (a never terminating statement), and if it does not change the state at all, it behaves as a *skip* statement (an empty statement).

The variables which are assigned within the action A are called the *write variables* of A, denoted by wA. The other variables present in the action A are called the *read variables* of A, denoted by rA. The write and read variables form together the *access set* vA of A:

$$vA \mathrel{\widehat{=}} wA \cup rA \qquad (access\ set)$$

Observe that the read and write sets of A are not necessary exclusive, that is, a variable can belong to both sets ($rA \cap wA \neq \emptyset$). If $rA \cap wA \neq \emptyset$ holds, then *a read-write conflict* may occur when the variable is read and written at the same time. This can be solved with careful system design.

The scope of a composition operator is indicated with parenthesis, for example: $A; ((B; C) \ [] \ D)$.

Example 2.1. As an example of an action, consider two variables x and y, which are added together and the result is written to the variable res. The action definition is of the form: $Add \mathrel{\widehat{=}} res := x + y$, where Add is a label for given action
End of example.

Notations A *substitution* operation within an action A, denoted by:

$$A[e'/e] \qquad (substitution)$$

where e refers to an element such as variables and predicates of the original action A and e' denotes the new element, which replaces e in A.

A *multiple (simultaneous) assignment* is defined by:

$$\begin{aligned} &x_1, \ldots, x_n := e_1, \ldots, e_n \\ &x_1, \ldots, x_n := e, \ldots, e \equiv x_1, \ldots, x_n := e \end{aligned} \qquad (multiple\ assignment)$$

where the former simultaneously assigns the expression e_i to variables x_i,

respectively, and the latter assigns the same expression e to all the variables x_i.

A *quantified composition* of actions is defined by:

$[\bullet\ 1 \leq i \leq n : A_i] \mathrel{\widehat{=}} A_1 \bullet \ldots \bullet A_n$ (*quantified composition*)

where the bullet $\bullet$ denotes any of the composition operators $[]$ (non-deterministic choice) or ; (sequential composition), and n is the number of actions.

2.2 Semantics of Actions

Actions and action compositions are considered to be *atomic*, which means that only the initial and final states are *observable* by the system. Therefore, when an action is selected for execution, it is completed without any interference from other actions.

The *total correctness* of an action A with respect to a precondition p and a post condition q is denoted by pAq and defined by:

$p\ A\ q \mathrel{\widehat{=}} p \Rightarrow \mathbf{wp}(A, q)$ (*total correctness*)

where $\mathbf{wp}(A, q)$ is the *weakest precondition* for the action A to establish the postcondition q.

$\mathbf{wp}(abort, q) = false$	(*abortion, non-termination*)
$\mathbf{wp}(skip, q) = q$	(*empty statement*)
$\mathbf{wp}(\{p\}, q) = p \wedge q$	(*assert statement*)
$\mathbf{wp}([p], q) = p \Rightarrow q$	(*assumption statement*)
$\mathbf{wp}(x := x'.r, q) = \forall x'.r \Rightarrow q[x'/x]$	(*non-deterministic assignment*)
$\mathbf{wp}(x := e, q) = q[e/x]$	(*(multiple) assignment*)
$\mathbf{wp}(p \rightarrow A, q) = p \Rightarrow \mathbf{wp}(A, q)$	(*guarded action*)
$\mathbf{wp}((A_1\ []\ A_2), q) = \mathbf{wp}(A_1, q) \wedge \mathbf{wp}(A_2, q)$	(*non-deterministic choice*)
$\mathbf{wp}((A_1; A_2), q) = \mathbf{wp}(A_1, \mathbf{wp}(A_2, q))$	(*sequential composition*)
$\mathbf{wp}(\lvert[\mathbf{var}\ x := x0; A]\rvert, q) = \forall x.\mathbf{wp}(A, q)$	(*block with variables*)
$\mathbf{wp}(\mathbf{do}\ A\ \mathbf{od}, q) = \exists k \geq 0.H_k$	(*iterative composition*)

where the conditions H_k are defined by:

$$H_k \mathrel{\widehat{=}} \begin{cases} q \wedge \neg gd(A) & ,k = 0 \\ gd(A) \wedge \mathbf{wp}(A, H_{k-1}) \vee H_0 & ,k > 0 \end{cases} \qquad (\textit{condition of iterative composition})$$

The weakest precondition of the iteration loop requires that after k repetitions of A the loop terminates. That is, A becomes disabled in a state where the postcondition q holds. If $k = 0$, A is disabled and the iteration behaves as a *skip* action. The Boolean condition $gd(A)$ is the guard of the action A, defined by:

$$gd(A) \mathrel{\widehat{=}} \neg \mathbf{wp}(A, false) \qquad (\textit{guard of an action})$$

Hence, $gd(A)$ is *true* in the states, where A does not behave miraculously. In the case of a guarded action $A \mathrel{\widehat{=}} p \rightarrow B$, the guard of A is $gd(A) = p \wedge gd(B)$. The guarded action can also be defined as the statement $[p]; B$.

An action is said to be *enabled* in states where its guard is *true*. Otherwise A is disabled. If the guard $gd(A)$ is invariantly *true*, the action is said to be *always enabled*: $\mathbf{wp}(A, false) = false$. Furthermore, if

$$\mathbf{wp}(A, true) = true$$

holds, the action A is said to be *always terminating*. The body $bd(A)$ of the action A is defined by:

$$bd(A) \mathrel{\widehat{=}} \{gd(A)\}; A \qquad (\textit{body of an action})$$

Prioritized composition A prioritized composition [77] of actions, denoted by $A_1 \mathbin{/\!\!/} A_2$ is a new composed action in which the execution order of enabled actions is prioritized. The highest priority belongs to the leftmost action in the composition; thus if both of the actions are enabled at the same time, the leftmost enabled action is always selected for execution. The prioritized composition is defined by:

$$A_1 \mathbin{/\!\!/} A_2 \mathrel{\widehat{=}} A_1 \mathbin{[\!]} \neg gd(A_1) \rightarrow A_2 \qquad (\textit{prioritized composition}) \quad (2.1)$$

where $gd(A_1)$ is the guard of the action A_1. The definition shows that if A_1 is enabled, then it is executed independently of the enabledness status of $A2$.

Synchronous composition A synchronous composition, denoted by $A_1 \vee A_2$, is used to model synchronous behavior of system components where all the enabled actions perform their operation at the same time within one atomic action. The synchronous composition is defined by:

$$\begin{aligned} A_1 \vee A_2 \mathrel{\widehat{=}} & [\![\mathbf{var}\ uA_1, uA_2 \qquad (\textit{synchronous composition}) \\ & ; gd(A_1) \vee gd(A_2) \rightarrow uA_1, uA_2 := wA_1, wA_2; \\ & ; [; 1 \leq i \leq 2 : A_i[uA_i/wA_i] \ /\!\!/ \ skip\ ] \\ & ; wA_1, wA_2 := uA_1, uA_2]\!] \end{aligned} \tag{2.2}$$

where only enabled actions are executed. The enabledness of an action A_i is evaluated based on its guard $gd(A_i)$. Those actions that are disabled at the time of execution perform the *skip* operation. The variable substitution resolves the possible read-write conflicts by storing the write variables ($wA_1 \cap wA_2 = \emptyset$) onto local, internal variables uA_1 and uA_2. The actions write on the local variables and after all the actions have been executed the write variables are updated. Thus, there is no possibility for read-write conflicts during the operation.

2.3 Non-Atomic Operation of Actions

The actions, and their compositions presented in the previous sections are all atomic. *The atomic compositions* are merely larger atomic entities composed of simpler ones. However, when dealing with the iterative composition, the **do - od** loop, whose execution may consist of several executions of its component actions, it is also useful to define the concept of a *non-atomic composition* of actions within the scope of the loop in question. In a such construct, the component actions are atomic entities of their own, but their composition is not. Non-atomicity means that also the intermediate states of the composite action can be observed unlike in the case of atomic construct.

A *non-atomic sequence* of atomic actions A_1 and A_2 is denoted by:

$$\mathbf{do}\ A_1 \ \boldsymbol{;}\ A_2\ \mathbf{od} \qquad (\textit{non-atomic sequence}) \tag{2.3}$$

where the execution order is from left to right. It can be defined in terms of non-deterministic choice and an auxiliary local program counter p (initialized to 1) as follows:

$$A_1 \ \boldsymbol{;}\ A_2 \mathrel{\widehat{=}} A_1^{\boldsymbol{;}} \ [\!]\ A_2^{\boldsymbol{;}}$$

where $A_1^{\boldsymbol{;}}$ and $A_2^{\boldsymbol{;}}$ are of the form:

$$A_1^{;} \mathrel{\widehat{=}} p = 1 \rightarrow (A_1; p := 2)$$

$$A_2^{;} \mathrel{\widehat{=}} p = 2 \rightarrow (A_2; p := 1)$$

A *parallel composition* of atomic actions A_1 and A_2 is denoted by:

---

do $A_1 \;\|\; A_2$ **od** (*parallel compostion*) (2.4)

---

where the execution order in the composition is of no importance. It can be defined in terms of non-deterministic choice and an auxiliary Boolean variable p_i (initialized to T) and a new auxiliary action $T_{\|}$ as follows:

$$A_1 \,\|\, A_2 \mathrel{\widehat{=}} A_1^{\|} \;\;[]\;\; A_2^{\|} \;\;[]\;\; T_{\|}$$

where $A_1^{\|}$, $A_2^{\|}$, and $T_{\|}$ are of the form:

$$A_1^{\|} \mathrel{\widehat{=}} p_1 \rightarrow (A_1; p_1 := F)$$

$$A_2^{\|} \mathrel{\widehat{=}} p_2 \rightarrow (A_2; p_2 := F)$$

$$T_{\|} \mathrel{\widehat{=}} \neg p_1 \wedge \neg p_2 \rightarrow (p_1, p_2 := T)$$

where it is required that $wA_1 \cap wA_2 \neq \emptyset$. Moreover, the above definitions extend to more than two actions as well, see for instance [67].

2.4 Properties of Actions

A predicate I is an *invariant over* an action A if the following condition holds:

$I \Rightarrow \mathbf{wp}(A, I)$ (*invariant*)

---

which is the same as the total correctness assertion $\{I\}A\{I\}$. If $\mathbf{wp}(A, I) =$ *true*, then the action A is said to establish the invariant I. It is also said that the action *preserves* the invariant I.

A predicate p *excludes* an action A if the following condition holds:

$p \Rightarrow \neg gd(A)$ (*excludes*)

---

and a predicate p *includes* an action A if the following condition holds:

$p \Rightarrow gd(A)$ (*includes*)

The way in which two actions A and B interact is captured by the following definitions:

$$A \textbf{ cannot enable } B \mathrel{\widehat{=}} \neg gd(B) \Rightarrow \mathbf{wp}(A, \neg gd(B)) \tag{2.5}$$
$$A \textbf{ can enable } B \mathrel{\widehat{=}} \neg(\neg gd(B) \Rightarrow \mathbf{wp}(A, \neg gd(B))) \tag{2.6}$$
$$A \textbf{ enables } B \mathrel{\widehat{=}} \neg gd(B) \Rightarrow \mathbf{wp}(A, gd(B)) \tag{2.7}$$
$$A \textbf{ cannot disable } B \mathrel{\widehat{=}} gd(B) \Rightarrow \mathbf{wp}(A, gd(B)) \tag{2.8}$$
$$A \textbf{ can disable } B \mathrel{\widehat{=}} \neg(gd(B) \Rightarrow \mathbf{wp}(A, gd(B))) \tag{2.9}$$
$$A \textbf{ disables } B \mathrel{\widehat{=}} gd(B) \Rightarrow \mathbf{wp}(A, \neg gd(B)) \tag{2.10}$$
$$A \textbf{ excludes } B \mathrel{\widehat{=}} gd(A) \Rightarrow \neg gd(B) \tag{2.11}$$
$$A \textbf{ includes } B \mathrel{\widehat{=}} gd(A) \Rightarrow gd(B) \tag{2.12}$$

2.5 Action Systems

An action system has the form:

sys $\mathcal{A}$ (**imp** p_I; **exp** p_E;)(g_A;) ::
|[
type
type: Def;
constraint
constraint: (B);
variable
l_A;
private procedure
p_I(**in** x : **out** y): (P_I);
public procedure
p_E(**in** x : **out** y): (P_E);
action
A_i: (aA_i);
initialization
$g_A, l_A := g_A0, l_A0$;
execution
forever do `composition of actions` A_i **od**
]|

where it is possible to identify three main sections: *interface*, *declaration* and *iteration*. The interface part declares those variables, g_A, that are visible

outside the action system boundaries and therefore accessible by other action systems. Global variables maybe of type input, output or bi-directional input-output, and the types are denoted by the following identifiers: **in**, **out** and **inout**, respectively. It also introduces *interface procedures* p_I and p_E that are imported in or introduced in and exported by the system. These are denoted by the **imp** and **exp** identifiers, respectively. If an action system has no interface variables or procedures, it is *a closed action system*, otherwise it is *an open action system*.

The declaration part introduces all new, local type definitions (**type**); constraints (**constraint**) that define restrictions on the functional behavior of the system including invariants; and the local variables l_A (**variable**). Furthermore, the declaration part introduces private p_I and public p_E procedures (**private procedure / public procedure**) and action definitions aA_i (**action**) that perform operations on local and global variables. Furthermore, a label A_i is given for every action definition.

The operation of an action system is started by the initialization in which the variables are set to predefined values. In the iteration part, the **execution** section, actions are selected for execution based on their composition and enabledness. In general, actions are selected for execution one at a time. Parallel execution of actions is considered when two or more actions, which are not in conflict, are executed simultaneously. The synchronous composition forms an exception, that is, it executes all the enabled actions in a composition as a one atomic action. This is continued until there are no enabled actions, whereupon the computation terminates. Hence, an action system is essentially an initialized block with a body that contains an iteration, that is, a statement which is repeatedly executed. In this thesis the iteration loop is of the form **forever do - od** instead of the conventional **do - od** loop. This is only notational difference, that is, the behavior is the same between these two loops. The former one is adopted to illustrate the *reactive* nature of Action Systems.

In general, the sets of the global and local variables of an action system $\mathcal{A}$ are denoted by $g_{\mathcal{A}}$ and $l_{\mathcal{A}}$, respectively. The access set of $\mathcal{A}$ is defined by:

$$v_{\mathcal{A}} \mathrel{\widehat{=}} g_{\mathcal{A}} \cup l_{\mathcal{A}} \qquad (\textit{access set of } \mathcal{A})$$

A predicate is *an invariant of an action system* $\mathcal{A}$, if the following conditions hold:

$$\textit{true} \Rightarrow \mathbf{wp}((g, l := g0, l0), I) \qquad (\textit{established by the initialization}) \tag{2.13}$$

$$I \Rightarrow \mathbf{wp}(A, I) \qquad (\textit{preserved by the action } A) \tag{2.14}$$

That is, an invariant is a Boolean condition that must hold in all possible execution states of action system $\mathcal{A}$.

2.5.1 Parallel action systems

Consider two actions systems $\mathcal{A}$ and $\mathcal{B}$ of the form:

sys $\mathcal{A}$ (g_A;) ::
|[
variable
l_A;
action
A: (aA);
initialization
$g_A, l_A := g_A0, l_A0$;
execution
forever do A **od**
]|

sys $\mathcal{B}$ (g_B;) ::
|[
variable
l_B;
action
B: (aB);
initialization
$g_B, l_B := g_B0, l_B0$;
execution
forever do B **od**
]|

where $l_A \cap l_B = \emptyset$ (distinct local variables).

The *parallel composition* [67]of $\mathcal{A}$ and $\mathcal{B}$, denoted by $\mathcal{A} \parallel \mathcal{B}$, is an action system that combines the state spaces of the constituent action systems such that the local variables are kept distinct and the global variables are a set $g_A \cup g_B$. Furthermore, it is required that the initializations of the global variables $(g_A \cap g_B)$ in the system $\mathcal{A}$ and $\mathcal{B}$ are consistent with each other so that their initial values are equivalent in the systems: $\forall v \in (g_A \cap g_B).(v_A0 = v_B0)$. Here $v_A0 \in g_A0$ and $v_B0 \in g_B0$.

The parallel composition of the above systems is of the form:

sys $\mathcal{A} \parallel B$ ($g_A \cup g_B$;) ::
|[
variable
$l_A \cup l_B$;
action
A: (aA);
B: (aB);
initialization
$(g_A \cup g_B), (l_A \cup l_B) := (g_A0 \cup g_B0), (l_A0 \cup l_B0)$;
execution
forever do A [] B **od**
]|

The *reactive components* $\mathcal{A}$ and $\mathcal{B}$ interact with each other via the global variables that are referenced in both components. The reactivity means that the action systems $\mathcal{A}$ and $\mathcal{B}$ do not terminate independently of each other, but termination is a global property of $\mathcal{A} \| \mathcal{B}$.

2.5.2 Hierarchical action systems

A *hierarchical action system* [67] is an action system that encapsulates other action system inside its boundaries. These action systems are called *sub-*

systems, and they are introduced in the declarative part of the hierarchical action system and their functionalities are defined elsewhere. A hierarchical action system $\mathcal{H}$ is of the form:

```
sys H (   g_H; ) ::                        sys A (   g_A; ) ::
|[                                         |[
 variable                                   variable
   l_H ∪ (g_A\g_H);                           l_A;
  subsystem                                  action
   I: A(g_A);                                  A: (aA);
  action                                     initialization
   H: (aH);                                    g_A, l_A := g_A0, l_A0;
  initialization                             execution
   l_H ∪ (g_A\g_H), g_H                        forever do A  od
        = (l_H0 ∪ (g_A0\g_H0)), g_H0;       ]|
  execution
    forever do H  od || I
]|
```

where $\mathcal{A}$ is the subsystem within the action system $\mathcal{H}$ and I its local instance name. The local variables of the hierarchical system $\mathcal{H}$ are the union of its own local variables l_H and the difference of the global variables g_A of the subsystem $\mathcal{A}$ and the global variables g_H of the hierarchical system $\mathcal{H}$. The subsystems local variables l_A are encapsulated inside the system $\mathcal{A}$.

The interpretation $\mathcal{H}_{flattened}$ of the hierarchical composition of the action systems $\mathcal{H}$ and $\mathcal{A}$ is of the form:

```
sys H_flattened (   g_H; ) ::
|[
 variable
   l_H ∪ I.l_A ∪ (g_A\g_H);
  action
   H: (aH);
   I.A: (aA);
  initialization
   l_H ∪ I.l_A ∪ (g_A\g_H) ∪ g_H
        = (l_H0 ∪ I.l_A0 ∪ (g_A0 ∪ g_H0) ∪ g_H0;
  execution
    forever do H  [] I.A  od
]|
```

where the local variables are the union of the local variables of the hierarchical system $\mathcal{H}$ and the local variables of the subsystem $\mathcal{A}$. Furthermore, the action clause consist of the actions of both systems.

2.6 Procedures

A procedure is either local, exported, or imported, and, furthermore, they must be distinct. The local and exported procedures are defined in the **private procedure** and **public procedure** clauses of action system, respectively. The private procedures are used by the system whereas the public procedures are called by other action systems. The imported procedures are introduced in and exported by other action systems and called by the system that imports them. Together the imported and exported procedures are called *interface procedures*.

Each procedure may have a set of input values **in**, a set of output values (result) **out**, and a set of bidirectional values **inout** as parameters. The parameters are accessible in the body of the procedure but not the outside, and they are replaced with the actual ones at each procedure call. The operation performed by the procedure (a procedure body) is considered a part of the calling atomic action, that is, a procedure is a parametrized subaction. A procedure is defined by:

$p(\ \textbf{in}\ x;\ \textbf{out}\ y;\ \textbf{inout}\ z) : P$ (*procedure p*)

---

where the body P of the procedure p is any atomic action A, possibly with some auxiliary local variables u initialized to u_0 every time the procedure is called. These variables are accessible only within the procedure. The action A may access the global and local variables g and l of the host/enclosing system and the formal parameters x, y, and z Hence, the body P can be generally defined by:

$$P \mathrel{\widehat{=}} |[\textbf{var}\ u; \textbf{init}\ u := u0; A(g, l, u, x, y, z)]|$$

If there are no local variables u, the begin-end brackets $|[\ ]|$ can be ignored and the body P is of the form:

$$P \mathrel{\widehat{=}} A(g, l, x, y, z)$$

If there are neither local variables nor parameters, the action A only accesses the global and local variables of the host system. Then the definition of the procedure p can be written as: **proc** $p : (A(g, l))$.

A call to a private procedure $p(\textbf{in}\ x;\ \textbf{out}\ y;\ \textbf{inout}\ z) : P$ with actual parameters is denoted by $p(a, b, c)$ and it is defined by:

$p(a, b, c) \mathrel{\widehat{=}} P[a/x, b/y, c/z]$ (*private procedure call p*)

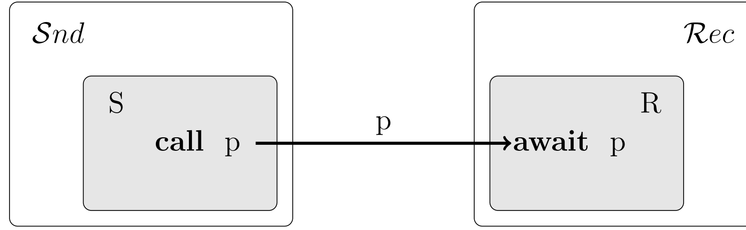


Figure 2.1: Action systems $\mathcal{S}nd$ and $\mathcal{R}ec$ communicates directly using public procedure p

where the formal parameters are replaced by the actual ones during procedure call. Thus, an action A calling a procedure p is semantically equivalent to $A(P[a/x, b/y, c/z]/p(a, b, c))$.

2.6.1 Procedure based communication

The procedure based communication, presented in [19, 68, 78], uses remote procedures to model communication channels between action systems. Consider action systems $\mathcal{S}nd$ and $\mathcal{R}ec$ (See Fig. 2.1) whose operation is defined by actions:

$$S \mathrel{\widehat{=}} \textbf{call } p(a, b, c)$$
$$R \mathrel{\widehat{=}} \textbf{await } p(a, b, c)$$

where S is a *caller* action in the system $\mathcal{S}nd$, which imports the public procedure p, and R is a *callee* action in the system $\mathcal{R}ec$, which exports p. The **execution**-clause of the composed system $\mathcal{S}nd \| \mathcal{R}ec$ has the form: **do** S $[\!]$ R **od**. The construct $S\ [\!]\ R$, where S calls p (**call** command) and R awaits such a call (**await** command), is regarded as a single atomic action SR, defined by:

$$SR \mathrel{\widehat{=}} S[R[P[a/x, b/y, c/z]/\textbf{ await } p]/\textbf{ call } p(a, b, c)] \qquad (\textit{public procedure})$$

---

where the action R is substituted for the **call** command in S, and the procedure body P with actual parameters is substituted for the **await** command in R. Hence, communication is based on sharing an action in which data is atomically passed from $\mathcal{S}nd$ to $\mathcal{R}ec$ by executing the body P of the procedure p hiding the details of the communication into the procedure call. Consider the following example:

Example 2.2. Consider the caller action S and the callee action R, defined by:

$$S \mathrel{\widehat{=}} x := x', x' > 0;\ \textbf{call } Trf(x)$$
$$R \mathrel{\widehat{=}}\ \textbf{await } Trf(x)$$

where the public procedure $Trf(x)$ is defined by:

$$Trf(\ \textbf{in}\ x : integer) : (buf := x)$$

where the caller action S transfers data by calling the public procedure $Trf(x)$, which stores the data into a local buffer buf. The callee action R awaits the procedure, and it can read the transferred integer value from the local buffer. According to the substitution principle the combined action SR is equivalent to:

$$SR \mathrel{\widehat{=}} (x := x', x' > 0; (buf := x))$$

End of example.

The type of the communication channel is identified in the definition of the procedure. A *push* channel between components is defined using the **in** specifier:

$$p(\ \textbf{in}\ x) : P$$

where data is transferred from *caller* action to *callee* action. The communication channel in Fig. 2.1 is a *push* channel. A *pull* channel is defined using the **out** specifier:

$$p(\ \textbf{out}\ x) : P$$

where data is transferred from *callee* action to *caller* action, and finally, a *biput* channel has both of the above mentioned specifiers or an **inout** specifier in the interface definition of a public procedure:

$$p(\ \textbf{inout}\ x) : P$$
$$p(\ \textbf{in}\ x,\ \textbf{out}\ y) : P$$

where data is transferred in either direction between the *caller* and the *callee* actions.

2.7 Refinement

Action systems are intended to be designed in a step wise manner within the *refinement calculus* framework [22]. The refinement calculus preserves the correctness of the actions during the process of refinement. The (atomic) action A is said to be *(correctly) refined* by action C, denoted $A \leq C$, if

$$\forall Q.\, (\textbf{wp}\ (A, Q) \Rightarrow \textbf{wp}\ (C, Q))$$

holds. This is equivalent to the condition

$$\forall P, Q.\, ((PAQ) \Rightarrow (PCQ))$$

which means that the *concrete* action C preserves every total correctness property of the abstract action A.

2.7.1 Data refinement

In a data refinement an abstract action A on the variables a and g is refined by a concrete action C on the variables c and g using an abstraction invariant $R(a, c, g)$ which is a Boolean relation between the abstract variables a and the concrete variables c. The action A is data refined by the action C, denoted $A \leq_R C$, if

$$\forall Q. (R \wedge \mathbf{wp}\ (A, Q) \Rightarrow \mathbf{wp}\ (C, \exists a.R \wedge Q))$$

holds. Note that the predicate $(C\exists a.R \wedge Q)$ is a Boolean condition on the variables a and c. The above definition can be written in terms of guards $gd(A)$, $gd(C)$ and bodies $bd(A)$, $bd(C)$ of the actions A and C as follows:

$$R \wedge gd(C) \Rightarrow gd(A) \qquad (\textit{guard}) \text{ (i)}$$
$$\forall Q. (R \wedge gd(C) \wedge \mathbf{wp}\ (bd(A), Q) \Rightarrow \mathbf{wp}\ (bd(C), \exists a.R \wedge Q)) \qquad (\textit{body}) \text{ (ii)}$$

The data refinement $A \leq_q C$ replaces a with c preserving the variables g. That is, the local variables a are made concrete in c, for instance, a local set a can become an array c, and the relation $R(a, c, g)$ captures the essentials of this transformation.

2.7.2 Refinement of action systems

Rather than going into the details of refinement of parallel and reactive programs [16], a commonly used method to prove the refinement step is reviewed. The presented data refinement method has been use to prove the correctness of the superposition refinement of action systems [21], as well as the trace refinement of action systems [14]. The trace refinement of action system preserves the trace, the sequence of global states (observable behavior) of the system in question, and the superposition refinement [46] enhances the behavior of a system model by adding new functionality into the model while preserving the old one. The method is extended in [78] to prove the correctness of data refinement of action systems with remote procedures. Next, the refinement of Action Systems is described formally after which informal descriptions are given for the required conditions.

Refinement rule

Consider a action systems:

```
sys 𝒜 (    g;  ) ::
|[
 variable
   a;
 action
   A: (aA);
 initialization
   g, a := g0, a0;
 execution
   forever do A  od
]|
```

```
sys 𝒞 (    g;  ) ::
|[
 variable
   c;
 action
   C: (aC);
 initialization
   g, c := g0, c0;
 execution
   forever do C [] X  od
]|
```

whose operating environment $\mathcal{E}nv$ is of the form:

```
sys ℰnv (    g;  ) ::
|[
 variable
   e;
 action
   E: (aE);
 initialization
   g, e, = g0, e0;
 execution
   forever do E  od
]|
```

Let $R(a, c, g)$ be an abstraction relation on the state variables a, c and g. The abstract system $\mathcal{A}$ is said to be correctly refined by the concrete system $\mathcal{C}$ denoted $\mathcal{A} \sqsubseteq \mathcal{C}$ if there exists a *data relation R(a,c,g)* on the state variables and the following conditions are satisfied:

$R(a_0, c_0, g_0) = holds$	(*initialization*)	(i)
$A \leq_R C$	(*main action*)	(ii)
$skip \leq_R X$	(*auxiliary action*)	(iii)
$R \wedge gd(A) \Rightarrow gd(C) \vee gd(X)$	(*continuation condition*)	(iv)
$R \Rightarrow \mathbf{wp}\ (\mathbf{do}\ X\ \mathbf{od}, true)$	(*internal convergence*)	(v)
$R \Rightarrow \mathbf{wp}\ (E, R)$	(*non-interference*)	(vi)

(i) The first condition says that the initialization of the systems $\mathcal{A}$ and $\mathcal{C}$ establish the abstraction relation R.

(ii) The second condition requires the abstract action A to be data-refined by the concrete action C using R.

(iii) The third condition, in turn, indicates that the auxiliary action X is obtained by data-refining a *skip* action. This basically means that X behaves like *skip* action with respect to the global variables.

(iv) The fourth condition requires that whenever the action A of the abstract system is enabled, assuming the abstract relation R holds, there must be enabled action in the concrete system $\mathcal{C}$ as well.

(v) The fifth condition states that if R holds, the execution of the auxiliary action X, taken separately, must terminate at some point.

(vi) The sixth condition guarantees that the interleaved execution of E actions preserves the abstract relation R.

The conditions (i)-(v) guarantee, in terms of global and local variables that the behavior of an abstract system is preserved in the concrete one, when the action system is executed in isolation. When the system is operating in a parallel composition with other action systems the last condition must also be validated.

2.7.3 Development of action systems

Stepwise development of action systems is based on the following properties of the refinement operator $'\sqsubseteq'$ [16, 22]:

- *Reflexivity*: A system is refined by itself:

$$\mathcal{A} \sqsubseteq \mathcal{A}$$

- *Equivalence*: If

$$\mathcal{A} \sqsubseteq \mathcal{B} \wedge \mathcal{B} \sqsubseteq \mathcal{A}$$

  then one can write $\mathcal{A} = \mathcal{B}$

- *Monotonicity* [20]: A refinement of component system implies a refinement of the whole composition

$$(\mathcal{A}_1 \sqsubseteq \mathcal{C}_1) \Rightarrow ((\mathcal{A}_1 \parallel \mathcal{A}_2 \parallel \ldots \parallel \mathcal{A}_n) \sqsubseteq (\mathcal{C}_1 \parallel \mathcal{A}_2 \parallel \ldots \parallel \mathcal{A}_n))$$

  provided that the applied invariant R is satisfied by the systems $\mathcal{A}_2, \ldots, \mathcal{A}_n$ as well. If R refer only to the local variables of $\mathcal{C}_1$, monotonicity holds trivially.

- *Transitivity*: A system obtained by successive refinement steps is the refinement of initial system.

$$(\mathcal{A} \sqsubseteq \mathcal{C}_1 \sqsubseteq \ldots \sqsubseteq \mathcal{C}_n) \Rightarrow (\mathcal{A} \sqsubseteq \mathcal{C}_n)$$

where the monotonicity and transitivity are especially important properties as they enable stepwise development of systems.

2.8 Chapter Summary

An overview of the Action System formalism was given in this chapter. It is a design language targeted to both SW and HW system design as demonstrated by several researchers [67, 76]. The Action Systems formalism provides tools such as abstraction and encapsulation as well as correctness preserving refinements, all of which are important instruments in system development. Furthermore, the formalism provides a stepwise development framework within which one is able to derive an initial, abstract system specification step by step towards a more concrete model in a correctness preserving manner. All of these properties offers a powerful modeling base for any type of concurrent system development, including SoC designs. However, time modeling is also instrumental in SoC design, and therefore the following chapter presents one Action System implementation towards time-awareness in concurrent system development.

Chapter 3

Timed Action Systems

To model the performance of SoCs, the lapse of time has an important role in terms of speed and power consumption. In Action Systems computation does not take time, a reaction is instantaneous, and therefore atomic in any possible sense. Atomicity means that only pre- and post-states of the actions are observable, and when they are chosen for execution they cannot be interrupted by external counterparts. However, in Action Systems, the verification is focused on logical properties. That is, timing behavior of the system is not reflected at any abstraction levels. Therefore, in this chapter the *Timed Action Systems* formalism is presented, which allows one to model the lapse of time. Observe that Timed Action System formalism is an extension from the Action Systems formalism. The detailed definition of the formalism can be found in [86]. This chapter describes the foundation of the formalism as well as those properties, which are utilized in the power modeling.

3.1 Timed Actions

To be able to describe the basic properties of timed actions, a time domain $\mathbb{T} = \mathbb{R}_{\geq 0}$ is defined to be dense, continuous, and transitive. Formally:

$\forall t_1, t_3 . \exists t_2 . (t_1 > t_2 > t_3)$	(*density*)
$\forall t_1, \exists t_2 . (t_1 < t_2)$	(*continuity*)
$\forall t_1, t_2, t_3 . (t_1 < t_2 < t_3 \Rightarrow t_1 < t_3)$	(*transitivity*)

where *density* defines that there exists always a time point between two time points; *continuity* defines that for every time point there is a time point that follows it; *transitivity*, as time is linear, says that there is a natural ordering between two adjacent time instants.

3.1.1 Interpretation of system behavior

A behavior of an action system can be given as a sequence ($\mathcal{SEQ}$) of states with two components:

$$\mathcal{SEQ} \mathrel{\widehat=} \langle (l_1, g_1), (l_2, g_2), \ldots \rangle$$

where l_i and g_i stands for local and global variables, respectively.

To define the behavior of a timed action system, a new global time variable gt is associated into the state of type $\mathbb{T}$. The state containing such a variable is called a timed state where the global time variable carries information about the time elapsed from the start of the system. The sequence $\mathcal{TSEQ}$ of timed states is a timed behavior:

$$\mathcal{TSEQ} \mathrel{\widehat=} \langle (gt_1, l_1, g_1), (gt_2, l_2, g_2), \ldots \rangle \quad (\textit{timed behavior of an action system})$$

where gt_i, l_i, and g_i denote the time of the state change and the states of the local and global variables, respectively. A timed trace is formed by removing the local variables from a timed state as in conventional Action Systems. The timed trace is defined by:

$$\mathcal{TTR} = \langle (gt_1, g_1), (gt_2, g_2), \ldots \rangle$$

3.1.2 Components of timed action

Time-aware actions are defined using a conventional action A, whose guard is denoted by $gd(A)$ and body by $bd(A)$: $A \mathrel{\widehat=} gd(A) \rightarrow bd(A)$. The operation in time domain is divided into three segments: *commence*, *time*, and *end*, which are introduced by the order of execution.

The *commence* segment of a timed action performs the operation of an action and it consists of a *start action* A_S that is of the form:

$$\begin{aligned} A_S \mathrel{\widehat=} \neg bA \wedge gd(A) \rightarrow\ & delayA := delayA'.dpA \\ & ; stateA := (wA, gt, gt + delayA) \qquad (\textit{timed action(start)}) \\ & ; bd(A)[stateA.wA/wA]; bA := T; \end{aligned}$$

where the Boolean variable bA sequences the operation of a timed action; $gd(A)$ is the guard of the conventional action A, which signals the enabledness of the timed action A; the non-deterministic assignment assigns the delay variable $delayA$ a value that satisfies the delay predicate dpA (discussed more detail in Sect. 3.5); $stateA$ is a record: **record**(wA; st; ft: $\mathbb{T}$, where wA denotes the write variables of A, st a start time, and ft a finish time of the current computation event. The start time is set to the global

time gt and the finish time is obtained by adding together the global time with the delay $delayA$.

Next in the execution sequence is the *time* segment in which the time is propagated by executing a *time propagation action* tp_A:

$$tp_A \mathrel{\widehat{=}} gt := stateA.ft \qquad (\textit{time propagation action})$$

where the global time is set to the finish time $stateA.ft$ of the timed action A.

Finally, the operation is terminated in the *end* segment in which the result of the operation is written onto the write variables wA of the conventional action A. Notice that a timed action whose operation is performed, but its write variables are not yet updated, is called a *scheduled timed action*. It consist of a *finish action* A_f, which is of the form:

$$\begin{aligned} A_f \mathrel{\widehat{=}} & \; bA \land gd(A) \land (gt = stateA.ft) \rightarrow \\ & \quad bA := F; wA := stateA.wA; \end{aligned} \qquad (\textit{timed action (finish)})$$

By combining these three segments, a *timed action* A is defined by:

$$\begin{array}{ll} A[\![dA]\!] \mathrel{\widehat{=}} & (\textit{timed action}) \\ \quad A_f & (\textit{finish}) \\ /\!/ \; A_s & (\textit{commence}) \\ /\!/ \; tp_A & (\textit{time}) \end{array}$$

where $[\![\;]\!]$ are called delay brackets, dA a label of a delay predicate dpA introduced in Sect 3.5. Observe that the execution of a timed action starts from the commence segment as its name suggest, although the end section has the highest priority. This prioritization order is required to ensure that all write variables are updated before any start action may commence its execution.

Non-deterministic composition of timed actions. The composition of timed actions is defined by:

$$\begin{array}{ll} [\; [\!] \; 1 \leq i \leq n : A_i[\![dA_i]\!]] \mathrel{\widehat{=}} & (\textit{timed action composition}) \\ \quad [\; [\!] \; 1 \leq i \leq n : A_{f,i}] & (\textit{finish}) \\ /\!/ \; [\; [\!] \; 1 \leq i \leq n : A_{s,i}] & (\textit{commence}) \\ /\!/ \; Pt & (\textit{time}) \end{array}$$

where n is the number of timed actions. Observe that the time propagation action Pt is shared amongst the timed actions A_i. The sharing preserves the linearity of time ensuring that the global time is not able to jump back and forth depending on the delays of the scheduled timed actions. The redefined propagation action is of the form:

$$Pt \mathrel{\widehat{=}} [\ [\!]\ 1 \le j \le n : min[i] \to gt := stateA_i.ft] \quad (\textit{time propagation action})$$

where the global time is updated to the nearest scheduled finish time determined by the guard $min[i]$:

$$\begin{aligned} min[i] \mathrel{\widehat{=}}\ & (stateA_i.ft > gt) && (\textit{guard min}) \\ & \wedge \big(\forall j : 1 \le j \le n : j \ne i : stateA_j.ft > gt \\ & \qquad \Rightarrow stateA_i.ft \le stateA_j.ft\big) \end{aligned}$$

which evaluates to true (T) if a finish time $stateA_i.ft$ of a timed action A_i is greater than gt (a requirement for a timed action being scheduled one). Thus, the guard chooses the smallest scheduled finish time greater than the global time, which then becomes a new global time in the time propagation action.

A special case occurs when the scheduled timed action becomes disabled before the write variables are updated. In other words, this situation may produce a *deadlocked* timed action. To avoid deadlocks, a *kill* action is defined, which is of the form:

$$A_{k,i} \mathrel{\widehat{=}} bA_i \wedge \neg gd(A_i) \to bA_i := F; \qquad (\textit{timed action(kill)})$$

which simply releases the scheduled and disabled timed action by setting the Boolean variable bA_i to false (F).

By examining the guards of the finish and kill actions $A_{f,i}$ and $A_{k,i}$, respectively, it is clearly seen that the conventional guard $gd(A_i)$ defines which action will be executed. Therefore these actions can be merged with the non-deterministic choice. The composition of timed actions A_i with the deadlock avoidance action, the kill action, is defined by:

$$\begin{aligned} & [\ [\!]\ 1 \le i \le n : A_i[\![dA_i]\!]] \mathrel{\widehat{=}} && (\textit{timed action composition}) \quad (3.1) \\ & \quad [\ [\!]\ 1 \le i \le n : A_{f,i}\ \ [\!]\ \ A_{k,i}] && (\textit{finish}) \\ & /\!/\ [\ [\!]\ 1 \le i \le n : A_{s,i}] && (\textit{commence}) \\ & /\!/\ Pt && (\textit{time}) \end{aligned}$$

where the end segment consist of two actions: the finish action $A_{f,i}$ that reveals the result of computation and the kill action $A_{k,i}$ that enables a disabled, scheduled timed action, and thus prevents a scheduled timed action being deadlocked forever.

3.1.3 Timed action state predicates

State predicates embody those system states in which timed actions are enabled, starting their operation, in operation, and ending their operation. These predicates are useful in the verification of the system as they explicitly indicate the state in which timed actions are. The state of the timed actions are observed on the grounds of their guards: The enabledness is determined by the guard $gd(A)$ of the conventional action A, the start time by the guard $gd(A_s)$ of the start action, the finish time by the guard $gd(A_f)$ of the finish action A_f, and, finally, a time point within the limits of operation time captured by the guard $gd(A_k)$ of the kill action A_k. The state predicates are defined by:

$$En(A) \mathrel{\widehat{=}} gd(A) \qquad (\textit{enabled})$$

$$St(A) \mathrel{\widehat{=}} gd(A_s) \qquad (\textit{start})$$

$$Op(A) \mathrel{\widehat{=}} \neg gd(A_s) \wedge gd(A) \wedge \neg gd(A_f) \qquad (\textit{operation})$$

$$Fi(A) \mathrel{\widehat{=}} gd(A_f) \qquad (\textit{finish})$$

$$K(A) \mathrel{\widehat{=}} gd(A_k) \qquad (\textit{kill})$$

3.1.4 Weakest precondition of a timed action

The weakest precondition of timed action is divided into two parts depending on its enabledness during execution: (a) a timed action is enabled throughout its execution time allowing the finish action to update the write variables, that is, in terms of the state predicates the kill predicate is false ($K() = F$) during the whole computation period of the timed action and (b) a timed action becomes disabled during execution preventing the update of the write variables. In this case the kill predicate evaluates to true ($K() = T$) at some time point during the operation time. This means that the execution of the kill action corresponds to the *skip* action. Thus, the weakest precondition of a timed action is:

$$\mathbf{wp}((A[\![dA]\!]), Q) = \qquad (\textit{weakest precondition of timed action})$$
$$\mathbf{wp}(A, Q[(gt + delay)/gt]) \wedge (\neg gd(A) \Rightarrow Q))$$

where the latter part is the weakest precondition of the *skip* action.

3.2 Non-atomic Operation of Timed Actions

The non-atomic sequence, defined in (2.3), whose operation is controlled by an auxiliary variable set by the actions themselves operates in the time domain as intended. However, the operation of the parallel composition, see (2.4), is not that apparent due to the use of the terminal action $T_{\|}$. The terminal action enables a new execution round by resetting all the Boolean control variables in one atomic step after all the actions in the composition have performed their operation in turn. Thus, the parallel composition is considered in more detail.

The parallel composition of timed actions A and B whose delays are dA and dB is of the following form (based on (2.4)).

$$\begin{aligned} A[\![dA]\!] \parallel B[\![dB]\!] \mathrel{\widehat{=}} & \left((A^{\|}_{f} \ \ [\!] \ \ A^{\|}_{k}) \ [\!] \ (B^{\|}_{f} \ \ [\!] \ \ B^{\|}_{k}) \ [\!] \ T_{\|}\right) \\ & /\!\!/ \ \left(A^{\|}_{s} \ [\!] \ B^{\|}_{s}\right) \\ & /\!\!/ \ Pt \end{aligned}$$

where the terminal action is placed into the end segment where it will be executed after its guard $gd(T_{\|}) \mathrel{\widehat{=}} \neg p_A \wedge \neg p_B$ evaluates to true. The execution of the terminal action $T_{\|}$ does not consume time, as it is part of the control structure. The definition of the parallel composition is:

$$\begin{aligned} & A_1[\![dA_1]\!] \parallel A_2[\![dA_2]\!] \mathrel{\widehat{=}} \\ & \quad ((A_{f,i} \ \ [\!] \ \ A_{k,i}) \ \ [\!] \ \ (A_{f,2} \ \ [\!] \ \ A_{k,2})) \\ & \quad /\!\!/ \ ((A_{s,1} \ \ [\!] \ \ A_{s,2})) \\ & \quad /\!\!/ \ Pt \end{aligned} \qquad (\textit{parallel composition}) \quad (3.2)$$

---

where it is required that the composed timed actions are independent ($vA \cap wB = \emptyset$ and $wA \cap vB = \emptyset$). Actually the above definition is the same as the composition of timed actions defined in the previous section. However, to reflect the untimed operation, a timed action is allowed to kill another one, allthough they are operating in parallel and they are required to be independent of each other. This is further illustrated in Chapter 4, when the performance modeling of parallel execution is discussed. For more detailed discussion, see, for example, [86].

3.3 Synchronous Operation

The synchronous composition for conventional actions, defined by (2.2) on page 17, does not include a situation in which, for instance, external asyn-

chronous control signal is willing to disable the composition or parts of it. However, in Timed Action Systems [86], the synchronous composition is defined in a way that this kind of behavior is possible, and thus the atomic timed synchronous composition is defined by:

$$
\begin{aligned}
& A_1[\![dA_1]\!] \vee A_2[\![dA_2]\!] \mathrel{\widehat{=}} (A_1 \vee A_2)^{\vee}_{f\&k} && (\textit{timed synchronous composition}) \\
& \ /\!/ \ \ (A_1 \vee A_2)^{\vee}_{s} && (3.3) \\
& \ /\!/ \ \ Pt
\end{aligned}
$$

where the clock signal is modeled by the fact that there is only one start action and the merged finish and kill actions. They are of the form:

$$
\begin{aligned}
(A_1 \vee A_2)^{\vee}_{s} \mathrel{\widehat{=}} & \neg bS \rightarrow \\
& delayS := delayS'.dpS \\
& ; stateS := (wA_1 \cup wA_2, gt, gt + delayS) \\
& ; [; 1 \leq i \leq 2 : gd(A_i) \rightarrow bd(A_i)[stateS.wA_i/eA_i] \ /\!/ \ skip] \\
& ; bS := T; \\
(A_1 \vee A_2)^{\vee}_{f\&k} \mathrel{\widehat{=}} & bS \wedge (gt := stateS.ft) \rightarrow \\
& ; [; 1 \leq i \leq 2 : gd(A_i) \rightarrow wA_i := statewA_i \ /\!/ \ skip] \\
& ; bS := F;
\end{aligned}
$$

where the synchronous composition $A_1 \vee A_2$ is denoted (for simplicity) by S. The *stateS* is the union of the states of the composed timed actions A_1 and A_2 ($stateS \mathrel{\widehat{=}} stateA_1 \cup stateA_2$). The write variables of the timed actions A_1 and A_2 are referred by $statewA_1$ and $statewA_2$, respectively. Observe that the above mentioned external interruption possibility is modeled by using merged finish and kill actions.

A *non-atomic synchronous composition* is a composition where the operation of the composed timed actions commence their operation at the same time, but these operations are finished individually by storing the result onto new variables until the slowest timed action has performed its operation after which the write variables are updated. The composition is of the form:

$$
\begin{aligned}
& A_1[\![dA_1]\!] \bowtie A_2[\![dA_2]\!] \mathrel{\widehat{=}} && (\textit{timed non-atomic synchronous composition}) \\
& \Big((A^{\bowtie}_{f,1} \vee A^{\bowtie}_{k,1}) \ [\!] \ (A^{\bowtie}_{f,2} \vee A^{\bowtie}_{k,2}) \ [\!] \ T_{\bowtie} \Big) && (3.4) \\
& \ /\!/ \ \Big(A^{\bowtie}_{s,1} \ [\!] \ A^{\bowtie}_{s,2} \Big) \\
& \ /\!/ \ Pt
\end{aligned}
$$

where the terminal action $T_{\bowtie}$ is executed after all the synchronously composed actions have performed their operations. The actions in the composition are defined using the non-deterministic choice; auxiliary, local Boolean variables pA_i (initialized to *false* (F)) whose responsibility is to guarantee that the write variables are updated simultaneously; an auxiliary local Boolean variable oA_i, which indicates that the timed action is a *skip* action; and, furthermore, auxiliary variables mA_i that store the results of the computation until the actual write variables are updated. The actions are of the form:

$$
\begin{aligned}
A^{\bowtie}_{s,i} \;\widehat{=}\; & \neg pA_i \wedge \neg bA_i \wedge \neg oA_i \rightarrow \\
& \quad delayA_i := delayA_i'.dpA_i \\
& \quad ; stateA_i := (wA_i, gt, gt + delayA_i) \\
& \quad (gd(A_i) \rightarrow bd(A_i)[stateA_i.wA/wA_i]; bA_i := T) \\
A^{\bowtie}_{f,i} \;\widehat{=}\; & gt = stateA_i.ft \rightarrow \\
& \quad ((bA_i \wedge gd(A_i) \rightarrow mA_i, bA_i := stateA_i.wA, F) \;[\!]\; oA_i \rightarrow skip) \\
& \quad ; pA_i := T; \\
A^{\bowtie}_{k,i} \;\widehat{=}\; & (bA_i \wedge \neg gd(A_i) \rightarrow bA_i := F; pA_i, kA_i := T) \\
T_{\bowtie} \;\widehat{=}\; & \textstyle\bigwedge_{i=1}^{2} pA_i \rightarrow [1 \leq j \leq 2 \colon (\neg kA_j \wedge \neg oA_j \rightarrow wA_j := mA_j \\
& \qquad\qquad\qquad\qquad\quad [\!]\; kA_j \rightarrow kA_j := F \\
& \qquad\qquad\qquad\qquad\quad [\!]\; oA_j \rightarrow oA_j := F) \\
& \qquad\qquad\qquad\qquad\quad ; pA_j := F]
\end{aligned}
$$

where the synchronous timed action $A^{\bowtie}_i$ disables itself after it has performed its operation. The execution of its operation depends on the enabledness of a timed action at the time when a new clock cycle starts. That is, if a timed action is disabled at that time it acts as a **wait** action, it does not consume the amount of time specified by its delay. The terminal action $T_{\bowtie}$ reveals the result of the computations after which the synchronous timed actions $A^{\bowtie}_i$ are enabled to next computation round.

Gated clock

Gated clocks are used to reduce power consumption in hardware systems by selectively halting the clock in portions of system wherein active operation is not being performed. In Timed Action Systems, the clock gating is implemented by expanding the non-atomic synchronous composition with a Boolean variable. A Boolean variable g when activated stops the update of the guarding variables pA_i, which, on the other hand, prevents the new execution round of the synchronously composed timed actions. The execution is prevented as long as the gating signal is active. Depending on the

model the activation signal may set the guarding signal either to $true(T)$ or $false(F)$. In the following definition the guarding signal is activated by setting it false. The gated non-atomic synchronous composition, denoted by $A_1 \overset{g}{\bowtie} A_2$, is defined by:

$$A_1[\![dA_1]\!] \overset{g}{\bowtie} A_2[\![dA_2]\!] \mathrel{\widehat{=}} \qquad (\textit{gated non-atomic synchronous composition})$$

$$\Big((A^{\bowtie}_{f,1} \ [\!]\ A^{\bowtie}_{k,1}) \ [\!]\ (A^{\bowtie}_{f,2} \ [\!]\ A^{\bowtie}_{k,2}) \ [\!]\ g \to T_{\bowtie}\Big) \tag{3.5}$$

$$/\!/ \ \Big(A^{\bowtie}_{s,1} \ [\!]\ A^{\bowtie}_{s,2}\Big)$$

$$/\!/ \ Pt$$

---

where the start, finish, and kill actions are presented as earlier, but the terminal action updates the write variables only when the guard g is active. That is, it blocks the update of the write variables corresponding the guarding of the clock signal in the real circuits.

3.4 Timed Action System

A timed action system has the form:

sys $\mathcal{A}$ (**imp** p; **exp** c;)(g_A;) ::
|[
type
$type$: Def;
delay
dp: $dp0, dC$: $[dC_{min}, dC_{max}], dA_i : dA_i0$;
constraint
$constraint$: (B);
variable
l_A;
private procedure
$p[\![dp]\!]$(**in** x : **out** y): (P_p);
public procedure
$c[\![dc]\!]$(**in** x : **out** y): (P_c);
action
$A_i[\![dA_i]\!]$: (aA_i);
initialization
$g_A, l_A := g_A0, l_A0$;
execution
forever do `composition of timed actions` A_i **od**
]|

where three main parts can be observed as in the conventional action system: *interface*, *declaration*, and *iteration*.

The interface part declares those variables g_A that are visible outside the timed action system boundaries, and thus accessible by other systems. Furthermore, it introduces interface procedures that are imported (p) by or introduced in and exported by (c) the system. If a timed action system does not have any interface variables or procedures, it is a *closed timed action system*, otherwise it is an *open timed action system*.

The declaration part introduces all the new type definition, local variables l_A action definitions aA_i with their labels A_i. A new element is the **delay** clause that describes the delays of the procedures and timed actions and their labels. A delay is an amount of time (not necessary very long), and it is used in delay predicates described in Sect. 3.5. Furthermore, in this thesis, delays are described using symbols, which can be later on replaced using actual values. The delays are joined within the timed actions using *delay brackets* $[\![\ ]\!]$. In addition, the declaration part introduce constraints that define conditions, which can be either *functional* or *temporal*, whose strict adherence is mandatory. Finally the initialization sets the system into a well defined state from where the system may safely start its operation.

The last item, the iteration part defines the reactive behavior of the system. It describes the composition of timed actions defined in the declaration part. The time when the computation is commenced is set in the initialization, but it is of no importance as only the relative ordering of timed action is important, The iteration part does not address delays for the timed actions, it only defines the behavior of the system.

Modeling aspect. Consider two timed action systems $\mathcal{A}$ and $\mathcal{E}nv$ whose local variables are distinct and the latter is the environment of the former. The parallel composition of these two systems is denoted by $\mathcal{A} \parallel \mathcal{E}nv$. The definition of the parallel composition follows from the conventional Action Systems notation defined in Sect. 2.5.1 where it is defined to be a another action system whose distinct global and local identifiers (variables and actions) consist of the identifiers of component systems. In addition to the conventional parallel composition, the time propagation is shared among the (sub)systems as it is shared among the timed actions in (3.1). By opening the timed action notation the **execution** clause of the parallel execution becomes:

$$\begin{aligned}\textbf{forever do}\ & [\ [\!]\ 1 \le i \le n : A_{f,i}\ \ [\!]\ \ A_{k,i}]\ \ [\!]\ \ [\ [\!]\ 1 \le j \le m : E_{f,i}\ \ [\!]\ \ E_{k,i}] \\ & /\!/\ \ [\ [\!]\ 1 \le i \le n : A_{s,i}]\ \ [\!]\ \ [\ [\!]\ 1 \le i \le m : E_{s,i}] \\ & /\!/\ \ Pt\ \textbf{od}\end{aligned}$$

where only one time propagation action exists. Furthermore, modeling the behavior of the system $\mathcal{A}$ and its environment $\mathcal{E}nv$, it is assumed that there is always at least one enabled timed action.

Hierarchical composition. The hierarchical composition of timed action systems follows the introduced composition in action systems. The hierarchical composition of timed action systems $\mathcal{H}$ and its subsystem $\mathcal{A}$ is defined to be another system whose **execution** clause has the the form:

$$
\begin{aligned}
\textbf{forever do } & [\ [\!] \ 1 \le i \le n : H_{f,i} \ \ [\!] \ \ H_{k,i}] \ \ [\!] \ \ [\ [\!] \ 1 \le j \le m : A_{f,i} \ \ [\!] \ \ A_{k,i}] \\
& /\!/ \ \ [\ [\!] \ 1 \le i \le n : H_{s,i}] \ \ [\!] \ \ [\ [\!] \ 1 \le i \le m : A_{s,i}] \\
& /\!/ \ \ Pt \ \textbf{od}
\end{aligned}
$$

3.4.1 Computation model

The execution order of timed actions progress is presented in Sect. 3.1.2. Observe that before the system may start its operation it has to be initialized after which the operation proceeds into the iteration part, the **forever do-od** loop of a timed action system. In the iteration part timed actions are sequentially selected for execution based on their composition and enabledness. The computation model is depicted in Fig. 3.1, where the variables (both global and local ones) are *initialized* into predefined values setting the system in a state from which the computation may safely begin. After initialization enabled actions are *executed*, and the operation proceeds to the *time segment*, where the *time propagation* action examines the finish times of the scheduled actions, and sets the *global time* to the nearest finish time. Then the operation proceeds towards the end segment, where several *finish or kill* actions are executed. The execution of the finish action reveals the result of computation, that is, the *write variables of the timed action are updated*, and the execution kill action releases a timed action for further executions. After all enabled finish and kill actions are executed one execution round is successfully completed, and the computation proceeds towards a new one. The operation continues as long as there is enabled finish or kill actions by propagating time again. In a timed action point of view there might occur a situation in which there are no such timed actions within the system whose computation may be commenced or finalized. In this case the system is said to be *temporarily delayed*, and its computation resumes execution when the environment invokes the system.

3.4.2 Timed action semantics

As stated above, the system behavior is a set of sequences. In Timed Action Systems, the behavior of a system is defined in terms of timed actions whose

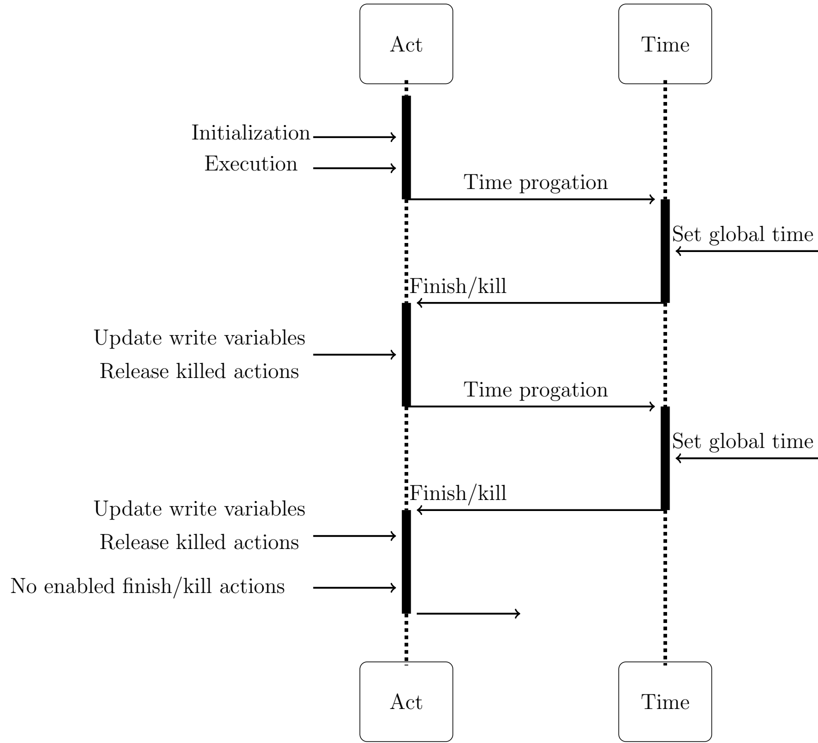


Figure 3.1: The computation model of Timed Action Systems

execution cause state changes by updating the values of the system variables. Therefore, the behavior of a timed action system $\mathcal{A}$ is defined in terms of timed actions by:

$\mathcal{TAS}(\mathcal{A}) = \{\mathcal{EX}_1(\mathcal{A}), \mathcal{EX}_2(\mathcal{A}), \ldots\}$ (*behavior of a timed action system*)

---

where $\mathcal{EX}_i(\mathcal{A})$ is an execution sequence of timed actions:

$$\begin{aligned}\mathcal{EX}_i(\mathcal{A}) &\mathrel{\widehat{=}} \langle A_1, A_2, \ldots\rangle \\ &= \langle A_i\rangle \text{ where } i \in \mathbb{N}, A_i \in \mathcal{A}\end{aligned}$$ (*execution sequence of system*)

Parallel. Actions Systems as well as Timed Action Systems are sequential by nature. Therefore, the parallel behavior of timed actions is modeled because parallelism is not built into the semantics. Actions are said to operate in parallel if they are enabled at the same state, and, furthermore,

they are independent of each other. Independency of timed actions can be described as follows:

$$\mathcal{I}(A, B \in \mathcal{A}) \mathrel{\widehat{=}} (vA \cup vB = \emptyset) \qquad (\textit{independency of timed actions})$$
$$\vee\, (rA \cup wB = \emptyset \wedge rB \cup wA = \emptyset wA \cup wB = \emptyset)$$

---

where the first condition defines trivially independent actions as they do not share any variables. The second condition, on the other hand, says the other action does not read the variables that are written by the other action, and, furthermore, they do not write on the same variables. Thus, the definition of parallel actions becomes:

$$\mathcal{P}(A, B \in \mathcal{A}) \mathrel{\widehat{=}} \mathcal{I}(A, B) \wedge \qquad (\textit{parallel timed actions})$$
$$\Big(\exists R, K \in \mathcal{BEH}(\mathcal{A}) : \exists r, k \in R \cap K :$$
$$s \Rightarrow \big(gd(A) \wedge gd(B) \wedge \textbf{wp}\ (A, \textbf{wp}\ (B, k))$$
$$\wedge\, \textbf{wp}\ (B, \textbf{wp}\ (A, k))\big)\Big)$$

---

where R and K are sequences of the system $\mathcal{A}$ and their intersection $R \cap K$ is performed in a set theory fashion, that is, the set of states that exists in both of the execution sequences is obtained. The condition states that when two actions are independent and there are two common states in two different sequences of the system $\mathcal{A}$, which are connected by the sequential execution of the independent actions in either order. That is, the execution order does not affect the result of computation.

Successor. Consider an execution sequence $\mathcal{EX}(\mathcal{A})$, A is a predecessor of B and B is a successor of A if B reads variables written by A ($wA \cup rB \neq \emptyset$), and thus, it is defined by:

$$\mathcal{SUCC}(A, B \in \mathcal{A}) \mathrel{\widehat{=}} \neg\mathcal{P}(A, B) \wedge \exists s \in \mathcal{SEQ}(\mathcal{A}) \qquad (\textit{successor})$$
$$\Rightarrow gd(A) \wedge \textbf{wp}\ (A, gd(B))$$

---

which defines that the timed actions A and B are not parallel and there exists a state in the sequence $\mathcal{SEQ}(\mathcal{A})$ in which A is enabled and executed leading into the next state in which B is enabled.

3.4.3 Computation path

Computation path, denoted by $\mathcal{CP}(A, B)$ is a finite sequence of immediate successors, which leads from A to B. An immediate successor is an action

whose execution is instantly followed by an execution of its predecessor in the path where finite loops are allowed. This extraction is very convenient in defining and verifying system behaviour. Furthermore, computation paths are a natural way to think SoC, which is modular by its nature. One important point of modularity is that it allows us to concentrate smaller design tasks one at a time and then interconnect smaller components together in a divide-and-conquer manner. Formally the computation path between two actions is of the form:

$$\mathcal{CP}(A,B) \mathrel{\widehat=} \big\langle A_1, A_2, \cdots, A_n \big\rangle \qquad (\textit{computation path}) \tag{3.6}$$

$$\begin{aligned} \text{where } & A_i \in \mathcal{EX}_k(\mathcal{A} \parallel \mathcal{E}nv) \\ & \wedge \forall i\colon 1 < i \leq n\colon \big(\mathcal{SUCC}(A_{i-1}, A_i)\big) \\ & \wedge A_1 = A \wedge A_n = B \end{aligned}$$

where the amount of the actions is limited to n. It is also important to observe that actions in a computation path are successors of each other, that is, executed in sequentially.

Observe that the above definition does not uniquely define one computation path as in embedded computing systems actions occur periodically, and therefore it is required to delimit a computation path further. Consider the following execution sequence:

$$\big\langle A, B, C, D, F, G, C, D, F, G, A, B, C, D, \cdots \big\rangle$$

where by defining a computation path $cp(A,G)$ the first two paths are: $\big\langle A, B, C, D, F, G \big\rangle$ and $\big\langle A, B, C, D, F, G, C, D, F, G \big\rangle$. Therefore, it is important that one have means to specify, which action is our target to be able to precisely specify a computation path in question. This can be done by using a superscript as follows: $\mathcal{CP}(A, B^k)$, where k defines which one of the actions B is the last action in the path. In the above example computation path the former path could be obtained by defining $\mathcal{CP}(A, G1)$, shortly $\mathcal{CP}(A, G)$, and the latter one by $\mathcal{CP}(A, G2)$. Two problems arises when consideting the definition given so far: How to pinpoint the first action in the path and how to gather all the paths between two points created by parallel behavior. The latter one can be solved by defining a set of computation paths, as follows:

$$cp(A,B) \mathrel{\widehat=} \{x | x = \mathcal{CP}(A,B)\} \tag{3.7}$$

The former problem regarding the beginning of a computation path is not that trivial because of the reactive system model: actions are triggered for execution in a non-deterministic or deterministic (e.g., periodic) manner.

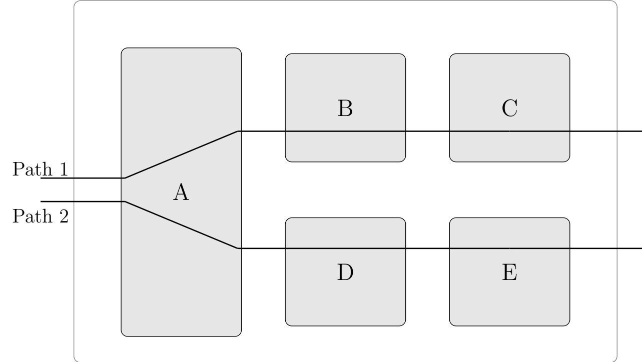


Figure 3.2: The computation paths of the composition $A;\big((B;C) \parallel (D;E)\big)$

This is solved by utilizing the modular structure of SoCs in a way that a beacon is set on the boundary of a system. This bounds the problem domain quite nicely and extracts the required information out of the execution sequence.

Based on the above reasoning, computation paths through a system, say $\mathcal{A}$, is a set of computation paths whose first action reads from the input variables of the system and whose last action writes on the output variables of the system. Actions that read from the input variables are called by *input actions* I and those actions that write on the output variables are called by *output actions* O. In conclusion, one gather all the paths between the system's *input-output* variables:

$$cp(\mathcal{A}) \mathrel{\widehat{=}} \{x|x = \mathcal{CP}(I, O) \wedge rI \in r\mathcal{A} \wedge wO \in w\mathcal{A}\} \tag{3.8}$$

Example 3.1. Consider the following composition:

$$A;\big((B;C) \parallel (D;E)\big)$$

For such a composition there exists two possible computation paths which are illustrated in Fig 3.2.
End of example.

3.5 Delay

Delays define the time that timed actions consume in their operation, and they are defined in the declaration part of a timed action system and associated with a timed action using the labels and delay brackets. Delays can be described as long or short depending on the operations performed in the actions.

3.5.1 Delay predicates

A *deterministic delay* defines a precise delay for a timed action A, and a *non-deterministic delay* predicate defines a delay whose value is chosen non-deterministically: The delay predicates are defined by

---

$dpA \mathrel{\widehat=} delayA' = dA0$ (*deterministic delay*)

$dpA \mathrel{\widehat=} dA_{min} \bullet delayA' \bullet dA_{max}$ (*non-deterministic delay*)

---

where dpA is a delay predicate that determines the delay of the timed action, which will be assigned to the delay variable $delayA$ in the start action A_s. The delay statement thus becomes: *delayA := delayA'.(delayA' = dA0).* The bullet $\bullet$ in the non-deterministic assignment is one of the following inequalities: $<$, $>$, $\leq$ and $\geq$.

In Timed Action System context, the delays are written as:

$\mathbf{delay}\ dA : dA0$ (*deterministic delay*)

$\mathbf{delay}\ dA : [dA_{min}, dA_{max}]$ (*non-deterministic delay*)

3.5.2 Delay calculation rules

The delay calculation rules needed for the timing analysis and performance estimation is presented [1]. These delay calculation rules and their compositions are defined by:

$\Delta(A) \mathrel{\widehat=} \quad dA$	(*action*)	(3.9)
$\Delta\{p\} \mathrel{\widehat=} \Delta([p]) = dp$	(*predicate*)	(3.10)
$\Delta(A_1; A_2) \mathrel{\widehat=} \Delta(A_1) + \Delta(A_2)$	(*sequential*)	(3.11)
$\Delta(g \rightarrow A) \mathrel{\widehat=} \Delta([g]; A) = \Delta([g]) + \Delta(A)$	(*guarded action*)	(3.12)
$\Delta(A\langle proc\rangle) \mathrel{\widehat=} \Delta(A) + \Delta(proc)$	(*procedure*)	(3.13)
$\Delta(A_1 \ [\!]\ A_2) \mathrel{\widehat=} \{\Delta(A_1), \Delta(A_2)\}$	(*alternative*)	(3.14)
$\Delta(A_1 \vee A_2) \mathrel{\widehat=} \mathbf{Max}(\mathbf{Max}(\Delta(A_1)), \mathbf{Max}(\Delta(A_2)))$	(*synchronous*)	(3.15)
$\Delta(A_1 \bowtie A_2) \mathrel{\widehat=} \mathbf{Max}(\mathbf{Max}(\Delta(A_1)), \mathbf{Max}(\Delta(A_2)))$	(*synchronous*)	(3.16)

---

where

(3.9) The *action delay* rule calculates the delay of a timed action by eval-

[1]For detailed study, the reader is urged to get familiarized with [86]

uating the delay statement. The delay equals the subtraction of the start and finish times of a timed action.

(3.10) The *predicate delay* rule calculates the time it takes to evaluate the predicate, either assert $\{g\}$ or assumption $[g]$.

(3.11) The *sequential delay* rule sums the delay of sequentially executed timed actions whose delays are calculated with (3.9).

(3.12) The *guarded action delay* rule calculates a delay for a guarded timed action. It consist of two components based on the definition of a guarded action: the evaluation of the guard $gd(A)$ and the time to perform functionality $bd(A)$.

(3.13) The *procedure delay* rule sums the delay of the calling action with the delay of the called procedure: $A\langle proc\rangle$ denotes a timed action that calls the procedure *proc* in its body.

(3.14) The *alternative delay* rule gives a set of delays each of which reflects an alternative delay (computation) path.

(3.15) The *atomic synchronous delay* defines the clock cycle time in a synchronous systems according the slowest action in the system.

(3.16) The *non-atomic synchronous delay* defines the clock cycle time in a similar manner as the atomic delay presented above. There is notational difference.

3.5.3 Procedure delay

Although procedures, both private and public, are introduced separately in the declaration part of a system, the operation performed by a procedure is considered to be a part of the calling action. In other words, a procedure is a parametrized subaction (see Sect. 2.6). The justification for having separate functionality and timing definitions for procedures and actions is modularity, which enables one to develop the procedures apart from the calling action. Nevertheless, the procedure delay, defined by (3.13), and the action delay adds up the total delay of the action.

3.5.4 Procedure based communication delay

The procedure based communication [68], defined in Sect, 2.6.1, uses remote procedures to model communication channels between action systems. In this section, the timing information is included into that model. Consider

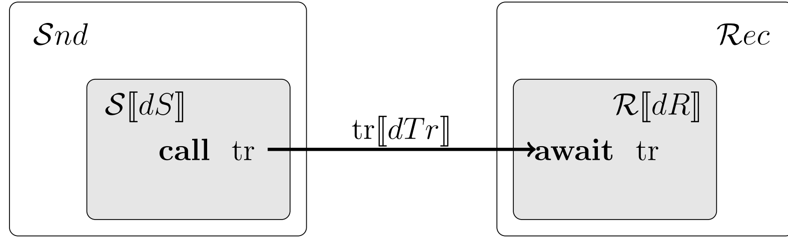


Figure 3.3: Timed action systems $\mathcal{S}nd$ and $\mathcal{R}ec$ communication directly with each other using a public procedure c

the timed action systems $\mathcal{S}nd$ and $\mathcal{R}ec$ whose internal activities are denoted with timed actions:

$$S[\![dS]\!] \mathrel{\widehat{=}} \textbf{call } tr(a,b,c)$$
$$R[\![dR]\!] \mathrel{\widehat{=}} \textbf{await } tr(a,b,c)$$

where the procedure is of the form:

$$tr[\![dTr]\!] \mathrel{\widehat{=}} P_{tr}$$

The delays of the actions are deterministic whereas the delay of the public procedure is non-deterministic, because its value depends on great deal of variables such as the state of the communication network and the communication protocol. Thus, the delays are defined by:

$$\begin{aligned} dS &\mathrel{\widehat{=}} dS0 \\ dR &\mathrel{\widehat{=}} dR0 \\ dTr &\mathrel{\widehat{=}} [dTr_{min}, dTr_{max}] \end{aligned}$$

where the non-deterministic communication delay consist of two components: a static delay $dTr0$ and the synchronization delay $dSync$ in way that:

$$\begin{aligned} dTr &\mathrel{\widehat{=}} [dTr_{min}, dTr_{max}] \\ &= [dTr0 + dSync_{min}, dTr0 + dSync_{max}] \\ &= dTr0 + dSync \end{aligned}$$

where the synchronization delay defines an amount of time required to get two communication parties to synchronize their operations. In this model the synchronization delay is included into the communication procedure, and therefore it can be considered to be caused by either of the communication parties.

To calculate the duration of the communication activities, the communication is regarded as a single atomic action SR based on the definition of the procedure based communication:

$$SR \mathrel{\widehat{=}} S[R[P_{tr}[a,b,c/x,y,z]/\ \textbf{await}\ Tr]/\ \textbf{call}\ Tr(a,b,c)]$$

whose delay can be calculated by using the sequential delay rule (3.11):

$$\begin{aligned}
\Delta_{comm} &\mathrel{\widehat{=}} \Delta(SR) \\
&= \Delta(S\langle tr\rangle) + \Delta(R) \\
&= \Delta(S) + \Delta(tr) + \Delta(R) \\
&= dS + dTr + dR
\end{aligned}$$

3.5.5 Computation path delay

A computation path delay between actions A and B is defined by:

$$\Delta\mathcal{CP}(A,B) \mathrel{\widehat{=}} \Delta\,(A; 0 \le i \le n \wedge \mathcal{SUCC}(A_i, A_{i+1}); B) \qquad (\mathcal{CP}\ \textit{delay}) \quad (3.17)$$

where the delay includes all the actions in the computation path, that is, A and sequence of successors leading from A to B, including A and B themselves.

The symbol * is used to indicate, by its position, whether the delay of a timed action is included into the computation path or not. The actions A_i are immediate successors in the path leading eventually to B. In the above definition, both delays are included into the computation path delay. Next both or either delay is excluded from the computation path delay, and thus, the definitions are of the form:

$$\begin{aligned}
\Delta\mathcal{CP}({}^*A, {}^*B) &\mathrel{\widehat{=}} \Delta\mathcal{CP}(A,B) - \Delta(B) \\
&= \Delta\,(A; 0 \le i < n \wedge \mathcal{SUCC}(A_i, A_{i+1}))
\end{aligned}$$

$$\begin{aligned}
\Delta\mathcal{CP}(A^*, B^*) &\mathrel{\widehat{=}} \Delta\mathcal{CP}(A,B) - \Delta(A) \\
&= \Delta\,(0 \le i < n \wedge \mathcal{SUCC}(A_i, A_{i+1}); B)
\end{aligned}$$

$$\begin{aligned}
\Delta\mathcal{CP}(A^*, {}^*B) &\mathrel{\widehat{=}} \Delta\mathcal{CP}(A,B) - \Delta(A) - \Delta(B) \\
&= \Delta\,(0 \le i < n \wedge \mathcal{SUCC}(A_i, A_{i+1}))
\end{aligned}$$

3.6 Chapter Summary

In this chapter, the time spiced Action Systems, Timed Action Systems was described. After describing the time domain, a basic component of the formalism, a timed action was introduced. Timed action is an action whose computation result is postponed by the duration specified by a delay associated with it. After introducing the basics of timed action, it was time to proceed to system level modeling issues, and therefore, timed action systems was defined. The form of the timed action system is similar to the action system, which makes the adoption of Timed Action System as easy as possible for a designer familiar with the Action Systems. Having introduced the timed action system, the delay predicates was described and a way to calculate static delays for timed action compositions and timed action systems was presented. The described constructs and properties of Timed Action Systems are adopted for the power modeling framework defined in the next chapter.

Chapter 4

Power Estimation

Methods to evaluate various performance metrics such as area, delay, and power consumption at all levels of the design hierarchy are an important part of the design process. While it is typically the case that lower level estimation tools offer higher estimation accuracy, their use to explore architectural trade-offs during higher-level design tends to be prohibitively time consuming. Therefore, to avoid costly redesign steps, power estimation techniques are required which can estimate the power consumption at a high level of abstraction, such as when a circuit is modeled with Boolean functions, as in [44, 45, 71, 88]. These modeling frameworks are targeted to operate in RTL. However, the initial system specifications in the timed action systems formalism have an even higher abstraction level, and therefore the model introduced here is not directly comparable with the RTL models. Furthermore, the study presented in this chapter includes a static power consumption model as well.

In the Action Systems formalism, modeling, verification, and analysis have been confined only to logical properties. However, to analyze power consumption, timing information is required. Therefore, the presented power modeling framework is targeted to Timed Action Systems, presented in Chapter 3, which can be used to analyze system's timing properties. The timing information is then exploited in power consumption modeling.

The average power consumption of an arbitrary action A is defined by:

$$P(A) \mathrel{\widehat{=}} P_{dyn}(A) + P_{stat}(A)$$

where the latter is *static power consumption* of the action A and the former is *dynamic power consumption* of the action A. Dynamic power consumption is related to system operation, and therefore it can be decreased, for instance by using low-power clocking techniques [47] such as clock gating, described in Section 3.3, or by using design methods such as asynchronous design [53]. In turn the static power consumption, is caused by the leakage

current I_{leak}, which is the combination of the subthreshold leakage (a weak inversion current across the device) and the gate-oxide leakage (a tunneling current through the gate oxide insulation) [49]. Detailed analysis of the leakage current can be found, for instance, in, [33, 49]. In general, both the subthreshold and the oxide leakage depends on the total gate width or more approximately the *gate count*, temperature (subthreshold leakage), supply voltage, and oxide thickness (oxide leakage). The static power consumption can no longer be ignored, since the power consumption due to chip leakage is approaching the dynamic power consumption, and the projected increases in off-state subthreshold leakage show it exceeding total dynamic power consumption as the technology drops below the 65 nm feature size [49]. Emerging techniques to moderate the gate-oxide tunneling effect could bring gate leakage under control by 2010.

This chapter concentrates on building a power modeling framework for Timed Action Systems. An area complexity model for timed actions is first presented and then extended to cover an action system. Furthermore, the chapter includes a model to evaluate instantaneous power dissipation in the Timed Action System context.

4.1 Area Complexity of an Action

The area complexity measure is adopted to estimate the complexity of a chip architecture, and thus the area of the chip. This measure has an impact on both dynamic and static power consumption. For dynamic power consumption, the area complexity can be used to model physical capacitance [51]. For static power consumption, the area complexity can be used to approximate the gate count of a system, and thus to estimate the amount of leakage [49]. To model the area complexity of an action, the following information is utilized:

- The sets of read and written variables of an action
- The non-deterministic assignment $x := x'.Q$.

where the read and write sets of an action describe the input and output variables of the action, respectively. The non-deterministic assignment is the generalization of the assignment operation, and therefore it can be used to describe any operations on variables in the action context. The preliminary constraint of the area complexity model follows from this property. That is, to simplify the presentation and the area complexity model, the non-deterministic assignment is selected as a base action. In other words, the functionality of an arbitrary action must be described using a non-deterministic assignment. Consider an action A, defined by:

$$A \widehat{=} x := x'.Q$$

where x' is the value that satisfies the predicate Q, which is assigned into the variable x. The hardware illustration of the action A is shown in Fig. 4.1, where the assignment $x := x'$ is shown as a register block. The predicate, on the other hand, is viewed as combinatorial logic, which in turn performs the actual computation. The term "combinatorial cloud" is often referred to the combinatorial logic part of the circuit at the RTL, because the exact logic structure is not yet determined. The presented area complexity model is even more challenging regarding the interpretation of Q because the targeted abstraction level is above the RTL. Therefore, prior defining any area complexity model, one should discuss what kind of hardware operations are usually modeled by the predicate Q. In general, the predicate Q is often used to model arithmetic operations such as addition and multiplication, logical operations like AND and OR, and store operations. Consider the following examples:

$$\begin{aligned} Q &\widehat{=} (x' = y + z) \\ Q &\widehat{=} (x' = y \vee z) \\ Q &\widehat{=} (x' = y) \end{aligned}$$

where the first one is the addition operation between the two variables x and y, the second one is the OR operation between these two variables, and the last one stores the value of the variable y. That is, in general the predicate Q is thought as a combinatorial logic and it is a Boolean expression, not necessary containing any Boolean variables.For example, the variables x and y can be integer type of variables or Boolean type of variables. From the fact that the targeted abstraction level is above the RTL, and due to the different type of variables follows that size evaluation methods targeted to Boolean functions are not applicable. A combinatorial logic forms usually a layered structure, which, in turn, is described using a tree structure. *Binary Decision Diagrams (BDD)*, discussed in Sect. 1.4, forms the basis for the area complexity model. Naturally, generating a BDD from an abstract Boolean expression is not directly possible. In addition to the tree model, the Shannon's size equation, presented in Sect. 1.4, is adopted with certain restrictions, which are discussed later on in this chapter. Both of these techniques relate to size estimation of Boolean functions, which was an important criterion when selecting the method to evaluate the area complexity of abstract logic models. It is assumed that as the abstraction level of the system description decreases during system development, and therefore the methods targeted to Boolean functions are more and more accurate. Thus,

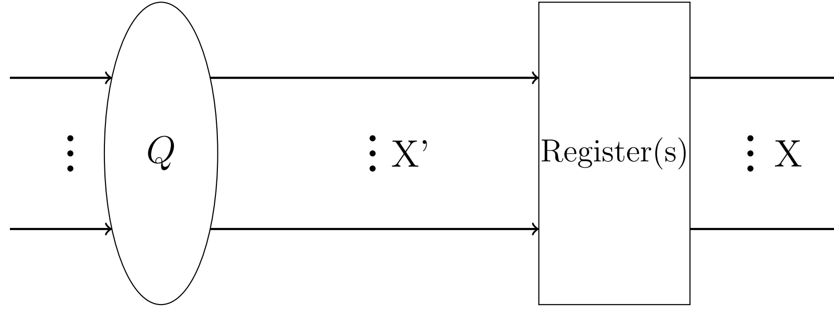


Figure 4.1: Illustration of a non-deterministic assignment $x := x'.Q$

in the end there should be enough information to generate BDD description for the system and use it as an area complexity model.

4.1.1 Area complexity of variables

In the abstract specification, the type of the variables (both local and global variables) can vary from Boolean variables to abstract type definitions, for instance, the type of the variable can be *Data*. Therefore, for the variables of an action, *a variable width* is defined, which is the minimum number of bits needed to represent a given variable. The width information is the basic necessity to model area complexity [85] due to the different variable types, typical to the high abstraction level. The area complexity of a variable x is defined by:

$$C(x) \mathrel{\widehat{=}} w_x$$

where w_x is the variable width. For a set of variables, say S, the area complexity is obtained by adding together the widths of the variables $x, x \in S$. Formally the area complexity of the set S is defined by:

$$C(S) \mathrel{\widehat{=}} \sum_{x \in S} w_x \qquad \text{(\textit{area complexity of a set of variables})} \quad (4.1)$$

4.1.2 Area complexity of non-deterministic assignment

Consider an action $A \mathrel{\widehat{=}} x := x'.Q$ whose access set is $vA \mathrel{\widehat{=}} rA \cup wA$, where rA is the set of read variables of A, and wA is the set of write variables of A. According to these sets, one can separate two parts in the area complexity model: the *assignment part* and the *predicate evaluation part*, where the former includes the analysis of the write set wA and the latter the analysis of the read set rA. At first, consider the assignment part or the write set of

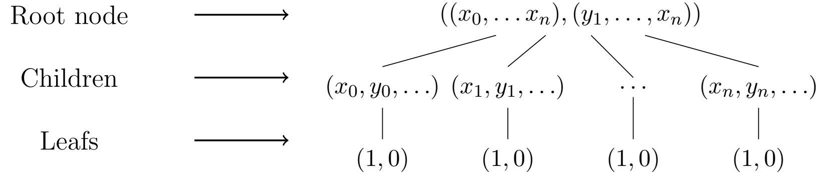


Figure 4.2: Tree model construction

the action A, illustrated as a register block in Fig. 4.1. Its area complexity is defined by:

$$C(wA) \mathrel{\widehat{=}} \sum_{x \in wA} w_x \qquad (\textit{area complexity of an assignment}) \quad (4.2)$$

---

where the wA is the write set of A. This definition is a direct application of (4.1).

To model the area complexity of a predicate Q one can consider the set vQ, which consist all the variables that appear in the predicate Q. In the case of the action A above, the set vQ equals the read set rA: $vQ = rA$. To construct the tree model for the area complexity, the set vQ is set as a root node, as shown in Fig. 4.2, where it is shown that the set vQ consists of two variables x and y, whose widths are equal: $w_x = w_y = n$ $(n > 0)$. To define the number of children in the tree, the area complexity $C(vQ)$ of the set vQ is defined using (4.1), after which it is divided by the cardinality of the set vQ:

$$\left\lceil \frac{C(vQ)}{|vQ|} \right\rceil$$

where $|vQ|$ is the cardinality of the set vQ. That is, the number of children is the average variable width in the set vQ, as shown in Fig. 4.2. This approach is selected because the computation of a predicate is carried out in a "bitwise" manner. For example, consider an addition operation between two four-bit variables, say x and y. The addition is carried out first by adding bits x_0 and y_0 together and then x_1, y_1 and possibly carry bit and so on. The tree model, presented so far, is an abstraction from a BDD tree, where the variables are replaced by the variable sets and subsets (children). Naturally, this tree model does not represent Boolean function as a directed acyclic graph, which is the case with BDDs. Therefore, the Shannon's size equation is used to analyze the area complexity of each child node. The Shannon's size equation is of form (for the ease of reference):

$$size = 2^m$$

where the m is the number of inputs in the Boolean function. In the tree model, shown in Fig. 4.2 each child node is considered as a Boolean function with $|vQ|$ input arguments ($m = |vQ|$) with two possible output values $'1'$ and $'0'$, which are denoted as *leafs* in the area complexity model. In this approach the generation of the tree model is done in a way that the exponential dependence of the Shannon's equation does not have significant negative effect to the model. In other words the model restricts the growth of the input arguments. The area complexity of the predicate evaluation is defined by:

$$C(Q) \mathrel{\widehat{=}} \left\lceil \frac{C(vQ)}{|vQ|} \right\rceil \cdot 2^{|vQ|} \qquad (\textit{area complexity of a predicate}) \quad (4.3)$$

where the first term calculates the average width of the input variables of the predicate. The average width describes the number of *children* in the tree model. The second term calculates the area complexity of the child node with two *leaf* nodes. The rounding becomes effective when the widths of the input variables are not equal. To summarize, the area complexity of the action $A \mathrel{\widehat{=}} x := x.'Q$ is defined by:

$$C(A) \mathrel{\widehat{=}} C(wA) + C(Q) \qquad (\textit{area complexity of an action } A) \quad (4.4)$$

where the first term is the area complexity of the assignment (4.2) and the second term is the area complexity of the predicate evaluation (4.3). A special case in the area complexity modeling of the non-deterministic assignment occurs when the predicate Q is of form $Q \mathrel{\widehat{=}} (x' = y)$, where y is a variable. For instance, consider an action $A \mathrel{\widehat{=}} z := z'.(z' = k)$, where z and k are variables of the same type. The area complexity of such A is

$$C(A) \mathrel{\widehat{=}} C(wA)$$

where the area complexity of the predicate Q (4.4) is zero as there is no computation but just the assignment. In other words, the action can be thought as a register block without any combinatorial logic.

Example 4.1. Assume that the predicate Q defines an integer addition: $Q \mathrel{\widehat{=}} (x' = a + b)$, where the variables a and b have the same width $w_A = w_B = 4$. The read set wQ is of form $wQ \mathrel{\widehat{=}} \{a, b\}$, which is set as a root in the tree model, shown in Fig. 4.3. Then the number of children is calculated by dividing the area complexity of the set vQ with the cardinality of the set vQ. The area complexity $C(vQ)$ is:

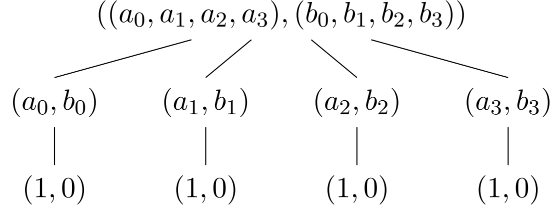


Figure 4.3: Example area complexity modeling

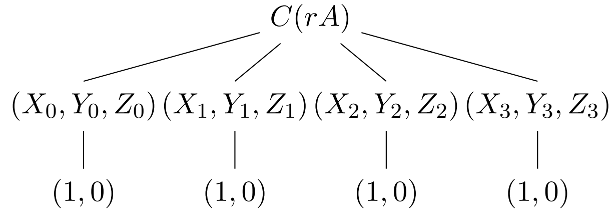


Figure 4.4: Example area complexity modeling

$$C(vQ) \overset{(4.1)}{=} \sum_{x \in vQ} w_x = w_A + w_B$$

where the area complexity of the set is calculated by adding together the widths of the variables in the read set. The number of children is then defined by: $\left\lceil \frac{C(vQ)}{|vQ|} \right\rceil = \left\lceil \frac{8}{2} \right\rceil = 4$. Each of these four children has two possible output values, and the area complexity of the child node is (according to Shannon's equation) $2^2 = 4$. The area complexity $C(Q)$ of the predicate Q is calculated by:

$$C(Q) \overset{(4.3)}{=} \left\lceil \frac{C(vQ)}{|vQ|} \right\rceil \cdot 2^{|vQ|} = \left\lceil \frac{8}{2} \right\rceil \cdot 2^2 = 16$$

In this example, both of the variables involved in the area complexity evaluation had the same width. Therefore, the area complexity modeling is further illustrated using a more general example.

End of example.

Example 4.2. The read set of a predicate $Q \mathrel{\widehat{=}} (x' = x+y+z)$ is $vQ \mathrel{\widehat{=}} \{x, y, z\}$. The variables x, y, and z are of the same type and their widths are $w_x = 3$, $w_y = 3$, and $w_z = 5$, respectively. The area complexity of the set vQ is

$$C(vQ) \overset{(4.1)}{=} \sum_{x \in rA} w_x = w_x + w_y + w_z = 3 + 3 + 5 = 11$$

Adopting $C(vQ)$ one can determine the area complexity of the predicate Q:

$$C(Q) \overset{(4.3)}{=} \left\lceil \frac{C(vQ)}{|vQ|} \right\rceil \cdot 2^{|VQ|} = \left\lceil \frac{11}{3} \right\rceil \cdot 2^3 = 32$$

where the first term is the number of children in the tree model whereas the second term calculates the area complexity of a child node. The tree structure of the model is shown in Fig. 4.4, where the number of child nodes is rounded because the widths of the variables in the set wQ are not equal. Therefore, based on the rounding, an approximation is generated where the width of each variables in the set vQ is four.
End of example.

4.1.3 Area complexity of a guarded action

Consider a guarded action of the form: $gd \to B$, where the action B is of form $B \mathrel{\widehat{=}} x := x'.Q$, and the guard gd is a condition that decides whether the action B is enabled or not. The definition of B follows directly from the assumption that the non-deterministic assignment is selected as a base action. The hardware illustration of the guarded action is shown in Fig. 4.5, where both the predicate Q and the guard gd are assumed to be combinatorial logic, and the write set forms the storage elements. That means that, the area complexity of the write set is defined by (4.2). To evaluate the area complexity of the predicate Q and the guard gd, consider an action $A \mathrel{\widehat{=}} gd \to x := x'.Q$. The read set of A consist of two type of variables: the variable(s) that appear in the guard and the variables that appear in the predicate, or in both. Because the guard is a condition that has to be true before the computation starts, it is fair to assume that gd and Q require separate logic components. Therefore, the read set rA of the action A is divided in two: vQ and vgd, where the former contains those variables that appear in the predicate Q, and the latter those variables that appear in the guard gd, $rA \mathrel{\widehat{=}} vQ \cup vgd$. The area complexity of these sets are evaluated separately for each set using (4.3). Thus, for guarded action A, the area complexity is formally defined by:

$$C(A) \mathrel{\widehat{=}} C(wA) + C(Q) + C(gd) \quad (\textit{area complexity of a guarded action } A) \tag{4.5}$$

4.1.4 Calculation rules for area complexity

Based on the definitions from the previous sections, the action level area complexity calculation rules are defined for various actions. These rules are also directly applicable for Timed Action Systems as well, and are needed

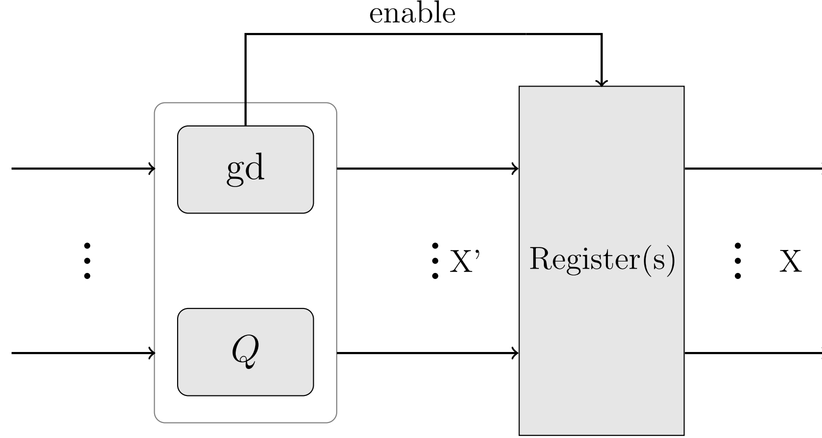


Figure 4.5: Illustration of a guarded non-deterministic assignment $gd \rightarrow x := x'.Q$

to analyze the area complexity of the system in the forthcoming sections. Consider the base action, the *non-deterministic assignment*, denoted by: $A \mathrel{\widehat{=}} x := x'.Q$. The area calculation rules for action A, and its compositions are defined by:

$C(A) \mathrel{\widehat{=}} C(wA) + C(Q)$	(*action*)	(4.4)
$C(g \rightarrow A) \mathrel{\widehat{=}} C(vg) + C(A)$	(*guarded action*)	(4.5)
$C(A\langle Proc\rangle) \mathrel{\widehat{=}} C(A) + C(Proc)$	(*procedure*)	(4.6)
$C(A_1 \; [] \; A_2) \mathrel{\widehat{=}} C(A_1) + C(A_2)$	(*non-deterministic choice*)	(4.7)
$C(A_1; A_2) \mathrel{\widehat{=}} C(A_1) + C(A_2)$	(*sequential*)	(4.8)
$C(A_1 \; /\!\!/ \; A_2) \mathrel{\widehat{=}} C(A_1) + C(A_2)$	(*prioritized*)	(4.9)

---

where

(4.4) The area complexity of a non-deterministic assignment based *action* is defined by calculating the area complexity of the read set and the area complexity of the predicate evaluation as defined in (4.4)

(4.5) The area complexity rule of the*guarded action* calculates the area complexity of the guard (4.5) and the action.

(4.6) The *procedure* rule defines the area complexity of the calling action and the called procedure: The notation $A\langle Proc\rangle$ denotes an action that calls a procedure in its body. The procedure can be an action, a guarded action or composition of actions. Its area complexity is

calculated, respectively, using the rules (4.4) and (4.5) or one of the rules (4.7), (4.8), or (4.9).

(4.7) The area complexity rule for a *non-deterministic choice* sums the area complexities of those actions that are part of the composition. The composition enables non-deterministically one of the two actions A_1 or A_2. However, when considering the area complexity of the system, one needs to note that both of these actions form a separate component in hardware, and therefore, their area complexities must be included.

(4.8) The *sequential* area complexity rule gives also a sum of the area complexities of those actions that are part of the composition. A sequential composition executes the actions in the composition consecutively, and therefore, each action in the composition models a hardware module, all included into the area complexity of the sequential composition.

(4.9) The *prioritized* area complexity rule sums the area complexities of those actions that are in the composition. This is identical with the sequential and the non-deterministic choice rules.

Example 4.3. Consider an action composition of the form:

$$(((B_1; B_2) \;[]\; D) \;/\!\!/\; E$$

where the actions B_1, B_2, D, and E have area complexities $C(B_1)$, $C(B_2)$, $C(D)$, and $C(E)$, respectively. Assume that these complexities are calculated using one of the rules (4.4), (4.5) or (4.6) depending on the definition of the action. The area complexity of the composition becomes:

$$\begin{aligned}
& C\big((B_1; B_2) \;[]\; D) \;/\!\!/\; E\big) \\
& \stackrel{(4.9)}{=} C\big((B_1; B_2) \;[]\; D\big) + C(E) \\
& \stackrel{(4.4)}{=} C\big((B_1; B_2)\big) + C(D) + C(E) \\
& \stackrel{(4.8)}{=} C(B_1) + C(B_2) + C(D) + C(E)
\end{aligned}$$

End of example.

4.1.5 Tuning the area complexity model

Consider (4.4), which defines the area complexity of a non-deterministic assignment. the formula consist of two parts, the area complexity of the write set and the area complexity of the predicate Q. One may question

what kind of components does the predicate describe. In this section, the purpose is to clarify on that aspect.

In general, the predicate Q is assumed to model combinatorial logic as shown in Fig. 4.1. That is, it could model logical operations such as AND or OR or arithmetic operations such as addition or multiplication. By looking at the area complexity of the predicate (4.3) one observes that the area complexity would be identical, for instance, for addition and multiplication because the read set of the multiplication action and the read set of the addition action contain the same variables. To illustrate this, consider actions A_1, A_2, and A_3:

$$\begin{aligned} A_1 &\mathrel{\widehat{=}} r1 := r1'.(r1' = d1 + d2) \\ A_2 &\mathrel{\widehat{=}} r2 := r2'.(r2' = d1 \wedge d2) \\ A_3 &\mathrel{\widehat{=}} r3 := r3'.(r3' = d1 * d2) \end{aligned}$$

where the actions A_1, A_2, and A_3 describe addition operation, AND operation, and multiplication operation, respectively. The area complexity of the predicate Q is of the form:

$$C(Q) \overset{(4.3)}{=} \left\lceil \frac{C(vQ)}{|vQ|} \right\rceil \cdot 2^{|vQ|}$$

where $vQ = \{d1, d2\}$. Assume that in all three actions defined above, the set vQ is identical, and that the variables $d1$ and $d2$ have equal widths: $w_{d1} = w_{d2} = 4$. Then, the area complexity of the predicate becomes:

$$C(Q) = \left\lceil \frac{8}{2} \right\rceil \cdot 2^2 = 16$$

One can see that the area complexity of the AND operation is same as the area complexity of addition and multiplication. In other words the logic needed to perform bitwise AND operation is the same (following the above calculations) with the logic needed to perform for instance multiplication. This is obviously wrong because multiplication is more complex than the logic AND operation [38]. To overcome this, a *complexity factor* $\phi \in \mathbb{R}^+$ for the predicate is defined:

$$C(Q) \mathrel{\widehat{=}} \left\lceil \frac{\sum_{v \in w_v} w_v}{|vQ|} \right\rceil \cdot (2^{|vQ|})^{\phi} \tag{4.10}$$

where the value of the complexity factor can be any positive real number, and, furthermore, adjustable by a designer. In this thesis, the complexity factor is adjusted so that it takes the high abstraction level into account, and thus there is no need to assume exact complexity factors. For instance,

the complexity factor for addition is assumed to be one ($\phi = 1$) and for multiplication it is assumed to be two ($\phi = 2$). This follows from the complexity relation between binary addition and multiplication as described in [38], where the complexity of binary addition is n and the complexity of a schoolbook multiplication is n^2. At the high abstraction level it is fair to assume the above values for the area complexity evaluation.

Example 4.4. Adopting the actions A_1 and A_3 defined above, where the first one defines an addition and the last one defines a multiplication. The predicate Q of action A_1, denoted Q_1, is of the form: $Q_1 \mathrel{\widehat{=}} x'.(d1 + d2)$, and the predicate Q of action A_3 is of the form: $Q_3 \mathrel{\widehat{=}} x'.(d1 * d2)$. The variables $d1$ and $d2$ are of same type and have same widths: $w_{d1} = w_{d2} = 4$. Based on (4.10) and on the above defined complexity factor:

$$C(Q_1) \overset{(4.10)}{=} \left\lceil \frac{4+4}{2} \right\rceil \cdot (2^2)^1 = 16$$

$$C(Q_3) \overset{(4.10)}{=} \left\lceil \frac{4+4}{2} \right\rceil \cdot (2^2)^2 = 64,$$

which seems to be more realistic.
End of example.

The complexity factor can be so that more efficient multiplier is selected that allow a designer to decrease the complexity factor. Next the logic operations such as bitwise AND defined by action A_2, are discussed. Assuming that the complexity factor is one ($\phi = 1$), and that the variables are of same type, and, furthermore, they have similar widths, for instance, those widths defined in Example 4.4. Then the area complexity of the predicate Q_2 is the same as the area complexity of the predicate Q_1. To check how accurate this assumption is, BDDs are adopted. The BDD environment used, and more detailed analysis on the area complexity model are defined in Sect. 4.6. In this section, a complexity factors for logic AND and logic XOR are explored. The AND operation is defined by action A_2, and the XOR is defined by action A_4:

$$A_4 \mathrel{\widehat{=}} r4 := r4'.(r4' = d1 \oplus d2)$$

where the bitwise XOR is calculated between the variables $d1$ and $d2$. Assuming that the variables are of same type, and that their widths are the same for both actions A_2 and A_4 ($w_{d1} = w_{d2} = 4$), the area complexities for the predicates in actions are calculated using two values for complexity factor. The result are shown in Table4.1. The result show that to get better accuracy, it is worthwhile to decrease the complexity factor for action $A2$, but for action $A4$, there is no need to change the factor. This kind of analysis can be done at different phases of the design process. Table (4.2)

Table 4.1: Tuning the complexity factor.

Action	C(Q) ($\phi = 1$)	C(Q)($\phi = 0.5$)	Area (BDD)
A2	16	8	10
A4	16	8	21

Table 4.2: Example operations and their complexity factor.

Operation	definition	Complexity factor
Integer Addition	$A + B$	$\phi = 1$
Integer Subtraction	$A - B$	$\phi = 1$
Integer Multiplication	$A * B$	$\phi = 2$
Integer Division	$A \div B$	$\phi = 2$
AND (bitwise)	$A \wedge B$	$\phi = \frac{1}{2}$
OR (bitwise)	$A \vee B$	$\phi = \frac{1}{2}$
XOR (bitwise)	$A \oplus B$	$\phi = 1$

summarizes the complexity factors discussed in this section. Furthermore, these factors are adopted in the forthcoming sections. Moreover, the accuracy of the area complexity model is further alleviated in the end of this Chapter in Sect.4.6

4.2 Average Power of Timed Actions

The power modeling starts by defining an energy model for a timed action after which the static and the dynamic power consumption are explored. The area complexity model presented in Sect.4.1, is instrumental in both of these models. In this section assume that the timed actions are atomic. Non-atomicity in power modeling is presented Sect. 4.4.

4.2.1 Energy

Every time a timed action is executed it dissipates energy in its operation. For an arbitrary timed action A, the energy dissipation is modeled by:

$$E(A) \mathrel{\widehat{=}} \alpha \cdot C(A) \cdot E^1 \qquad \textit{(energy dissipation of timed action } A\textit{)} \quad (4.11)$$

where $C(A)$ is the area complexity of the timed action A, α is the switching probability parameter, and E^1 is the energy of an unit action A^1. The unit action defines a basic logic gate, an inverter:

$$A^1 \mathrel{\widehat{=}} x := x'.(x' = \neg y) \qquad (\textit{an unit action } A^1) \quad (4.12)$$

---

where x and y are Boolean variables. The energy of the unit action (inverter) in CMOS can be calculated by:

$$E^1 = \frac{1}{2} \cdot C_L \cdot V_{DD}^2$$

where V_{DD} is a supply voltage and C_L is the output capacitance of the unit size inverter driving another unit size inverter in a given CMOS technology [69]. Therefore, an arbitrary action A has a load capacitance: $C(A) \cdot C_L$. Observe that the supply voltage V_{DD} and the load capacitance of the unit size inverter C_L are technology dependent parameters, and thus, their values are not defined.

The switching probability parameter α is defined for every action in the composition, and its value is specified by a designer. In general, the switching probability parameter defines the probability that subactions in the atomic composition are enabled. Therefore, it is required that $0 < \alpha \leq 1$, and, furthermore, in this study, the value of the parameter is one ($\alpha = 1$) unless otherwise stated. Consider the following example:

Example 4.5. An atomic action A defined by:

$$A \mathrel{\widehat{=}} (A_1; (A_2 \ [\!] \ A_3); A_4)$$

whose one is able to monitor pre and post conditions of A. Due to the atomicity, one is not able to observe the intermediate states of the composition, for example, which one of the subactions A2 or A3 is executed. In terms of area complexity, the estimation is carried out for the whole composition as described by the area calculations rules (4.4) and (4.8), and thus the area complexity of the composition A becomes:

$$C(A) \overset{(4.4)(4.8)}{=} C(A_1) + C(A_2) + C(A_3) + C(A_4)$$

where the $C(A)$ is the area complexity of the action A. Adopting this area complexity and the energy dissipation of timed action (4.11) would give the worst-case energy consumption of the timed action A:

$$E(A) \overset{(4.11)}{=} \alpha \cdot C(A) \cdot E^1$$

where the switching probability parameter is defined to be one for all actions in the composition $\alpha = 1$. Often it is useful to evaluate the average case or even the best case energy consumption. To do this, a designer can adjust the switching probability parameter, for instance, by assuming that the actions A_2 and A_3 have equal likelihood to be selected for execution, and therefore the energy consumption $E(A)$ is:

$$E(A) = (\alpha_1 \cdot C(A_1) + \alpha_2 \cdot C(A_2) + \alpha_3 \cdot C(A_3) + \alpha_4 \cdot C(A_4)) \cdot E^1$$

where the switching probability parameters $\alpha_1 = \alpha_4 = 1$ and $\alpha_2 = \alpha_3 = 0.5$. These parameters indicate that subactions A_1 and A_4 are selected for execution every time the action A is enabled, whereas the actions A_2 and A_3 are selected with 50 % likelihood. Adopting the latter area complexity clause to evaluate energy consumption one is able to estimate the average case or even best the best case energy consumption of the timed action A
End of example.

4.2.2 Dynamic Power Consumption

Consider an arbitrary timed action A. Its dynamic power is defined by:

$$P_{dyn}(A) \mathrel{\hat{=}} \frac{E(A)}{\Delta(A)} \quad (\textit{dynamic power consumption of timed action } A) \tag{4.13}$$

where $E(A)$ is the energy consumption of the action, and $\Delta(A)$ is the duration of a single execution of A ($\Delta A \mathrel{\hat{=}} A.ft - A.st$). The energy consumption is illustrated in Fig. 4.6, where the dotted curved line demonstrates the actual energy consumption of the action A, whereas the straight line is the energy estimate used in this study. That is, the energy consumption of timed action A is related to the duration of execution reaching its final value when the timed action finishes its execution. Moreover, from the linear energy model it follows that at time t ($t \in [A.st, A.ft]$), shown in Fig. 4.6, the energy of A is

$$E(A) \mathrel{\hat{=}} E^{*t}(A) + E^{t*}(A)$$

where

$$E^{*t}(A) \mathrel{\hat{=}} (t - A.st) \cdot P_{dyn}(A) \tag{4.14}$$
$$E^{t*}(A) \mathrel{\hat{=}} (A.ft - t) \cdot P_{dyn}(A) \tag{4.15}$$

where $E^{*t}(A)$ describes the energy consumption of the timed action A from initialization to time t, and $E^{t*}(A)$ is the energy dissipation from the time

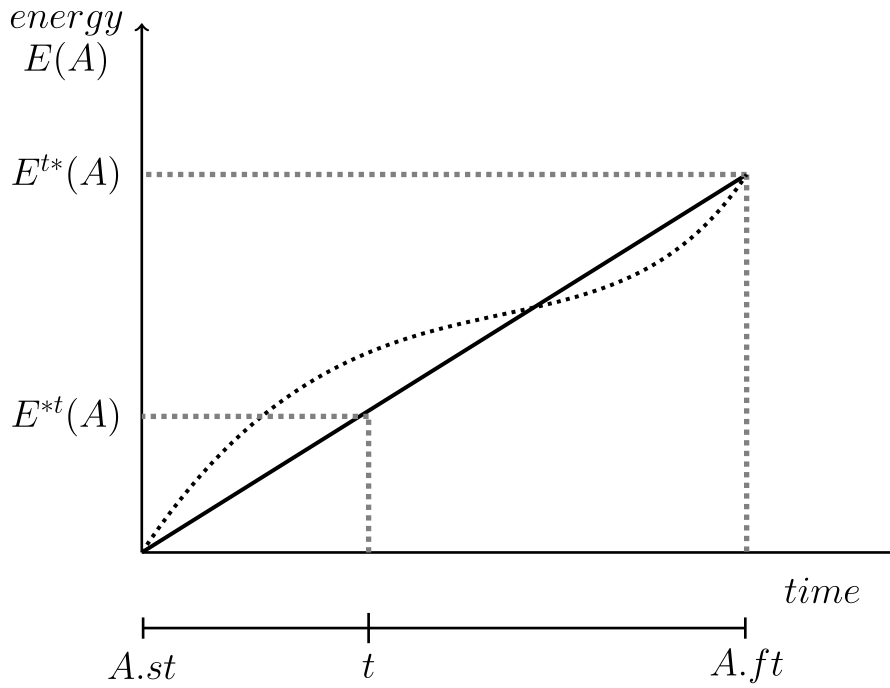


Figure 4.6: Dynamic power of a timed action

t to the finish time of the action A. Observe that in the above calculations the dynamic power P_{dyn} is constant due to the selected linear energy model.

In cases when the power is not constant, that is, it is a function of time, the energy consumption of the timed action A is:

$$E(A) \mathrel{\widehat{=}} \int_{A.st}^{A.ft} P_{dyn}(t)dt$$

where $A.st$ and $A.ft$ are the start and the finish times of the timed action A, respectively.

4.2.3 Static power consumption

To model the static power consumption of a timed action A, the unit action A^1,defined by (4.12), is adopted. The static power dissipation of the unit action is:

$$P^1_{stat} \mathrel{\widehat{=}} V_{DD} \cdot I^1_{leak} \qquad (\textit{static power of an unit action } A^1) \quad (4.16)$$

---

where I^1_{leak} is the leakage current of the unit action, and V_{DD} is the supply voltage. Observe that both the supply voltage V_{DD} and the leakage current I_{leak} are technology dependent parameters, and therefore, their values are not defined in this abstraction level. Adopting the static power consumption P^1_{stat} of an unit action A^1 the timed action A causes a static power loss $P_{stat}(A)$:

$$P_{stat}(A) \mathrel{\widehat{=}} C(A) \cdot P^1_{stat} \qquad \textit{(static power consumption of timed action A)} \tag{4.17}$$

where $C(A)$ is the area complexity of the timed action A. That is, the static power loss of the timed action A is directly proportional to its area. Kim et. al. in [49] stated that both the sub threshold and the oxide leakage depends on the total gate width or more approximately the *gate count*, temperature, and oxide thickness. The last two are not considered in this model but the first one, the gate count, can be modeled using the area complexity model for timed actions.

4.2.4 Power consumption of timed action

The total power consumption of a timed action A is obtained by adding the dynamic power consumption (4.13) and the static power consumption (4.17)of the timed action A together, and thus the power consumption of the timed action A is defined by:

$$P_{tot}(A) \mathrel{\widehat{=}} P_{dyn}(A) + P_{stat}(A) \qquad \textit{(power consumption of timed action A)} \tag{4.18}$$

4.3 Power Modeling at System Level

In this Section, the area complexity and the power consumption are discussed at the system level. The area complexity and power models for timed actions are applied in a timed action system context. Furthermore, the area model can be adopted for conventional Action Systems as well because no timing information is required. The modeling environment, shown in Fig. 1.1 on page 4, is defined to be a parallel composition $\mathcal{A} \| \mathcal{E}nv$, that is another action system whose distinct global and local identifiers (procedures, variables, actions) consist of the identifiers of the component systems $\mathcal{A} \| \mathcal{E}nv$. The **execution** clause is of the form:

$$\textbf{forever do} \; [\; [\!] \; 1 \leq i \leq k : A_i] \;\; [\!] \;\; [\; [\!] \; 1 \leq j \leq l : E_j] \; \textbf{od}$$

where A_i and E_j are actions (with distinct labels) operating on the state variables of $\mathcal{A}$ and $\mathcal{E}nv$, respectively. Constituent systems communicate via their shared interface variables and public procedures. Furthermore, the actions operate in a *non-final* manner meaning that there is always at least one enabled action in the established system composition.

4.3.1 Area complexity of Action Systems

Consider the above introduced system $\mathcal{A} \parallel \mathcal{E}nv$ where the latter models the environment of the former. Furthermore, let $\mathcal{A}$ denote all the actions of the system $\mathcal{A}$, and $\mathcal{E}nv$ denote all the actions in the system $\mathcal{E}nv$. It is easy to see that $\mathcal{A} \parallel \mathcal{E}nv = \mathcal{A} \cup \mathcal{E}nv$, where $\mathcal{A}\|\mathcal{E}nv$ consist of all the actions in the parallel composition.

The area complexity of the system $\mathcal{A}$ is defined by

$$C(\mathcal{A}) \mathrel{\widehat{=}} \sum_{A \in \mathcal{A}} C(A)$$

where the area complexity is calculated for all the actions A in the system $\mathcal{A}$. From this follows that the area complexity of the environment $\mathcal{E}nv$ is of the form:

$$C(\mathcal{E}nv) \mathrel{\widehat{=}} \sum_{E \in \mathcal{E}nv} C(E)$$

By adopting the above area complexities for the systems $\mathcal{A}$ and $\mathcal{E}nv$ one is able to define the area complexity of the systems $\mathcal{A} \parallel \mathcal{E}nv$:

$$C(\mathcal{A} \parallel \mathcal{E}nv) \mathrel{\widehat{=}} C(\mathcal{A}) + C(\mathcal{E}nv) \qquad (\textit{area complexity of systems}) \quad (4.19)$$

where the area complexity of the parallel composition $\mathcal{A} \parallel \mathcal{E}nv$ is the sum of the area complexity $C(\mathcal{A})$ of the system $\mathcal{A}$ and the area complexity $C(\mathcal{E}nv)$ of the system $\mathcal{E}nv$.

4.3.2 Average power of Timed Action Systems

To model the average power of a timed action system $\mathcal{A}$, one needs to define an observation period: $T \mathrel{\widehat{=}} [T.st, T.ft]$ ($T \in \mathbb{T}$), where $T.st$ and $T.ft$ denote the start and finish times of the interval, respectively. The duration of the period is obtained as follows: $\Delta(T) \mathrel{\widehat{=}} T.ft - T.st$. During the observation period, a timed action system executes a set $\mathcal{A}_T$ of timed actions.

$$\mathcal{A}_T \mathrel{\widehat{=}} \big\{A | (A \in \mathcal{A}) \wedge \big(\exists t : T.st < t < T.ft : gd(A)(t)\big)\big\}) \quad (\textit{set of } \mathcal{A}_T) \quad (4.20)$$

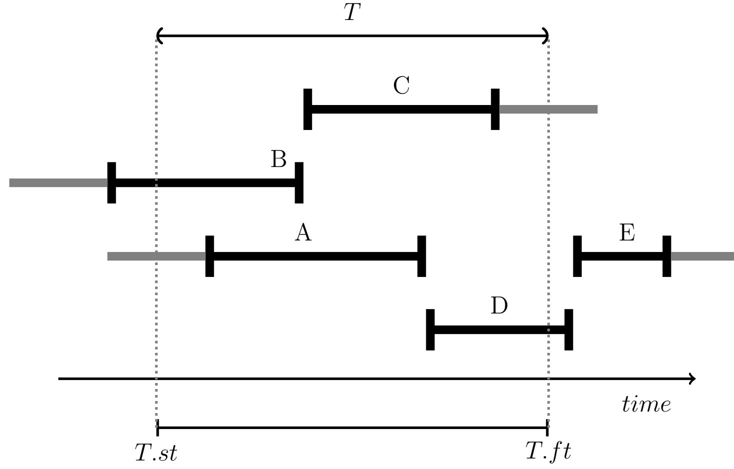


Figure 4.7: Example set of timed actions $\mathcal{A}_T$

where $gd(A)(t)$ refers to the guard of a timed action A at the given time t. Hence, the set $\mathcal{A}_T$ contains all the actions from the execution loop of the system $\mathcal{A}$ that are enabled during the observation period T. This includes actions that are started, finished, or started and finished within the observation period T. Consider an example system $\mathcal{A} \parallel \mathcal{E}nv$. In Fig. 4.7, actions that are executed by the system $\mathcal{A}$ are illustrated as black lines whereas the gray lines describes those actions that are executed by the environment system $\mathcal{E}nv$. The environment system $\mathcal{E}nv$ is excluded from the average power evaluation to capture the average power dissipation of the system $\mathcal{A}$. The observation period T, illustrated in Fig. 4.7, determines the set of actions $\mathcal{A}_T = \{A, B, C, D\}$, where according to Figure, the action E is excluded due to the definition of the set $\mathcal{A}_T$. In general, however, the observation period is defined in a way that it contains, for example, a single execution of a computation path with in a system. That is, the observation period is determined in a way that it contains the desired functionality for power estimation, for instance, a single computation cycle.

The energy dissipation of a timed action system $\mathcal{A}$ must be defined before the average power can be evaluated. The energy dissipation of the system $\mathcal{A}$ over the observation period T, denoted by $E_T(\mathcal{A}_T)$, is defined by:

$$E_T(\mathcal{A}) \mathrel{\widehat{=}} \sum_{A \in \mathcal{A}_T} E_T(A) \qquad (\textit{energy of timed action system } \mathcal{A}) \quad (4.21)$$

where $E_T(A)$ denotes the energy consumption of action A during the observation period T. The way action A is executed during the observation period T has an effect on the energy calculations. That is, there are five

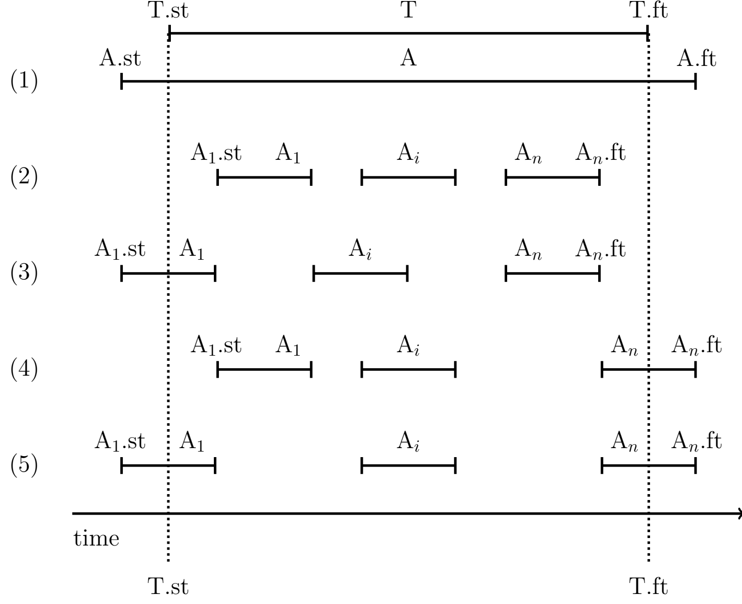


Figure 4.8: Execution of timed action A during observation period T

different cases, shown in Fig. 4.8, which are listed and explained below (the numbers in the figure correspond with the numbers in the lists below):

(1) $E_T(A) = E(A) - E^{*T.st}(A) - E^{T.ft*}(A),$
$(A.st \leq T.st \wedge A.ft \geq T.ft)$

(2) $E_T(A) = n \cdot E(A),$
$(A_1.st \geq T.st \wedge A_n.ft \leq T.ft \wedge n \geq 1)$

(3) $E_T(A) = (n-1) \cdot E(A) - E^{*T.st}(A_1),$
$(A_1.st \leq T.st \wedge A_n.ft \leq T.ft \wedge n \geq 1)$

(4) $E_T(A) = (n-1) \cdot E(A) - E^{T.ft*}(A_n),$
$(A_1.st \geq T.st \wedge A_n.ft \geq T.ft \wedge n \geq 1)$

(5) $E_T(A) = (n-2) \cdot E(A) - E^{*T.st}(A_1) - E^{T.ft*}(A_n),$
$(A_1.st \geq T.st \wedge A_n.ft \geq T.ft \wedge n \geq 2)$

where

(1) The start and finish times of the timed action A are outside the observation period T.

(2) The timed action is executed n ($n \geq 1$) times inside the observation period T.

(3) The first execution of timed action, denoted A_1, starts its operation outside the observation period T, but finishes it inside the observation period T, after which the action is executed $(n-1)$ times $(n \geq 1)$.

(4) The timed action A is executed $(n-1)$ times $(n \geq 1)$ inside the observation period T. The last execution, denoted by A_n falls partly outside the observation period T.

(5) The first (A_1), and the last (A_n) execution of the timed action A are partly outside the observation period T. In between these execution the timed action is executed $(n-2)$ times $(n \geq 2)$.

The above expressions for energy consumption requires that the $E(A)$ is constant. For atomic compositions, like the one presented in Sect. 4.2.1, the constant energy value is guaranteed by using the switching probability parameter α. This property is illustrated by the following example:

Example 4.6. Consider a timed action $A \mathrel{\widehat{=}} Mult \; [] \; Add$, where the action $Mult$ describes a multiplication operation, and the action Add describes an addition operation. The area complexity of the action A is defined by adding the area complexities of the subactions together: $C(A) = C(Mult) + C(Add)$, where the $C(Mult)$ and the $C(Add)$ denote the area complexities of the actions $Mult$ and Add, respectively. However, to illustrate its dynamic behavior, the area complexity $C(A)$ of the action A gives the worst-case result. Assuming that the switching probabilities for these subactions are known and defined by $\alpha_{Mult} = 0.6$ and $\alpha_{Add} = 0.4$. Adopting these switching probabilities for energy consumption evaluation results a constant energy for the action A, and the presented energy calculation rules can be adopted. The energy consumption is of the form:

$$E(A) \overset{(4.11)}{=} (0.6 \cdot C(Mult) + 0.4 \cdot C(Add)) \cdot E^1$$

Naturally the action A can be implemented using two separate actions $Mult$ and Add. which will lead directly to constant energies $E(Mult)$ and $E(Add)$. Of course the drawback is that one requires two action definitions instead of one, which may complicate the definition in large systems.
End of example.

Adopting (4.21), the dynamic power consumption $P_{T,dyn}(\mathcal{A})$ during observation period T is defined by:

$$P_{T,dyn}(\mathcal{A}) \mathrel{\widehat{=}} \frac{E_T(\mathcal{A})}{\Delta(T)} \qquad (\textit{dynamic power of system } \mathcal{A}) \quad (4.22)$$

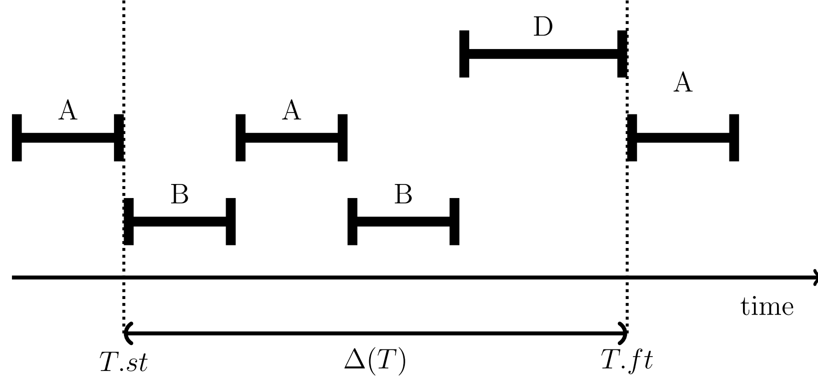


Figure 4.9: Illustration of the selected observation period

where $\Delta(T)$ is the duration of the observation period. The static power consumption is related to the area complexity of the system. Adopting (4.19) the static power consumption of the system $\mathcal{A}$ becomes:

$$P_{stat}(\mathcal{A}) \mathrel{\widehat{=}} C(\mathcal{A}) \cdot P^1_{stat} \qquad (\textit{static power of system } \mathcal{A}) \quad (4.23)$$

where the area complexity of the above system is calculated in a way that all actions in the system are included. That is the leakage is dependent of the area complexity of the whole system whereas the dynamic power includes only those actions that are enabled during the selected observation period.

Adopting (4.22) and (4.23), the average power consumption $P_{T,avg}(\mathcal{A})$ during the observation period T of a timed action system $\mathcal{A}$ is defined by:

$$P_{T,avg}(\mathcal{A}) \mathrel{\widehat{=}} P_{T,dyn}(\mathcal{A}) + P_{stat}(\mathcal{A}) \qquad (\textit{average power of system } \mathcal{A}) \quad (4.24)$$

Naturally, the average power consumption of a parallel composition of timed action systems is obtained by adding together the average power of the individual systems: for instance, the average power consumption of the system $\mathcal{A} \parallel \mathcal{E}nv$ is:

$$P_{T,avg}(\mathcal{A} \parallel \mathcal{E}nv) \mathrel{\widehat{=}} P_{T,avg}(\mathcal{A}) + P_{T,avg}(\mathcal{E}nv)$$

where the $P_{T,avg}(\mathcal{A})$ is the average power consumption of the system $\mathcal{A}$ and $P_{T,avg}(\mathcal{E}nv)$ is the average power consumption of the system $\mathcal{E}nv$ both of which are calculated using (4.24).

Example 4.7. Consider a timed action system $\mathcal{A}$ whose execution loop is of the form:

$$\textbf{forever do } (A\,;B)\ \ []\ \ D \textbf{ od}$$

The average power consumption of the system is calculated by evaluating the dynamic power consumption and the static power consumption of the system separately defined in (4.24). To evaluate dynamic power consumption, one have to set an observation period $T = [T.st, T.ft]$, where the start time $T.st$ and the finish time $T.ft$ are defined in a way that the system executes all the timed actions in the execution loop at least once. The set of actions $\mathcal{A}_T$ can be, for instance, of the form: $\mathcal{A}_T \mathrel{\widehat{=}} \{A, B, D\}$, as shown in Fig. 4.9. The dynamic power consumption:

$$\begin{aligned} P_{T,dyn}(\mathcal{A}) &\overset{(4.22)}{=} \frac{E(\mathcal{A}_T)}{\Delta(T)} \\ &\overset{(4.21)}{=} \frac{E(A) + 2 \cdot E(B) + E(D)}{\Delta(T)} \end{aligned}$$

and the static power consumption for the system $\mathcal{A}$ is:

$$\begin{aligned} P_{stat}(\mathcal{A}) &\overset{(4.23)}{=} C(\mathcal{A}) \cdot P^1_{stat} \\ &\overset{(4.19)}{=} (C(A) + C(B) + C(D)) \cdot P^1_{stat} \end{aligned}$$

Thus, the average power dissipation of the system $\mathcal{A}$ is:

$$P_{T,avg}(\mathcal{A}) \overset{(4.24)}{=} P_{T,dyn}(\mathcal{A}) + P_{stat}(\mathcal{A})$$

End of Example.

4.3.3 Power dissipation in computation paths

During the development of a digital system, a designer make choices between different implementation methods and styles. For instance, it can be useful to introduce additional pipeline stages or to operate in a fully parallel manner within given requirements. Based on a *computation path* one is able to gain information on different computational "suboperations" inside a system. Therefore, to obtain more detailed information on timing and power consumption in the system, one reconsider the computation path, defined in (3.8).

Consider a computation path $\mathcal{CP}(A, B)$, which is defined by (3.6) on page 44 (denoted below for the ease of reference):

$$\mathcal{CP}(A, B) \mathrel{\widehat{=}} \langle A_1, A_2, \dots A_n \rangle$$

where $A = A_1$ and $A_n = B$. The area complexity of the computation path $\mathcal{CP}(A, B)$ is defined by:

$$C(\mathcal{CP}(A,B)) \;\widehat{=}\; \sum_{i=1}^{n} C(A_i) \qquad (\textit{area complexity of a computation path}) \tag{4.25}$$

---

where the area complexity includes all the actions in the computation path, that is, A and sequence of successors leading from A to B, including A and B themselves.

As discussed above, the computation paths are used to explore different computational operations within a system. To compare power consumption between computation paths, it is enough to compare the dynamic power consumption of different paths. This is due to the fact that the static power consumption of the system does not alter because one is not increasing or decreasing the amount of actions in the system, only analyzing different subsets of actions in the system. The power consumption of the computation path is:

$$P(\mathcal{CP}(A,B)) \;\widehat{=}\; \frac{E(\mathcal{CP}(A,B))}{\Delta\mathcal{CP}(A,B)} \qquad (\textit{power consumption of a computation path}) \tag{4.26}$$

---

where $E(\mathcal{CP}(A, B))$ is the energy consumption of the computation path and $\Delta\mathcal{CP}(A, B)$ is the computation path delay, defined by (3.17) on page 49.

The symbol * is used to indicate if A or B or both A and B are excluded from the computation path. Compare the notations with the computation path delay on page 49. The energy consumption of the computation path is of form (both A and B included):

$$E(\mathcal{CP}(A,B)) \;\widehat{=}\; \sum_{i=1}^{n} E(A_i) = E(A_1) + E(A_2) + \ldots E(A_n)$$

where the energy consumption is first calculated for each action A_i in the computation path separately and then added together.

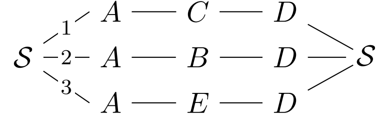
$$E(\mathcal{CP}({}^*A, {}^*B)) \;\widehat{=}\; E(\mathcal{CP}(A,B)) - E(B)$$
$$E(\mathcal{CP}(A^*, B^*)) \;\widehat{=}\; E(\mathcal{CP}(A,B)) - E(A)$$
$$E(\mathcal{CP}(A^*, {}^*B)) \;\widehat{=}\; E(\mathcal{CP}(A,B)) - E(A) - E(B)$$

adopting the energy clauses, and the computation path delays, presented on page 49, and adopting (4.26), one is able to calculate the power consumption for all the four cases presented above.

Example 4.8. The execution loop of a timed action system $\mathcal{S}$ is of the form:

$$\textbf{forever do } A\textbf{;}\big((B \;[\!]\; C) \parallel E\big)\textbf{;}D \textbf{ od}.$$

Three possible computation paths are illustrated using a tree structure shown below:



where the computation path 1 and 2 are alternative paths from action A to action D whereas the computation path 3 is partially parallel (action E) with the paths 1 and 2. For power analysis purposes it is necessary to compare the execution time and the area of each path. The computation path delays are:

$$\begin{aligned}\Delta\mathcal{CP}_1(A, D) &\overset{(3.17)}{=} \Delta(A; C; D)\\ &\overset{(3.11)}{=} \Delta(A) + \Delta(C) + \Delta(D)\end{aligned}$$

$$\begin{aligned}\Delta\mathcal{CP}_2(A, D) &\overset{(3.17)}{=} \Delta(A; B; D)\\ &\overset{(3.11)}{=} \Delta(A) + \Delta(B) + \Delta(D)\end{aligned}$$

$$\begin{aligned}\Delta\mathcal{CP}_3(A, D) &\overset{(3.17)}{=} \Delta(A; E; D)\\ &\overset{(3.11)}{=} \Delta(A) + \Delta(E) + \Delta(D)\end{aligned}$$

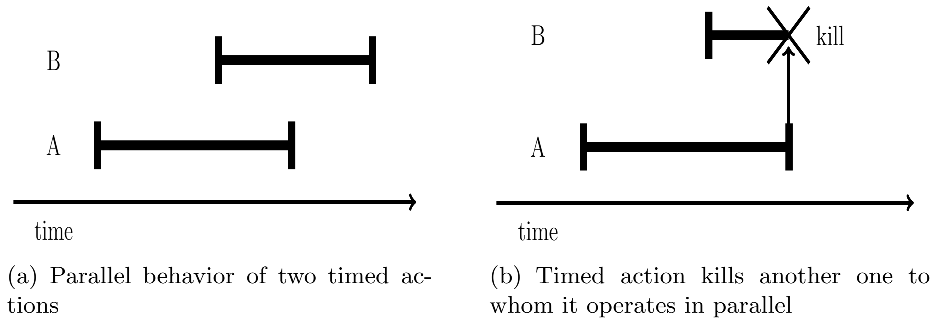
where one can see that the difference in delays between the three computation paths depends on the delays of the actions C, B, and E. This is due to the fact that three paths have actions A and D whose delays are not altered between computation paths. Thus, if $\Delta(E)$ is the smallest delay then the computation path $\mathcal{CP}_3$ is the fastest.

In a similar manner, the area complexity is calculated for all three computation paths:

$$C(\mathcal{CP}_1(A, D)) \overset{(4.25)}{=} C(A) + C(C) + C(D)$$

$$C(\mathcal{CP}_2(A, D)) \overset{(4.25)}{=} C(A) + C(B) + C(D)$$

$$C(\mathcal{CP}_3(A, D)) \overset{(4.25)}{=} C(A) + C(E) + C(D)$$



(a) Parallel behavior of two timed actions

(b) Timed action kills another one to whom it operates in parallel

Figure 4.10: Parallel behavior of timed actions.

where one can see that the area complexity of the actions C, B, and E determines which one of the three computation paths is smallest.

The power dissipation for each computation path is evaluated by adopting the delay and the area complexity information and (4.26):

$$
\begin{aligned}
P(\mathcal{CP}_1(A,D)) &= \frac{E(\mathcal{CP}_1(A,D))}{\Delta\mathcal{CP}_1(A,D)} = \frac{E(A)+E(C)+E(D)}{\Delta\mathcal{CP}_2(A,D)} \\
P(\mathcal{CP}_2(A,D)) &= \frac{E(\mathcal{CP}_2(A,D))}{\Delta\mathcal{CP}_2(A,D)} = \frac{E(A)+E(C)+E(D)}{\Delta\mathcal{CP}_2(A,D)} \\
P(\mathcal{CP}_3(A,D)) &= \frac{E(\mathcal{CP}_3(A,D))}{\Delta\mathcal{CP}_3(A,D)} = \frac{E(A)+E(C)+E(D)}{\Delta\mathcal{CP}_3(A,D)}
\end{aligned}
$$

By looking the above power equations, the difference in power dissipation depends on the timed actions C,B,and E. For instance, if ($C(B)$,$C(C)$) $\geq C(E)$ and $(\Delta(B), \Delta(C)) \leq \Delta(E)$ then the power consumption of the computation path 3 is smaller than the power dissipation of the computation paths 1 and 2.
End of example.

4.4 Power Consumption of Killed Timed Actions

The parallel behavior of timed actions, defined in Sect.3.4.2 on page 41, requires that the actions are enabled at the same state, and, furthermore, they are *independent* of each other (no write-write and read-write conflicts between the actions executed in parallel). However, in conventional Actions Systems, *non-independent* actions cannot be executed in parallel. In the Timed Action Systems formalism, the non-independent actions can operate partially parallel if timed actions are killed. Consider two timed actions, say A and B, are operating in parallel such that A has started its execution

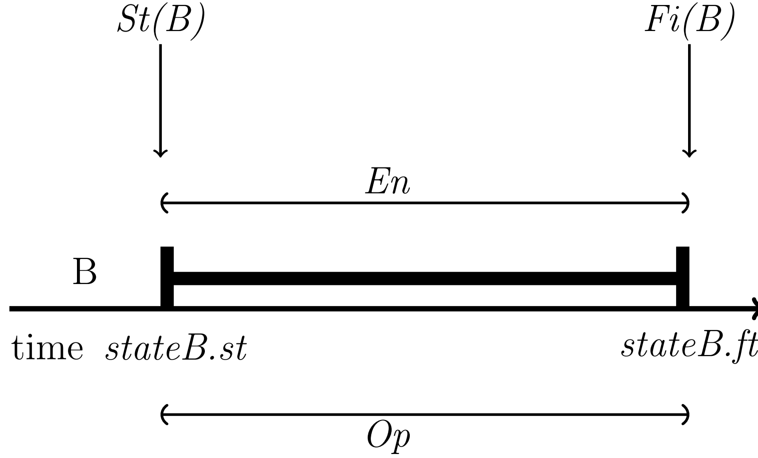


Figure 4.11: Timed action state predicates

before B and A ends its operation before B, shown in Fig 4.10(a). This reflects the operation of two independent parallel actions A and B. Next, consider a case when A and B are not independent. Timed action A, at the end of its operation updates its write variables some of which belongs to the read set of B ($wA \cap rB \neq \emptyset$). Assume that the update of these variables disables the timed action B. This, in turn, kills the operation of the timed action B as shown in Fig. 4.10(b). The killing of a timed action results in the execution of the skip action creating the same outcome in the timed and untimed domains. In other words, the execution of the conventional action A would disable the conventional action B, and therefore B would not be executed in the first place. Adopting the skip operation is not appropriate for power modeling purposes, because at the same moment when a timed action commences its operation it consumes energy. To further illustrate the difference, consider Fig. 4.11, where the operation of the action B is shown using timed action state predicates (discussed in Sect. 3.1.3 on page 35). In the beginning of the execution, the read variables of a timed action are read, then the timed action continues its operation, after which the results are written onto the write variables by the finish action. If the timed action B is killed the operation is interrupted and the write variables are not updated as shown in Fig. 4.12. In power estimation point of view, power is dissipated in the beginning of the execution ($St(B)$) and during the time that a timed action has been in operation ($Op(B)$). In other words the timed action performs computation but the results of the computation are not stored into the write variables.

The area complexity of timed actions operating in parallel is calculated using (4.4) (for each action separately). Hence, whether a timed action is

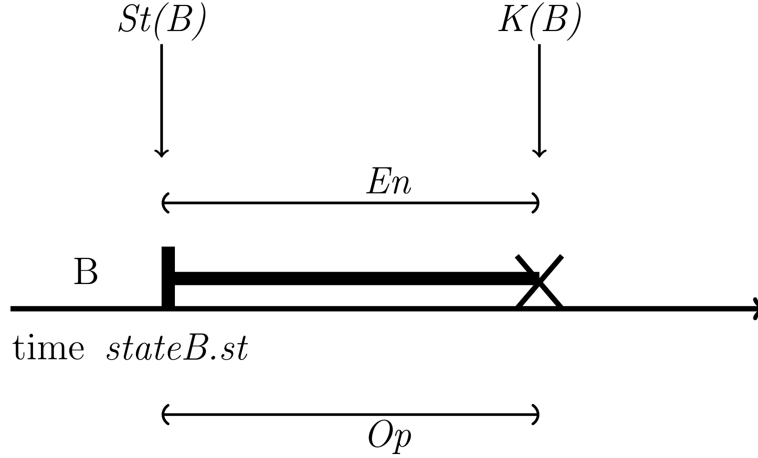


Figure 4.12: Timed action state predicates kill operation

killed or not does not affect on its area complexity. Furthermore, the area complexity is the only variable in the presented static power consumption model, and therefore, the static power consumption is evaluated as discussed in Sect. 4.3.2. However, the dynamic power consumption is dependent of a computation time, and therefore if a timed action is killed, the dynamic power has to be re-calculated. The dynamic power evaluation is started by calculating the energy consumption of an independent timed action B:

$$E^K(B) \overset{(4.14)}{=} E^{*t}(B) = (t - B.st) \cdot P_{dyn}(B) \qquad (\mathit{Energy(killed\ action)}) \tag{4.27}$$

where $E^{*t}(B)$ is the partial energy discussed on page 65 and P_{dyn} is the dynamic power consumption of the action B whose operation is not killed. The dynamic power consumption of the killed timed action B is defined by:

$$P^K_{dyn}(B) \widehat{=} \frac{E^K(B)}{\Delta^K(B)} \qquad (\mathit{dynamic\ power(killed\ action)})$$

where $\Delta^K(B)$ is the operation time of the timed action B, and it is defined by:

$$\Delta^K(B) \widehat{=} B.kt - B.st \qquad (\mathit{delay(killed\ action)})$$

The average power consumption of the killed action B is defined by:

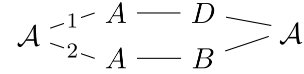
$$P^K(B) \mathrel{\widehat{=}} P^K_{dyn}(B) + P_{stat}(B) \qquad (\textit{average power(killed action)}) \quad (4.28)$$

where the first term evaluates the dynamic power consumption and the second term calculates the static power consumption using (4.17). This is due to the fact that the static power estimate depends on the area of the action, which is not altered in the killing process, and therefore it can be calculated using the static power equation for timed actions. Consider the following two examples where the first one describes a timed action system, whose behavior is evaluated using computation paths, and, furthermore, the timed actions in the system are assumed to be independent. The second one describe a timed action system whose operation is modeled using non-independent actions, and therefore the power analysis must take the possible kill action into account.

Example 4.9. Consider a timed action system $\mathcal{A}$ whose execution sequence is of the form:

$$\textbf{forever do } A\textbf{;}(B \;[\!]\; D) \textbf{ od}$$

where the execution of the timed action A is followed by the non-deterministic choice between actions A and B. The system has two possible execution sequences, which are illustrated using computation paths shown below:



where the action A is executed first after which either the action B or the action D is selected (non-deterministically) for execution. Assuming that kill operation does not take place during the execution of the computation paths. The computation path delays are of the form:

$$\begin{aligned}\Delta\mathcal{CP}_1(A, B) &\overset{(3.17)}{=} \Delta(A; B) \\ &\overset{(3.11)}{=} \Delta(A) + \Delta(B)\end{aligned}$$

$$\begin{aligned}\Delta\mathcal{CP}_2(A, D) &\overset{(3.17)}{=} \Delta(A; D) \\ &\overset{(3.11)}{=} \Delta(A) + \Delta(D)\end{aligned}$$

In a similar manner the area complexities are calculated for the computation paths:

$$C(\mathcal{CP}_1(A,B)) \overset{(4.25)}{=} C(A) + C(B)$$

$$C(\mathcal{CP}_2(A,D)) \overset{(4.25)}{=} C(A) + C(D)$$

Adopting the above timing and area information the power equations for the computation paths $\mathcal{CP}_1$ and $\mathcal{CP}_2$ are:

$$\begin{aligned} P(\mathcal{CP}_1(A,B)) &\overset{(4.26)}{=} \frac{E(\mathcal{CP}_1(A,B))}{\Delta\mathcal{CP}_1(A,B)} \\ &= \frac{(C(A)+C(B)) \cdot E^1}{\Delta(A)+\Delta(B)} \end{aligned}$$

$$\begin{aligned} P(\mathcal{CP}_2(A,D)) &\overset{(4.26)}{=} \frac{E(\mathcal{CP}_2(A,D))}{\Delta\mathcal{CP}_2(A,D)} \\ &= \frac{(C(A)+C(D)) \cdot E^1}{\Delta(A)+\Delta(D)} \end{aligned}$$

where one can see that both computation paths include the action A, and therefore its time and area dissipation are the same in both paths. The difference in power consumption depends on the size and speed difference between the actions B and D.
End of example.

Example 4.10. Consider the timed actions system $\mathcal{A}$ described in Example 4.9, whose execution sequence is of the form:

$$\textbf{forever do } A;(B \; [\!] \; D) \textbf{ od}$$

where the execution of the timed action A is followed by the non-deterministic choice between actions B and D. In the Timed Action System context, it is possible, however, that the actions B and D are simultaneously enabled. This is illustrated in Fig. 4.13, where one can see that both the action B and D are enabled at the same time. The sequential composition with the action A, however, force the faster action to kill the slower one. The faster action is the action whose operation is finished earlier. In this case, the action B kills the action D. Therefore, the operation of the system, in this particular case, can be defined using the computation path $\mathcal{CP}_1(A,D)$, described in the previous example. To model power consumption for the above described behavior, the presence of the kill operation has to be taken care of. Only dynamic power is discussed since, one analyze a subset of the model, and therefore the total area of the system remains the same.

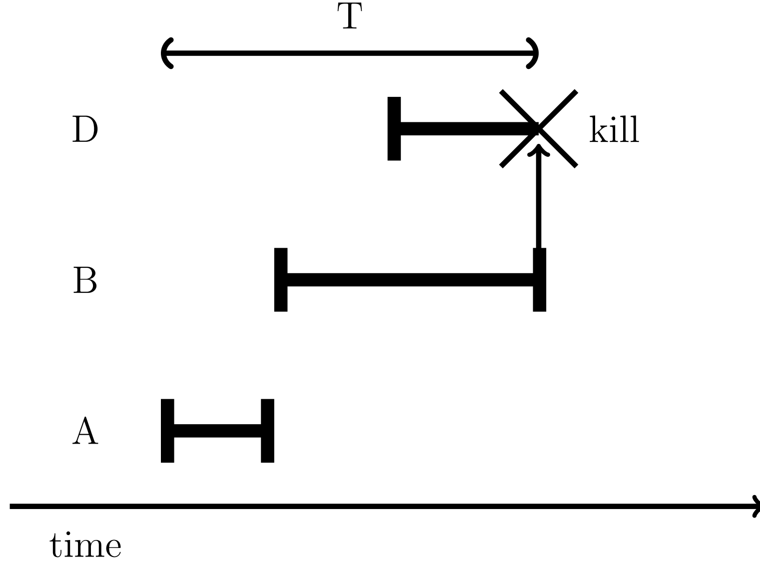


Figure 4.13: Example composition with kill

To evaluate the dynamic power consumption for the behavior, shown in Fig 4.13, an observation period T is defined by: $T \mathrel{\widehat{=}} [T.st, T.ft]$, where the start time $T.st$ and the finish time $T.ft$ are defined in a way that both the actions A and B are executed sequentially once. The dynamic power consumption becomes:

$$\begin{aligned} P_{T,dyn}(\mathcal{A}) &\overset{(4.22)}{=} \frac{E(\mathcal{A}_T)}{\Delta(T)} \\ &\overset{(4.21)}{=} \frac{E(A) + E(B) + E(D)}{\Delta(T)} \\ &\overset{(4.27)}{=} \frac{E(A) + E(B) + ((t - D.st) \cdot E(D))}{\Delta(T)} \end{aligned}$$

where the energy consumption is calculated for the killed action D as well. The delay of the observation period corresponds with the delay of the computation path $\mathcal{CP}_1(A, B)$ discussed in preceding example. Thus, the above power equation becomes:

$$P_{T,dyn}(\mathcal{A}) = \frac{E(A) + E(B) + ((t - D.st) \cdot E(D))}{\Delta(A) + \Delta(B)}$$

Comparing the above equation with the power dissipation of computation path $P(\mathcal{CP}_1(A, B))$, described in the previous example, one can see that the difference in power dissipation consist of the energy consumption of the action D. The example showed that, in the presence of kill operation, the

fact that the killed action behaves as the skip action does not justify to exclude it from power estimation.
End of example.

4.5 Instantaneous Power Dissipation

As stated in Sect. 1.4, in many applications, the average power estimate may not be enough. Often it is important to be able to estimate instantaneous power as well. In general, this type of analysis is needed to analyze the power and ground bus networks for finding DC-voltage drop problems [36] [1], which lead to a reduced circuit speed due to the lowered supply voltage. Another obvious application is in noise analysis because glitches on the power supply are coupled into the circuit leading to noisy and possibly erroneous signals. The presented model, as it is targeted to above RTL, cannot give a detailed waveform; but the model can be used to find relative information between system models.

Consider a set $\mathcal{A}_T$ (4.20) where the observation period is divided into time segments T_i $(1 \leq i \leq m)$ that satisfy the following condition.

$$
\begin{aligned}
&\big(T_i \subseteq T\big) \wedge \big(i \geq 1\big) && \text{(p1)}\\
\wedge\ &\big(\forall t : T_i.st < t < T_i.ft : (\forall A\!: A \in \mathcal{A}_T : t \notin \{A.st, A.ft\})\big) && \text{(p2)}\\
\wedge\ &\Big(\exists m : m \geq i : \\
&\qquad \big(i = 1 \Rightarrow T_i.st = T.st\big)\\
&\qquad \wedge \big(1 \leq i < m \Rightarrow T_i.ft = T_{i+1}.st\big)\\
&\qquad \wedge \big(\exists A : A \in \mathcal{A}_T : (T_i.ft \in \{A.st, A.ft\})\big) && \text{(p3)}\\
&\qquad \wedge \big(i = m \Rightarrow T_i.ft = T.ft\big)\Big)
\end{aligned}
$$

Hence, the start time of a time segment T_i is either the start time $T.st$ of the observation period T or the start or finish time of an action $A \in \mathcal{A}_T$ (*p1,p3*). Analogously, the finish time of a time segment T_i is either the finish time $T.ft$ of the whole observation period T or the start or finish time of an action $A \in \mathcal{A}_T$. Furthermore, the finish time of a segment T_i is the start time of the next segment T_{i+1} for $1 \leq i < m$ and $m > 1$. No action $A \in \mathcal{A}_T$ is started or finished *inside* a time segment T_i, i.e. when $T_i.st < t < T_i.ft$ (*p2*).
The definition of time segments is illustrated in Fig. 4.14, where three timed actions A, B, and D are executed once during the observation period T.

[1] "IR" drop according to Ohm's law: $(V = R \cdot I)$, where R is the equivalent path DC resistance between the source location and the device location and I is the average current

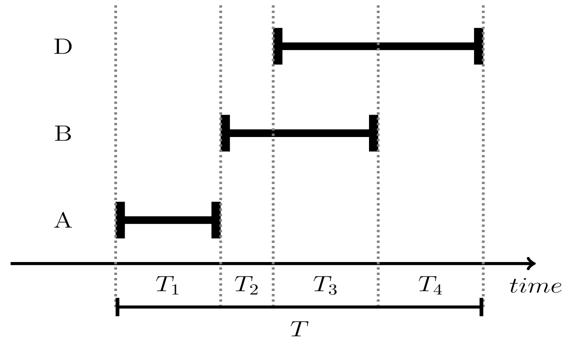


Figure 4.14: Illustration of time segments

The observation period is divided into four time segments according to the definition above.

The instantaneous power consumption is defined for each timed segment T_i by adding together the power consumptions of all the enabled actions in that particular segment. This follows from the linear energy model, depicted in Fig. 4.6, where all the timed actions have a "constant power". For instance, the power consumption of an arbitrary action A is $P(A) = \frac{E(A)}{\Delta(A)}$ (4.13). Therefore, the instant power consumption is:

$$P_{inst}(T_i) \mathrel{\widehat{=}} \sum_{A \in \mathcal{A}_{T_i}} P_{dyn}(A) \qquad (\textit{instant power consumption}) \quad (4.29)$$

where P_{dyn} is the dynamic power consumption of the enabled actions in time segment T_i. The set $\mathcal{A}_{T_i}$ defines those actions that are enabled during the time segment T_i ($\mathcal{A}_{T_i} \subset \mathcal{A}_T$). Observe that only the dynamic power is estimated because the instant power dissipation analysis evaluates the simultaneous switching activity inside system(s).

Example 4.11. Consider the situation presented in Fig. 4.14 where three actions A, B, and D are executed during the observation period T. The instant power for time segments 1, 2, and 4 is:

$$\begin{aligned} P_{inst}(T_1) &\overset{(4.29)}{=} P_{dyn}(A) \\ &\overset{(4.13)}{=} \frac{E(A)}{\Delta(T_1)} = \frac{E(A)}{\Delta(A)} \end{aligned}$$

the chip draws from the supply

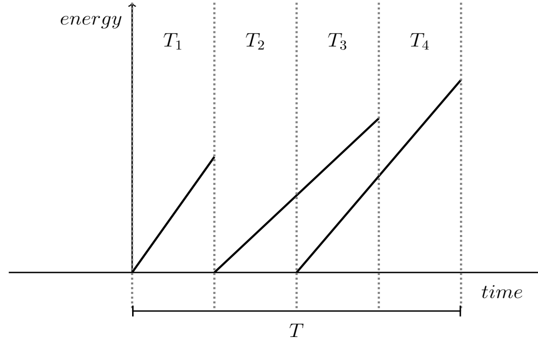


Figure 4.15: Instantaneous power estimation per time segments

$$P_{inst}(T_2) \overset{(4.29)}{=} P_{dyn}(B) \overset{(4.13)}{=} \frac{E(B)}{\Delta(T_2)} = \frac{E(B)}{\Delta(B)}$$

$$P_{inst}(T_4) \overset{(4.29)}{=} P_{dyn}(D) \overset{(4.13)}{=} \frac{E(D)}{\Delta(T_4)} = \frac{E(D)}{\Delta(D)}$$

The time segment 3 contains two enabled actions, and therefore the instant power consumption is calculated by:

$$P_{inst}(T_3) \overset{(4.29)}{=} P_{dyn}(B) + P_{dyn}(D) \overset{(4.13)}{=} \frac{E(B)}{\Delta(B} + \frac{E(D)}{\Delta(D)}$$

The energy dissipation during the observation period T, and the way it is distributed among time segments $T_1 - T_4$ is shown in Fig. 4.15.
End of example.

4.6 Discussion on Models

The accuracy of the proposed area complexity model was compared with the model provided by BDDs. A detailed study on the feasibility analysis can be found in [84]. As stated in Sect. 1.4, several tools exists to use and manipulate BDDs. The introduced area complexity model was evaluated using Verification Interacting with Synthesis (VIS) tool [10] together with selected benchmark circuits from ACM/SIGDA benchmark set [1]. These benchmark circuits are written in Berkeley Logic Interchange Format (BLIF) format. The generous number of available benchmark circuits and especially their documentation were the main reasons to select this tool.

Table 4.3: Results (area complexity).

Design	VIS-2.1	Action model	%
16-bit multiplier	198	272	72
32-bit multiplier	374	544	69
16-bit adder	48	72	67
32-bit adder	96	160	60
8-bit ALU (+ control)	447	250	178
Arbiter with 3 clients	323	327	99
GCD	768	967	79

The functionality of the benchmark circuits were described using the Action Systems formalism according to their documentation after which their area complexities were evaluated. In the area complexity calculations, the complexity factors shown in Table 4.2 were adopted. The results were compared with the results gained from the VIS tool, shown in Table 4.3, where the first column specifies the name of the benchmark circuit, the second column the size of BDD, the third one area complexity of the actions model, and the fourth one the percentual difference between BDD and the area complexity model presented in this chapter. If the percent is less than hundred, the BDD model was smaller than the corresponding action systems model. The comparison was made in a way that *root* and *child nodes* of the presented model are thought as BDD nodes, and the *leaf nodes* in both cases are the same, that is logic zero or logic one. Thus, the node count and the area complexity were compared directly.

The direct comparison is possible because the basis of the area complexity model is on BDDs, and, furthermore, as the abstraction level decreases, the BDD generation from the action system description becomes possible. That is, at higher abstraction levels, the generation of Boolean functions from the action system description is difficult. In general the accuracy of the model varied most commonly between 60% and 70% from the BDD model. Furthermore, in most cases the area of the formal system description was larger than the Boolean ones. This supports the idea that as the level of abstraction decreases, the area complexity analysis will be more and more accurate as the formal system descriptions become closer to the Boolean level.

The presented accuracy is for the area complexity model, and therefore direct comparisons cannot be made with the existing high level power models. The macromodel based approach, presented in [45], the range of maximum errors varied from 12 % to 33 %. The presented model has an average error rate in between 30% to 40%, which is a satisfactory result. This

is due to the fact that the presented model is only for the area complexity, and therefore direct comparison between existing high level power models is out of focus. Furthermore, the abstraction level of the presented model is above RTL, and thus the accuracy of the model is likely to suffer because of that. In other words, as the abstraction level increases, the accuracy of the model decreases, and the relative estimates becomes more and more important.

In power estimation the area complexity is used to estimate the physical capacitance in dynamic power modeling and as a gate count estimate, which is needed in static power consumption. The timing information is gained from the base formalism, Timed Action Systems [86]. Adopting the area complexity and the time-aware formalism one is able to built a power modeling framework. Observe that the other variables affecting to the power consumption are assumed to be parameters, which are updated as the level of abstraction decreases. Furthermore, the power estimation framework presented in this chapter do not cover communication structures, which is discussed in Chapter 7.

4.7 Chapter Summary

This chapter presented a power estimation framework for Timed Action Systems. At first, the sources of power consumption were discussed after which the area complexity model for a timed action was defined. Adopting the area complexity model, one is able to reason on the relative sizes between timed actions. In addition, the area complexity model is used to estimate both dynamic power (load capacitance estimate) consumption and static power consumption (gate count). After the power estimation is defined for timed action, the chapter proceeds into more complex analysis environment, the system level environment, where the area complexity and power consumption are defined for timed action systems. Once the basic power modeling framework is set for systems, the chapter defines different methods to reason on power consumption inside a system. That is, it defines ways to compare the power consumption of different computation paths within the system. Furthermore, the parallel behavior between two or more independent timed actions is discussed, and its effect to the power estimation. Finally the instant power modeling is presented, which analyzes the power consumption in a more detailed manner, detecting possible hot spots in the system. Finally, the presented area complexity model was compared with a BDD tool. The next chapter introduces system constraints, which are essential for system verification. These constraints can be both functional and non-functional properties.

Chapter 5

Specification of System Constraints

The previous chapter introduced a power aware design framework for Timed Action Systems. The presented framework consists of techniques to estimate both area and power characteristics of the system under design by utilizing the properties of the underlying formalism. The power characteristics are verified against the power requirements, which are defined in the system's specification. These requirements are in fact constraints, i.e., restrictions posed either by the systems environment or by a designer. In this chapter, various constraint types and their specification are explored. A constraint can be defined as a Boolean condition that indicates how an involved timed action must operate; i.e., a violation of such conditions causes an unpredictable computation. If the condition defined by a constraint evaluates to *true*, then constraint holds, otherwise it evaluates to *false*, which means that the constraint is not satisfied. In general, the verification of a constraint takes place either before or after the execution of the action.

5.1 Deadline

A *deadline* [86] defines the maximum time that a sequence of timed actions may consume with their operation. The deadline is defined using the computation path concept $\mathcal{CP}(A, B)$, and it is of the form:

$$\mathcal{D}\big(\mathcal{CP}(A, B), d\big) \mathrel{\widehat{=}} \big(\Delta(\mathcal{CP}(A, B) \leq d)\big) \qquad (\textit{deadline}) \quad (5.1)$$

where it is required that the computation path delay $\Delta(\mathcal{CP}(A, B))$ is less than or equal with d. That is, a deadline $\mathcal{D}$ is regarded as a logical condition in proofs. The deadline is evaluated to *true* ($\mathcal{D} = T$) when $\Delta(\mathcal{CP}(A, B)) \leq d$ and to *false* ($\mathcal{D} = F$) when $\Delta(\mathcal{CP}(A, B)) > d$.

5.2 Area Complexity

An *area* constraint defines the maximum area complexity that a timed action or a timed action system may consume. For a single timed action, the area constraint is of the form:

$$\mathcal{C}(A, c) \mathrel{\widehat{=}} (C(A) \leq c) \qquad (\textit{area}) \quad (5.2)$$

---

where it is required that the area complexity of the action A is less than or equal with the area constraint c. The area constraint c is set by a designer. The area constraint is defined for timed action systems as follows:

$$\mathcal{C}(\mathcal{A}, c) \mathrel{\widehat{=}} (C(\mathcal{A}) \leq c) \qquad (\textit{area (system)}) \quad (5.3)$$

---

where the area complexity of the system is compared with the area constraint c. Similarly to the deadline constraint, the area constraints of the system $\mathcal{A}$ evaluates to *true* if $C(\mathcal{A}) \leq c$ hold and to *false*, if $C(\mathcal{A}) > c$ fail. The area constraint of an action behaves similarly.

The $\mathcal{C}(\mathcal{A})$ denotes the conjunction all the area complexity constraints of the system $\mathcal{A}$. Therefore, it is required that all the area constraints in the system hold before one is able to state that $\mathcal{C}(\mathcal{A})$ holds:

$$\mathcal{C}(\mathcal{A}) \mathrel{\widehat{=}} \mathcal{C}_1 \land \mathcal{C}_2 \land \ldots \land \mathcal{C}_n$$

where n is the number of constraints in the system $(1 \leq i \leq n)$ and C_i is a constraint defined for a single action or for a subsystem of $\mathcal{A}$.

5.3 Power Consumption

A *power* constraint defines the maximum allowable power dissipation that a sequence of timed actions may consume in its operation. The definition of the power constraint is divided into two: First, the power constraint of single timed action is discussed, and second the constraint is defined at system level.

5.3.1 Timed action

The power constraint of a single timed action is of the form

$$\mathcal{P}(A, p) \mathrel{\widehat{=}} (P_{tot}(A) \leq p) \qquad (\textit{power}) \quad (5.4)$$

---

where it is required that the average power P_{tot} of the action A is less than or equal with the maximum allowed power dissipation p. In general this value is defined by a designer. In this thesis, the reasoning on whether the power constraint hold or not is done using the deadline and area constraints set for the action. Consider a timed action A, the average power, defined by (4.18), depends on both time and area complexity (defined below for the ease of reference):

$$\begin{aligned} P_{tot}(A) &= P_{dyn} + P_{stat} \\ &\overset{(4.13)(4.17)}{=} \frac{E(A)}{\Delta(A)} + C(A) \cdot P^1_{stat} \\ &= \frac{C(A) \cdot E^1}{\Delta(A)} + C(A) \cdot P^1_{stat} \end{aligned}$$

where $\Delta(A)$ is the delay of the action, and $C(A)$ is the area complexity of the action. Therefore, if the maximum allowed power consumption p is defined using the maximum allowed delay and area complexity values, the power constraint is validated as follows:

- The delay of the action is less than or equal with the defined deadline constraint (5.1).
- The area complexity of the action is less than or equal with the maximum allowed area complexity of the action (5.2).

5.3.2 Timed action system

A power constraint of a timed action system $\mathcal{A}$ is of the form:

$\mathcal{P}_T(\mathcal{A}, p) \mathrel{\widehat{=}} (P_{T,avg}(A) \le p)$ *(power(system))* (5.5)

---

where it is required that the average power of the timed action system $\mathcal{A}$ during observation period T is less than or equal to the maximum allowable power dissipation p. The limit value p is defined by a designer. In this thesis, the reasoning on whether the constraint holds or not is based on the area and timing constraint set for the system. Therefore, the first requirement is that the area constraint ($\mathcal{C}(\mathcal{A})$) (5.3) of the system holds. The average power of timed action system is calculated for a particular observation period, defined in (4.20), which is a monitoring window set by a designer. The observation period is usually determined in a way that it includes one or more computation cycles of the system. Therefore, if one wants to compare the average power consumption between, for instance,

two system descriptions, the observation period should consists of similar amount of computation cycles, otherwise the results are not comparable in terms of average power consumption. Consider a timed action system $\mathcal{A}$, whose execution loop is of the form:

$$\textbf{forever do } A;B \textbf{ od}$$

where the actions A and B are executed sequentially so that A enables B. To constrain the execution time of these actions, the deadline is defined for both actions $\mathcal{D}(A, d_1)$, and $\mathcal{D}(B, d_2)$. Furthermore, it is assumed that both of these deadline constraints are satisfied during the refinement of timing constraints. To estimate the average power of a single computation cycle, that is, both actions are executed once, the observation period T is defined: $T = [A.st, B.ft]$. The delay information can be used to validate that the observation period is set correctly. That is, the delay of the observation period $\Delta(T)$ have to be the sum of the delays of the actions: $\Delta(T) = \Delta(A) + \Delta(B)$. The properties and refinement of the observation period is further discussed in the next Chapter.

The system level power constraint $\mathcal{P}(\mathcal{A}, p)$ denotes all the power constraints in the system $\mathcal{A}$ (the similar condition was set for the system level area constraint as well):

$$\mathcal{P}(\mathcal{A}) \mathrel{\widehat{=}} \mathcal{P}_1 \wedge \mathcal{P}_2 \wedge \ldots \wedge \mathcal{P}_n$$

where i is the number of constraints in the system $\mathcal{A}$ $(1 \leq \imath \leq n)$, and $\mathcal{P}_i$ is a constraint defined for an action or a subsystem of $\mathcal{A}$.

5.4 Chapter Summary

This chapter introduced the typical constraints for the power aware modeling environment. A constraint defines a restriction whose strict adherence is mandatory and violating such conditions causes an unpredictable computation. From Timed Action Systems, a *deadline* constraint is introduced because it is utilized in the definition and analysis of power constraints. Addition to the deadline constraint, an *area* and *power* constraints are defined. The former one can be used to constraint the area complexity of a single action or an action system. The latter one is used to constraint the average power of a timed action or timed action system. The introduced constraints are essential in the verification of the SoC designs, not only logically but also with respect to given area and power constraints. Therefore, in the next chapter these constraints are used to ensure the correctness of the system in terms of area and power dissipation.

Chapter 6

Development of Systems

Conventional Action Systems, and their time-aware extension Timed Action Systems, are meant to be designed in a stepwise manner within the *refinement calculus framework* [22] as discussed in Chapter 2. The refinement calculus preserves the correctness of actions during the refinement procedure. Both functional and non-functional (time) characteristics of an abstract system specification can be transformed towards more concrete system description by adopting the time aware refinement calculus [86]. The scope of this study is to further extend the time-aware refinement calculus to preserve the correctness of area and power characteristics analyzed in the timed action context. The presented refinement methodology describe performance related conditions that must be satisfied during the transformation into a more concrete design. In other words, the purpose is to give tools to start the design process from a high level specification and to develop it in a stepwise manner towards an implementable specification. The correctness preserving transformations ensure that the implementable specification reflects the abstract specification, and, furthermore, the time, area, and power related conditions are satisfied.

6.1 System Requirements

System development in Timed Action Systems requires monitoring both functional and non-functional behavior. In this study, the time-aware refinement calculus is further extended to analyze the power consumption of a system. The objective of this extension is to include requirements into the refinement framework that are sufficient to prove the correctness of the transformation. The selected approach validates the power related constraint by showing that requirements given for an abstract system are satisfied by the concrete system as well. This approach follows directly from the time aware refinement approach. Timing information plays an impor-

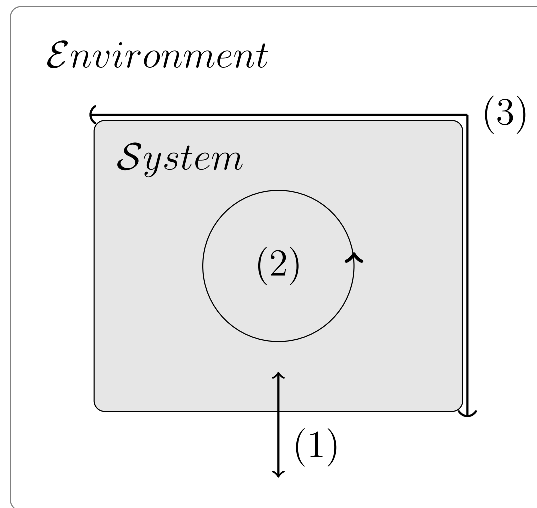


Figure 6.1: System / environment interaction

tant role in the power analysis. In this chapter, the scope is to discuss those requirements that are set for the system under development for power analysis. In Fig. 6.1, the most commonly posed requirements for system under design are as follows:

(1) *Interaction between system and its environment.* Environment communicates with the system at some frequency and provides input data for the system. The amount of input data determines the computational activity of the system. Furthermore, the system's output data is read by the environment. Moreover, environment might disable the system if there is no data to process, for example, for power saving purposes.

(2) *Computation time*, for example, a time interval between the update of input and output variables. In power analysis purposes, this affects especially dynamic power consumption.

(3) *The area of the system*, the amount of logic needed to perform the computation. The area of the system is adopted to evaluate both dynamic and static power consumption of the system.

6.2 Refinement

Action systems are meant to be developed in a stepwise manner within the *refinement calculus* framework as discussed in Sect. 2.7.2 on page 26. In this

section, the refinement calculus is extended by first describing refinement of Timed Action Systems [86] and further extending it to cover the power analysis. As stated in Chapter 4, the high level power estimation model is based on timing and area estimates. Therefore, these properties dominate in power aware refinement.

6.2.1 Refinement of Timed Action Systems

Timed Action Systems applies and extends the commonly used refinement method presented in Sect. 2.7.2, which used data refinement on establishing the foundation for refinement of an action system.

Consider timed action systems $\mathcal{A}$ and $\mathcal{C}$:

sys $\mathcal{A}$ (g;) ::
|[
delay
$dA : dA0$;
constraint
$\mathcal{T}(\mathcal{A})$;
variable
a;
action
$A[\![dA]\!] : (aA)$;
initialization
$g, a := g0, a0$;
execution
forever do A **od**
]|

sys $\mathcal{C}$ (g;) ::
|[
delay
$dC : dC0; dX : dX0$;
constraint
$\mathcal{T}(\mathcal{C})$;
variable
c;
action
$C[\![dC]\!] : (aC)$;
$X[\![dX]\!] : (aX)$;
initialization
$g, c := g0, c0$;
execution
forever do C [] X **od**
]|

where aA, aC, and aX are any of the atomic actions defined earlier, and $\mathcal{T}$ defines the time constraints posed on the abstract and concrete systems $\mathcal{A}$ and $\mathcal{C}$, respectively. The above systems operate in an environment $\mathcal{E}nv$, which is of the form:

sys $\mathcal{E}nv$ (g;) ::
|[
delay
$dE : dE0$;
constraint
$\mathcal{T}(\mathcal{E}nv)$;
variable
e;
action
$E : (aE)$;
initialization
$g, e, = g0, e0$;
execution
forever do E **od**
]|

Let $R(a, c)$ be an abstraction relation between the local variables a and c. The abstract timed action system $\mathcal{A}$ is refined by the concrete timed action system $\mathcal{C}$, denoted by $\mathcal{A} \sqsubseteq \mathcal{C}$, if there exists such an abstraction relation $R(a, c)$ that the conditions, defined below, (i) -(vi) hold. Furthermore, let R_T be an abstraction relation of the timing behavior between the concrete system $\mathcal{C}$ and the abstract system $\mathcal{A}$. The abstract timed action system $\mathcal{A}$ is refined by the concrete system $\mathcal{C}$, if there exists such an abstraction relation R_T that the condition (vii) hold.

$R(a_0, c_0) = true$	(*initialization*)	(i)
$A \leq_R C$	(*main action*)	(ii)
$skip \leq_R X$	(*auxiliary action*)	(iii)
$R \wedge gd(A) \Rightarrow gd(C) \vee gd(X)$	(*continuation condition*)	(iv)
$R \Rightarrow \mathbf{wp}\ (\mathbf{do}\ X \mathbf{od}, true)$	(*internal convergence*)	(v)
$R \wedge \mathbf{wp}\ (E, true) \Rightarrow \mathbf{wp}\ (E, R)$	(*non-interference*)	(vi)
$R_T \wedge \mathcal{T}(\mathcal{A}) \Rightarrow \mathcal{T}(\mathcal{C})$	(*timing behavior*)	(vii)

(i) The first condition says that the initialization of the systems $\mathcal{A}$ and $\mathcal{C}$ establish the abstraction relation R.

(ii) The second condition requires the abstract action A to be data-refined by the concrete action C using R.

(iii) The third condition, in turn, indicates that the auxiliary action X is obtained by data-refining a *skip* action. This basically means that X behaves like a *skip* action with respect to the global variables.

(iv) The fourth condition requires that whenever the action A of the abstract system is enabled, assuming the abstract relation R holds, there must be enabled action in the concrete system $\mathcal{C}$ as well.

(v) The fifth condition states that if R holds, the execution of the auxiliary action X, taken separately, must terminate at some point.

(vi) The sixth condition guarantees that the interleaved execution of the environment actions E preserves the abstract relation R.

(vii) The seventh condition requires that all the time constraints are met in the concrete timed action system $\mathcal{C}$, preserving the constraints set for the abstract timed action system $\mathcal{A}$.

The conditions (i)-(vi) are adopted as such from the refinement of Action Systems. This is justified by the fact that Timed Action Systems defines a

delay that determines the duration after which the result of computation is written onto write variables. Therefore, the operation part, the functionality of an action is not altered. The seventh condition (*vii*) guarantees that the timing constraints are met in the concrete timed action $\mathcal{C}$. The functionality of a timed action is refined using the data refinement of conventional actions keeping the delay unaltered. The timed action A is data-refined by the timed action C if the condition $A \leq_R C$ holds:

$$A \leq_R C \Rightarrow A[\![dP]\!] \leq_R C[\![dP]\!]$$

where it is required that the timing constraint set for the abstract system are satisfied in the concrete system as well. The proof of this condition can be found in [86].

6.2.2 Power aware refinement

To refine the power characteristics in the timed action system context, it is required that the above presented conditions (*i*) - (*vii*) hold. The six first conditions are related to the refinement of the conventional actions and their tenability must be ensured only if the functionality of the system is changed during refinement. The seventh condition relates to timing aspects, ensuring that the timing behavior of the system is validated. The presented power aware refinement framework introduced below is build upon Timed Action Systems, and therefore it is required that the conditions (*i*) - (*vii*) are validated before the area condition (*viii*) and power refinement condition (*ix*). Observe that the area constraint can be validated for conventional Action Systems as well. In that case it is required that the conditions (*i*) - (*vi*) and (*viii*) are satisfied, but then one is not able to constraint timing and power characteristics of the system.

The area complexity of the abstract action system $\mathcal{A}$ is refined by the concrete action system $\mathcal{C}$ if there exists such an abstraction relation R_C that the following condition (*viii*) holds.

$R_C \wedge \mathfrak{C}(\mathcal{A}) \Rightarrow \mathfrak{C}(\mathcal{C})$ (*area complexity*) (viii)

---

where it is required that the area complexity constraints set for the abstract system $\mathcal{A}$ hold for the concrete system $\mathcal{C}$. That is, the reasoning about the correctness of the area complexity is mainly to ensure that the size of the system does not exceed the given limit due to the performed refinement step. Therefore, for simplicity, the delays are excluded. Consider an abstract system $\mathcal{A}$ whose execution loop is of the form:

forever do A_1**;**A_2 **od**

The area complexity of the system is:

$$\begin{aligned} C(\mathcal{A}) &= C(A_1 \;[]\; A_2) \\ &\overset{(4.8)}{=} C(A_1) + C(A_2) \end{aligned}$$

Furthermore, the system has an area constraint, which is:

$$\mathcal{C}(\mathcal{A}, c) = C(\mathcal{A}) \leq c$$

where it is required that the area complexity of the abstract system $\mathcal{A}$ is less than or equal with the maximum allowable area complexity c.

After the refinement step, the constraint set for the abstract system $\mathcal{A}$ must hold in the concrete system $\mathcal{C}$ too. The concrete system $\mathcal{C}$ is of the form:

$$\textbf{forever do } C_1\textbf{;}C_2 \textbf{ od}$$

and the system has an area constraint:

$$\mathcal{C}(\mathcal{C}, c) = C(\mathcal{C}) \leq c$$

where it is required that the area complexity constraint set to the concrete system $\mathcal{C}$ hold. Observe that both of these conditions hold as long as the area complexity of the abstract system and the area complexity of the concrete system are smaller than the maximum allowable area (c) for the system. Moreover, it is assumed that $C(\mathcal{A})$ and $C(\mathcal{C})$ are non-zero because if a system exist, its size has to be non-zero.

Assume an abstraction relation $R_C \mathrel{\widehat{=}} (C(C_1) \leq C(A_1) \wedge C(C_2) \leq C(A_2))$ hold. Then

$$\begin{aligned}
&(\mathcal{C}(\mathcal{A}, c)) \\
&\Leftrightarrow \{\text{definition of area constraint (5.3)}\} \\
&(C(\mathcal{A}) \leq c)) \\
&\Leftrightarrow \{\text{definition of area complexity (4.8)}\} \\
&((C(A_1) + C(A_2)) \leq c) \\
&\Rightarrow \{\text{abstraction relation } R_C\} \\
&((C(C_1) + C(C_2)) \leq c) \\
&\Leftrightarrow \{\text{definition of area complexity (4.8)}\} \\
&(C(\mathcal{C}) \leq c)) \\
&\Leftrightarrow \{\text{definition of area constraint (5.3)}\} \\
&(\mathcal{C}(\mathcal{C}, c))
\end{aligned}$$

To refine the average power characteristics of the timed action system, let R_p be an abstraction relation. The power constraints of the timed action system $\mathcal{A}$ are refined by the concrete timed action system $\mathcal{C}$ if there exists such an abstraction relation R_p that the following condition hold:

$R_p \wedge \mathcal{P}_T(\mathcal{A}) \Rightarrow \mathcal{P}_T(\mathcal{C})$ (*power*) (ix)

---

where it is required that the power constraints set for the abstract system $\mathcal{A}$ hold for the concrete system $\mathcal{C}$ as well. Assume that the abstract system is of the form:

forever A **od**

and the concrete system is of the form:

forever C **od**

The abstract system $\mathcal{A}$ has a power constraint:

$$\mathcal{P}_T(\mathcal{A}, p) = P_{T,avg}(\mathcal{A}) \leq p$$

where it is required that the average power of the abstract system $\mathcal{A}$ does not exceed the maximum allowable average power consumption p. After a system development, the concrete system $\mathcal{C}$ must also satisfy the power constraint:

$$\mathcal{P}_T(\mathcal{C}, p) = P_{T,avg}(\mathcal{C}) \leq p$$

To validate the above constraints, assume an abstraction relation
, $R_p \mathrel{\widehat{=}} (P_{tot}(C) \leq P_{tot}(A))$. The abstraction relation is *true* if the total power consumption of the action C in the concrete system $\mathcal{C}$ is less than or equal with the total power consumption of action A in the abstract system $\mathcal{A}$. The total power for both actions can be evaluated using (4.18). To reason whether the relation hold or not, both area and timing characteristics must be taken into account. That is, area complexity has an effect on both dynamic and static power consumption of an action, whereas the time has an effect to the dynamic power consumption. The area constraint (condition (*viii*)) holds if the following abstraction relation holds: $R_C \mathrel{\widehat{=}} C(C) \leq C(A)$. By assuming that the area constraint hold, one is able to state that the static power dissipation of the abstract system $\mathcal{A}$ is greater than or equal to the static power dissipation of the concrete system $\mathcal{C}$. In the beginning of this section, it was required that the timing behavior is validated before the

power aware refinement is performed, and therefore the deadline constraint is satisfied, for instance, the deadline can be of the form $\Delta(C) \leq \Delta(A)$. Assuming that both the area and the deadline constraints hold, one is able to reason on the validity of the power constraint. Naturally, at this abstraction level this is not an exact operation, but assuming that the static power consumption is at least half of the total power consumption, one can state that the power constraint holds as well.

At system level, however, the power evaluation is carried out during a particular observation period, which usually consists of several executed actions. The observation period is usually defined, by a designer, in a way that it contains, for instance, a single computation cycle. That is, during refinement, one must make sure that the observation period defined for the abstract system and the observation period in the concrete system contains the same functionality. In general, the observation period is either fixed or relative:

- Fixed observation period: $T_{\mathcal{A}} = [T.st, T.ft] \wedge T_{\mathcal{C}} = T_{\mathcal{A}}$.
- Relative observation period: $T_{\mathcal{A}} = [A_i.st, A_j.ft] \wedge T_{\mathcal{C}} = [C_i.st, C_j.ft]$

where in the first case, the fixed observation period, requires that the observation period is same in both the abstract and concrete systems, and the second case, the relative observation period requires that the functionality of the abstract and the concrete systems must be the same inside the observation period, but the start and finish times do not necessary have to be same between the observation period set for the abstract system $T_{\mathcal{A}}$ and the observation period $T_{\mathcal{C}}$ set for the concrete system.

Consider the abstract system, presented in the definition of the refinement of area complexity:

$$\textbf{forever do } A_1\textbf{;}A_2 \textbf{ od}$$

and the concrete system is of the form

$$\textbf{forever do } C_1\textbf{;}C_2 \textbf{ od}$$

The abstract system has a power constraint:

$$\mathcal{P}_T(\mathcal{A}, p) = P_{T,avg}(\mathcal{A}) \leq p$$

where it is required that the average power of the abstract system $\mathcal{A}$ cannot exceed the maximum allowable power dissipation p set for the system. After the system development, the concrete system $\mathcal{C}$ must also satisfy the power constraint:

$$\mathcal{P}_T(\mathcal{C}, p) = P_{T,avg}(\mathcal{C}) \leq p$$

to validate that the power constraint hold between the abstract and the concrete system, one have to show that the following abstraction relation hold:

$$R_p \mathrel{\widehat{=}} (P_{T,avg}(\mathcal{C}) \leq P_{T,avg}(\mathcal{A}))$$

As stated above, to validate the power constraint, one have to reason on both area and timing behavior of these systems. It was shown, that the area constraints posed for these systems hold, when the following abstraction relation hold: $R_C \mathrel{\widehat{=}} (C(C_1) \leq C(A_1) \wedge C(C_2) \leq C(A_2))$. Therefore the area requirement of the power constraint refinement is satisfied. Assume that the observation periods defined for both the abstract and the concrete systems are fixed, and they are of the form:

$$T_{\mathcal{A}} = [A_1.st, A_2.ft]$$
$$T_{\mathcal{C}} = [C_1.st, C_2.ft]$$

where is required to show that $T_{\mathcal{A}} = T_{\mathcal{C}}$. This can be done using the deadline constraints. That is, one must show that $\Delta(A_1) + \Delta(A_2) = \Delta(C_1) + \Delta(C_2)$. The deadlines set for the timed actions A_1 and A_2 in the abstract system are of the form: $\mathcal{D}(A_1, d_1)$ and $\mathcal{D}(A_2, d_2)$, where the delay constraints d_1 and d_2 are set in a way that they equal the delays of the actions A_1 and A_2, respectively. Therefore, to show that the new timed actions satisfy the deadline constraints $\mathcal{D}(C1, d_1)$ and $\mathcal{D}(C2, d_2)$ one is able to validate that the fixed observation period hold for the concrete system. Due to the fixed observation periods, the validation of the power constraint depends on the validation of the area complexity constraint, and thus in this case the power constraint is satisfied.

In general, the above described use of the fixed observation period restricts the development of the systems at high abstraction levels, because, as shown above, tight timing constraints are required. Therefore, it is often more convenient to adopt the relative observation period, where the equal start and finish timed are not required. The benefit of this approach is that more variability is allowed in the execution times of the timed actions. In other words, the delay of the action can be significantly lower than the given deadline. This on the other hand makes the reasoning of the power constraint more difficult due the high abstraction level, where exact delay values are unknown. Thus, in this case, the power evaluation relies more on the area complexity evaluation.

6.2.3 Refinement of constraints

After a refinement step the restrictions of the original constraints must be satisfied by the new system as well. The refinement of constraint is carried out using abstraction relations R_T (time), R_C (area), and R_p (power), where in general, a constraint can be either *decomposed* and *reallocated*, or *updated* to meet the new timed action definitions. The former approach introduces two or more constraints and the latter approach retains the number of constraints unchanged. Naturally, the specification of the above mentioned abstraction relations must be done carefully to avoid relations that cannot hold between the abstract and concrete system descriptions.

Example 6.1. Consider a timed action system $\mathcal{S}$ whose execution loop is of the form:

$$\textbf{forever do } A \textbf{ od}$$

where the action A is defined by:

$$A[\![dA]\!] \mathrel{\widehat{=}} A_1; A_2$$

and, furthermore, whose operation is constrained by [1]:

(1) deadline $(\mathcal{D}(A, d))$

(2) area complexity $\mathcal{C}(\mathcal{S}, c)$

(3) power $\mathcal{P}_T(\mathcal{S}, p)$

During the development of timed action system the action A is decomposed into several atomic actions whose operation is sequenced by the non-atomic sequence operation (**;**) such that A_1 enables A_2. After refinement, a new timed action system $\mathcal{S}'$ is defined whose functionality is given by the execution loop:

$$\textbf{forever do } A_1\textbf{;}A_2 \textbf{ od}$$

On showing the correctness of the refinement, the nine $((i) - (ix))$ refinement conditions needs to be satisfied. The functionality of the actions is not determined, and therefore the functional refinement, (conditions *i*-*vi*), is not presented in a detailed manner. In this context, it is enough to show that the fifth condition (v), the internal convergence hold. In the refinement, the action A is decomposed into two new timed actions A_1 and A_2, which are sequentially composed in a way that A_1 enables A_2. This is the abstraction relation R, and it is assumed to be valid, as stated earlier in this example. The fifth condition requires that the auxiliary action, taken separately, must

[1]the numbers in front of the constraints denote the order of discussion

terminate at some point. Therefore, assume that A_1 is the auxiliary action X, and A_2 is the concrete action C in the new system. According to the abstraction relation the auxiliary action enables (always) the concrete action, and therefore the fifth condition holds. In other words, taken separately, the auxiliary action will terminate after single execution.

In the refinement of timing behavior (1), the original delay dA is reallocated among the new timed actions A_1 and A_2 using the abstraction relation R_T: $R_T \mathrel{\widehat{=}} dA_1 + dA_2 = dA$. Therefore, the refinement of timing behavior becomes:

$$A[\![dA]\!] \leq_{R_T} A_1[\![dA_1]\!]\mathbf{;}A_2[\![dA_2]\!]$$

where one have to show that the delay of the two new actions is less than or equal to the maximum operation time d. To accomplish this, the decomposition and reallocation procedure is selected. The deadline is generated for each new action:

$$\begin{aligned}
&\mathcal{D}(A_1, f \cdot d)\\
&\mathcal{D}(A_2, (1-f) \cdot d)
\end{aligned}$$

where the fraction f defines the portion in which the maximum operating time is shared amongst the new actions. Observe that $f \cdot d + (1-f) \cdot d = d$.

In the decomposition, one has to show that both of these new deadlines hold:

$$\begin{aligned}
&\mathcal{D}(A_1, f \cdot d) = true\\
&\mathcal{D}(A_2, 1 - f \cdot d) = true
\end{aligned}$$

which means that the delay of the actions A_1 and A_2 must be less than or equal to the maximum operation time of the original action. The computation path of the new composition becomes:

$$\mathcal{CP}(A_1, A_2) = \langle A_1, A_2 \rangle \tag{6.1}$$

One way to show the correctness of the new constraint(s) [86] is to adopt the above computation path.

$$\begin{aligned}
\Delta(\mathcal{CP}(A_1, A_2)) &\leq f \cdot d + (1-f) \cdot d\\
&\overset{(6.1)}{\Leftrightarrow} \Delta(A_1; A_2) \leq d\\
&\overset{(3.11)}{\Leftrightarrow} (dA_1 + dA_2) \leq d\\
&\overset{R_T}{\Leftrightarrow} (dA \leq d)\\
&= \mathcal{D}(A, d)
\end{aligned}$$

Next the refinement of the area complexity constraint (2) is described. The area complexity constraint set for the system $\mathcal{S}$ is $\mathcal{C}(\mathcal{S}, c)$, where the area complexity of the system $\mathcal{S}$ is:

$$\begin{aligned} C(\mathcal{S}) &= C(A) \\ &\overset{(6.1)}{=} C(\mathcal{CP}(A_1, A_2)) = C(A_1; A_2) \\ &\overset{(4.8)}{=} C(A_1) + C(A_2) \end{aligned}$$

In the refinement, the area complexity of the timed action A is divided between the two new timed actions according to the abstraction relation:

$$R_C \mathrel{\widehat{=}} C(A_1) + C(A_2) \leq c$$

where the maximum allowed area complexity is $c = C(A)$. The area constraints for the two new actions are

$$\begin{aligned} &\mathcal{C}(A_1, c \cdot x), (0 < x < 1) \\ &\mathcal{C}(A_2, c \cdot (1 - x)), (0 < x < 1) \end{aligned}$$

where x defines the portion in which the maximum area complexity is shared amongst the new timed actions. Naturally $c = x \cdot c + (1 - x) \cdot c$. To validate the new constraints, the computation path (6.1) is adopted:

$$\begin{aligned} C(\mathcal{CP}(A_1, A_2)) &\leq x \cdot c + (1 - x) \cdot c \\ &\overset{(6.1)}{\Leftrightarrow} C(A_1 \mathbin{;} A_2) \leq c \\ &\overset{(4.25)}{\Leftrightarrow} C(A_1) + C(A_2) \leq c \\ &\overset{R_C}{\Leftrightarrow} \mathcal{C}(\mathcal{S}', c) \end{aligned}$$

The power constraint (3) is set for the system to reason on average power dissipation between development phases. To validate the constraint, one must show that the following abstraction relation hold:

$$R_p \mathrel{\widehat{=}} P_{T,avg}(\mathcal{S}') \leq P_{T,avg}(\mathcal{S})$$

where it is required that the average power of the new timed action system $\mathcal{S}'$ less than or equal with the average power dissipation of the system $\mathcal{S}$. As stated in Sect. 6.2.2, both area and timing behavior have to be refined before the validation of the power constraint. The refinement of the area complexity is shown above (2), and therefore, it is enough to validate the observation

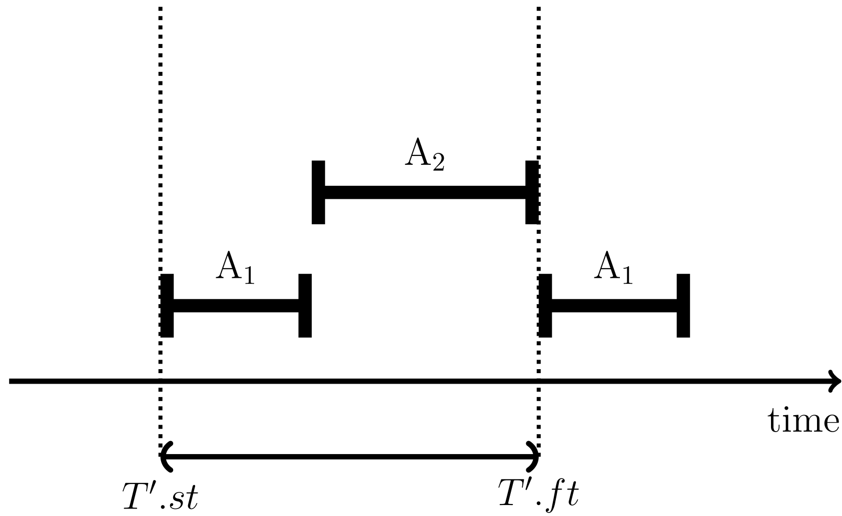


Figure 6.2: Observation period for the system $\mathcal{S}'$

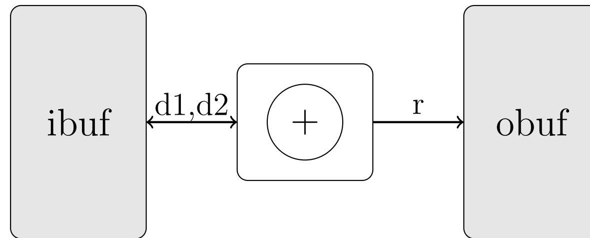


Figure 6.3: Closed timed action system $\mathcal{A}$

period used in power calculations. In this case, the fixed observation period is adopted because the delay dA of the action A is reallocated between the new actions A_1 and A_2 according to the abstract relation R_T. Therefore the observation period is: $T = [A.st, A.ft]$, and for the new actions it is defined in a way that both actions in the system are executed once, and thus the new fixed observation period is $T' = [A_1.st, A_2.ft]$. The new observation period is shown in Fig. 6.2. To prove the observation period of the power constraint, one have to show that: $T = T'$. As stated above, the observation period is fixed, due to the re-allocation of delays, and therefore it is enough to show that the deadline constraint hold (1). That is, to show that $\Delta(T) = \Delta(A_1) + \Delta(A_2)$ is true. Thus, the power constraint is satisfied.

End of Example.

Example 6.2. Consider the following timed action system $\mathcal{A}$:

```
sys 𝒜 (    ) ::
|[
  type
    Buffer : set of Data;
  delay
    dOp : dOp0;
    dAdd : dAdd0;
  constraint
    𝒟(Op⟨Add⟩), D);
    𝒞(𝒜, c);
    𝒫_T(𝒜, p);
  variable
    ibuf, obuf : Buffer;
    d1, d2, r : Data;
  private procedure
    Add[[dAdd]](in x1, x2 : out y) : (y := y'.(y' = x1 + x2));
  action
    Op[[dOp]] : (d1, d2 := d1', d2'.(d1', d2' ∈ ibuf)); Add(d1, d2, r);
                obuf := obuf ∪ {r};
  initialization
    ibuf, obuf := ibuf0, obuf0;
    d1, d2, r := d10, d20, r0;
  execution
    forever do Op  od
]|
```

where the timed action *Op* reads data from the input buffer *ibuf* after which it calls the local procedure *Add*. The procedure perform an addition operation on the passed values, and finally, the result is written into the output buffer *obuf*. The system $\mathcal{A}$ is illustrated in Fig. 6.3. Observe that the action *Op* is executed by the system $\mathcal{A}$ forever because its guard is invariantly *true*. Furthermore, the system is closed action system which can be seen from the Fig. 6.3, because there is no communication between the system and its environment.

The deadline $\mathcal{D}$ defines the maximum allowable operation time for the action *Op*. The duration of the computation is obtained by adding together the delay of the action *Op* and the delay of the procedure *Add*. That is, the sum of these two delays must be less than the given limit D:

$$\mathcal{D}(Op\langle Add\rangle), D)$$

The area constraint $\mathcal{C}$ defines the maximum allowed area complexity for the system. The area complexity of the system $\mathcal{A}$ is obtained by adding the area complexity of the timed action *Op* and the area complexity of the

procedure *Add* together (rule (4.6)). Furthermore, the sum of these must be less than the given limit c:

$$\mathcal{C}(\mathcal{A}, c) = C(\mathcal{A}) \le c$$

To evaluate the area complexity of the timed action *Op* and the procedure *Add*, assume that variables of type *Data* have an equal width $w = w_{Data}$, and the variables of type $Buffer$ have an equal width $w = w_{Buffer}$. First, the area complexity of the procedure is defined by (4.4):

$$\begin{aligned} C(Add) &\overset{(4.4)}{=} C(wA) + C(rQ) \\ &= w_{Data} + \frac{w_{Data} + w_{Data}}{2} \cdot 2^2 = 5w_{data} \end{aligned}$$

and the area complexity of the action *Op*:

$$\begin{aligned} C(Op) &\overset{(4.4)}{=} C(wA) + C(Q) \\ &= C(wA) = w_{Data} + w_{Data} + w_{Buffer} \\ &= 2w_{data} + w_{Buffer} \end{aligned}$$

Assuming that the output buffer can store only one result r at a time then one can assume that because the width of the variable r is w_{Data} then $w_{Buffer} \equiv w_{Data}$. Thus, the area complexity of the system becomes:

$$\begin{aligned} C(\mathcal{A}) &= C(Add) + C(Op) \\ &= (5w_{data}) + (2w_{data} + w_{buffer}) \\ &\overset{w_{Data} = w_{Buffer}}{=} 8w_{data} \end{aligned}$$

The maximum area complexity value allowed for the system is defined by a designer and in this case, it is assumed to be twice the value of the area complexity of the system $\mathcal{A}$:

$$c = 2 \cdot C(\mathcal{A}) \tag{6.2}$$

The power constraint $\mathcal{P}_T(\mathcal{A}, p)$ states that the average power of the system cannot exceed the limit value p. The fixed observation period for the power evaluation is defined to contain one computation cycle of the system. In other words, a single execution of the action *Op*. Thus, the power constraint is satisfied if the deadline constraint and area constraint are satisfied. **End of example.**

Example 6.3. Consider the timed action system $\mathcal{A}$ introduced in Example 6.2. The *Op* reads data from the input buffer and performs the addition operation after which it writes the results to the output buffer. To separate the read and write operations, the action *Op* (defined below for ease of reference):

$$
\begin{aligned}
Op[\![dOp]\!] \mathrel{\widehat{=}} & (d_1, d_2 := d_1', d_2'.(d_1', d_2' \in ibuf));\\
& Add(d_1, d_2, r);\\
& obuf := obuf \cup \{r\};
\end{aligned}
$$

is broken into two new timed actions ROp and Wr as follows:

$$
\begin{aligned}
ROp[\![dROp]\!] \mathrel{\widehat{=}} & \neg b \to d_1, d_2 := d_1', d_2'.(d_1', d_2' \in ibuf));\\
& Add(d_1, d_2, r);\\
& s, b := r, T;\\
Wr[\![dWr]\!] \mathrel{\widehat{=}} & b \to obuf := obuf \cup \{s\}; b := F;
\end{aligned}
$$

where the new data variable s stores the result of the computation temporarily, and the Boolean variable b ($b \in \{T, F\}$) is used to sequence the operation between the new timed actions.

The delays of the two new actions ROp and Wr are defined in a way that:

$$R_T \mathrel{\widehat{=}} dOp = dROp + dWR$$

where the delay of the timed action Op is reallocated between the two new actions. Observe that the delay of the procedure Add is not altered because its functionality is not changed in this refinement. In the new, refined timed action system, the delay of the procedure is included into the delay of the action ROp.

The area complexities of these new actions must satisfy the constraint:

$$R_C \mathrel{\widehat{=}} C(ROp) + C(Wr) \leq c$$

where c is the area constraint defined by (6.2). The area of the procedure Add is included into the area of the action ROp, and, furthermore, the area of the procedure is not altered because its functionality is not changed.

The power constraint of these new actions is defined in a way that:

$$R_p \mathrel{\widehat{=}} P_{tot}(ROp) + P_{tot}(Wr) \leq p$$

where it is required that the observation period of the new system contains a single execution of the actions ROp and Wr in a way that ROp enables Wr. Furthermore, the new (fixed) observation period T' have to be equal with the one defined for the system $\mathcal{A}$ ($T = T'$). In addition it is required that the area constraint is validated the power constraint presented above.

The behavior of the concrete system $\mathcal{A}'$ is of the form:

$$\textbf{forever do } ROp \ [\!] \ Wr \textbf{ od}$$

On showing the correctness of the refinement, the nine refinement conditions $((i) - (ix))$ need to be satisfied:

(i) *Initialization.* The initialization of the timed action system $\mathcal{A}$ and the new timed action system $\mathcal{A}'$, do not contradict.

(ii) *Main action.* The goal is to prove that $Op \leq_R Wr$ where R is an abstraction relation of the form $R \mathrel{\widehat{=}} b \Rightarrow r = s$.

$$
\begin{aligned}
\textit{Guard: } & R \wedge gd(Wr) \Rightarrow gd(Op) \\
& \Leftrightarrow R \wedge b \Rightarrow T \\
& \Leftrightarrow R
\end{aligned}
$$

$$
\begin{aligned}
\textit{Body: } & R \wedge gd(Wr) \wedge \mathbf{wp}\ (bd(Op), Q) \Rightarrow \mathbf{wp}\ (bd(Wr), Q \wedge R) \\
& \Leftrightarrow \{\text{weakest precondition of } bd(Op) \text{ and } bd(Wr)\} \\
& R \wedge b \wedge Q[obuf \cup \{r\}/obuf] \\
& \Rightarrow (b \Rightarrow r = s)[F/b] \wedge Q[obuf \cup \{s\}/obuf] \\
& \Leftrightarrow \{\text{the relation } R \mathrel{\widehat{=}} b \Rightarrow r = s\} \\
& (b \Rightarrow r = s) \wedge b \wedge Q[obuf \cup \{r\}/obuf] \\
& \Rightarrow (b \Rightarrow r = s)[F/b] \wedge Q[obuf \cup \{s\}/obuf] \\
& \Leftrightarrow \{\textit{logic}\} \\
& (b \Rightarrow r = s) \wedge b \wedge Q[obuf \cup \{r\}/obuf] \\
& \Rightarrow T \wedge Q[obuf \cup \{s\}/obuf] \\
& \Leftrightarrow \{\textit{logic}\} \\
& b \wedge (r = s) \wedge Q[obuf \cup \{r\}/obuf] \\
& \Rightarrow Q[obuf \cup \{s\}/obuf] \\
& \Leftrightarrow \{s = r\} \\
& b \wedge (r = s) \wedge Q[obuf \cup \{s\}/obuf] \Rightarrow Q[obuf \cup \{s\}/obuf] \\
& \Leftrightarrow \{\textit{logic}\} \\
& T
\end{aligned}
$$

Thus, the reasoning showed that $Op \leq_R Wr$ holds.

(iii) *Auxiliary action.* Because the auxiliary action *ROp* writes onto the variables s and b and it preserves the relation R, it behaves like a *skip* with respect to this kind of variables.

(iv) *Continuation condition.* There is always either of the new timed actions *ROp* or *Wr* enabled when the original timed action *Op* is enabled.

(v) *Internal convergence.* Holds trivially as the new auxiliary action disables itself.

(vi) *Non-interference.* Holds trivially as the new timed action system is closed.

(vii) *Timing behavior.* The deadline constraint $\mathcal{D}$ must be satisfied to show the correctness of the (vii) condition: $R_T \wedge \mathcal{D}(\mathcal{A}) \Rightarrow \mathcal{D}(\mathcal{C})$, where:

$$\mathcal{D}(\mathcal{C}) = (\mathcal{CP}(ROp, Wr), D)$$
$$\mathcal{D}(\mathcal{A}) = (Op\langle Add\rangle, D)$$

and the abstraction relation is $R_T \mathrel{\widehat{=}} dOp = DROp + dWR$.

To prove the above condition, a computation path is defined:

$$\mathcal{CP}(ROp, Wr) = \langle ROp, Wr\rangle \tag{6.3}$$

where the computation path definition follows from the functional behavior of the system. That is, the action *ROp* contains the first part of the action *Op* whereas the latter part is located in the action *Wr*. The correctness of the timing requirement is shown by calculating the computation path delay and comparing it with the delay of the original action *Op*. The computation path delay is:

$$\begin{aligned}
\Delta(ROp, Wr) &\overset{(6.3)}{=} \Delta(ROp\langle Add\rangle, Wr) \\
&\overset{(3.11)}{=} \Delta(ROp\langle Add\rangle) + \Delta(Wr)) \\
&\overset{(3.13)}{=} \Delta(ROp) + \Delta(Add) + \Delta(Wr) \\
&\overset{(3.9)}{=} dROp + dAdd + dWr \\
&\overset{R_T}{=} dOp + dAdd \\
&= \Delta(Op\langle Add\rangle)
\end{aligned}$$

according the original requirement $dOp + dAdd \leq D$ and the reasoning above, one may conclude that $dROp + dWr \leq dOp + dAdd$, and thus the timing requirement is satisfied.

(viii) *Area complexity.* The area complexity constraint is satisfied if the relation $R_C \mathrel{\widehat{=}} C(ROp) + C(Wr) \leq c$ hold. This is validated by calculating the area complexities of the new actions ROp and Wr, and then compare the result with area complexity constraint c. Notice that the functionality of the procedure *Add* is not changed during this procedure step, and therefore its area complexity is not altered. However, recalculation of the area complexity is required for the actions ROp and Wr, where the width of the Boolean variable b is assumed to be one $w_b = 1$ and the width of the auxiliary variable s is $w_s = w_{Data}$:

$$
\begin{aligned}
C(ROp) &\overset{(4.6)}{=} C(ROp) + C(Add) \\
&\overset{(4.5)}{=} \frac{w_b}{w_b} \cdot 2^{w_b} + w_{Data} + w_{Data} + w_{Data} + w_b + C(Add) \\
&\overset{(w_b=1)}{=} 3 + 4w_{Data} + C(Add) \\
C(Wr) &\overset{(4.5),(4.7)}{=} C(rgd) + C(wWr) + C(rQ) \\
&= \frac{w_b}{w_b} \cdot 2^{w_b} + w_{Buffer} + w_b \\
&\overset{(w_b=1)}{=} 3 + w_{Buffer} \\
&\overset{(ass. w_{Buffer}=w_{Data})}{=} 3 + w_{Data}
\end{aligned}
$$

where the width of the output buffer w_{Buffer} is assumed to be equal with w_{Data} ($w_{Buffer} = w_{Data}$), that is, this assumption is similar with the one that was made in the previous example.

In the previous example, it was shown that the abstract system $\mathcal{A}$ satisfies the area condition: $C(Op) + C(Add) \leq c$, where $c = 2 \cdot C(\mathcal{A})$. Therefore, to prove that the area condition is satisfied by the concrete system $\mathcal{A}'$, the abstract relation must hold:

$$
\begin{aligned}
R_C &\mathrel{\widehat{=}} C(ROp) + C(Wr) \leq c \\
&= C(ROp) + C(Add) + C(Wr) \leq c
\end{aligned}
$$

This condition holds if:

$$\frac{c}{C(ROp) + C(Add) + C(Wr)} \geq 1$$

Adopting the area complexities calculated for the new system and for the original one:

$$\frac{2 \cdot (8w_{data})}{5w_{data} + 3 + w_{data} + 3 + 4w_{data}} \geq 1$$

where it is assumed that the width w_{Data} (and multiples of it) are assumed to be of same type, which allow one to calculate the similar widths together. Furthermore, the integer widths are all from Boolean variables, and therefore, these widths are also added together. The inequality becomes:

$$\frac{16w_{data}}{10w_{data} + 6} \geq 1$$

where one can see that even if the value of the w_{data} is one ($w_{Data} = 1$) the inequality holds. However, the w_{Data} describes a data transfers, which usually implies that the numerical value of the width w_{data} is, for instance, 32 or 64. Thus, the area complexity requirement is satisfied.

(ix) *Power.* To satisfy the power requirement, one must show that the abstract relation R_p:

$$R_p \mathrel{\widehat{=}} P_{tot}(ROp) + P_{tot}(Wr) \leq p$$

holds. The area and timing constraints set for the system holds, and therefore it is enough to reason whether the new observation period includes the same functionality than the old one. The fixed observation period set for the average power calculation of the abstract system $\mathcal{A}$ is $T = [Op.st, Op.ft]$. To show that the new fixed observation period T' is valid for the concrete system $\mathcal{A}'$, one need to show that

$$\Delta(T') \mathrel{\widehat{=}} \Delta(ROp) + \Delta(Proc) + \Delta(Wr) = \Delta(T)$$

where the duration of the observation period T is defined to consist the computation path $\mathcal{CP}(ROp, Wr) = (ROp, Wr)$, and therefore one need to show that the following condition holds.

$$\Delta(ROp, Wr) = \Delta(Op\langle Add \rangle)$$

The above condition hold because of the re-allocation of the delay between new actions satisfied the condition (vii). From this it follows that the new fixed observation period for the new actions hold as well due the definition of the observation period. Thus, the power constraint is satisfied.

That is, all the refinement conditions ((i) - (ix)) are satisfied, and one has performed a refinement $\mathcal{A} \sqsubseteq \mathcal{A}'$, where the $\mathcal{A}'$ is of the form:

sys $\mathcal{A}'$ ( ) ::
|[
type
$Buffer$: **set of** $Data$;
delay
$dROp$: $dROp0$;
dWr: $dWr0$;
$dAdd$: $dAdd0$;
constraint
$\mathcal{D}((CP)(ROp, Wr)), D)$;
$\mathcal{C}((CP)(ROp, Wr)), c)$;
$\mathcal{P}_T((CP)(ROp, Wr)), p)$;
variable
$ibuf, obuf$: $Buffer$;
d_1, d_2, r: $Data$;
b: $Boolean$;
private procedure
$Add[\![dAdd]\!]($**in** x_1, x_2 : **out** $y)$: $(y := y'.(y' = x_1 + x_2))$;
action
$ROp[\![dROp]\!]: \neg b \rightarrow d_1, d_2 := d'_1, d'_2.(d'_1, d'_2 \in ibuf))$;
$Add(d_1, d_2, r); s, b := r, T$;
$Wr[\![dWr]\!]: b \rightarrow obuf := obuf \cup \{s\}; b := F$;
initialization
$ibuf, obuf := ibuf0, obuf0$;
$b := F$;
$d_1, d_2, r, s := d_10, d_20, r0, s0$;
execution
forever do ROp [] Wr **od**
]|

where the constraints are defined using the computation path, which form is congruent with the new action definitions.
End of example.

6.2.4 Decomposition

Decomposition allows one to form smaller systems from a large, complex system. During the decomposition a communication medium is generated between these new, smaller systems. This can be implemented by using either the variable or procedure based communication. The latter one abstracts away the detailed communication events, and therefore it is preferred at high abstraction levels. This idea has already been used in [68] where pro-

cedure based communication is used between decomposed systems.

The decomposition process is based on transforming local actions into procedures and in the end eventually turn these newly created local procedures into communication procedures. This is straightforward refinement as the atomicity is preserved and other actions are left intact. These actions are then grouped to reflect the forthcoming division of the action system. The decomposition of timed action systems is presented in [86] where it is stated that the timing requirements of these new and decomposed (sub) systems, naturally, must fulfil the requirements of the original one. The requirements are either reallocated or distributed among the new (sub) systems. The chosen approach depends on how the functionality of the original system is decomposed between the new (sub) systems, because, a constraint follows the timed action whose operation it bounds. Thus, the decomposition process of timed action systems consist of the following phases:

(a) Assort timed actions.

(b) Transform those timed actions into procedures which will act as communication procedures in the new system.

(c) Use the definition of parallel composition inversely to form timed action systems operating in parallel and communicating using the newly created communication procedures.

(d) Rename variables to indicate in which system they are located.

(e) Ensure the temporal correctness.

The above properties cover the functional and temporal behavior of the system. It does not, however cover the aspects related to the power analysis. Naturally, the power aware refinement framework requires that the these new decomposed (sub) systems fulfil the requirements set for the original system. The requirements are distributed between the new (sub) systems according to their functionality. The constraint is targeted to timed action system, and it has a maximum value, which it cannot exceed. Therefore, requirements are set for the new (sub) systems in a way that the maximum allowed value for the constraint in the original system holds. Thus, the final step of the decomposition procedure is:

(f) Ensure the correctness of the area and power constraints.

where the condition (f) requires that the area complexity, selected observation period, and the selected parameters are validated as a part of the power analysis process.

6.3 Chapter Summary

This chapter introduced a method to develop, in a stepwise manner, an abstract system module towards a more concrete one. The presented method lays its foundation into the refinement calculus framework defined for the Action Systems formalism. This framework concentrates on validating functional properties of systems, and therefore physical properties such as time, area, and power are not validated. Timed Action Systems extends this refinement by including methods to constrain and validate timing behavior of the system. The direct extension is possible because the functional behavior of the system is not altered. In a similar manner the presented power aware refinement extends, in turn, the Timed Action Systems refinement framework. The benefits of this approach is that it extends the well known Action Systems refinement framework, and, furthermore, the Timed Action Systems can be adopted directly without any modifications. The presented refinement rules for power, and time allow one to start development of SoC designs from abstract specification and refine them towards more concrete synchronous or asynchronous systems while ensuring their temporal and power characteristics. After introducing the power aware refinement framework for timed action systems, the decomposition of timed action systems is discussed. The decomposition bases its foundation on defining procedures out of actions. These will be used as communication procedures between the newly decomposed action systems The possibility to decompose a system into two or more subsystems or parallel systems is an important property in large scale SoC designs. Observe that refinement and decomposition are methods classically used together for developing systems, and the presented approach fits well with the findings presented in [11]. The following Chapter focus on evaluating the average power from synchronous as well as asynchronous systems.

Chapter 7

Analyzing System Models

So far the thesis has concentrated on introducing a framework to analyze area complexity and power consumption in Timed Action Systems. Moreover a framework to transform an abstract model into more concrete one in a correctness preserving manner is introduced. This chapter presents various system models, and, furthermore, extends the analysis of these models towards area complexity and power consumption. The chapter proceeds as follows: First, a synchronous system model is introduced whose operation is sequenced by a clock signal. Second, an asynchronous system model is introduced where the order of events is sequenced using specific control signals, often called as handshakes. Finally, the power estimation of large on-chip communication channels is discussed.

7.1 Synchronous Systems

Systems that use a periodic synchronization signal as a time reference for data transfers are called *synchronous systems*. Commonly this periodic signal is known as a *clock signal*. The maximum clock frequency of such a system, the clock cycle time, is determined by the slowest component. In an ideal synchronous system, the clock signal is at the same phase at all the points in the circuit. However, in real circuits the clock is never ideal. Nevertheless, in the synchronous models presented below the real clock behavior is abstracted away, and therefore it behaves as an ideal clock.

Clock distribution is challenging design issue nowadays due to the performance requirements. In general, a synchronous system consist of several clock domains rather than one global clock domain. The disadvantage of one global clock is that, for example, timing requirements increases the size of the clock distribution network as it needs more and more clock buffers to deliver the clock signal all around the circuit as close to ideal clock as possible. These timing uncertainties are commonly known as *skew* and *jit-*

Table 7.1: Clock distribution performance metrics

Power [W/GHz]		Area [% *of total area*]			
Pre-Global Stages	Final Driver + Grid	M1-M4 (Devices)	M5	M6	M7
0.75	1.75	0.25	2	3	5

ter [1]. To overcome these timing requirements set for the clock distribution network often leads to an increase in power dissipation. For example, the power dissipation of a clock distribution network can be 30 - 50 % of the overall power consumption of a system [13, 31]. Another significant design metric is area [42, 43, 52]. It affects on power consumption because larger clock tree requires bigger drivers to fulfill the timing requirements. This, in turn, increases the total power dissipation of the system. Therefore, it is often more convenient to divide the system into several specific clock domains, where each domain have its own local clock instead of a single global one. To highlight the performance metrics in a modern microprocessor, consider a scalable (scalability up to 5 GHz) microprocessor manufactured using 90 nm technology [28]. The microprocessor's power consumption and area usage are summarized in Table 7.1, where the power consumption of the clock distribution network is described in the first two columns and the last four columns describe the percentual amount of area consumed by the distribution in different metal layers ($M1 - M7$). Therefore, when creating formal power model for synchronous systems it is not reasonable to abstract all the physical properties away because the clock signal is a major contributor to the overall power consumption of a synchronous system.

7.1.1 Power analysis of synchronous timed actions

The timed synchronous composition in (3.3) on page 37 do not explicitly define a specific clock signal to sequence the operation. The synchronous behavior is ensured by controlling the operation of the synchronous composition by means of new operators. To explicitly create a clock signal one is required to have clock generator(s) and to alter the presented synchronous model. This kind of approach would evoke design challenges such as latches and flip-flops, which are not in the scope of this thesis.

Consider a timed synchronous composition (3.3):

$$S \mathrel{\widehat{=}} A[\![dA]\!] \vee [\![dB]\!]$$

[1] Clock skew describes the spatial variations in the clock signal, whereas the jitter describes the temporal variations. See for instance [69].

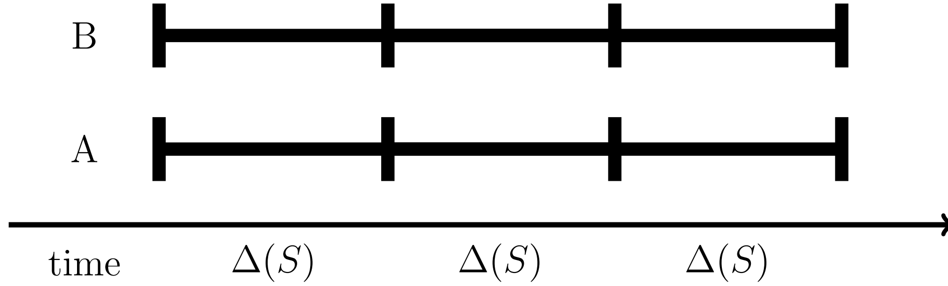


Figure 7.1: An execution sequence of atomic synchronous composition $A \vee B$

The synchronous delay is defined by (3.15), and for the composition S it is of the form:

$$\Delta(S) = \mathbf{Max}(\mathbf{Max}(\Delta(A)), \mathbf{Max}(\Delta(B)))$$

where the synchronous delay $\Delta(S)$ is the delay of the slowest action in the composition. In other words the synchronous delay is the clock frequency, which is selected according to the slowest action in the composition. That is, the actions in the composition forms a clock domain. The illustration of the synchronous composition S in time domain is shown in Fig. 7.1. The figure illustrates the fact that the atomic synchronous composition unifies the execution time of the subactions.

The area complexity model of the synchronous composition S is defined in a similar manner as the compositions presented by (4.7) - (4.9) on page 59. That is, the area complexity of the synchronous composition S is defined by adding the area complexities of actions in the composition together, and it is of the form:

$$C(S) = C(A \vee B) \mathrel{\widehat{=}} C(A) + C(B)$$

where $C(A)$ and $C(B)$ are the area complexities of timed actions A and B, respectively. However, as stated earlier in this section, a synchronous system requires clock distribution network, and even though this network is not modeled at this abstraction level, it should be taken into account when modeling power consumption. Several RTL power models exists where the power dissipation caused by the clock distribution network is included into the logic model of the component. For instance, in [56], the power dissipation caused by the clock was divided between two power models of the power estimation framework: a data path model and a wire model. In the datapath model, the input capacitances caused by the incoming clock signal are included into the component's model whereas the wires required by the clock distribution network are modeled as any other bus, except that it is accessed every clock cycle. On the other hand, Gupta et. al. [44] included the clock power dissipation into the macro model of a particular

component. In this study, however, the selected approach is kept as simple as possible, due to the high abstraction level. That is, the area required for clock distribution is a percentual portion of the total area of the synchronous composition defined by:

$$C(A \vee B) \mathrel{\widehat{=}} C(A) + C(B) + C(clk) \qquad (area(synchronous)) \quad (7.1)$$

where the area of the actions A and B are denoted by $C(A)$ and $C(B)$, respectively. $C(clk)$ denotes the area estimate reserved for the clock tree later on during system development, and it is defined by:

$$C(clk) \mathrel{\widehat{=}} \beta \cdot (C(S)) \qquad (area(clock)) \quad (7.2)$$

where β is coefficient $(0 < \beta < 1)$ which defines the percentual portion of the area required for the clock distribution network. With this model the area of the clock distribution network grows linearly with respect to the area of the synchronous composition. This approach is suitable for this abstraction level, and moreover the model dedicated to asynchronous systems is extended in a minimalistic manner. Thus, problems related to large clock distribution networks are left for future studies.

In traditional synchronous systems, components switch their state at every rising clock edge regardless whether they have data to process or not. Therefore, the dynamic power consumption for the synchronous composition cannot directly adopt (4.22), where the dynamic power of the action composition is calculated for those actions that are enabled and executed during the observation period. That is, actions that are not executed during an observation period are excluded from the dynamic power validation. In synchronous composition, the observation period is replaced by the synchronous delay $\Delta(S)$, and all timed actions in the synchronous composition are executed. Therefore, the average power of the synchronous composition is of the form:

$$\begin{aligned} P_{avg}(S) &\mathrel{\widehat{=}} P_{avg}(S) + P_{stat}(S) \\ &= \frac{E(S)}{\Delta(S)} + C(S) \cdot P^1_{stat} \\ &= \frac{E^1 \cdot C(S)}{\Delta(S)} + C(S) \cdot P^1_{stat} \end{aligned} \qquad (power(synchronous)) \quad (7.3)$$

where $\Delta(S)$ is the synchronous delay (clock cycle) of the composition, $C(S)$ is the area complexity of the composition, and $E(S)$ its energy consumption. Observe that in synchronous composition the energy consumption is calculated for all the actions in the composition.

The non-atomic timed synchronous composition, defined in (3.4) on page 37, is of the form (for the ease of reference):

$$S \mathrel{\widehat{=}} A[\![dA]\!] \bowtie B[\![dB]\!]$$

In the above composition the results are written into an auxiliary variables until the slowest action in the composition has finished its operation after which the actual write variables are updated. This approach allows a designer to gather information on individual action delays. With this information the designer is able to find the critical path, which determines the clock frequency, and possibly make design changes, which, in turn, make the delay of the critical path smaller. This kind of information is not available when using atomic composition. The synchronous delay for non-atomic composition is defined by (3.16), where the value of the delay is determined by the slowest action of the composition. Hence, the actual synchronous delay, the clock frequency, is the delay of the slowest action in atomic- and non-atomic composition. Furthermore, the amount of actions in the composition does not depend on the type of synchronous composition, and therefore one can adopt the area complexity rule (7.1) for the non-atomic composition as well. These time and area estimates are adopted to model average power consumption, and thus the average power of atomic actions, defined by (7.3), is adopted. While there is no difference between the atomic synchronous composition and the non-atomic synchronous composition with respect to the non-atomic composition is used because it can describe low-power design techniques such as clock gating described below.

Gated clocks

The non-atomic timed synchronous composition is extended to model the operation of gated clocks as defined by (3.5) on page 39. The gated non-atomic synchronous composition enables to selectively shut down the clock signal in those portions of the system where active operation is not performed, and thus decrease the power consumption of the system. Consider a gated non-atomic synchronous composition S, defined by (3.5):

$$S \mathrel{\widehat{=}} A[\![dA]\!] \overset{g}{\bowtie} B[\![dB]\!]$$

where the operation of the actions A and B is disabled if there is no data to process. If the clock signal is not disabled, the composition operates exactly in a similar manner than the non-atomic synchronous composition,

presented earlier in this chapter, and therefore its area complexity, synchronous delay, and power consumption are evaluated using (7.1), (3.16), and (7.3), respectively. However, if the gating option is enabled, the power consumption of the composition must be re-evaluated. Consider the equation (7.3):

$$P_{avg}(S) = \frac{E(S)}{\Delta(S)} + C(S) \cdot P^1_{stat}$$

where the synchronous delay $\Delta(S)$ describes the clock cycle time. If the clock signal is disabled, the dynamic power component $\frac{E(S)}{\Delta(S)}$ becomes zero, and thus the average power is the static power loss in the composition S:

$$P_{avg}(S) = C(S) \cdot P^1_{stat}$$

Power gating

The gated synchronous composition (3.5) can be used to describe a power reduction technique called *power gating* [47]. Power gating is used to decrease both the static and the dynamic power consumption of a circuit by shutting down a power supply from inactive parts of the circuit. Naturally, careful design process is required to decide, which entities are valid as power domains. Good candidates are, for example, processors and accelerators. Furthermore, in the presence of memory and register components, a designer must determine how much data to retain or to lose when power is shutdown. Moreover, power domains inside the system should be isolated to prevent unstable data propagation.

In Timed Action Systems context, power gating can be modeled using the gated synchronous composition. To separate the gated clock composition from the power gating, the Boolean variable g is replaced by another Boolean variable gp whose operation is not altered. Adopting this new notation for the Boolean variable, the power gated non-atomic synchronous composition becomes:

$$A_1[\![dA_1]\!] \overset{gp}{\bowtie} A_2[\![dA_2]\!] \mathrel{\widehat{=}} ((\textit{power gated non-atomic synchronous composition}))$$

$$\Big((A^{\bowtie}_{f,1} \;[\!]\; A^{\bowtie}_{k,1}) \;[\!]\; (A^{\bowtie}_{f,2} \;[\!]\; A^{\bowtie}_{k,2}) \;[\!]\; gp \rightarrow T_{\bowtie}\Big) \tag{7.4}$$

$$/\!/ \;\Big(A^{\bowtie}_{s,1} \;[\!]\; A^{\bowtie}_{s,2}\Big)$$

$$/\!/ \; Pt$$

---

where the Boolean variable gp prevents the update of the write variables if it is set to $false$. Otherwise the composition behaves as a non-atomic

synchronous composition. The new notation for the Boolean variable is required to indicate that the power gating is adopted. Naturally, if the power supply is shut down, the clocks are shut down as well, but often they do not form the same entities in the system. For example, a system with a single gated clock may have a subsystem, which is also power gated. Furthermore, at lower abstraction levels the power gating often requires software support, which is not in the scope of this thesis. However, this property motivates to define separate compositions for power and clock gating because software part will be inserted to the power gated composition in the future.

Similarly as with the gated clock, the power gated non-atomic synchronous composition has its own power characteristics. In this case, however, the gating affects on both dynamic and static power consumption of the composition. Consider a power gated synchronous composition S:

$$S \mathrel{\widehat{=}} A[\![dA]\!] \overset{gp}{\bowtie} B[\![dB]\!]$$

where the operation of the actions A and B can be disabled if there is no data to process. The average power of the composition S during operation is defined by:

$$\begin{aligned} P_{avg}(S) \mathrel{\widehat{=}} & \frac{E(S)}{\Delta(S)} + C(S) \cdot P^1_{stat} \\ & \frac{E^1 \cdot C(S)}{\Delta(S)} + C(S) \cdot P^1_{stat} \end{aligned}$$

where the unit energy E^1 and the unit static power consumption P^1_{stat} both include the supply voltage parameter (V_{DD}) as defined by (4.12) and (4.16), respectively. Therefore, if the power supply is shut down (V_{DD} becomes zero), no power is consumed into that particular power domain. Observe that in a bigger system description, parts of the system would be shut down, which, in turn, would reduce the overall power dissipation of the system. In Timed Action System context, for instance, in a hierarchical system construct, one would be able to shut down the power supply from subsystems that are not active during particular time period by adopting this composition.

7.1.2 Synchronous timed action systems

The synchronous system level behavior is modeled using the non-atomic synchronous composition, because power modeling between non-atomic and atomic timed synchronous compositions did not differ. Furthermore, the non-atomic synchronous composition has the gated clock and power gating expansions, which can be used to reduce power consumption at system level, too. Consider a synchronous timed action system $\mathcal{A}$:

sys $\mathcal{A}$ (g_A;) ::
|[
delay
$dA_1 : dA_1 0; dA_2 : dA_2 0;$
variable
l_A;
action
$A_1[\![dA_1]\!] : (aA_2);$
$A_2[\![dA_2]\!] : (aA_2);$
initialization
$g_A, l_A := g_A 0, l_A 0;$
execution
forever do $A_1 \bowtie A_2$ **od**
]|

where the timed actions A_1 and A_2 define the functionality of the system. The clock cycle time of the synchronous system $\mathcal{A}$ is

$$\begin{aligned} T_{ClkA} &= \Delta(A_1 \bowtie A_2) \\ &\overset{(3.16)}{=} \mathbf{Max}\ (\mathbf{Max}(\Delta(A_1)), \mathbf{Max}(\Delta(A_2))) \\ &\overset{(3.9)}{=} \mathbf{Max}\ (dA_1, dA_2) \end{aligned}$$

The area complexity model of the system $\mathcal{A}$ is:

$$\begin{aligned} C(\mathcal{A}) &= C(A_1 \bowtie A_2) \\ &\overset{(7.1)}{=} C(A_1) + C(A_2) + C(Clk) \end{aligned}$$

The power dissipation per clock cycle is:

$$P_{T_{ClkA},avg}(\mathcal{A}) \overset{(4.24)}{=} \frac{E(A_1) + E(A_2)}{T_{ClkA}} + P^1_{stat} \cdot C(\mathcal{A})$$

where the energy consumptions of timed actions in the composition are added together and then divided by the clock cycle.

To highlight the behavior of the power gating, a hierarchical synchronous systems $\mathcal{B}$ is defined:

sys $\mathcal{B}$ (g_B;
exp c;) ::
$|[$
delay
$dB_1 \colon dB_10; dB_2 \colon dB_20;$
$dc \colon dc0;$
variable
$l_B;$
public procedure
$c[\![dc]\!]$: **inout** $trf := data;$
subsystem
$\mathcal{G}$(**imp** c); g_G;
action
$B_1[\![dB_1]\!] \colon (aB_1);$
$B_2[\![dB_2]\!] \colon (aB_2);$
initialization
$g_B, l_B := g_B0, l_B0;$
execution
forever do $B_1 \bowtie B_2 \parallel \mathcal{G}$ **od**
$]|$

where the operation of the system is defined by the actions B_1 and B_2, and its subsystem $\mathcal{G}$. The communication between the system and its subsystem is carried out using the communication procedure c, which is awaited by $\mathcal{B}$ and called by $\mathcal{G}$. The detailed action descriptions are left out to keep the system descriptions simple and to concentrate on highlighting the power gating approach. The subsystem $\mathcal{G}$ is defined by:

sys $\mathcal{G}$ (g_G;
imp c;) ::
$|[$
delay
$dG_1 \colon dG_10; dG_2 \colon dG_20;$
variable
$l_G;$
action
$G_1[\![dG_1]\!] \colon (aG_2);$
$G_2[\![dG_2]\!] \colon (aG_2);$
initialization
$g_G, l_G := g_G0, l_G0;$
execution
forever do $G_1 \overset{gp}{\bowtie} G_2$ **od**
$]|$

where the power gated non-atomic synchronous composition is adopted to

shut down the subsystem. The operation of $\mathcal{G}$ is defined by the actions G_1 and G_2. The clock cycle for the system $\mathcal{B}$ is:

$$\begin{aligned} T_{ClkB} &= ((\Delta(B_1 \bowtie B_2)), \Delta(G_1 \bowtie G_2)) \\ &\overset{(3.16)}{=} \mathbf{Max}\ (\mathbf{Max}(\Delta(B_1)), \mathbf{Max}(\Delta(B_2)), \mathbf{Max}(\Delta(G_1)), \mathbf{Max}(\Delta(G_2))) \\ &= \mathbf{Max}\ ((dB_1), (dB_2), (dG_1), (dG_2)) \end{aligned}$$

where the slowest action determines the system delay (clock cycle).

By adopting the area complexity of systems, defined by (4.19) on page 68, the area complexity of the system $\mathcal{B}$, and its subsystem $\mathcal{G}$ becomes:

$$\begin{aligned} C(\mathcal{B}) &= \sum_{B \in \mathcal{B}} C(B) \\ &= \sum_{B \in \mathcal{B}} C(B) + \sum_{G \in \mathcal{G}} C(G) \end{aligned} \tag{7.5}$$

where the area complexity of the system $\mathcal{B}$ is the sum of the area complexities of the actions in the system $\mathcal{B}$. That is the first line in the above definition contains also the actions in the subsystem $\mathcal{G}$. To adopt the above equation, the area complexity of the system $\mathcal{B}$ is:

$$C(\mathcal{B}) \overset{(7.1)}{=} C(B_1) + C(B_2) + C(c) + C(G_1) + C(G_2) + C(clk_{\mathcal{B}}) + C(clk_{\mathcal{G}})$$

where the area complexities of the clocks $C(clk)$ and $C(clk_{\mathcal{G}})$ are:

$$\begin{aligned} C(clk_{\mathcal{B}}) &\overset{(7.2)}{=} \beta \cdot C(\mathcal{B}) \\ C(clk_{\mathcal{G}}) &\overset{(7.2)}{=} \beta \cdot C(\mathcal{G}) \end{aligned}$$

Adopting the area complexity and the synchronous delay, the average power consumption per clock cycle without power gating is defined by:

$$P_{T_{ClkB},avg}(\mathcal{B}) \overset{(4.24)}{=} \frac{E(\mathcal{B})}{T_{ClkB}} + P^1_{stat} \cdot C(\mathcal{B})$$

where the energy consumption of the system $E(\mathcal{B})$ is calculated using (4.21) on page 69, where the observation period is the synchronous delay, the clock cycle, and therefore the energy consumption becomes:

$$E(\mathcal{B}) = C(\mathcal{B}) \cdot E^1$$

where the energy consumption per clock cycle is the energy consumption of all actions in the system, including the subsystem $\mathcal{G}$.

Consider a case when the subsystem $\mathcal{G}$ is shut down. The area complexity for the power evaluation is:

$$C(\mathcal{B}_{gp}) = C(B_1) + C(B_2) + C(c) + C(clk_{gp})$$

where the area of the subsystem $\mathcal{G}$ is excluded because its power is shutdown, and therefore it does not consume power. Therefore, the area complexity of the clock distribution is of the form:

$$\begin{aligned} C(clk_{gp}) &= \beta \cdot C(\mathcal{B}) \\ &= \beta \cdot (C(B_1) + C(B_2) + C(c)) \end{aligned}$$

where one can see that the area complexity of the clock $C(clk)$ is the area complexity of the clock for system $\mathcal{B}$ defined above ($C(clk_{gp}) = C(clk_{\mathcal{B}})$). Observe that the total area complexity $C(\mathcal{B})$ of the system still includes the subsystem $\mathcal{G}$ as defined above. However, to evaluate the average power during the time the power gating is active, the subsystem $\mathcal{G}$ is excluded because its power is shut down, and therefore it does not consume any power. The power consumption per clock cycle T_{ClkB} when the power gating is active is:

$$\begin{aligned} P_{T_{ClkB},avg}(\mathcal{B}_{gp}) &\overset{(4.24)}{=} \frac{E(\mathcal{B})}{T_{ClkB}} + P^1_{stat} \cdot C(\mathcal{B}_{gp}) \\ &= \frac{E^1 \cdot (C(B_1) + C(B_2) + C(c) + C(clk_{gp}))}{T_{ClkB}} \\ &+ P^1_{stat} \cdot (C(B_1) + C(B_2) + C(c) + C(clk_{gp})) \end{aligned}$$

The power reduction gained using this approach is:

$$P(reduced) \mathrel{\widehat{=}} P_{T_{ClkB},avg}(\mathcal{B}) - P_{T_{ClkB},avg}(\mathcal{B}_{gp})$$

where the reduction in dynamic power consumption:

$$\begin{aligned} P(reduced)_{T_{ClkB},dyn} &= \left(\frac{E^1 \cdot (C(B_1) + C(B_2) + C(c) + C(\mathcal{G}) + C(clk))}{T_{ClkB}} \right) \\ &- \left(\frac{E^1 \cdot (C(B_1) + C(B_2) + C(c) + C(clk_{gp}))}{T_{ClkB}} \right) \\ &= \frac{E^1 \cdot (C(G) - C(clk) - C(clk_{gp}))}{T_{ClkB}} \end{aligned}$$

In a similar manner the reduction in static power consumption can be evaluated by:

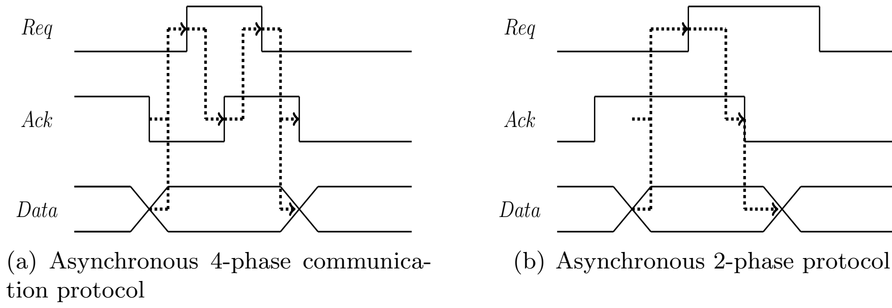
$$P(reduced)_{stat} \mathrel{\widehat{=}} P^1_{stat} \cdot (C(G) - C(clk) - C(clk_{gp}))$$

where the static power reduction is the relative change in the area complexity of the whole system $C(\mathcal{B})$ and the area complexity of the power gated system $C(\mathcal{B})_{gp}$. Furthermore, the change in area complexity due to the power gating causes the reduction in energy consumption of the system, and therefore the decrease of dynamic power consumption is dependent on the change in area complexity values as well.

7.2 Asynchronous Systems

Most digital circuits designed and fabricated today are synchronous. In essence, they are based on two fundamental assumptions that greatly simplify their design: (1) all signals are binary and (2) all components share a common notion of time, as defined by the clock signal distributed throughout the circuit. Asynchronous systems on the other hand also assumes binary signals but *there is no common discrete time*. Instead these circuits use handshaking between their components in order to perform the necessary synchronization, communication, and sequencing of operations. This difference gives asynchronous circuits inherent characteristics to design circuits with very interesting performance parameters in terms of power, timing, and noise [53, 80].

Asynchronous modules communicate with each other via asynchronous communication channels or by procedure based communication described in Sect. 2.6.1. The asynchronous communication channel consists of two control variables *req* and *ack*, and data variable *data*, where the former ones set and close down the communication channel and the latter one is used to transfer data. The setup and closing of communication channels are controlled using certain communication protocols such as *four-phase* and *two-phase* signaling protocols. In the four phase signaling protocol, shown in Fig. 7.2(a), the term 4-phase refers to the number of communication actions: (1) The sender issues data and sets the request high, (2) the receiver absorbs the data and sets acknowledge high, (3) the sender responds by setting the request low (at which point data is no longer valid), and (4) the sender acknowledges this by setting the acknowledgement low. At this point a new communication cycle may begin. The drawback of this protocol is superfluous return-to-zero transitions that cost unnecessary time and energy [80]. This can be avoided by using *two-phase* signaling protocol, shown in Fig. 7.2(b), where the information on the request and acknowledge lines is encoded as signal transitions and there is no difference between $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions. Hence, they both present a signal event. Ideally the 2-phase protocol leads into faster circuits than the 4-phase one, but often the implementation of circuits responding to event is complex, and there is no general answer to the question of which protocol is best [80].



(a) Asynchronous 4-phase communication protocol

(b) Asynchronous 2-phase protocol

Figure 7.2: Asynchronous signaling protocols

The procedure based communication, introduced in Sect. 2.6.1 and whose timing characteristic was discussed in Sect. 3.5.4, uses remote procedures to model communication between systems. This approach allows a designer to hide the communication details (a communication event is considered to be a single atomic action) that are visible onto asynchronous communication channels, and to concentrate on other important modeling issues. Therefore, the communication related design issues, for example, the protocol selection, can be left for lower abstraction levels. Next, consider an example that illustrates the procedure based asynchronous communication.

Example 7.1. An arbiter $\mathcal{A}rb$ grants bus accesses to two master systems, say $\mathcal{A}$ and $\mathcal{B}$. The arbiter is of the form:

sys $\mathcal{A}rb$ (**in** $req_A, req_B : Boolean;$
out $gr_A, gr_B : Boolean;$) ::
|[
delay
$dA_1 : dA_1 0; dA_2 : dA_2 0;$
action
$A_1[\![dA_1]\!] : (req_A \wedge \neg gr_B \rightarrow gr_A := T;$
$[\!] \;\neg req_A \wedge gr_A \rightarrow gr_A := F);$
$A_2[\![dA_2]\!] : (req_B \wedge \neg gr_A \rightarrow gr_B := T;$
$[\!] \;\neg req_B \wedge gr_B \rightarrow gr_B := F);$
execution
forever do $A_1 \;[\!]\; A_2$ **od**
]|

where req_A and req_B are Boolean variables that request an access, and gr_A and gr_B inform a gained access to the shared resource (the grant signals are comparable with the acknowledgement signals used above). The shared resource, the bus, is of the form:

```
sys 𝓑us (   exp Comm; ) ::
|[
  delay
    dComm: [dComm_min, dComm_max];
  public procedure
    Comm⟦dComm⟧(...): (Trf);
  action
    gC⟦dgC⟧: (await Comm);
  execution
    forever do Comm  od
]|
```

where the communication between the masters and the bus is modeled using the public procedure *Comm*. The master systems $\mathcal{A}$ and $\mathcal{B}$ are of form:

```
sys 𝓐 (   out req_A: Boolean;
          in gr_A: Boolean;
          imp Comm; ) ::
|[
  delay
    dReq: dReq0;
    dComm_A: dComm_A0;
    dAck: dAck0;
  action
    Req⟦dReq⟧: (¬gr_A →
                 req_A := T);
    C_A⟦dComm_A⟧: (gr_A →
                 call Comm());
    Ack⟦dAck⟧: (req_A, gr_A := F);
  execution
    forever do Req;C_A;Ack  od
]|
```

```
sys 𝓑 (   out req_B: Boolean;
          in gr_B: Boolean;
          imp Comm; ) ::
|[
  delay
    dReq: dReq0;
    dComm_B: dComm_B0;
    dAck: dAck0;
  action
    Req⟦dReq⟧: (¬gr_B →
                 req_B := T);
    C_B⟦dComm_B⟧: (gr_B →
                 call Comm());
    Ack⟦dAck⟧: (req_B, gr_B := F);
  execution
    forever do Req;C_B;Ack  od
]|
```

where the bus access is first requested from the arbiter using the asynchronous communication channel (the action *Req*). When the access is granted a master calls the communication procedure *Comm* (call takes place inside the timed action C_A). After the data transfer the master releases the request variable (by the timed action Ack).

The non-deterministic choice (in $\mathcal{A}rb$) is used to grant the bus access to either of the masters ($\mathcal{A}$ or $\mathcal{B}$). If the master $\mathcal{A}$ is requesting bus access and the master $\mathcal{B}$ is not or the other way around, the granting decision is trivial. It is trivial also when the access is not requested by either of masters. The non-trivial case takes place when both masters requests bus access at the same time, because the arbiter can grant only one bus access at a time. This situation is illustrated in Fig 7.3, where both of the masters requests the bus access at the same time ($req_A = T$ and $req_B = T$). Because of this both of the actions A and B are enabled. After the delay $dArb$, the bus access is granted to the master $\mathcal{B}$ based on the non-deterministic choice.

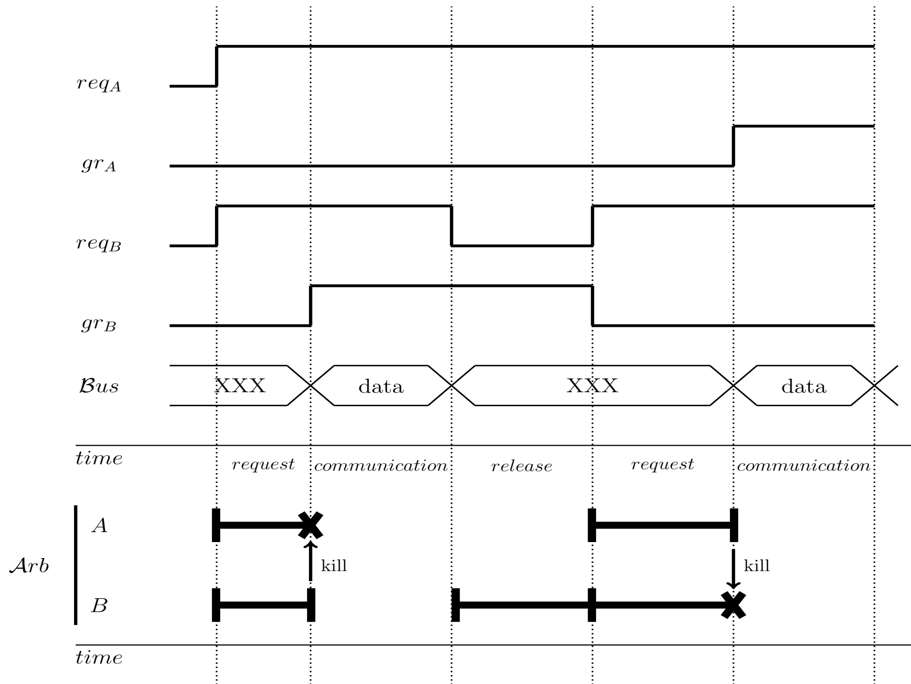


Figure 7.3: Asynchronous operation of systems $\mathcal{A}rb$ and $\mathcal{B}us$

At the same time the action A that is trying to get bus access is killed. The winner of the arbitration, the master $\mathcal{B}$, starts the communication with the bus by executing action C_B which calls the communication procedure $Comm$. After all communication activities are completed, $\mathcal{B}$ releases the request signal, and thus releases the shared resource. A new arbitration cycle may begin as no module has control over the shared resource. This time the module $\mathcal{A}$ is granted.

In terms of power modeling the asynchronous design approach does not pose any new conditions, that is, the area complexity and power modeling framework presented in Chapter. 4 is adopted. Furthermore, the power gating, described in the previous section, can be implemented for asynchronous systems as well. Consider a closed timed action system $\mathcal{S}$, which consist of an arbiter, a bus, and two master components (components are defined above), and, furthermore, the execution clause of $\mathcal{S}$ is of the form

execution

$$\mathcal{A}rb \parallel \mathcal{B}us \parallel \mathcal{A} \parallel \mathcal{B}$$

The area complexity of the system is:

$$C(S) \overset{(4.19)}{=} C(\mathcal{A}) + C(\mathcal{B}) + C(\mathcal{B}us) + C(\mathcal{A}rb)$$

The average power of the system is:

$$P_{T,avg}(\mathcal{S}) \overset{(4.24)}{=} \frac{E_T(S)}{\Delta(T)} + C(\mathcal{S}) \cdot P^1_{stat}$$

where $E_T(\mathcal{S})$ is the energy consumption of the system $\mathcal{S}$ during observation period T. The static power consumption of the system is defined by multiplying the unit static power dissipation P^1_{stat} with the area complexity of the system $C(\mathcal{S})$.

Assume that the master $\mathcal{B}$ is idle for long time periods, and therefore a designer wants to know how much static power is saved if the master $\mathcal{B}$ is shutdown. In asynchronous context, no specific composition is required to model power gating. The area complexity of the power gated the system is denoted by $C(\mathcal{S})$ is:

$$C(\mathcal{S}) \overset{(7.5)}{=} C(\mathcal{A}rb) + C(\mathcal{B}us) + C(\mathcal{A})$$

where the area complexity of the master $\mathcal{B}$ is excluded from the area complexity of the system, and thus the relative static power reduction gained from shutting down the master $\mathcal{B}$ is the area complexity C($\mathcal{B}$) of the master $\mathcal{B}$ multiplied by the unit static power dissipation P^1_{stat}.

To model the average power of such system, the observation period T is selected in a way that the master $\mathcal{B}$ is disabled. Recall the definition (4.20), which states that only those actions which are enabled during observation period is included into the energy estimation. Therefore, adopting the above definition for the observation period, the set of actions S excludes those actions, which are defined by the system $\mathcal{B}$. The average power of gated system $\mathcal{S}$ becomes:

$$P_{T,avg}(\mathcal{S}) \overset{(4.24)}{=} \frac{E_T(S)}{\Delta(T)} + C(\mathcal{S}) \cdot P^1_{stat}$$

where $E_T(\mathcal{S})$ is the energy dissipation during the observation period T. Observe that the actions in master system $\mathcal{B}$ are excluded from the energy consumption evaluation of the system due to the definition of T. Furthermore, they are also excluded from the area complexity calculations, and therefore from the static power estimation, as described above. Thus, shutting down a power domain in the system reduces both the static and the dynamic power dissipation

End of example.

Comparing the area complexity modeling between the asynchronous and synchronous systems, one can see that the area complexity model of the synchronous systems includes the model for clock tree which is not included into the asynchronous one. This is due to the fact that in asynchronous circuits, the control logic is local whereas at synchronous systems it is global. That is, there is no need to build network that is able to drive the timing reference for each component in the circuit. However, this does not imply that the asynchronous systems would be by default smaller. Instead, the local handshaking logic usually causes some amount of area overhead comparing with its synchronous counterpart [80]. On the other hand the lack of a clock distribution network offers potential for low-power and low-noise solutions because the components in the system switch only when they have data to process, and, furthermore, the overall simultaneous switching activity is reduced, which, in turn affects the amount of on-chip noise [53].

7.3 Communication Networks

Interconnects are used in a digital system for communication and for distribute power and clock. Furthermore, interconnects dominate a modern digital system in terms of speed, power, and cost [39, 69]. That is, the time required to drive signals over wires is often the largest factor in determining the cycle time, and a bulk of power in many systems is dissipated to drive these wires. Moreover, an economically achievable wire density is a major influence on the architecture of a system. Therefore, assuming ideal wire models is not suitable approach for power analysis in a communication networks.

In this section, general communication models are discussed. The communication within action systems is assumed to be local communication, and therefore their power dissipation is assumed to be part of the power consumption of the action system. However, the communication between action systems is considered to use longer interconnects and drivers, and therefore a more detailed discussion on modeling power issues is required. The presented model estimates the parasitic properties (capacitance, resistance, inductance) of an interconnect using an area complexity estimate. Timing related aspects are described using the properties of the Timed Action System formalism, and the routing aspects are left intact because of the high abstraction level. That is, the estimation of floorplanning and placement is difficult (or even impossible) above RTL.

Consider the procedure based communication, presented in Sect. 2.6.1 on page 24. It uses procedures to transfer data between systems. The area complexity of the procedure based composition is obtained by calculating the area complexity of those actions and procedures that participate the communication activities [83]. The area complexity of the procedure based

communication, is denoted by *C(Comm)* and defined by:

$$C(Comm) \mathrel{\hat{=}} C(SR) = C(S) + C(p) + C(R) \qquad (\textit{area complexity}) \quad (7.6)$$

where $C(S)$ is the area complexity of the sender action S, $C(p)$ the area complexity of the communication procedure p, and $C(R)$ the area complexity of the receiver action. The area complexities of the actions and the communication procedure are calculated using (4.4) - (4.9) on page 56. The area complexity of the sender $C(S)$ consists the logic needed to send data, and, in a similar manner, the area complexity of the receiver $C(R)$ consists the logic that is required to receive the data that is transferred over the communication channel. The area complexity of the communication procedure p presents the cost of the actual communication channel between the sender and receiver. The delay of the procedure based communication is discussed in 3.5.4 on page 47, and denoted by Δ_{comm}. The above presented area complexity describes the size of a short communication link between two system blocks. However, in modern large scale systems the communication structures are usually more complex, and therefore the power evaluation of such structures is discussed next.

7.3.1 Modeling communication networks

Consider a simple communication channel, henceforward called point-to-point network, whose only responsibility is to call a receiver after a communication call from a sender has arrived. The communication is initiated by calling the communication procedure, which is introduced in and exported by the point-to-point network system. The network system is illustrated in Fig. 7.4, where the communicating parties $\mathcal{S}nd$ and $\mathcal{R}ec$ are similar to those systems described in the definition of the procedure based communication in Sect. 2.6.1 on page 24. The point-to-point network $\mathcal{N}$ is of the form:

```
sys N ( imp p;
        exp n; ) ::
|[
  delay
    dn;
  public procedure
    n[[dn]](in d : data) : call p(d);
  action
    Trf : await n;
  execution
    forever do Trf od
]|
```

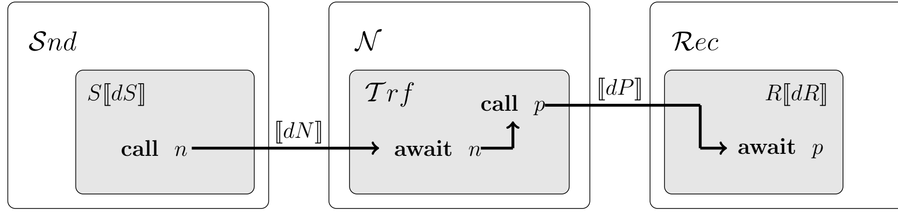


Figure 7.4: Communication through point-to-point network $\mathcal{N}$

The communication activities proceeds as follows: The sender $\mathcal{S}nd$ calls the imported procedure n whose call is awaited by the action $\mathcal{T}rf$ in the point-to-point network $\mathcal{N}$. Then the procedure n calls the communication procedure p introduced in and exported by the receiver system $\mathcal{R}ec$ and imported by the network $\mathcal{N}$. After all the participating actions and communication procedures are free for execution, the communication takes place. The communication activities are performed within a single atomic action, as was done in the procedure based communication presented in Sect. 2.6.1. Thus, **call** / **await** construct of the point-to-point network model $\mathcal{N}$ is:

$$STrfR \mathrel{\hat{=}} S[R[P[d/x]/ \ \textbf{await} \ p]/ \ \textbf{call} \ n(d)]$$

where P denotes the body of the communication procedure p, n is the communication procedure, and S and R are part of the calling and awaiting actions in the sender and receiver systems $\mathcal{S}nd$ and $\mathcal{R}ec$, respectively.

The presented point-to-point network model did not store any data during communication, which makes it unsuitable for larger networks due to congestion and fairness issues. Therefore, a network that stores the sent data onto its own memory element is introduced. In such network the entire communication cycle takes several atomic steps caused by the decomposition of the communication procedure. In general, when using a single atomic communication action, the whole communication medium is reserved before the communication may start. Therefore, the sender cannot continue its operation before the whole communication cycle is completed. In the buffering network, the sender can start a new operation after the network has accepted the sent data item. Furthermore, this approach allows two communication activities at the same time. Naturally, the integrity of the data must be taken care of. The buffering point-to-point network $\mathcal{N}_B$ model is illustrated in Fig. 7.5 and defined by the action system $\mathcal{N}_B$, system description is introduced on next page, where the communication call is initiated by the sender $\mathcal{S}nd$, and awaited by the timed action $\mathcal{R}ec$. After $\mathcal{N}_B$ receives the communication call, it stores data onto its own local buffer variable $ibuf$ whose content is then copied to the output buffer $obuf$ by the action Mov. This, in turn, activates the sending action Snd, which calls the communication procedure p exported by the receiver $\mathcal{R}ec$. When the receiver is ready,

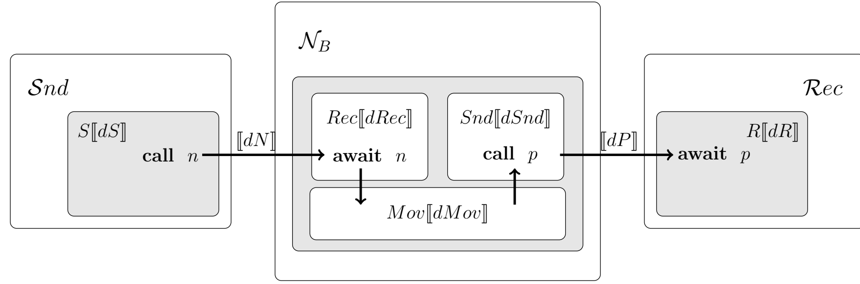


Figure 7.5: Communication through point-to-point network $\mathcal{N}_B$ that stores data item onto its local buffer

the sent data is transferred to its destination.

sys $\mathcal{N}_B$ (**imp** $p($ **in** $x : Data)$; **exp** $n($ **in** $y : Data)$;) ::
|[
delay
$dN : dN0; dRec : dRec0;$
$dMov : dMov0; dSnd : dSnd0;$
variable
$ibuf, obuf : data; g, s : Boolean;$
public procedure
$n[\![dN]\!]$: (**in** $y : data$) : $ibuf := y$;
action
$Rec[\![dRec]\!] : \neg g \rightarrow$ **await** $n; g := T$;
$Mov[\![dMov]\!] : g \wedge \neg s \rightarrow obuf, g, s := ibuf, F, T$;
$Snd[\![dSnd]\!] : s \rightarrow$ **call** $p(obuf); s := F$;
initialization
$ibuf, obuf, g, s := ibuf0, obuf0, F, T$;
execution
forever do Rec [] Mov [] Snd **od**
]|

The communication between the sender and the receiver consist of two atomic communication components: (1) the **call** / **await** construct $SRec$ and (2) the **call** / **await** construct $SndR$:

$$SRec \mathrel{\widehat{=}} S[Rec[P_N[y/ibuf]/\ \textbf{await}\ n]\ \textbf{call}\ n(y)]$$

$$SndR \mathrel{\widehat{=}} Snd[R[P[obuf/d]/\ \textbf{await}\ p]\ \textbf{call}\ p(obuf)]$$

where the first one transfers data between the sender and the network, and the second between the network and the receiver.

A communication delay for the point-to-point network and for the buffered network is:

$$\Delta(\mathcal{N}) \mathrel{\widehat{=}} \Delta STrfR \qquad (\textit{delay(network)}) \tag{7.7}$$
$$= \Delta(S) + \Delta(n) + \Delta(p) + \Delta(R)$$

$$\Delta(\mathcal{N}_B) \mathrel{\widehat{=}} \Delta(SndR) + \Delta(Mov) + \Delta(RecS) \qquad (\textit{delay(network, buffer)})$$
$$= \Delta(S) + \Delta(Snd) + \Delta(Mov) \tag{7.8}$$
$$+ \Delta(Rec)\Delta(p) + \Delta(n) + \Delta(R)$$

and the area complexities are calculated in a similar manner as the area complexity of the direct communication defined earlier in this section. Thus, the area complexities for the above presented network models are:

$$C(\mathcal{N}) \mathrel{\widehat{=}} C(STrfR) \qquad (\textit{area(network)})$$
$$= C(S) + C(n) + C(p) + C(R) \tag{7.9}$$

$$C(\mathcal{N}_B) \mathrel{\widehat{=}} C(SndR) + C(Mov) + C(SRec) \qquad (\textit{area(network, buffer)})$$
$$= C(S) + C(Snd) + C(n) \tag{7.10}$$
$$+ C(Mov) + C(p) + C(R) + C(Rec)$$

Consider the buffered network $\mathcal{N}_B$, where a single communication cycle can be described using the computation path $\mathcal{CP}_N(SndR, RecS)$. The area complexity of the computation path equals the area complexity of the buffered network presented above:

$$C(\mathcal{CP}_N) = C(\mathcal{N}_B))$$

To evaluate the power consumption of a single data transfer from sender to receiver one adopts both the delay and area information of the system. The communication delay is the delay $\Delta(N)_B$, and therefore the observation period T is $T = [SRec.st, SndR.ft]$, where the start time is the time when the data transfer is initialized by the sender $\mathcal{S}nd$, and the finish time is the time when the receiver system $\mathcal{R}ec$ has received the transferred data. The power consumption during the observation period is calculated using (4.24) on page 72:

$$P_{T,avg}(\mathcal{N}_B) \mathrel{\widehat{=}} P_{T,dyn}(\mathcal{N}_B) + P_{stat}(\mathcal{N}_B) \qquad (\textit{power(network,buffer)}$$
$$= \frac{E_T(\mathcal{N}_B)}{\Delta(T)} + P^1_{stat} \cdot C(\mathcal{N}_B)$$

Adopting the buffered point-to-point network model $\mathcal{N}_B$, a general communication network model for systems with k sender(s) and l receiver(s) $(k, l > 0)$, shown in Fig. 7.6, is given below.

```
sys N_B^x (    imp w(1..l)(in x : Data); exp n(1..k)(in y : Data; Dst);
             in Addr : [k, l];  ) ::
|[
  delay
    dN : [dN_min, dN_max], dRec : dRec0;
    dMov : dMov0, dSnd : dSnd0;
  variable
    ibuf[k], obuf[l] : Data;
    dstbuf[k] : Addr;
    ifull[k], ofull[l] : Boolean;
  public procedure
    n(i)⟦dN⟧ : (in dst : Addr, x : data) : (ibuf[i], dstbuf[i] := d, dst);
  action
    Rec(i)⟦dRec⟧ : ¬ifull[i] → await n(i)(x, dst); ifull[i] := T;
    Mov(i)⟦dMov⟧ : ¬ofull[dstbuf[i]] ∧ ifull[i] →
     obuf[dstbuf[i]], ifull[i], ofull[dstbuf[i]] := ibuf[i], F, T;
    Snd(j)⟦dSnd⟧ : ofull(j) → call w(j)(obuf[j]); ofull[j] := F;
  initialization
    ibuf[k], obuf[l], dstbuf[k] := ibuf0[k], obuf0[l], dstbuf0[l];
    ifull[k], ofull[l] := F, F;
  execution
     forever do [ [] 1 ≤ i ≤ k : Rec(i)  []  Mov(i)]
                  []  [ [] 1 ≤ j ≤ l : Snd(j)]  od
]|
```

where the *dstbuf[k]* is the buffer onto which the destination address is stored, *ifull* and *ofull* indicate whether the input and output buffers, *ibuf* and *obuf*, respectively, are booked up or not.

As might be expected, the communication delay and the area complexity in the above model equals with the buffered network communication delay and area complexity presented in (7.8) and (7.10), respectively. This is due to the fact that their communication schemes are equivalent. The only difference is the amount of senders and receivers in the network. The area complexity and delay models are further elaborated by considering the distance between the communicating parties to obtain more accurate delay and area complexity estimates.

Communication delay. Large cover a vast area of silicon, hence their communication delay depends greatly on the distance of the communication parties. This, on the other hand, means that to connect the communica-

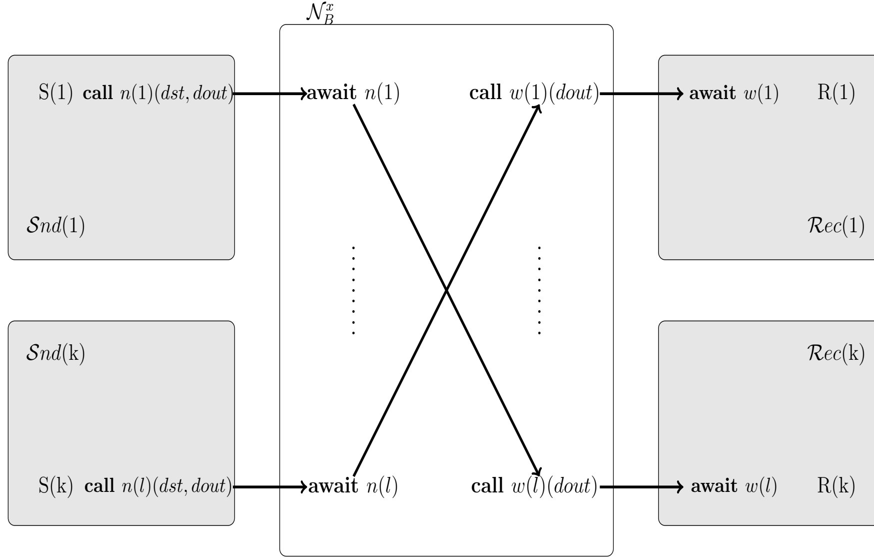


Figure 7.6: General communication network

tion parties apart from each other it is required to have long wires, which, naturally cause long communication delays. In [86], a unit wire delay Δ^1_{wire} is defined whose multiple defines the distance of the parties [87]. Its value can be adjusted at lower abstraction levels when more detailed information of the system is available. To find a factor for the unit wire delay, one need a tool to estimate the distance of the communication parties. As depicted in Fig.7.6 and modeled in the network model $\mathcal{N}^x_B$ each of the communicating parties have an index, also referred as its address. Assuming that the address of the modules, in some respect, correlates with the location of the modules in the system, the addresses can be used to calculate relative distance, denoted by rd, between two modules $\mathcal{S}_i$ and $\mathcal{R}_j$:

$$rd \mathrel{\widehat{=}} |i - j| \qquad (\textit{relative distance(general network)}) \quad (7.11)$$

The above definition cannot be considered as a general approximation of the distance between two communicating parties, as there exists a wide variety of network topologies. Best approximates can be obtained from regular networks such as a 2-D mesh or a ring based communication networks. By multiplying the relative distance by the unit wire delay, one obtain an approximation of the communication delay for the high-level network model:

$$\Delta(\mathcal{N}^x_B) \mathrel{\widehat{=}} rd \cdot \Delta^1_{wire} + \Delta(\mathcal{N}_B) \qquad (\textit{delay(general network)}) \quad (7.12)$$

Area complexity. In a large communication network, the area required by the wires has significant impact on system's performance. In the general network model, shown in Fig. 7.6, the length of the wires between the communication parties is estimated using the relative distance in (7.11):

$$l(wire)^{i,j} \mathrel{\widehat{=}} rd \cdot l^1_{wire}, \quad i \neq j$$
$$l(wire)^{i,j} \mathrel{\widehat{=}} l^1_{wire}, \qquad i = j$$

where l^1_{wire} is a unit wire length, which is a measure of the shortest wire in the network. The length of the wires, other than unit lengths ($i = j$), is calculated by multiplying the unit wire length with the relative distance. The area complexity of a wire in the general network model for is defined by:

$$C(wire)^{i,j} \mathrel{\widehat{=}} l(wire)^{i,j} \cdot w(wire) \qquad (\textit{wire area(general network)}) \quad (7.13)$$

where $l(wire)^{i,j}$ and $w(wire)$ denotes the length and width of the wire, respectively. In general both of theses parameters are hard to evaluate without any information from the target technology. Therefore, for the abstract network model, the length of the wire is defined with the aid of relative distance as stated above, and the width of the wire is introduced as a technology related parameter. Adopting (7.13), the area complexity of the general network model is defined by:

$$C(\mathcal{N}^x_B) \mathrel{\widehat{=}} k \cdot (C(SRec) + C(Mov)) + l \cdot C(SndR) + C(wire)^{i,j} \quad (7.14)$$
$$(\textit{area(general network)})$$

where k and l are the number of receivers and senders in the network, respectively ($k, l > 0$).

Power consumption. To model the average power consumption of the general communication network $\mathcal{N}^x_B$, an observation period T is defined using (4.20) on page 68. During the observation period T, the system executes a set $\mathcal{N}^x_B$ of timed actions. The set $\mathcal{N}^x_B$ consists of n ($n > 1$) communication cycles where it is assumed that a communication cycle is defined using the following computation path by:

$$\mathcal{CP}(SndR, SRec) = \langle SRec(i), Mov(i), SndR(j) \rangle$$

where the communication procedure $n(i)$ is included into the action $SRec(i)$. The average power during the observation period T is calculated using (4.24) on page 72:

$$P_{T,avg}(\mathcal{N}_B^x) = P_{T,dyn}(\mathcal{N}_B^x) + P_{stat}(\mathcal{N}_B^x) \qquad (\textit{average power(general network)}$$

where $P_{dyn}(\mathcal{N}_B^x)$ is the dynamic power consumption of the network and $P_{stat}(\mathcal{N}_B^x)$ is the static power consumption. The dynamic power consumption is defined by (4.22) on page 71:

$$P_{T,dyn}(\mathcal{N}_B^x) = \frac{E_T(\mathcal{N}_B^x)}{\Delta(T)} \qquad (\textit{dynamic power(general network)}$$

where the energy consumption of the network is defined by:

$$E_T(\mathcal{N}_B^x) \mathrel{\widehat{=}} E_T(\mathcal{N}_B^x) + E(wire^{i,j}) \qquad (\textit{energy(general network)} \qquad (7.15)$$

where the energy consumption of the system $\mathcal{N}_B^x$ is calculated using (4.21) and the energy consumed by the wire is defined by:

$$E(wire)^{i,j} \mathrel{\widehat{=}} E_{wire}^1 \cdot C(wire)^{i,j} \qquad (\textit{energy(wire)}) \quad (7.16)$$

where the energy consumption of an unit wire E_{wire}^1 is multiplied by the area complexity of the wire (7.14). Observe that the difference between (4.21) and (7.15) is that the former does not include any model for communication channels whereas the latter one does. The static power consumption of the general network system is defined by:

$$P_{stat}(\mathcal{N}_B^x) \mathrel{\widehat{=}} C(\mathcal{N}_B^x) \cdot P_{stat}^1 + C(wire)^{(i,j)} \cdot P_{stat}^1 \qquad (7.17)$$
$$(\textit{static power(general network)})$$

where the first term calculates the static power consumption caused by the actions in the system $\mathcal{N}_B^x$, and the second one evaluates the static power consumption of the interconnects.

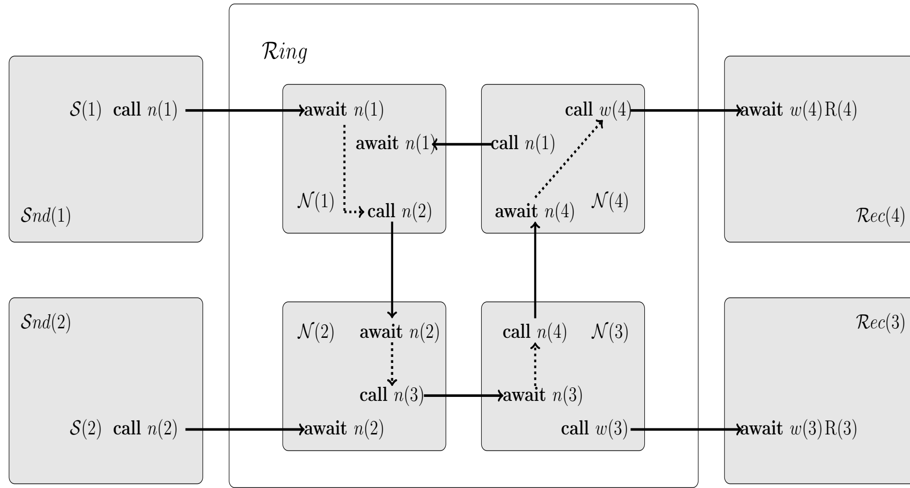


Figure 7.7: Ring based communication and an example communication sequence between $\mathcal{S}_1$ and $\mathcal{R}_4$

7.3.2 Ring based communication network

A ring based communication network, shown in Fig 7.7, is based on parallel operating transceivers, called network nodes $\mathcal{N}(i)$, each of which connected to one host system and two network nodes. All data items not directed to a network node host system are forwarded by calling the adjacent node. The network itself is defined by a system $\mathcal{R}ing$:

$$
\begin{aligned}
&\textbf{sys } \mathcal{R}ing\ (\ \textbf{imp } w(1..l)(\ \textbf{in } x : Data);\\
&\qquad\qquad \textbf{exp } n(1..l)(\ \textbf{in }\ : Dst : Addr);\\
&\qquad\qquad \textbf{in } [l : Addr];\ \ ) ::\\
&|[\\
&\ \textbf{public procedure}\\
&\quad n(i)[\![dN]\!](\textbf{in}Dst : Addr; x : Data) : ibuf, dstbuf := d, dst;\\
&\ \textbf{subsystem}\\
&\quad \mathcal{N}ode(i) :\\
&\qquad \mathcal{N}(i)\Big(\textbf{imp } w(i)(\ \textbf{in } x : Data)\\
&\qquad\qquad n((i+1) \textbf{ mod } l)(\ \textbf{in } Dst : Addr; d : Data)\\
&\qquad\qquad n(i)(\textbf{in } Dst : Addr; x : Data)\Big);\\
&\ \textbf{execution}\\
&\qquad \Big[\ \|\ 1 \le i \le l \colon \mathcal{N}ode(i)\Big]\\
&]|
\end{aligned}
$$

where the communication procedures $w(1 \ldots l)$ and $n(1 \ldots l)$ map the network nodes together. Observe that in the above model the communication

between network nodes is performed within the system, and only the communication between network node and its host system is visible outside the system. The network node is defined by:

$$
\begin{array}{l}
\textbf{sys } \mathcal{N}(i) \ (\ \textbf{imp } w(i)(\textbf{ in } x : Data); \\
\qquad\qquad\qquad n((i+1) \textbf{ mod } l)(\textbf{ in } dst : Addr; x : Data); \\
\qquad\qquad \textbf{imp } n(i)(\textbf{in } x : Data; Dst : Addr); \ ) :: \\
|[\\
\quad \textbf{delay} \\
\quad\quad dN : [dN_{min}, dN_{max}], dRec : dRec0; \\
\quad\quad dMov : dMov0, dSnd : dSnd0; \\
\quad \textbf{variable} \\
\quad\quad ibuf, obuf, sysbuf : Data; \\
\quad\quad dstbuf : Addr; \\
\quad\quad ifull, ofull, sysfull : Bool; \\
\quad \textbf{public procedure} \\
\quad\quad n(i)[\![dN]\!] : (\textbf{in } dst : Addr; y : Data) : (ibuf[i], dstbuf[i] := x, dst); \\
\quad \textbf{action} \\
\quad\quad Rec[\![dRec]\!] : \big(\neg ifull[i] \rightarrow \textbf{await } n(i); ifull[i] := T\big); \\
\quad\quad Mov[\![dMov]\!] : ifull[i] \rightarrow \\
\quad\quad\quad ofull \wedge dstbuf = i \rightarrow \\
\quad\quad\quad \big(obuf, ofull := ibuf, T) \\
\quad\quad\quad\quad [\!] \ \neg sysfull \wedge dstbuf \neq i \rightarrow (onbuf, sysfull := ibuf, T\big) \\
\quad\quad\quad ; ifull[i] := F; \\
\quad\quad Snd[\![dSnd]\!] : ofull \rightarrow (\textbf{call } w(i)(obuf); ofull := F) \ [\!] \ sysfull \rightarrow \\
\quad\quad\quad (\textbf{call } n((i+1) \textbf{ mod } l)(dstbuf, sysbuf); sysfull := F); \\
\quad \textbf{initialization} \\
\quad\quad ibuf, obuf, dstbuf, sysbuf := ibuf0, obuf0, dstbuf0, sysfull0; \\
\quad\quad ifull, ofull, sysfull := F, F, F; \\
\quad \textbf{execution} \\
\quad\quad\quad \textbf{forever do } Rec \quad [\!] \quad Mov \ [\!] \quad Snd \ \ \textbf{od} \\
]|
\end{array}
$$

where the communication proceeds as follows: The *Rec* action awaits the communication call and copies the sent data onto its local buffer *ibuf* and the destination address to the buffer *dstbuf*. The data is transferred towards the next network node ($dst \neq i$) and copied into the output buffer *obuf* or towards the synchronous system module ($dst = i$). If the data is transferred to the host system, it is copied onto output buffer *sysbus*.

Applying the notation introduced in the preceding section, the delay caused by the described ring based communication network is:

$$rd(i) \cdot \Delta(\mathcal{N}_B ring) = rd(i) \cdot \big(\Delta^1_{wire} + \Delta(\mathcal{N}_B)\big)$$

where the relative distance $rd(i)$ is defined by:

$$rd(i) \mathrel{\widehat{=}} |i-j| mod\ l \qquad (\textit{relative distance(ring network)}) \quad (7.18)$$

where l denotes the number of IP blocks. The relative distance defines the number of network nodes (excluding the one to which the requesting synchronous island is connected to) between the sender and receiver (a hop count). The communication delay of the ring architecture becomes:

$$\Delta(ring) \mathrel{\widehat{=}} rd(i) \cdot \Delta(\mathcal{N}_B ring) \qquad (\textit{delay(ring network)}) \quad (7.19)$$

Area complexity and power consumption. The area complexity of the ring based network is evaluated according the guidelines given for the general network model and it is defined by:

$$\begin{aligned} C(\mathcal{R}ing) &= C(\mathcal{N}ode(i)) = i \cdot C(\mathcal{N}(i)) \qquad (\textit{area(ring network)}) \\ &= i \cdot (C(Rec) + C(Mov) + C(Snd)) + C(wire)^{(i,j)} \qquad (7.20) \end{aligned}$$

where the area complexity of the wires $C(wire)^{(i,j)}$ is calculated using (7.13). Observe that the relative distance in (7.13) is replaced by (7.18).

To model average power dissipation, an observation period is defined in a similar manner as for a general network. The observation period T consist of n ($n > 1$) communication cycles, defined by computation path:

$$\mathcal{CP}(Rec, Snd) \mathrel{\widehat{=}} \langle Rec, Mov, Snd \rangle$$

The n communication cycles form a set $\mathcal{R}ing$ of timed actions. Adopting the above presented area complexity model, (7.16), and (4.24), the average power of the ring based network is defined by:

$$P_{T,avg}(\mathcal{R}ing)) \mathrel{\widehat{=}} P_{T,dyn}(\mathcal{R}ing) + P_{stat}(\mathcal{R}ing) \ (\textit{average power(ring network)}) \quad (7.21)$$

where the dynamic power consumption is:

$$\begin{aligned} P_{T,dyn}(\mathcal{R}ing) &= P_{T,dyn}(\mathcal{N}(i)) \\ &\overset{(7.15)}{=} \frac{E_T(\mathcal{N}(i))}{\Delta(T)} + \frac{E(wire)}{\Delta(T)} \end{aligned}$$

where the former calculates the dynamic power of i network nodes ($i > 0$), and the latter one calculates the dynamic power of the wires in the network. The static power consumption is:

$$P_{stat}(\mathcal{R}ing) \overset{(7.17)}{=} C(\mathcal{N}(i)) \cdot P^1_{stat} + C(wire)^{i,j} \cdot P^1_{stat}$$

Example 7.2. Consider an example call (denoted by dash arrows) depicted in Fig. 7.7, where the sender $\mathcal{S}(1)$ communicates with the receiver $\mathcal{R}(4)$. A computation path for such a communication activity is as follows:

$$\begin{aligned}\mathcal{CP}(\mathcal{S}(1), \mathcal{R}(4)) &= \langle \mathcal{S}(1), \mathcal{N}(i), \mathcal{R}(4) \rangle \\ &= \mathcal{S}(1), \mathcal{N}(1), \mathcal{N}(2), \mathcal{N}(3), \mathcal{N}(4), \mathcal{R}(4)\end{aligned}$$

where the relative distance between the source and destination can be calculated using (7.18):

$$rd(i) = |4 - 1| mod\ 4 = 3$$

and the communication delay is:

$$\Delta(\mathcal{CP}(\mathcal{S}(1), \mathcal{R}(4))) = 3 \cdot \Delta(\mathcal{N}_b ring)$$

The area complexity of the computation path is defined by calculating the area complexity of the systems in the computation path together:

$$\begin{aligned}C(\mathcal{CP}(\mathcal{S}(1), \mathcal{R}(4))) &\overset{(7.20)}{=} C(\mathcal{S}(1)) + C(\mathcal{N}(1)) + C(\mathcal{N}(2)) \\ &+ C(\mathcal{N}(3)) + C(\mathcal{N}(4)) + C(\mathcal{R}(4)) + C(wire)^{i,j}\end{aligned}$$

where $C(wire)^{i,j}$ is the area complexity of the wires in the computation path:

$$\begin{aligned}C(wire)^{i,j} &\overset{(7.13)}{=} l_{wire} \cdot w_{wire} = rd \cdot l^1_{wire} \cdot w_{wire} \\ &= 3 \cdot l^1_{wire} \cdot w_{wire}\end{aligned}$$

where rd is the relative distance, l^1_{wire} the unit wire length, and w_{wire} the width of the wire. To model average power consumption consider an observation period $\Delta(T) = 3 \cdot \Delta_{N_b ring}$, which is the delay of one communication cycle as stated above. Thus, the average power becomes:

$$\begin{aligned}P_{T,avg}\big(\mathcal{CP}(\mathcal{S}(1), \mathcal{R}(4))\big) &= \frac{E_T\big(\mathcal{CP}(\mathcal{S}(1), \mathcal{R}(4))\big)}{\Delta(T)} + \frac{E_{wire}}{\Delta(T)} \\ &+ P^1_{stat} \cdot C\big(\mathcal{CP}(\mathcal{S}(1), \mathcal{R}(4))\big)\end{aligned}$$

End of example.

7.4 Chapter Summary

Various system models were described in this chapter, and, furthermore, their demands to the power analysis were emphasized. At first, synchronous systems were discussed. Their operation is paced with a clock signal, which is significant contributor to the total power consumption of system. Therefore, area complexity and average power models were introduced for synchronous systems, where the size of the clock distribution network is taken into account. In addition to synchronous composition, the synchronous composition with gated clocks is analyzed in terms of area complexity and power consumption. The synchronous gated clock composition was also adopted to model power gating because it prevents the execution of actions in the composition if gating is enabled. Thus, the power gating and its effect to power consumption was also analyzed.

From synchronous systems the chapter proceeds to asynchronous systems, which do not have the clock signal, and therefore the power analysis can be done as discussed in Chapter 4. On-chip communication networks were then illustrated and analyzed in terms of area and power consumption. The communication network modeling started with a direct point-to-point link, after which a simple network model was presented, which included a local buffer where data could be stored. This model was further extended into two larger network models. The longer the communication link is the more it poses demands on the model. In this thesis, the distance between two links were analyzed using relative distance. All the presented communication models were analyzed in terms of time, area complexity and power. The presented network structures illustrated the emphasis on physical parameters, such as wire length, in timing and power analysis.

Chapter 8

Experiment

The power aware modeling techniques presented in previous sections are used to model and analyze the area and power dissipation of a co-processor system targeted to acceleration of Java programs. The presented model is a simplified author adaptation of an existing co-processor system described in [75]. First, the background of the co-processor is highlighted after which the formal model is introduced. The analysis part is categorized into four sections, which covers the most power hungry areas of the system. Finally, a framework for multicore system is introduced and analyzed.

8.1 Background

Java is emerging as a standard execution environment for portable devices, such as mobile phones and Personal Digital Assistants (PDAs), due to its security, portability, mobility, and network support. Java execution techniques, such as Just In Time compilation (JIT), have limitations in both storage space and computation power, which makes them unsuitable in mobile application domain. However, consumers demand faster systems with more capabilities and longer battery lives. Therefore, several methodologies exists to reduce the overhead in Java execution. These overheads are due to the fact that Java applications are not written, nor compiled, for any given hardware device. They are written and compiled for a *Java Virtual Machine (JVM)*, where Java applications are executed by emulating JVM on the host system. Clearly, extra emulation layers causes overheads, but they also provide opportunities for improvements like increased security and platform independent programming modes. More interestingly, a power management can be included into the virtual layer, where slower and more energy effective routines are selected when battery is running low.

Java code is first compiled into a bytecode, which is then run on JVM. JVM acts as an interpreter from the bytecode to a native microcode or uses

JIT to obtain the same result a bit faster at the cost of increased memory usage. This software only approach is inefficient in terms of power consumption and execution time. Mainly these problems rise from the fact that executing one Java bytecode instruction requires several native instructions [75].

Another approach would be to use a full standalone Java processor in the system. The problems with this approach is that JVM has an extra layer between hardware and software, and it often is not suitable for low level hardware control, device drivers, and hard real time parts of a given embedded system. In other words, this approach would be likely to require a general purpose processor for the low level accesses, which, in turn, would make the system integration both difficult and expensive.

One way to solve problems is to take the best parts from the full software and the full hardware approaches. In [75], an asynchronous co-processor architecture (REALJava) for an efficient Java execution is introduced. At the moment, the implementation of this architecture is synchronous and running on Field Programmable Gate Array (FPGA). The existing architecture provides an easy integration with existing systems, with the execution speed of the hardware standalone JVM [1]. Furthermore, the resulting system needs no special concerns related to accessing I/O devices and other services since they are produced by the general purpose host processor. In forthcoming sections, the modeling of the hardware parts of the co-processor is discussed.

8.2 The Formal Model of the Co-Processor

In this section, a high level formal model for REALJava [75] co-processor is introduced. The presented model is based on an asynchronous architecture, and, furthermore, it concentrates on the HW parts of Java execution because software modeling in power and time aware context is out of the scope of this thesis.

8.2.1 Preliminaries

The co-processor system, denoted by $\mathcal{S}$, consist of two parts: the host system and the co-processor system, shown in Fig. 8.1, where the host system consist of Central Processing Unit (CPU) and system memory. The execution of Java methods can take place in the host system, in the co-processor system, or in both. Before introducing the formal description of the system, consider an execution sequence shown in Fig. 8.2, where the execution of Java method needs both the host system and the co-processor system. First, $\mathcal{H}ostsystem$ invokes the Java method and transfers it to $\mathcal{C}o - Proc$

[1] Results of current REALJava implementation can be found in [8]

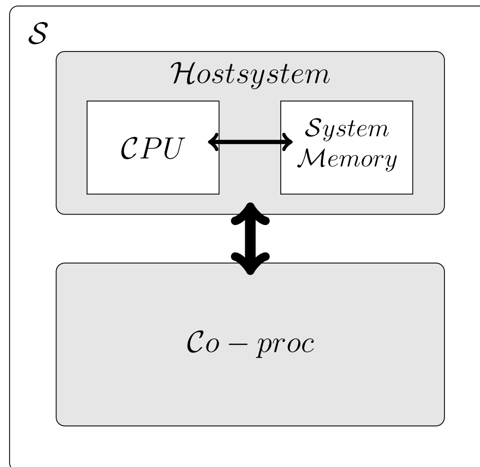


Figure 8.1: Illustration of the initial system construct $\mathcal{S}$

for execution. $\mathcal{C}o - Proc$ continues the execution until it detects an instruction that is not implemented by $\mathcal{C}o - Proc$, and therefore it shifts the method execution to $\mathcal{H}ostsystem$ using Interrupt Request (IRQ). Unknown instructions are processed by $\mathcal{H}ostsystem$ after which the method execution is transferred back to $\mathcal{C}o - Proc$. Finally, $\mathcal{C}o - Proc$ notifies that it has completed the method execution to $\mathcal{H}ostsystem$.

The presented model consists of co-processor with its environment (host system), which is assumed to provide necessary stimulus for the co-processor system, but the functionality of the environment is not described. The timed action system $\mathcal{S}$ is a closed system, which encapsulates two subsystems as shown in Fig. 8.1. $\mathcal{S}$ is defined by:

sys $\mathcal{S}$ ( ) ::
|[
type
method: *Java Method*;
subsystem
$\mathcal{C}o - Proc$: $\big(\mathcal{C}o - Proc(\textbf{imp}\ invoke(\textbf{in}\ d : Method));$
$\textbf{exp}\ IRQ(\textbf{out}\ x : Data);$
$\textbf{in}\ halt\colon Boolean\big);$
$\mathcal{H}ostsystem$: $\big(\mathcal{H}ostsystem(\textbf{exp}\ invoke(\textbf{in}\ d : Method));$
$\textbf{imp}\ IRQ(\textbf{out}\ x : Data);$
$\textbf{out}\ halt\colon Boolean\big);$
execution
$\mathcal{C}o - Proc \parallel \mathcal{H}ostsystem$
]|

where an abstract interface of both the host system and the co-processor

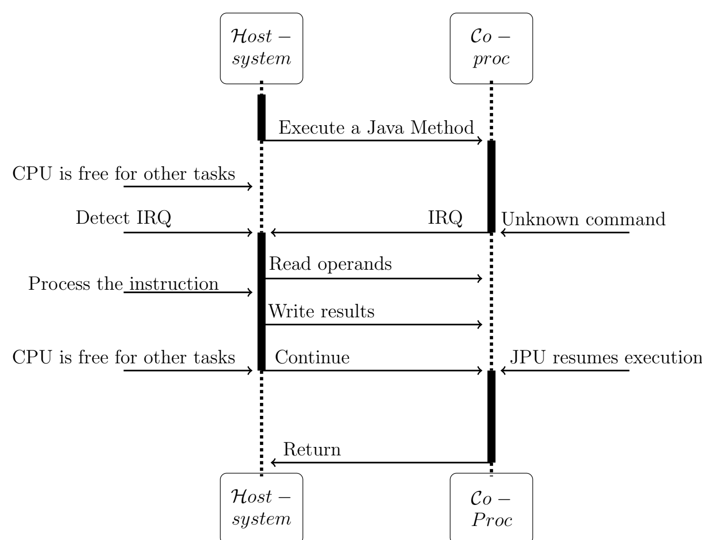


Figure 8.2: Example execution sequence between $\mathcal{H}ostsystem$ and $\mathcal{C}o{-}Proc$

system is described. The subsystem's interface includes the communication between the subsystems. The public procedure *invoke* is exported by $\mathcal{H}ostsystem$, and imported by $\mathcal{C}o - Proc$ because $\mathcal{H}ostsystem$ initiates the communication. Allthough $\mathcal{H}ostsystem$ is not defined in this thesis, the definition of *invoke* procedure is given:

$$invoke[\![dInvoke]\!] : (\mathbf{in}\ d : Method) : \ local_{mem} := local_{mem} \cup \{d\}$$

where the type of the variable d is *Java method*, which is written on the co-processors local memory area, $local_{mem}$. More detailed definition of the local memory area is given during the definition of the co-processor system. The second communication procedure IRQ is exported by the co-processor system and imported by the host system because the co-processor system initiates an IRQ. Furthermore, a Boolean variable *halt* is defined, which allows the host system to stop the co-processor system.

8.2.2 Co-processor system

The co-processor system $\mathcal{C}o - Proc$ consists of three subsystem blocks: a local memory ($\mathcal{M}em$), a communication unit ($\mathcal{C}omm$), and an execution unit ($\mathcal{E}xec$) as shown in Fig. 8.3. In Fig. 8.2 illustrated an example Java method execution, where the communication is initiated by $\mathcal{H}ostsystem$ using procedure *invoke*, which, in turn, writes the Java method to the co-processors local memory area. In general, local memory consists of several local elements such as stack and local variables on data side, and bytecode segments

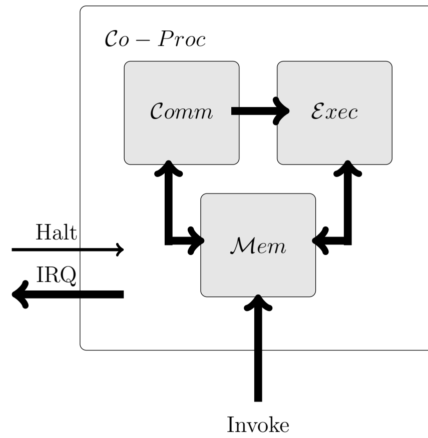


Figure 8.3: Block diagram of the system $\mathcal{C}o - Proc$

of the methods on the instruction side. The virtual machine needs to do some preparations before the co-processor can execute bytecode segment, for example, it has to check that the current thread's stack frame and the current method are loaded into the co-processors local memory area. These preparations are done by $\mathcal{H}ostsystem$, and therefore it is assumed that the host system handle all the necessary memory updates needed for the bytecode execution. Once this is done, the co-processor system enables the subsystem $\mathcal{C}omm$, which reads the physical memory addresses from the local memory after which an instruction is fetched from the instruction cache. The instruction is then transferred to the execution unit defined by the subsystem $\mathcal{E}xec$, which executes the instruction. The result of the operation is stored onto the local memory area, for instance, onto the top of the stack. The co-processor system is defined by $\mathcal{C}o - Proc$, where the action $J1$ awaits a Java method from $\mathcal{H}ostsystem$ after which the method is written onto the local memory area $local_{mem}$, which is located in the subsystem $\mathcal{M}em$. Once the Java method is written onto the local memory, the subsystem $\mathcal{C}omm$ is activated by $J2$. The subsystem $\mathcal{C}omm$ reads the physical addresses from the local memory area, after which it searches the requisite instruction from the instruction cache. In case of a cache miss, the instruction is fetched from the local memory of the co-processor system after which $\mathcal{C}omm$ indicates to the $\mathcal{C}o - Proc$ system that there is instruction to be executed. The action $J3$ enables the execution unit $\mathcal{E}xec$. Once the execution of the instruction is completed, the co-processor system returns to the *idle* state (action $J4$) and is ready to either continue method execution or to accept new method from $\mathcal{H}ostsystem$. The action $J5$ returns the system into *idle* state if the host system issues *halt* command to the co-processor system.

$\textbf{sys}\ \mathcal{C}o - Proc\ (\quad \textbf{imp}\ invoke(\textbf{in}\ d\colon Method);$

$\qquad\qquad \textbf{exp}\ IRQ(\textbf{out}\ x\colon Data);$

$\qquad\qquad \textbf{in}\ halt\colon Boolean;$

$\qquad\qquad \textbf{inout}\ req_{mem}, req_{Comm}, req_{exec}\colon Boolean;\ \ ) ::$

$|[$

$\textbf{type}$

$\quad Status\colon \{idle, op, addr, exec, irq_{request}\};$

$\quad Memory\colon \textbf{set of}\ Data;$

$\quad Instruction\colon Java\ Instruction;$

$\quad Address\colon \textbf{record}(icache\ address, Instruction);$

$\textbf{delay}$

$\quad dJ1, dJ2, DJ3, DJ4\colon dJ10, dJ20, dJ30, dJ40;$

$\quad dJ5\colon dJ50, dJ6\colon dJ60;$

$\quad dIRQ\colon [dIRQ_{min}, dIRQ_{max}];$

$\textbf{variable}$

$\quad sys_{stat}\colon Status;$

$\quad irq_{data}\colon Data;$

$\textbf{public procedure}$

$\quad IRQ[\![dIRQ]\!](\textbf{out}\ x : data)\colon x\ to\ \mathcal{H}ostsystem;$

$\textbf{subsystem}$

$\quad \mathcal{E}xec\colon \mathcal{E}xec\big(\textbf{imp}\ d_{alu}(\textbf{in}\ x : Data); \textbf{exp}\ d_{exec}(\textbf{in}\ x_1, x_2\colon Data);$

$\qquad\qquad \textbf{in}\ instr\colon Instruction\big);$

$\quad \mathcal{C}omm\colon \mathcal{C}omm\big(\textbf{imp}\ getAddr(\textbf{out}\ x\colon Address);$

$\qquad\qquad \textbf{imp}\ getInstr(\textbf{in}\ x\colon Address;$

$\qquad\qquad \textbf{out}\ instr\colon Instruction);$

$\qquad\qquad \textbf{inout}\ req_{comm}, req_{instr}\colon Boolean;$

$\quad \mathcal{M}em\colon \mathcal{M}em\big(\textbf{imp}\ invoke\colon (\textbf{in}\ d\colon Method);$

$\qquad\qquad \textbf{exp}\ getAddr\colon (\textbf{out}\ x\colon Address);$

$\qquad\qquad \textbf{exp}\ getInstr\colon (\textbf{in}\ x\colon Address; \textbf{out}\ y\colon Instruction)$

$\qquad\qquad \textbf{exp}\ flush\colon (\textbf{in}\ x\colon Memory);$

$\qquad\qquad \textbf{inout}\ req_{instr}\colon Boolean\big);$

$\textbf{action}$

$\quad J1[\![dJ1]\!]\colon sys_{stat} = idle \wedge reg_{mem} \rightarrow sys_{stat} := op;$

$\quad J2[\![dJ2]\!]\colon sys_{stat} = op \wedge \neg req_{mem} \rightarrow req_{comm} := T; sys_{stat} := addr;$

$\quad J3[\![dJ3]\!]\colon sys_{stat} = addr \wedge \neg req_{comm} \rightarrow sys_{stat} := exec; req_{exec} := T;$

$\quad J4[\![dJ4]\!]\colon sys_{stat} = exec \wedge \neg req_{exec} \rightarrow sys_{stat} := idle;$

$\quad J5[\![dJ5]\!]\colon halt = T \rightarrow sys_{stat} := idle; halt := F;$

$\quad J6[\![dJ6]\!]\colon sys_{stat} = irq_{request} \rightarrow \textbf{await}\ IRQ(irq_{data}); sys_{stat} := idle;$

$\textbf{initialization}$

$\quad sys_{stat} := idle; irq_{data} := irq_{data}0;$

$\quad req_{mem}, req_{comm}, req_{exec}, halt := F;$

$\textbf{execution}$

$\quad \textbf{forever do}\ [\ [\!]\ 1 \leq i \leq 6 : J_i]\ \ \textbf{od} \parallel \mathcal{E}xec \parallel \mathcal{C}omm \parallel \mathcal{M}em$

$]|$

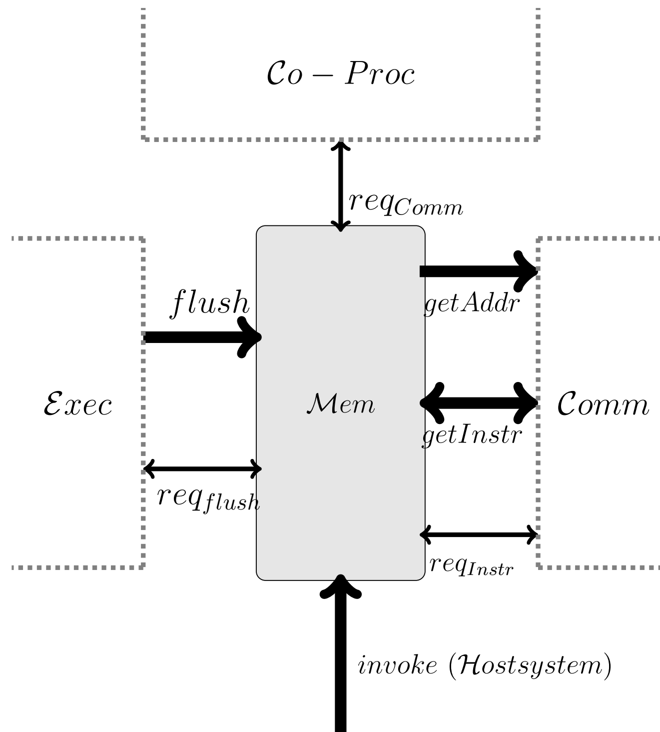


Figure 8.4: Subsystem $\mathcal{M}em$ and its interface towards other system blocks

The action $J6$ transfers IRQ to the host system using the communication procedure IRQ. In this study, the IRQ is not raised because it is assumed that all Java methods are executable by the co-processor system. The co-processor system returns to *idle* state ($J1$), after IRQ, because after the host system has handled the IRQ, the necessary information for the co-processor to proceed the execution is written onto the local memory area. In REALJava and IRQ can be a notification to the host system that the co-processor system cannot handle certain instruction(s), and the execution of that instruction(s) is then transferred to the host system. Although the host system is not modeled in this study, the *halt* and IRQ were modeled to make the comparison between the presented model and the HW parts of $REALJava$ more accurate. Observe that the $\mathcal{C}o - Proc$ system has four global variables ($halt, req_{mem}, req_{comm}$ and req_{exec}), whose values can be altered by $\mathcal{H}ostsystem$ and subsystems $\mathcal{E}xec$, $\mathcal{C}omm$, and $\mathcal{M}em$.

Memory block. The memory block ($\mathcal{M}em$) contains the local memory area of the co-processor system. $\mathcal{M}em$ and its communication with $\mathcal{C}o{-}Proc$ and the subsystems $\mathcal{E}xec$ and $\mathcal{C}omm$ is illustrated in Fig. 8.4, where the di-

rection of data in communication channels is indicated using arrows: $\leftrightarrow$ is used to denote the biput (**inout**) communication channels, where data is transferred in either direction. Otherwise the direction of the arrow head defines the direction of data transfer (**in** or **out**). Furthermore, thick arrows are used to denote those data transfers where the width of the communication channel is over one bit (data buses), and the thinner arrows denote the control variables (one bit Boolean variables). The $\mathcal{M}em$ is of the form:

sys $\mathcal{M}em$ (**imp** $Invoke$: (**in** d: $Method$);
  exp $getAddr$: (**out** x: $Address$);
  exp $getInstr$: (**in** x: $Address$; **out** y: $Instruction$);
  exp $Flush$: (**in** x: $Memory$);
  inout req_{instr}, req_{flush}: $Boolean$;) ::

$|[$
delay
$dM1, dM2, dM3, dM4\colon dM10, dM20, dM30, dM40;$
$dAddr\colon [dAddr_{min}, dAddr_{max}];$
$dInstr\colon [dInstr_{min}, dInstr_{max}];$
$dF\colon [dF_{min}, dF_{max}];$
variable
$local_{mem}\colon Memory;$
public procedure
$getAddr[\![dAddr]\!]$(**out** $x\colon Address)\colon x \in local_{mem};$
$getInstr[\![dInstr]\!]$(**in** $x\colon Address$; **out** $y\colon Instruction)\colon$
$y := local_{mem}(x);$
$Flush[\![dF]\!]$(**in** $x\colon Memory)\colon local_{mem} := local_{mem} \cup x;$
action
$M1[\![dM1]\!]\colon req_{mem} \rightarrow$ **call** $Invoke(d); req_{mem} := F;$
$M2[\![dM2]\!]\colon req_{comm} \rightarrow$ **await** $getAddr(I_{addr}); req_{instr} := T;$
$M3[\![dM3]\!]\colon req_{comm} \wedge \neg req_{instr} \rightarrow$ **await** $getInstr(I_{addr}, Instr);$
$M4[\![dM4]\!]\colon req_{flush} \rightarrow$ **await** $flush(dcache); req_{flush} := F;$
initialization
$req_{instr}, req_{flush} := F;$
$local_{mem} := local_{mem}0;$
execution
forever do $M1$ $[\!]$ $M2$ $[\!]$ $M3$ $[\!]$ $M4$ **od**
$]|$

where the action $M1$ calls *invoke* procedure when $\mathcal{H}ostsystem$ sets req_{mem} to *true*. Once the Java method is written to the local memory, $\mathcal{M}em$ sets the req_{mem} to *false*, which indicates to $\mathcal{C}o - Proc$ that the execution of the Java method may begin. As mentioned earlier, in REALJava, the local memory consists of several local elements such as stack and local variables on data side, and bytecode segments of the methods on the instruction

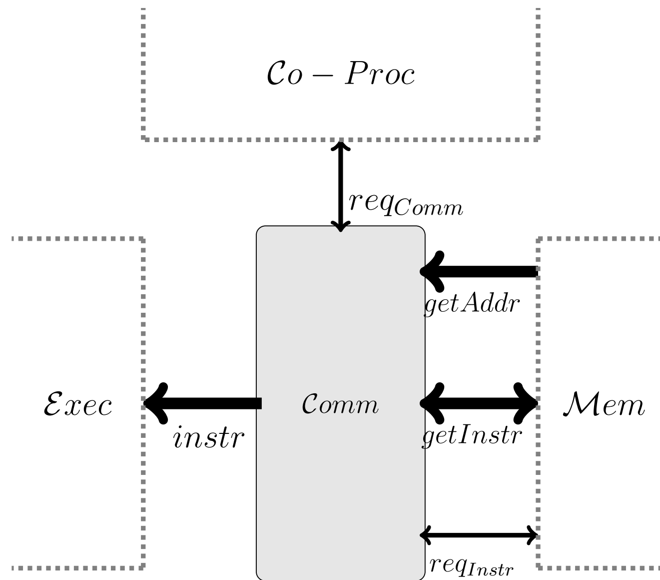


Figure 8.5: $\mathcal{C}omm$ and its interface towards other system blocks

side. In this thesis, however, the stack and cache memories are included into those system descriptions that uses them. For example, an instruction cache is located into the communication module ($\mathcal{C}omm$), which handles the instruction fetching. The local memory, on the other hand, is define to be an abstract memory area, which contains the necessary information to execute Java methods. Therefore, it is assumed that $\mathcal{H}ostsystem$ has performed the following operations before the Java method is executed: The instruction fetching buffer, located in the co-processor local memory area, generates the physical memory addresses to the local memory. These addresses are used to fetch requisite instructions, and the addresses are generated in the internal registers of the co-processor. In general, these registers contain configuration data, and, furthermore, they are used to control the execution. In this context, the definition of these registers is not given because the registers are controlled by $\mathcal{H}ostsystem$. That is, the presented model assumes that the required addresses for execution can be read directly from the fetching buffer, which is located in the local memory area of the co-processor. The action $M2$ searches the instruction cache address from the local memory area using the public procedure $getAddr$. If the instruction is not located in the instruction cache, define by $\mathcal{C}omm$, the instruction is fetched from the memory by the public procedure $getInstr$. The action $A4$ performs the data flush operation, if requested by subsystem $\mathcal{E}xec$.

Communication block. The communication block ($\mathcal{C}omm$), and its interaction with $\mathcal{C}o-Proc$ and subsystems $\mathcal{E}xec$ and $\mathcal{M}em$ is illustrated in Fig. 8.5. The notations in the figure are similar with those in Fig. 8.4, and $\mathcal{C}omm$ is defined by:

$$
\begin{array}{l}
\textbf{sys}\ \mathcal{C}omm\ (\quad \textbf{imp}\ getAddr(\textbf{out}\ x\colon Address);\\
\qquad\qquad\quad \textbf{imp}\ getInstr(\textbf{in}\ x\colon Address; \textbf{out}\ y\colon Instruction);\\
\qquad\qquad\quad \textbf{inout}\ req_{comm}, req_{instr}\colon Boolean;\\
\qquad\qquad\quad \textbf{out}\ instr\colon Instruction;\ )::\\
|[\\
\quad \textbf{type}\\
\quad\quad I_{mem}\colon \textbf{set of}\ Address;\\
\quad \textbf{delay}\\
\quad\quad dC1\colon dC10; dC2\colon dC20; dC3\colon dC30;\\
\quad \textbf{variable}\\
\quad\quad icache\colon I_{mem};\\
\quad\quad I_{Addr}\colon Address;\\
\quad \textbf{action}\\
\quad\quad C1[\![dC1]\!]\colon req_{comm} \rightarrow \textbf{call}\ getAddr(I_{addr});\\
\quad\quad C2[\![dC2]\!]\colon req_{instr} \wedge I_{addr} \in icache \rightarrow\\
\qquad\qquad\quad instr := icache(I_{addr}); req_{comm}, req_{instr} := F;\\
\quad\quad C3[\![dC3]\!]\colon \neg req_{instr} \wedge I_{addr} \notin icache \rightarrow\\
\qquad\qquad\quad \textbf{call}\ getInstr(I_{Addr}, instr); req_{comm} := F;\\
\quad \textbf{initialization}\\
\quad\quad I_{addr} := I_{addr}0;\\
\quad\quad icache := icache0;\\
\quad\quad instr := instr0;\\
\quad \textbf{execution}\\
\quad\quad \textbf{forever do}\ C1\ [\!]\ C2\ [\!]\ C3\ \ \textbf{od}\\
]|
\end{array}
$$

where the action $C1$ reads the physical memory address from the local memory (instruction fetching buffer) using the public procedure *getAddr*, which returns the instruction cache address (from subsystem $\mathcal{M}em$). After receiving the instruction cache address, the action $C2$ searches the corresponding instruction from the instruction cache. If the address is found from the cache, then the action indicates to $\mathcal{C}o-Proc$ that there is an instruction to be executed by setting the control variable req_{comm} to *false*. The instruction cache is defined to be of type *set of Address*, where the Address is **record** containing both *icache address* and *Instruction*, as defined in $\mathcal{C}o-Proc$. The *icache address* field defines the instruction cache address and the *Instruction* field is the Java instruction. This structure is an abstraction from the cache model defined in [82]. If the address is not found in the instruction cache,

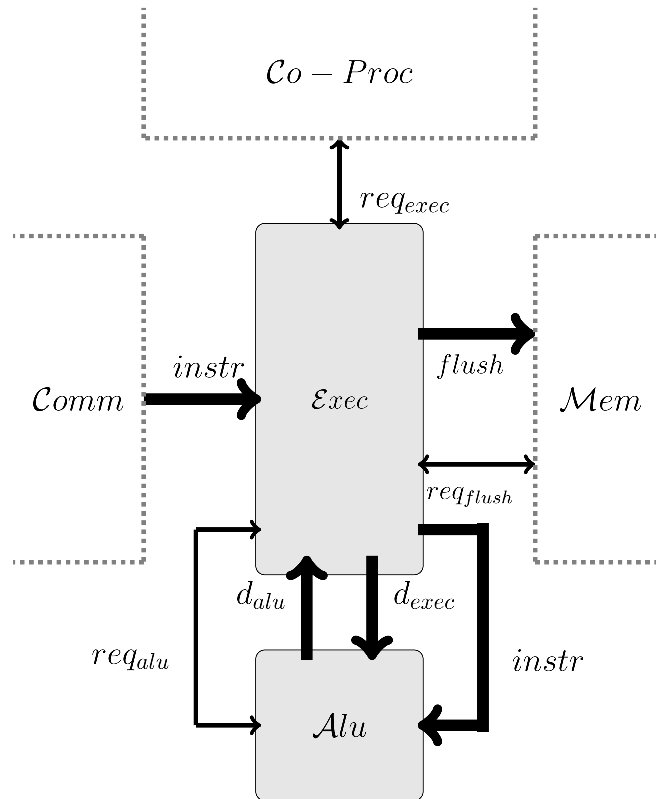


Figure 8.6: Block diagram of the subsystem $\mathcal{E}xec$

it is requested ($C3$) from the local memory located using the public procedure *GetInstr*, which receives the physical address as a parameter, and returns the instruction ($instr$). Finally, the action $C3$ notifies the $\mathcal{C}o - Proc$ system that the instruction fetched from the memory is ready by setting the variable req_{comm} to *false*. This indicates to the $\mathcal{C}omm$ system that an executable instruction is available.

Execution unit. The execution unit consists of the arithmetic logic unit, which is modeled as a subsystem $\mathcal{A}LU$, and two memory elements: stack and data cache. $\mathcal{E}xec$, and its communication with other subsystems of $\mathcal{C}o - Proc$ is illustrated in Fig. 8.6, and defined by $\mathcal{E}xec$ (next page). The operation of $\mathcal{E}xec$ proceeds as follows: $\mathcal{C}o - Proc$ activates its subsystem $\mathcal{E}xec$ by setting the variable req_{exec} to *true*, which indicates that there is an instruction to be executed. Furthermore, as stated earlier, it is required that the necessary operand and register data are written by $\mathcal{H}ostsystem$ into the co-processors local memory area before bytecode execution.

```
sys ℰxec (  imp d_{Alu}⟦dalu⟧(out x : Data);
            imp d_{Flush}⟦dF⟧(in x : Memory);
            exp d_{exec}⟦dexec⟧(in x_1, x_2 : Data);
            in instr : Instruction;
            inout req_{exec}, req_{alu}, req_{flush} : Boolean; ) ::
|[
  constant
    size : stack(max);
  delay
    dexec : [dexec_{min}, dexec_{max}]; dFlush : [dF_{min}, dF_{max}];
    dE1 : dE10; dE2 : dE20; dE3 : dE30; dE4 : dE40; dE5 : dE50;
    dP1 : [dP1_{min}, dP1_{max}]; dP2 : [dP2_{min}, dP2_{max}];
  variable
    stack, dcache : Memory;
    op_1, op_2, buf1, buf2 : Data;
    sz : Integer;
  public procedure
    d_{exec}(in x_1, x_2 : Data) : buf1, buf2 := x_1, x_2;
  private procedure
    Pop⟦dP1⟧(out y_1, y_2 : Data) : y_1, y_2 := y'_1, y'_2.(y'_1, y'_2 ∈ stack);
    Push⟦dP2⟧(in x : Data) : x := stack ∪ {x};
  subsystem
    𝒜LU : 𝒜LU(imp d_{exec}⟦dexec⟧(in x_1, x_2 : Data)
              exp d_{alu}⟦dalu⟧(out x : Data);
              in instr : Instruction; inout req_{alu} : Boolean;
  action
    E1⟦dE1⟧ : req_{exec} ∧ ¬req_{alu} → Pop(op_1, op_2);
              await d_{exec}(op_1, op_2); req_{alu} := T;
    E2⟦dE2⟧ : ¬req_{alu} → call d_{alu}(res); wr_{result} := T;
    E3⟦dE3⟧ : wr_{result} ∧ sz < size → Push(res);
              sz := sz'.(sz' = sz' + 1);
    E4⟦dE4⟧ : sz = size → dcache := dcache ∪ stack;
              req_{flush} := T; call flush(dcache); sz := 1;
    E5⟦dE5⟧ : wr_{result} ∧ sz ≠ size →
              wr_{result}, req_{alu}, req_{exec} := F;
  initialization
    req_{alu}, req_{exec}, wr_{result} := F;
    stack := stack0; dcache := dcache0;
    op_1, op_2, rbuf := op_10, op_20, rbuf0;
    sz := 1;
  execution
    forever do E1 [] E2 [] E3 [] E4 [] E5 od || 𝒜LU
]|
```

The action $E1$ reads the operands from the stack. The read operation from *stack* is modeled using the private procedure *Pop*, which returns data operands $Op1, Op2$ from the stack. After data is read from *stack*, the action $E1$ calls the public procedure d_{exec}, which transfers data from $\mathcal{E}xec$ to $\mathcal{A}LU$. The operation of $\mathcal{A}LU$ is defined later on in this section. The system $\mathcal{E}xec$ waits the result of the computation, which is transferred using the public procedure d_{alu} (defined in and exported by $\mathcal{A}LU$ and imported by $\mathcal{E}xec$). Furthermore, the Boolean variable wr_{result} is set *true* after which the result is written onto *stack*. The timed action $E4$ monitors the state of the stack by increasing the counter *sz* every time data is written onto *stack*. Once the stack is full ($E5$), the stack's data is written into the data cache (*dcache*), and then flushed into local memory (action $M4$ in $\mathcal{M}em$), by calling the public procedure $flush$.

Arithmetic logic unit. The Java bytecode instruction set in the original co-processor structure [75] includes 201 instructions, which are executed either in hardware or in software or in both. This abstract system description concentrate only on discussing a small subset of these instructions, and, furthermore, all of the instructions are performed in hardware. Table 8.1 shows the selected instructions: the mnemonics are directly adopted into the system description of $\mathcal{A}LU$. However, these are not listed in the system description. In addition, operations like memory read and write are implemented using procedure calls to maintain the high abstraction level of the system description as well as the length of the co-processor system in control. The subsystem $\mathcal{A}LU$ is defined by: where whenever $\mathcal{A}LU$ is in an idle state ($sel = idle$) it awaits that $\mathcal{E}xec$ sets the Boolean variable req_{alu} to *true* after which it calls the procedure d_{exec}. The procedure transfers operands from the stack to $\mathcal{A}LU$. At the same time the instruction variable *instr* is stored onto the select variable *sel* to perform the desired operation. The arithmetic and logic operations are defined by the actions $A2 - A8$, and, furthermore, they set the select *sel* variable into a *send* state, which enables the action $A9$, where the communication procedure d_{alu} is awaited, which transfers data back to $\mathcal{E}xec$.

$\textbf{sys}\ \mathcal{A}LU\ (\quad \textbf{imp}\ d_{exec}(\textbf{in}\ x_1, x_2 \colon Data);$
$\quad\textbf{exp}\ d_{alu}(\textbf{out}\ x : Data);$
$\quad\textbf{in}\ instr \colon Instruction;$
$\quad\textbf{inout}\ req_{alu} \colon Boolean;\ ) ::$

$|[$

delay

$dA1 \colon dA10; dA2 \colon dA20; dA3 \colon dA30;$
$dA4 \colon dA40; dA5 \colon dA50; dA6 \colon dA60;$
$dA7 \colon dA70; dA8 \colon dA80; dA9 \colon dA90;$
$dalu \colon [dA_{min}, dA_{max}];$

variable

$i_1, i_2, r, res \colon Data;$
$sel \colon Instruction;$

public procedure

$d_{alu}[\![dalu]\!](\textbf{out}\ x\ :\ Data) \colon res := res'.(res' = x);$

action

$A1[\![dA1]\!] \colon sel = idle \land req_{alu} \to \textbf{call}\ d_{exec}; i_1, i_2 := buf1, buf2;$
$\qquad sel := instr;$
$A2[\![dA2]\!] \colon sel = iadd \to r := r'.(r' = i_1 + i_2); sel := send;$
$A3[\![dA3]\!] \colon sel = isub \to r := r'.(r' = i_1 - i_2); sel := send;$
$A4[\![dA4]\!] \colon sel = imult \to r := r'.(r' = i_1 * i_2); sel := send;$
$A5[\![dA5]\!] \colon sel = idiv \to r := r'.(r' = i_1 / i_2); sel := send;$
$A6[\![dA6]\!] \colon sel = iand \to r := r'.(r' = i_1 \land i_2); sel := send;$
$A7[\![dA7]\!] \colon sel = ior \to r := r'.(r' = i_1 \lor i_2); sel := send;$
$A8[\![dA8]\!] \colon sel = ixor \to r := r'.(r' = i_1 \oplus i_2); sel := send;$
$A9[\![dA9]\!] \colon sel = send \to \textbf{await}\ d_{alu}(res); req_{alu} := F; sel := idle;$

initialization

$sel := idle;$
$i_1, i_2, r, res := i_10, i_20, r0, res0;$

execution

$\textbf{forever do}\ [\ [\!]\ 1 \le i \le 9 \colon A_i]\ \ \textbf{od}$

$]|$

8.3 Analysis

The area and power analysis presented in this section is divided into four subsections: first, the memory blocks of the co-processor system are discussed and evaluated; second, the arithmetic logic unit is modeled and analyzed in terms of area and power consumption; third, the communication structures inside the co-processor systems are discussed. Finally, the whole co-processor system is analyzed in terms area consumption and power dissipation.

Table 8.1: Operations performed in $\mathcal{ALU}$.

Mnemonic	Description
iadd	Add integer
isub	Subtract integer
imult	Multiply integer
idiv	Divide integer
iand	Compute integer bit wise AND
ior	Compute integer bit wise OR
ixor	Compute integer bit wise XOR
idle	Wait new instruction
send	Send data

8.3.1 Analyzing memory components

Power consumption in a memory chip consists of three major sources: the memory cell array, the decoders (row, column, block), and the periphery [69]. The power consumption can be divided into two parts: active and inactive. The former is due to memory accesses and the latter is the power dissipation caused by data retention in inactive parts of the memory chip. In general, the power dissipation of the memory chip is dominated by the cell array. As should be expected, the power dissipation of the memory is proportional to the size of the memory (n, m). Dividing the memory into subarrays, and keeping n and m small is essential to keep power within bounds. This approach is effective only if the standby dissipation of inactive memory modules is substantially lower. That is, the active power dissipation of peripheral circuits is small compared with the other components. The stand-by power of peripheral circuits can be high requiring that peripheral circuits such as sense amplifiers are turned off when not in action. The decoder charging current is also negligibly small in modern Random Access Memorys (RAMs) especially if care is taken that only one out of n or m nodes is charged at every cycle [69].

Power estimation models targeted to logic circuits, such as the model presented in Chapter 4, give rather optimistic power estimates for memory blocks, especially when dealing with large memory units. Therefore, a separate model for memory components is often included into high-level power estimation environments. For instance, the model presented in [54] evaluates the power dissipation of the memory cell array using:

$$P_{memcell} \widehat{=} \frac{2^k}{2} \left(c_{int} \cdot l_{column} + 2^{n-k} C_{tr} \right) \cdot V_{dd} \cdot V_{swing} \cdot f$$

where 2^k is the number of cells in a row, C_{int} is a wire capacitance per unit

length, l_{column} is a memory column length, 2^{n-k} is the number of cells in a column, C_{tr} is a minimum size drain capacitance, and V_{swing} is a bit line voltage swing.

In this thesis, the memory issues arises from the power consumption analyses of the Java co-processor, which contains a local on-chip memory unit consisting of both data and method areas. Furthermore, the co-processor contains a stack memory and cache-memories for both data and instructions. The power consumption profiles presented in [50] indicate that around 70% of the energy consumed during the execution of Java applications is used on memory accesses. This estimate includes the whole system $\mathcal{S}$, presented in Fig. 8.1, which consists of $\mathcal{H}ostsystem$ and $\mathcal{C}o - Proc$. However, the system modeled in this study consists only of the co-processor and, furthermore, only a small portion of the instruction set is implemented. Therefore, comparing with $REALJava$ the size of the presented co-processor model is small, and one is able to use of the area complexity model introduced in Sect. 4.1 on page 52 to modeling the memory components. Next the memory accesses area analyzed in terms of power consumption while the size of the memory blocks will be discussed in Sect. 8.3.4.

Instruction cache. The delay of the instruction fetching is:

$$\begin{aligned}\Delta(C1 \ [] \ C2 \ [] \ C3) &\overset{(3.11)}{=} \Delta(C1\langle getAddr\rangle) + \Delta(C2) + \Delta(C3\langle getInstr\rangle) \\ &\overset{(3.13)}{=} \Delta(C1) + \Delta(getAddr) + \Delta(C2) \\ &+ \Delta(getInstr) + \Delta(C3)\end{aligned}$$

where the delay of the procedure calls are defined to be non-deterministic in the system description due to the fact that the delay of the instruction fetching can vary depending on whether the instruction is fetched from the cache or from the main memory. In general, the invoked Java methods are under 512 bytes in length, and research carried out in [70] showed that the size of the methods rarely exceeds 40 bytes. From this property follows that most of the instructions are located in the cache instead of main memory, and, furthermore, the address space required to generate these instruction address can be smaller. Therefore the variability in memory access times will be smaller. Thus, in this context, the delays of memory accesses are modeled using deterministic delays.

The area complexity of instruction fetching is:

$$\begin{aligned}C(C1 \ [] \ C2 \ [] \ C3) &\overset{(4.7)}{=} C(C1)\langle getAddr\rangle + C(C2) + C(C3)\langle getInstr\rangle \\ &\overset{(4.6)}{=} C(C1) + C(getAddr) + C(C2) \\ &+ C(C3) + C(getInstr)\end{aligned}$$

where $C(C1)$, $C(2)$, and $C(C3)$ are the area complexities of the actions $C1$, $C2$ and $C3$, respectively. To calculate the average power of a single instructions fetch, one defines two computation paths: $\mathcal{CP}_1 = (C1, C3)$ and $\mathcal{CP}_2 = (C1, C2)$, where the first one describes the situation where the instruction is fetched from the local memory, and the latter one describes the situation where the instruction is in the instruction cache. The computation path delays are:

$$\Delta(\mathcal{CP}_1) = \Delta(C1, C3) = \Delta(C1 \;[]\; C2 \;[]\; C3)$$
$$\Delta(\mathcal{CP}_2) = \Delta({}^*C1, {}^*C3)$$

where the latter delay does not include the delay of $C3$ and the delay of the procedure $getInstr$. Thus, the delay of the second computation path is:

$$\begin{aligned}\Delta(\mathcal{CP}_2) &= \Delta({}^*C1, {}^*C3) \\ &= \Delta(C1 \;[]\; C2 \;[]\; C3) - (\Delta(C3) + \Delta(getInstr)) \\ &= \Delta(C1) + \Delta(getAddr) + \Delta(C2)\end{aligned}$$

To evaluate the average power of these two computation paths, assume a fixed observation periods T_1 and T_2. The observation period delays are $\Delta(T_1) = \Delta(\mathcal{CP}_1)$ and $\Delta(T_2) = \Delta(\mathcal{CP}_2)$ for the observation periods T_1 and T_2, respectively. The average power consumption for the first computation path is of the form:

$$\begin{aligned}P_{T,avg}(\mathcal{CP}_1) &\overset{(4.24)}{=} P_{T,dyn}(\mathcal{CP}_1) + P_{stat}(\mathcal{CP}_1) \\ &\overset{(4.22),(4.23)}{=} \frac{E_T(\mathcal{CP}_1)}{\Delta(T_1)} + P^1_{stat} \cdot C(\mathcal{CP}_1)\end{aligned}$$

and for the second computation path:

$$\begin{aligned}P_{T,avg}(\mathcal{CP}_2) &\overset{(4.24)}{=} P_{T,dyn}(\mathcal{CP}_2) + P_{stat}(\mathcal{CP}_2) \\ &\overset{(4.22),(4.23)}{=} \frac{E_T(\mathcal{CP}_2)}{\Delta(T_2)} + P^1_{stat} \cdot C(\mathcal{CP}_2)\end{aligned}$$

where the area complexity of the computation path $\mathcal{CP}_1$ is larger than the area complexity of the computation path $\mathcal{CP}_2$. This indicates that $\mathcal{CP}_1$ has higher static power dissipation, and larger capacitive load when compared with $\mathcal{CP}_2$. Furthermore, $\mathcal{CP}_1$ consist of two memory accesses: one to the instruction cache, and one to the local memory of the co-processor, whereas $\mathcal{CP}_2$ consist only the instruction cache access, and thus the average power dissipation of the computation path $\mathcal{CP}_1$ is higher than the average power dissipation of the computation path $\mathcal{CP}_2$.

Stack and data cache. Next, the average power of memory write operation, defined by $\mathcal{E}xec$ is analyzed. $\mathcal{E}xec$ receives the result from $\mathcal{ALU}$, and stores it onto *stack* (actions $E4$ and $E5$). If the stack is full ($E5$), the data is also written onto the data cache and then flushed into the local memory. That is, the system adopts the write through policy to keep the data consistent between memory units. However, if the stack is not full, then the result is written only onto *Stack* ($E4$). Therefore, the delay of the memory write operation is defined either by the delay of the action $E3$ or the delay of the computation path $\mathcal{CP}(E3, E5)$. In a similar manner with the analysis of the instruction cache, the delays are assumed to be deterministic.

$$\begin{aligned}
\Delta(E3\langle P2\rangle) &\overset{(3.13)}{=} \Delta(E3) + \Delta(P2) \\
\Delta\mathcal{CP}(E3, E5) &\overset{(3.17)}{=} \Delta(E3\langle P2\rangle) + \Delta(E4\langle F\rangle) + \Delta(E5) \\
&\overset{(3.13)}{=} \Delta(E3) + \Delta(P2) + \Delta(E4) + \Delta(F) + \Delta(E5)
\end{aligned}$$

where the first delay describes the case when data is written only onto the stack, and the second delay describes the situation when data is written onto the stack, to data cache, and to local memory.

To evaluate the average power of these two write operations, one needs to evaluate the area complexity of the timed action $E3$ and the area complexity of the computation path $\mathcal{CP}(E3, E5)$:

$$\begin{aligned}
C(E3\langle Push\rangle) &\overset{(4.6)}{=} C(E3) + C(Push) \\
C(\mathcal{CP}(E3, E5)) &\overset{(4.25)}{=} C(E3\langle Push\rangle) + C(E4\langle Flush\rangle) + C(E5) \\
&\overset{(4.6)}{=} C(E3) + C(Push) + C(E4) + C(Flush) + C(E5)
\end{aligned}$$

To evaluate the average power consumption of the memory write operations, one needs to define an observation period. In this case, the observation period is defined to consists of single memory write operation. That is, it consists either the execution of the action $E3$ or the execution of the computation path $\mathcal{CP}(E3, E5)$. The observation periods are defined by $T_1 \mathrel{\widehat{=}} [E3.st, E3.ft]$ and $T_2 \mathrel{\widehat{=}} [E3.st, E5.ft]$. Adopting the defined observation periods and the area complexity information the average power clauses are of the form:

$$
\begin{aligned}
P_{T_1,avg}(E3) &\overset{(4.18)}{=} P_{T_1,dyn}(E3) + P_{stat}(E3) \\
&\overset{(4.11)}{=} \frac{C(E3)\cdot E^1}{\Delta(T_1)} + C(E3)\cdot P^1_{stat} \\
&= \frac{C(E3)\cdot E^1}{\Delta(E3)} + C(E3)\cdot P^1_{stat} \\
P_{T_2,avg}(\mathcal{CP}(E3,E5)) &\overset{(4.18)(4.26)}{=} P_{T_2,dyn}(\mathcal{CP}(E3,E5)) + P_{stat}(\mathcal{CP}(E3,E5)) \\
&= \frac{C(\mathcal{CP}(E3,E5))\cdot E^1}{\Delta(T_2)} + C(\mathcal{CP}(E3,E5))\cdot P^1_{stat} \\
&= \frac{(C(E3)+C(E5))\cdot E^1}{\Delta(\mathcal{CP}(E3,E5))} \\
&+ (C(E3)+C(E4)+C(E5))\cdot P^1_{stat}
\end{aligned}
$$

where the former calculates the average power of the stack write operation and the latter calculates the average power in those cases when the stack is full and data is written onto the data cache and to the local memory. The average power dissipation is naturally larger when writing onto two memories ($\mathcal{CP}(E3, E5)$), which can be seen from the above equations. That is, the area complexity of the computation path $\mathcal{CP}(E3, E5)$ is larger than the area complexity of the action $E3$, which, in turn, causes larger static power component for the average power of the computation path. The dynamic power estimate is determined by the relation between the area complexity and the delay. Considering the dynamic power estimates defined above, one can see that the computation path $\mathcal{CP}(E3, E5)$ has larger delay and area complexity than the action $E3$. Larger area complexity indicates larger load capacitance, and the increase in delay shows, in this context, that the computation path performs more memory accesses, and therefore on is able to reason that the dynamic power component is larger.

8.3.2 Arithmetic logic unit

The area complexity evaluation for addition and multiplication were analyzed in Sect. 4.6. A similar approach is applied to the arithmetic and logic operations of $\mathcal{A}LU$.

The action $A4$ is defined by:

$$
A4[\![dA4]\!]: sel = imult \rightarrow r := r'.(r' = i_1 * i_2); sel := send
$$

which is a guarded action. The area complexity calculation rule for guarded action is of the form (re-written for the ease of reference):

$$C(gd \to A) \overset{(4.5)}{=} C(gd) + C(A)$$
$$\overset{(4.4)}{=} C(gd) + C(wA) + C(Q)$$

The read set $rA4$ is $rA4 = \{sel(8), i_1(32), i_2(32)\}$ and the write set is $wA4 = \{sel(8), r(32)\}$, where the number in parenthesis after each variable is the variable width. The width of the operand variables i_1 and i_2 is defined to be 32 because the implementation in [75] use 32-bit operands. Furthermore, it is worth noticing that Java specifies the instructions to produce 32-bit result, so that the lowest 32-bits are returned and the rest are ignored without any kind of overflow or exception. Therefore, the width of the result variable r is 32. The width of the variable sel is 8 because the width of the Java instructions is one byte. Furthermore, the read set $rA4$ is divided into two sets, namely rgd and rQ, where the former contains those variables that appear in the guard and the latter those variables that appear in the predicate. The sets are defined by: $rgd \mathrel{\widehat{=}} \{sel(8)\}$ and $rQ \mathrel{\widehat{=}} \{i_1(32), i_2(32)\}$. Adopting the above presented information, the area complexities are:

$$C(gd) \overset{(4.3)}{=} \left\lceil \frac{C(rgd)}{|rgd|} \right\rceil \cdot 2^{|rgd|} = \left\lceil \frac{4}{1} \right\rceil \cdot 2^1 = 8$$
$$C(Q) \overset{(4.3)}{=} \left\lceil \frac{C(rQ)}{|rQ|} \right\rceil \cdot (2^{|rQ|})^{\phi} = \left\lceil \frac{64}{2} \right\rceil \cdot (2^2)^2 = 512$$
$$C(wA4) \overset{(4.2)}{=} 32 + 4 = 36$$

where ϕ is the complexity factor and it is defined to be two ($\phi = 2$) for multiplication as discussed in Sect. 4.6 on page 84.

$\mathcal{A}LU$ includes three 32-bit logic operations: AND, OR, and XOR. To evaluate the area complexity of these operations, the complexity factor ϕ is adjusted according to the information gained from BDDs. For AND and OR operations, the complexity factor is set to zero $\phi = 0$. The result of this modification gives an area complexity value, which is the number of nodes in the BDD. Observe that this approach is valid only for the predicate evaluation, and therefore the assignment and comparison (guard) part have to be estimated using the guidelines given in Chapter 4 ($\phi = 1$). For the XOR operation the complexity factor is set to one $\phi = 1$. This gives a reasonable accuracy (76 %) compared to BDDs, as shown in Table 4.1 on page 63. The area complexities of the system $\mathcal{A}LU$ are summarized in Table 8.7.

The area complexity of the whole system is:

$$
\begin{aligned}
C(\mathcal{A}LU) &\overset{(4.19)}{=} C(A1) + \ldots + C(A9)\langle d_{alu}\rangle \\
&\overset{(4.6)}{=} C(A1) + \ldots + C(A9) + C(d_{alu})
\end{aligned} \tag{8.1}
$$

where the area complexity of the system $\mathcal{A}LU$ sums the area complexities of the system's actions together. The read and write sets of each timed action in the system $\mathcal{A}LU$ and their area complexities are listed in Table 8.7. Observe that the area complexity of the public procedure defined in the system is included into the area complexity of the action $A9$ whereas the imported procedure is excluded. Thus, the area complexity of the imported procedure is included into that action systems area, which defines it.

The average power consumption of $\mathcal{A}LU$ is calculated by:

$$
\begin{aligned}
P_{T,avg}(\mathcal{A}LU) &\overset{(4.24)}{=} P_{T,dyn}(\mathcal{A}LU) + P_{stat}(\mathcal{A}LU) \\
&\overset{(4.22),(4.23)}{=} \frac{E(\mathcal{A}_T)}{\Delta T} + C(\mathcal{A}LU) \cdot P^1_{stat}
\end{aligned}
$$

where the static power consumption adopts the area complexity of $\mathcal{A}LU$ defined by (8.1). The dynamic power consumption is estimated during the observation period T, which is a time period defined by a designer. The observation period should include several execution cycles of the system in order to get good average power estimate.

8.3.3 Communication structures

The communication between systems and subsystems is carried out using the procedure based communication, described in Sect. 2.6.1 on page 24. The delay calculations of the procedure based communication is introduced in Sect. 3.5.4 on page 47. Consider the communication procedure d_{exec}, which transfers data from $\mathcal{E}xec$ to $\mathcal{A}LU$. The procedure is of the form (defined below for ease of reference):

$$
\begin{aligned}
d_{exec}[\![dexec]\!]: (\mathbf{in}\ x_1, x_2\ :\ Data)\ : \\
buf_1, buf_2 := x_1, x_2;
\end{aligned}
$$

where the procedure d_{exec} receives two data operands as parameters and stores the values of the operands into buffers (buf_1, buf_2). These buffers are then read by $\mathcal{A}LU$.

The procedure delay is non-deterministic and defined by:

$$
dexec\colon [dexec_{min}, dexec_{max}]
$$

where $dexec_{min}$ is the best case communication delay whereas $dexec_{max}$ is the worst case communication delay. In case of synchronous computation the clock frequency of the bus would be determined according to the worst case behavior whereas in asynchronous communication the communication delay may vary between minimum and maximum, hence the term average case performance [80].

In the co-processor system the action $E1$ in $\mathcal{E}xec$ awaits the communication procedure, and action $A1$ in $\mathcal{ALU}$ calls it. To extract the communication model, consider the definition of direct procedure based communication, defined in Sect. 2.6.1 on page 24:

$$Comm_1 \mathrel{\widehat{=}} E1A1 \mathrel{\widehat{=}} E1[A1[d_{exec}[x_1, x_2/buf_1, buf_2]/\ \textbf{await}\ d_{exec}]/\ \textbf{call}\ d_{exec}]$$

where the atomic action $E1A1$ transfers data from $\mathcal{E}xec$ (the sender $E1$) to $\mathcal{ALU}$ (the receiver $A1$).

The communication delay of the communication channel is denoted by $\Delta(Comm)$ and defined by (3.11) on page 46 and it is of the form:

$$\begin{aligned}\Delta(Comm) &\mathrel{\widehat{=}} \Delta(E1A1) = \Delta(E1\langle d_{exec}\rangle) + \Delta(A1) \\ &= \Delta(E1) + \Delta(d_{exec}) + \Delta(A1) \\ &= dE1 + d_{exec} + dA1\end{aligned}$$

and the area complexity of the communication channel is denoted by $C(Comm_1)$, and it is defined by (7.6) on page 132:

$$\begin{aligned}C(Comm) = C(E1A1) &= C(E1) + C(d_{exec}) + C(A1) \\ &\overset{(4.4)}{=} C(E1) + C(d_{exec}) + C(A1)\end{aligned}$$

where $C(E1)$ and $C(A1)$ are the area complexities of the sender action $E1$ and the receiver action $A1$, respectively. Furthermore, the operation of the communication channel is constrained by the following constraints:

(1) deadline : $\mathcal{D}(Comm, d)$

(2) area : $\mathcal{C}(Comm, c)$

(3) power : $\mathcal{P}(Comm, p)$

where d is a deadline constraint, which defines the maximum allowed duration of a single communication cycle (one data transfer from sender to receiver), and it requires that $\Delta(Comm) \leq d$. The area constraint c is defined by $C(Comm) \leq c$, where c is the maximum allowed area complexity of

the communication channel presented above. The third condition requires that the average power dissipation of the communication channel does not exceed the maximum allowed value p during the observation period T. The observation period is fixed, and it contains a single communication cycle from $\mathcal{E}xec$ to $\mathcal{A}LU$. That is, $\Delta(T) = \Delta(Comm)$.

Decomposing the communication channel. Subsystem $\mathcal{E}xec$ transfers data to $\mathcal{A}LU$ using the communication procedure d_{exec}. The procedure is awaited by the action $E1$ in $\mathcal{E}xec$, and called by the action $A1$ in $\mathcal{A}LU$. The actions are of the form:

$$\begin{aligned} E1[\![dE1]\!] &: req_{exec} \wedge \neg req_{alu} \rightarrow Pop(op_1, op_2); \\ &\qquad req_{alu} := T; \textbf{await } d_{exec}(op_1, op_2); \\ A1[\![dA1]\!] &: sel = idle \wedge req_{alu} \rightarrow \textbf{call } d_{exec}; \\ &\qquad i_1, i_2 := buf1, buf2; sel := instr \end{aligned}$$

and the communication procedure d_{exec} is of the form:

$$d_{exec}[\![dexec]\!](\textbf{in } x_1, x_2 \colon Data) \colon buf_1, buf_2 := x_1, x_2;$$

The action $E1$ consist of two procedure calls, the private procedure call ($Pop()$) and the public procedure call ($d_{exec}()$). To enhance the communication in terms of area and power dissipation, a single procedure call is adopted. That is, the stack access is included into the communication procedure. The single procedure call approach reduces area because only one communication procedure is defined instead of two. Furthermore, two procedure calls versus one procedure call takes more time. Since both time and area are decreases, one is able to assume that the power consumption of the communication channel is decreased as well. The new public procedure is of the form:

$$\begin{aligned} & d'_{exec}[\![dexec']\!](\textbf{inout } x_1, x_2, x_3, x_4 \colon Data) \colon \\ & \qquad x_1, x_2 := x'_1, x'_2.(x'_1, x'_2 \in stack); x_3, x_4 := x_1, x_2; \end{aligned}$$

and the new data transfer actions $E1'$ and $A1'$ are of the form:

$$\begin{aligned} & E1'[\![dE1']\!] \colon req_{exec} \wedge \neg req_{alu} \rightarrow \\ & \qquad req_{alu} := T; \ \textbf{await } d'_{exec}(op_1, op_2, buf1, buf2); \end{aligned}$$

$$\begin{aligned} & A1'[\![dA1']\!] \colon sel = idle \wedge req_{alu} \rightarrow \\ & \qquad \textbf{call } d'_{exec}(op_1, op_2, buf1, buf2); i_1, i_2 := buf1, buf2, sel := instr; \end{aligned}$$

where the communication procedure d'_{exec} is bidirectional, that is, it is able to transfer data in both directions.

Due to the decomposition, the delay and the area complexity information are also decomposed between the new elements. Naturally, at this abstraction level this is not an exact operation but rather an estimate of the duration and size of different operations. Based on an earlier information on delays and area complexities, more accurate delay and area allocation could be performed even at high abstraction levels. Furthermore, delay and area information provided by the circuit manufacturer can be exploited to some components of the system.

To ensure that overall timing behavior is not violated, it is required that the sum of the delays of the new actions is less than or equal to the abstract delay $dComm$:

$$R_{delay} \mathrel{\widehat{=}} \left(dE1' + dexec' + dA1'\right) \leq dComm$$

where $dComm$ describe the delay of one communication cycle. This delay describes the maximum duration of a data transfer between the systems $\mathcal{E}xec$ and $\mathcal{A}LU$. Two new deadlines are formed:

$$\mathcal{D}(E1'\langle d'_{exec}\rangle, d_1)$$
$$\mathcal{D}(A1', d_2)$$

where the delay of the communication procedure is included into the timed action $E1'$. The above defined deadlines must satisfy the following abstraction relation:

$$R_d \mathrel{\widehat{=}} d_1 + d_2 \leq \Delta(Comm)$$

where it is required that the sum of the deadlines is less than or equal to the communication delay $\Delta(Comm)$. The communication cycle from $\mathcal{E}xec$ to $\mathcal{A}LU$ is described using a computation path:

$$\mathcal{CP}(E1', A1') \mathrel{\widehat{=}} E1'\langle d'_{exec}\rangle, A1' \tag{8.2}$$

Therefore, to prove that the abstraction relation R_d holds, one needs to show that:

$$
\begin{aligned}
&\Delta(\mathcal{CP}(E1'\langle d'_{exec}\rangle, A1')) \leq (d_1 + d_2)) \\
&\Leftrightarrow \{(8.2) \text{ and the abstraction relation } R_d\} \\
&\Delta(E1'\langle d'_{exec}\rangle, A1') \leq \Delta(Comm) \\
&\Leftrightarrow \{\text{the delay calculation rule (3.11)}\} \\
&\big(\Delta(E1'\langle d'_{exec}\rangle) + \Delta(A1'\big) \leq \Delta(Comm) \\
&\Leftrightarrow \{\text{the delay calculation rule (3.13)}\} \\
&\big(\Delta(E1') + \Delta(dexec') + \Delta(A1') \leq \Delta(Comm)\big) \\
&\Leftrightarrow \{\text{the abstraction relation } R_{delay}\} \\
&\big(dE1' + dexec' + dA1'\big) \leq dComm
\end{aligned}
$$

To ensure that the overall area complexity is not exceeded, it is required that the area complexity $C(Comm)$ is reallocated between the new actions:

$$R_{area} \mathrel{\widehat{=}} C(E1') + C(d'_{exec}) + C(A1') \leq C(Comm)$$

where it is required that the area complexity of the two new actions is less than or equal to the abstract area complexity $C(Comm)$. New area constraints are granted for the two new actions:

$$
\begin{aligned}
&\mathcal{C}(E1'\langle d'_{exec}\rangle, c_1) \\
&\mathcal{C}(A1', c_2)
\end{aligned}
$$

The above defined constraint values c_1 and c_2 must satisfy the following abstraction relation:

$$R_c \mathrel{\widehat{=}} c_1 + c_2 \leq C(Comm)$$

where it is required that the sum of the constraint values is less than or equal with the area complexity of the communication $C(Comm)$. To prove the above condition, the computation path defined by (8.2) is adopted.

$$
\begin{aligned}
& C(\mathcal{CP}(E1', A1')) \leq c_1 + c_2 \\
& \Leftrightarrow \{(8.2) \text{ and the abstraction relation} R_c\} \\
& C(E1'\langle d'_{exec}\rangle, A1') \leq C(Comm) \\
& \Leftrightarrow \{\text{the area complexity rule } (4.8)\} \\
& \big(C(E1'\langle d'_{exec}\rangle) + C(A1')\big) \leq C(Comm) \\
& \Leftrightarrow \{\text{the area complexity rule } (4.6)\} \\
& \big(C(E1') + C(d'_{exec}) + C(A1')\big) \leq C(Comm) \\
& \Leftrightarrow \{\text{the abstraction relation } R_{area}\} \\
& C(E1') + C(d'_{exec}) + C(A1') \leq C(Comm)
\end{aligned}
$$

The power constraint is satisfied if the area complexity constraint is validated, and, furthermore, the timing requirements hold. As shown above, the area constraint holds for the new communication channel. The communication delay was divided between the new actions $E1'$ and $A1'$, and therefore if the deadline constraints hold, the fixed observation period set for the power constraint hold, because the observation period delay was defined to be $\Delta(T) = \Delta(Comm)$. Thus, the power constraint set for the communication channel is satisfied.

8.3.4 System level power modeling

Consider the closed timed action system $\mathcal{S}$, defined in Sect. 8.2.1, which encapsulates the operation of the $\mathcal{H}ostSystem$ and of the $\mathcal{C}o - proc$ system. Its area complexity is the sum of the area complexities of its subsystems:

$$
C(\mathcal{S}) \overset{(4.19)}{=} C(\mathcal{C}o - Proc) + C(\mathcal{H}ostSystem) \tag{8.3}
$$

where $C(\mathcal{C}o - Proc)$ is the area complexity of the co-processor and $C(\mathcal{H}ostSystem)$ is the area complexity of the host system. The area complexity of the co-processor system is further specified:

$$
C(\mathcal{C}o - Proc) = C(\mathcal{M}em) + C(\mathcal{C}omm) + C(\mathcal{E}xec) + C(\mathcal{A}LU)
$$

where the total area complexity is the sum of the area complexities of subsystems under $\mathcal{C}o - Proc$. The area complexities are calculated for each subsystem, separately.

Area complexity of the co-processor system. The area complexities of the co-processor system are listed in Table 8.2, where the first column describes the name of the action or procedure, the second and third columns

Table 8.2: Area Complexities of the system $\mathcal{C}o - proc$.

Action	Read Set	Write set	Area Complexity
IRQ	-	$\{IRQ(32)\}$	$C(IRQ) = 32$
$J1$	$\{req_{mem}(1), sys_{stat}(3)\}$	$\{sys_{stat}(3)\}$	$C(J1) = 3 + \lceil \frac{4}{2} \rceil \cdot 2^2 = 11$
$J2$	$\{req_{mem}(1), sys_{stat}(3)\}$	$\{sys_{stat}(3), req_{comm}(1)\}$	$C(J2) = 4 + \lceil \frac{4}{2} \rceil \cdot 2^2 = 12$
$J3$	$\{sys_{stat}(3), req_{comm}(1)\}$	$\{sys_{stat}(3), req_{exec}(1)\}$	$C(J3) = 4 + \lceil \frac{4}{2} \rceil \cdot 2^2 = 12$
$J4$	$\{sys_{stat}(3), req_{exec}(1)\}$	$\{sys_{stat}(3)\}$	$C(J4) = 3 + \lceil \frac{4}{2} \rceil \cdot 2^2 = 11$
$J5$	$\{halt(1)\}$	$\{sys_{stat}(3), halt(1)\}$	$C(J5) = 4 + \lceil \frac{1}{1} \rceil \cdot 2^1 = 6$
$J6$	$\{sys_{stat}(3)\}$	$\{sys_{stat}(3)\}$	$C(J6) = 3 + \lceil \frac{3}{1} \rceil \cdot 2^1 = 9$
$total$			93

lists the read and the write sets of the action or procedure, respectively, and the fourth column illustrates the area complexity calculation. If a procedure is defined and exported by the system, its area complexity is included into the system's area complexity. For instance, the area complexity of $\mathcal{C}o - Proc$ includes the procedure IRQ but not *invoke*, because it is defined by $\mathcal{H}ostsystem$, and its area complexity should be included into the $\mathcal{H}ostsystem$'s area complexity.

Memory block. The size of the memory blocks in the system is evaluated adopting the information presented in [70, 75, 82]. At first, the physical sizes of the memory components are discussed. The Java instruction set consists of 201 instructions, and the size of a single instruction is one byte. Allthough the instruction set adopted for the experiment is small, the instruction width is also set to one byte. This allows more accurate comparison with *REALJava* in terms of area and power. Furthermore, this approach alleviates future expansions to the model, for instance, if one wants to add a software support, the width of the instructions are correct. The instruction cache is (defined by subsystem $\mathcal{C}omm$):

$$icache\colon I_{mem}$$

where I_{mem} is of type **set of** Address, where the address is of type **record**(Address, Instruction). In [82], the size of the instruction cache was 32 memory locations.

Table 8.3: Area Complexities of $\mathcal{M}em$.

Action	Read Set	Write set	Area Complexity
$M1$	$\{req_{mem}(1)\}$	$\{req_{mem}(1)\}$	$C(M1) = 1 + \lceil \frac{1}{1} \rceil 2^1 = 3$
$M2$	$\{req_{comm}(1)\}$	$\{req_{instr}(1)\}$	$C(M2) = 1 + \lceil \frac{1}{1} \rceil 2^1 = 3$
$M3$	$\{req_{comm}(1), req_{instr}(1)\}$	-	$C(M3) = \lceil \frac{1}{1} \rceil \cdot 2^1 = 2$
$M4$	$\{req_{flush}(1)\}$	$\{req_{flush}(1)\}$	$C(M4) = 1 + \lceil \frac{1}{1} \rceil \cdot 2^1 = 3$
$flush$		$\{x(32)\}$	$C(flush) = 32$
$getAddr$		$\{x(32)\}$	$C(getAddr) = 32$
$getInstr$		$\{x(5), (8)\}$	$C(getInstr) = 13$
$total$			88

In this case, the described co-processor system is much smaller, and therefore the required size for instruction cache can be smaller. That is, the instruction cache consists of 16 memory locations ($C(location) = 16$). The width of each instruction is, as discussed above, one byte ($C(instruction) = 8$), and the width of the address part is set to 5 ($C(Address) = 5$), which is required to give unique address to all memory locations in instruction cache. Thus, the area complexity of the *icache* is of the form:

$$\begin{aligned} C(icache) &\overset{(4.1)}{=} (C(instructions) + C(address)) \cdot C(location) \\ &= (8 + 5) \cdot 16 = 208 \end{aligned}$$

The rest of the memory sizes are relative to the sizes described in [75], however, scaled down due to the size difference between the presented formal model and REALJava. The size of *stack* is estimated in way that it can hold (atleast) operands and results of two consecutive computation cycles. The data cache is twice the size of the stack and the local memory data area is three time as big as the data cache. The method area of the local memory unit is assumed to be same size as the data area. These area complexities are listed in Table 8.4.

The area complexities described in Table 8.4 defines the physical size for each memory block in the co-processor system. For example, a procedure call to a memory unit inside the system is the area complexity of the communication between the memory and the subsystem. That is, the area complexity of those procedures that are requesting memory accesses can be used to evaluate average power of memory accesses inside the co-processor system. Thus, the area complexities of $\mathcal{M}em$ are listed in Table 8.3.

Table 8.5 illustrates the area complexity calculation of the communication block. $\mathcal{C}omm$ requests memory accesses several times, but because

Table 8.4: Area Complexities of the Memory Units.

Memory	Size [Byte]	Width
stack	32	256
dcache	64	512
$local_{mem}$ *data area*	192	1536
$local_{mem}$ *method area*	192	1536

Table 8.5: Area Complexities of $\mathcal{C}omm$.

Action	Read Set	Write set	Area Complexity
$C1$	$\{req_{comm}(1)\}$	-	$C(C1) = \lceil \frac{1}{1} \rceil \cdot 2^1 = 2$
$C2$	$\{req_{instr}(1), I_{Addr}(5)\}$	$\{instr(8), req_{comm}(1), req_{instr}(1)\}$	$C(C2) = 10 + \lceil \frac{6}{2} \rceil \cdot 2^2 = 22$
$C3$	$\{req_{instr}(1), I_{Addr}(5)\}$	$\{req_{comm}(1)\}$	$C(C3) = 1 + \lceil \frac{6}{2} \rceil \cdot 2^2 = 13$
total			37

Table 8.6: Area Complexities of $\mathcal{E}xec$.

Action	Read Set	Write set	Area Complexity
$E1$	$\{req_{exec}(1), req_{alu}(1)\}$	$\{req_{alu}(1)$	$C(E1) = 1 + \lceil \frac{2}{2} \rceil \cdot 2^2 = 5$
$E2$	$\{req_{alu}(1)\}$	$\{wr_{result}(1)\}$	$C(E2) = 1 + \lceil \frac{1}{1} \rceil \cdot 2^1 = 3$
$E3$	$\{sz(3), wr_{result}(1)\}$	$\{sz(3)\}$	$C(E3) = 3 + \lceil \frac{4}{2} \rceil \cdot 2^2 = 11$
$E4$	$\{sz(3)\}$	$\{sz(3), req_{flush}(1), stack(32)\}$	$C(E4) = 36 + \lceil \frac{3}{1} \rceil \cdot 2^1 = 42$
$E5$	$\{wr_{result}(1), sz(3)\}$	$wr_{result}(1), req_{alu}(1), req_{exec}(1)\}$	$C(E5) = 3 + \lceil \frac{4}{2} \rceil \cdot 2^2 = 11$
$Push$	-	$\{x(32)\}$	$C(Push) = 32$
Pop	-	$\{y_1(32), y_2(32)\}$	$C(Pop) = 64$
d_{exec}	-	$\{buf_1(32), buf_2(32)\}$	$C(d_{exec}) = 64$
total			232

the public procedures are defined by $\mathcal{M}em$, their area complexities are included into $\mathcal{M}em$. Finally, area complexities of the execution unit and its

subsystem $\mathcal{ALU}$ are shown in Table 8.6 and Table 8.7.

Table 8.7: Area Complexities $\mathcal{ALU}$.

Action	Write Set	Read set	Area Complexity
$A1$	$\{i_1(32), i_2(32), sel(8)\}$	$\{sel(8)\}$	$74 + \lceil \frac{8}{1} \rceil \cdot 2^1 = 88$
$A2$	$\{r(32), sel(8)\}$	$\{sel(8), i_1(32), i_2(32)\}$	$C(A2) = 40 + \lceil \frac{8}{1} \rceil \cdot 2^1 + \lceil \frac{64}{2} \rceil \cdot (2^2) = 184$
$A3$	$\{r(32), sel(8)\}$	$\{sel(8), i_1(32), i_2(32)\}$	$C(A3) = 40 + \lceil \frac{8}{1} \rceil \cdot 2^1 + \lceil \frac{64}{2} \rceil \cdot (2^2) = 184$
$A4$	$\{r(32), sel(8)\}$	$\{sel(8), i_1(32), i_2(32)\}$	$C(A4) = 40 + \lceil \frac{8}{1} \rceil \cdot 2^1 + \lceil \frac{64}{2} \rceil \cdot (2^2)^2 = 568$
$A5$	$\{r(32), sel(8)\}$	$\{sel(8), i_1(32), i_2(32)\}$	$C(A5) = 40 + \lceil \frac{8}{1} \rceil \cdot 2^1 + \lceil \frac{64}{2} \rceil \cdot (2^2)^2 = 568$
$A6$	$\{r(32), sel(8)\}$	$\{sel(8), i_1(32), i_2(32)\}$	$C(A6) = 40 + \lceil \frac{8}{1} \rceil \cdot 2^1 + \lceil \frac{64}{2} \rceil \cdot 2^0 = 88$
$A7$	$\{r(32), sel(8)\}$	$\{sel(8), i_1(32), i_2(32)\}$	$C(A7) = 40 + \lceil \frac{8}{1} \rceil \cdot 2^1 + \lceil \frac{64}{2} \rceil \cdot 2^0 = 88$
$A8$	$\{r(32), sel(8)\}$	$\{sel(8), i_1(32), i_2(32)\}$	$C(A8) = 40 + \lceil \frac{8}{1} \rceil \cdot 2^1 + \lceil \frac{64}{2} \rceil \cdot (2^2) = 184$
$A9$	$\{r(32) sel(8)\}$	$\{sel(8)\}$	$C(A9) = 40 + \lceil \frac{8}{1} \rceil \cdot 2^1 = 46$
total			1998

The average power of $\mathcal{C}o - Proc$ is calculated over observation period T, and it is of the form:

$$
\begin{aligned}
P_{T,avg}(\mathcal{C}o - Proc) =& P_{T,avg}(\mathcal{M}em) + P_{T,avg}(\mathcal{C}omm) \\
& + P_{T,avg}(\mathcal{E}xec) + P_{T,avg}(\mathcal{ALU}) \\
\overset{(4.24)}{=} & P_{T,dyn}(\mathcal{M}em) + P_{T,dyn}(\mathcal{C}omm) \\
& + P_{T,dyn}(\mathcal{E}xec) + P_{T,dyn}(\mathcal{ALU}) \\
& + P_{stat}(\mathcal{M}em) + P_{stat}(\mathcal{C}omm) \\
& + P_{stat}(\mathcal{E}xec) + P_{stat}(\mathcal{ALU}) + P_{stat}(Blocks)
\end{aligned}
$$

where the average power of subsystems are added together. Notice that the static power $P_{stat}(blocks)$ denotes the power dissipation caused by the physical size of the memories. This is due to the fact that the adopted model does not have any specific area and power modeling techniques to

model the power consumption of memories. Therefore, the physical size of the memories is modeled using the static power dissipation $P_{stat}(Block)$ Furthermore, the power dissipation caused by the memory accesses are the power consumption of the public procedures that access those memories. As shown in Tables 8.2-8.7, the memory blocks cause the biggest power dissipation in the co-processor system. This is a similar result with the results presented in [75, 50], where in both cases the most power was consumed in the memory units. In the presented case the memory units were small, and therefore, by increasing their sizes it is fair to assume that the memory is one of the biggest design challenges in terms of power consumption. Excluding the memory units from the power evaluation, the largest power dissipation turned out to be with the Arithmetic Logic Unit (ALU). This is also consistent with the results described in [75].

8.4 Multicore Processor Approach

A multicore processor combines two or more processor cores, for instance, several CPU cores, in a single Integrated Circuit (IC). Observe that multiple processors in the same chip do not have to be the identical [34]. For example, a very powerful means to accelerate multimedia processing is to adapt programmable processors to specific, frequently occurring, high-complexity operations. The general trend in processor design have been for multicore to many cores, that is, from dual-, quad, octo- core chips towards chips that integrate tens or even hundreds of cores. For instance, dual and quad core processors can be found on desktop computers whereas Tilera TILE64 TM[9] is an example of many core processor architecture implementing 64 processor cores on a single chip.

The amount of gained performance depends on the problem being solved and on the algorithms being used as well as on their implementation in SW. Hence, SW has to be designed to take the advantage of the available parallelism. For instance, to execute Java programs in a multicore environment, the application must be programmed using multiple threads [59]. Otherwise the speed up in program execution is insignificant. Naturally, the processor will multitask better because it can execute two separate programs at once. A dual core version of *REALJava* resulted in 52 % speed improvement between a single core and the dual core version [75]. At the moment, *REALJava* is operating with three co-processor cores.

Communication between two processor cores in the same chip is naturally more effective than, for instance, communication between two single chip cores. Within a dual core processor systems, the communication between processors is on-chip communication, and therefore the signal quality and transmission speed is improved. Increasing the number of processor cores

poses pressure for communication structures. Especially with many processor systems an efficient communication is required. For instance, in Tilera 64 the on-chip communication is implemented using a Network-on-Chip (NoC) architecture, where the buses are replaced with an on-chip network, which is similar to Internet [23]. However, the presented multicore processor model contains four cores, and therefore, the traditional SoC approach is selected.

8.5 Multicore Framework

In this section, a formal framework for a multi-co-processor system is introduced. At first, the coding requirements for SW in a multicore co-processor framework are discussed. Then the multiple co-processor model is constructed using the co-processor model presented earlier in this chapter.

8.5.1 Threads

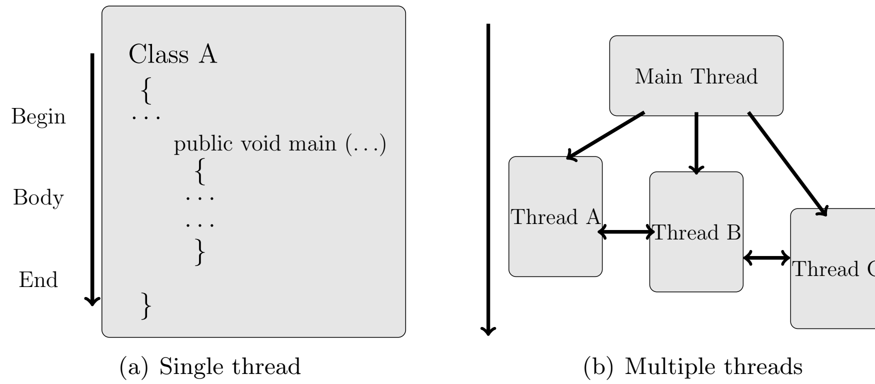
Nearly all operating systems support the concept of processes - independently running programs that are separated from each other to some degree. Threading is a facility allowing multiple activities to exist within single process, and in most cases thread(s) are located inside the process. Furthermore, multiple threads can exist (typically) within same process sharing resources such as memory while different processes do not share this data.

In Java, threads are treated like processes, i.e.,as independent, concurrent paths of execution through a program. Each thread has its own stack, its own program counter, and its own local variables. However, threads within a process are less insulated from each other than separate processes are. They share memory, file handles, and other per-processing states. Therefore, multiple threads within the process share the same memory address base, which means that they have access to the same objects and variables. While this makes it easy for threads to share information with each other, a designer must make sure that they do not interfere with other threads in the same process.

8.5.2 Forming the multiple co-processor model

In Sect. 8.2, the simplified co-processor model was presented. In this section, the purpose is to form an abstract multicore structure in which the co-processor model is adopted. The single core co-processor model is asynchronous, and therefore the presented multi-core model is asynchronous, too. Again the modeling concentrates on the HW parts of the system, whereas the SW parts are left for future studies.

The abstract network model combines a master ($\mathcal{H}ostsystem$) and co-processor units, which are controlled by the master. The multicore frame-



(a) Single thread (b) Multiple threads

Figure 8.7: Example threading schemes

work is defined by the closed timed action system $\mathcal{M}$, which defines an encapsulating system around its subsystems operating in parallel. The closed system model is adopted to avoid unnecessary complicate implementation details. The system $\mathcal{M}$ defined by:

sys $\mathcal{M}$ ( ) ::
|[
type
$Method$: $Java\ Method$;
subsystem
$\mathcal{C}o - Proc_i$: $\big(\mathcal{C}o - Proc(\mathbf{imp}\ invoke(\mathbf{in}\ d : method));$
$\mathbf{exp}\ IRQ(\ \mathbf{out}\ x : Data)$
$\mathbf{in}\ halt : Boolean\big);$
$\mathcal{H}ostSystem$: $\big(\mathcal{H}(\mathbf{exp}\ invoke(\mathbf{out}\ d : Method));$
$\mathbf{imp}\ IRQ(\ \mathbf{out}\ x : Data);$
$\mathbf{out}\ halt : Boolean\big);$
execution
$[\| \ 1 \leq i \leq n : \mathcal{C}o - Proc_i] \ \| \ \mathcal{H}ostSystem$
]|

where i is the core number. In the multicore co-processor model, $\mathcal{H}ostsystem$ acts as a central arbiter, controlling the operation of four co-processor cores, and thus, the execution clause of the system $\mathcal{M}$ is of the form:

execution
$\mathcal{C}o - Proc_1 \ \| \ \mathcal{C}o - Proc_2 \ \| \ \mathcal{C}o - Proc_3 \ \| \ \mathcal{C}o - Proc_4 \ \| \ \mathcal{H}ostSystem$

where four identical co-processor cores operates with the host system. A block diagram of the four core system is shown in Fig. 8.8.

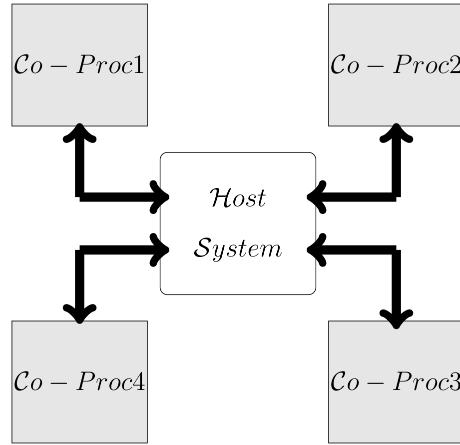


Figure 8.8: Illustration of the multicore co-processor scene

8.6 On Analyzing Power Consumption

The average power estimation of the timed action system $\mathcal{M}$ can be divided into three parts: communication, hardware, and software. The first two are highlighted in this section. Furthermore, the average power dissipation of the co-processor was defined earlier in this chapter, and therefore the analysis presented in this section concentrates on the communication between the host system and the co-processors. The co-processor system has two communication procedures *invoke*, *IRQ*, and a global Boolean variable *halt*:

$$\begin{aligned}&\textbf{exp } invoke[\![dInvoke]\!]: (\textbf{out } d : Method): d \in local_{mem}\\&\textbf{imp } IRQ[\![dIRQ]\!]: (\textbf{out } x : Data)\\&\textbf{in } halt: Boolean\end{aligned}$$

where the procedure $Invoke$ transfers the Java method d to the co-processor local memory area ($d \in local_{mem}$). The Boolean variable *halt* is used by the host system to disable the operation of the co-processor systems. The single Boolean variable is included into the IRQ procedure because it is not effective to form own communication channel for that. Therefore, the communication procedure IRQ in a multicore system is of the form:

$$IRQ[\![dIRQ]\!]: (\textbf{in } halt : Boolean; \textbf{out } x : Data)$$

where the $Halt$ operation is generated by the host system, and the IRQ is generated by the co-processor system. This is natural choice because

the host system might select to halt the co-processor system after an IRQ request.

The procedure based communication model in Sect. 2.6.1 on page 24 is adopted for the communication channels. The communication procedure models a *pull* type communication channel in which the host system transfers the Java method to the co-processor unit. The communication delays are defined using network's communication delay (7.7) on page 135.

$$\Delta_{invoke} = \Delta(Snd) + \Delta(invoke) + \Delta(Rec)$$
$$\Delta_{IRQ} = \Delta(Snd) + \Delta(IRQ) + \Delta(Rec)$$

where $\Delta(Snd)$ and $\Delta(Rec)$ are the delays of the sender and the receiver components whereas the procedure delay defines the duration of the actual data transfer. In the previous sections, the communication between subsystems inside the co-processor were modeled using this procedure based direct communication model. In SoC, the length of the wire must be taken into account when modeling the performance of the communication channel because the length of the communication links may vary significantly, which, in turn, affects the area, power, and timing measures. Section 7.3 discussed area and power modeling of different communication networks, where an relative distance was used as a measure of the communication link. The communication delay in the star network is defined by (one channel):

$$\Delta_{Link1} \widehat{=} \Delta_{invoke} + l_{wire} \cdot \Delta^{1}_{wire}$$
$$\Delta_{Link} \widehat{=} \Delta_{IRQ} + l_{wire} \cdot \Delta^{1}_{wire}$$

where l_{wire} is the length of the wire, Δ^{1}_{wire} is the unit wire delay, and Δ_{invoke} and Δ_{IRQ} the communication delays defined above.

The area complexity of communication channels in a star network is defined by (according to the guidelines presented in Sect. 7.3) :

$$C(Link1) \widehat{=} C(Snd) + C(invoke) + C(Rec) + C(wire) \qquad (8.4)$$
$$C(Link2) \widehat{=} C(Snd) + C(IRQ) + C(Rec) + C(wire) \qquad (8.5)$$

where $C(Snd)$, $C(Rec)$, and $C(invoke)$, $C(IRQ)$ are the area complexities of the sender action, receiver action, and the communication procedures, respectively; $C(wire)$ is the area complexity of the wire defined by (7.13) on page 138.

The power consumption of communication channels in a multicore system is modeled by adopting the area complexities (8.4) and (8.5), and is of the form:

$$
\begin{aligned}
P_{T,avg}(Link1) &\overset{(4.24)}{=} P_{T,dyn}(Link1) + P_{stat}(Link1) \\
&\overset{(4.22)(4.23)}{=} \frac{E_T(Link1)}{\Delta_{Link1}} + P^1_{stat} \cdot C(Link1) \\
&\overset{(4.21)}{=} \frac{E^1 \cdot \Big(C(S) + C(Invoke) + C(R)\Big) + C(wire) \cdot E^1_{wire}}{\Delta_{Link1}} \\
&+ P^1_{stat} \cdot C(Link1)
\end{aligned}
$$

$$
\begin{aligned}
P_{T,avg}(Link2) &\overset{(4.24)}{=} P_{T,dyn}(Link2) + P_{stat}(Link2) \\
&\overset{(4.22)(4.23)}{=} \frac{E_T(Link2)}{\Delta_{Link2}} + P^1_{stat} \cdot C(Link2) \\
&\overset{(4.21)}{=} \frac{E^1 \cdot \Big(C(S) + C(IRQ) + C(R)\Big) + C(wire) \cdot E^1_{wire}}{\Delta_{Link2}} \\
&+ P^1_{stat} \cdot C(Link2)
\end{aligned}
$$

where the delay of single communication cycle is used as an observation period. Therefore, the observation periods are of the form: $\Delta(T_1) = \Delta_{Link1}$ and $\Delta(T_2) = \Delta_{Link2}$ for communication channels $Link1$ and $Link2$, respectively. Furthermore, the unit wire energy is defined by (7.16).

To analyze the area complexity of the action system $\mathcal{M}$, where four co-processor cores are connected to $\mathcal{H}ostsystem$, shown in Fig. 8.8, the area complexity of the co-processor system with a single core (8.3) is adopted and extended:

$$
\begin{aligned}
C(\mathcal{M}) \mathrel{\widehat{=}} & \sum_{i=1}^{n} \big(C(\mathcal{C}o - Proc)_i + C(Link1)_i + C(Link2)_i\big) \\
& + C(\mathcal{H}ostsystem)
\end{aligned} \tag{8.6}
$$

where i is the number of the co-processors in the system. Observe that the area complexity of the co-processor blocks can be calculated by multiplying the area complexity of a single co-processor by i. However, the communication links have to be evaluated separately because the size of the link may vary, which, in turn, affects on the area complexity of the communication channel, and thus to the power dissipation of the entire system.

The average power of the multicore system is defined over observation period T, which is defined by a designer. The average power is:

$$P_{T,avg}(\mathcal{M}) \overset{(4.24)}{=} P_{T,dyn}(\mathcal{M}) + P_{stat}(\mathcal{M}) \tag{8.7}$$

$$\overset{(4.22)(4.23)}{=} \frac{E_T(\mathcal{M})}{\Delta(T)} + P^1_{stat} \cdot C(\mathcal{M}) \tag{8.8}$$

where the energy consumption is calculated for those actions that are enabled during the observation period T. The energy is calculated in two parts: First it is calculated for the co-processor system and the host system using (4.21), and second it is calculated for wires using (7.16). This approach is similar with the energy consumption of a general network model (7.15) on page 139

8.7 Chapter Summary

This chapter experimented the proposed power aware formal design framework. At first the selected case study, a Java co-processor was described after which the discussion proceeded to introduce the system under design. The hardware parts of the co-processor were the center of the design process. The initial system module consists of the host system and co-processor system, which communicated using communication procedures. The initial co-processor system was divided into subsystems, which were specified further. The target of the development process was to model the hardware parts in an asynchronous manner, after which those parts were modeled and analyzed to estimate the power consumption of the co-processor system. The hardware parts of the co-processor were implemented in a way that it was able to execute small subset of instructions, which all related to arithmetic and logic operations.

The selected case study is a good candidate especially because: it consists of different types of hardware structures, which all posed requirements to the power model. The power analysis was divided into three categories: memory components, logic, and communication. In the communication section, a communication channel in the co-processor system was decomposed into smaller parts. Each of these transformation steps adhered to the refinement guidelines discussed earlier. After these three categories were discussed, the overall power consumption of the co-processor system was calculated and analyzed. The results were similar to the existing Java co-processor implementations: The memory blocks are the most power hungry devices, whereas the communication structures contribute only small amount of power. If the memory blocks are excluded then the largest contributor to overall power consumption is the arithmetic logic unit.

In the end of the chapter, a multicore approach is introduced. The purpose of this model is to sketch the multicore co-processor approach and to show that proposed power framework can support the multiple core design approach regarding the hardware parts. Furthermore, the experiment showed that both the single core and the multicore models are lacking a model for the software parts.

Chapter 9

Conclusion

This thesis introduced a formal, power aware design framework for the modeling of SoC designs. The framework is built upon the Timed Action Systems formalism, which is an extension of the classical Action Systems framework. The thesis commenced by introducing the basic building blocks and their semantics in terms of weakest preconditions. The introduction proceeded to more complex structures such as systems, parallel operation of systems, and requisite for transferring information between systems. The introduced procedure based communication model ensures data integrity during communication by encapsulating the communication activities within one atomic action, which means that there is no possibility for an external counterpart to intervene the communication procedure. The importance of the procedure based communication at high abstraction level comes from the fact that it hides the complicated communication details, and thus a designer is able to concentrate on modeling the functionality of the system in question. Furthermore, one of the advantages of the chosen formalism is its associated refinement calculus framework within which more abstract system models are transformed in a stepwise manner towards a more concrete system models.

After describing the basic properties of the Action Systems, the introduction proceeded to its time-aware extension, Timed Action System. First, the time domain and its semantics were introduced. Second, the basic building block, a timed action is described after which the discussion proceeded to system level modeling and parallel operation of systems. Third, a concept of computation path is introduced, which can be used to describe different computational paths located, for instance, inside a system. Fourth, the delays, which were used to model the execution time of timed actions are described. Finally, the timed version of the procedure based communication, which is a direct extension from the one described above, is introduced. That is, the timed procedure based composition only adds the delays of the participating actions and procedures.

Based on the above described formal basis, the power modeling framework is introduced in Timed Action System context. The chapter commenced by discussing the power dissipation of CMOS devices, which is divided into two components: the static and dynamic component. Studying the related work in a field of high level power modeling, one is able to notice that the power models at RTL often includes two components, that is, they rely on area and timing estimates. The difference with the related work is that the presented model is targeted to higher abstraction levels than these RTL techniques, and therefore, a new estimation method for area is required. The model inherits properties from the size modeling of Boolean functions. However, it cannot directly adopt the model developed for Boolean functions because of the high abstraction level. The fundamental requirement for the presented model is that actions are defined using non-deterministic assignment. This restriction was selected to make the action definitions as consistent as possible, and therefore, to simplify the area complexity modeling. After the area complexity of action was defined, the rules for action compositions are derived after which the area complexity analysis is carried out at the system level. Naturally, the timing information is gained from the underlying formalism.

Adopting the area complexity estimate and the timing information one is able to construct a dynamic power consumption model of the timed action. The dynamic power consumption requires both the energy and the timing information of the action. The energy consumption of the timed action was defined using the area complexity, which estimates the size of the load capacitance, and the delay (execution time) of the timed action. The static power consumption of the timed action was estimated using the area complexity measure, which in this context, approximates the gate count of the timed action, and therefore, the amount of leakage that the timed action produces. Adopting both the dynamic and static power models, the total power of the timed action was defined by adding these two components together. After the area complexity and the power estimation of the timed action are introduced, the discussion proceeds towards the system level area complexity and power models. The system level area complexity is calculated by adding the area complexity of all actions in the system together. However, at system level the dynamic power consumption is defined for those actions that are enabled during the selected observation period. That is, the system level area complexity cannot be used unless all the actions in the system are executed during the observation period. Otherwise the physical capacitance estimate is the sum of the area complexities of those actions that are enabled during the observation period. However, for static power estimate, the area complexity of the system is adopted because the static power consumption does not depend on the activity of the system like the dynamic power consumption does. The average power dissipation of the

timed action system was estimated by adding the static and dynamic power consumptions together. The parallel behavior of actions was also analyzed for average power consumption after which the instantaneous power modeling framework is presented. This framework is suitable to model possible power peaks in the system. The framework divides the observation period into smaller units, denoted by time segments in this thesis, and thus the average power is analyzed separately for each time segment allowing more detailed power information of the system.

The behavior of timed actions can be restricted using constraints. This is an important issue, because the specification requirements of the system are needed to guide its development. The constraints are defined as Boolean expressions. The refinement calculus of Timed Action Systems is an extension from the refinement calculus of Actions Systems, and therefore it is easily adoptable if one is familiar with Action Systems. This robust development environment is adopted and extended for the power aware development framework. That is, functional and timing behavior of timed action are developed under the existing time aware refinement framework, which is then extended to cover the power dissipation as well.

Before experimenting the proposed power aware design framework and its development methods, two system models are introduced: synchronous and asynchronous. The functional modeling of these systems is described, after which the discussion concentrates on conditions and requirements that these system models pose to the power modeling framework. In synchronous systems, the clock signal is a major contributor to the overall power consumption of a digital system, and therefore, the power model for the clock distribution network is presented. Observe that the general power model, depicted in Chapter 4, does not take the power consumption of clock into account, and therefore, it is introduced as a separate model. Furthermore, methods to reduce power consumption such as clock and power gating were introduced, and defined in Timed Action Systems context. For asynchronous systems, the presented power modeling framework was suitable. In addition to these two system models, a power modeling framework for communication networks was presented, which concentrates on long on-chip communication lines and their power modeling.

To demonstrate the properties of the proposed power modeling framework, a simplified model of a co-processor system was defined. The experiment provided various challenging tasks to the proposed power modeling framework. The power analysis was carried out for arithmetic operations, for memory accesses and sizes, and for communication structures. A communication channel was analyzed and developed in a stepwise manner from abstract model towards a more concrete one. During the development step the time and power constraints were evaluated in order to preserve the physical correctness of the system. The selected case study showed many draw-

backs that have to be taken into account before the model can be used to larger systems. First, all though the development of systems was carried out only for small parts of the system, it required large amount of manual work. Applying the refinement framework for the entire system would have been extremely time consuming as well as tedious process. Therefore, tool support for both the modeling and performance evaluating processes is needed. Second, the selected case study highlighted the need to extend the power modeling framework to handle software components as well. Both of these issues are left for future work, discussed in the following Section in more detail.

9.1 Future Work

The formal modeling framework introduced in this thesis has no tool support at the moment. A tool would help during the development steps to visualize a system, to ease the proof obligations, to simulate the system, and to transform an implementable system specification to a hardware description language. Furthermore, the case study described in Chapter 8 showed that the area complexity calculation and the average power modeling are very time consuming for larger designs. Thus, a tool that could automate this manual work away is a challenge of outmost importance for the future work.

Another improving step for the introduced power aware modeling framework is to include a method to model and evaluate both software and hardware. Action Systems have support for both software and hardware modeling. The challenge is to decide how to model software components in the power aware design framework, and, furthermore, how to identify intrinsic properties of software and hardware components? Moreover, at a high abstraction level, a designer may want to change the implementation from HW to SW or vice versa. Thus, one needs to now what kind of component properties need to be taken care of when turning a module, for example, from HW to SW.

Based on the software side, reconfigurability is a new area in the introduced modeling framework, and it is becoming more and more important in modern embedded systems. In general, reconfigurability means that one is able to change the objectives of the hardware modules. The decision to reconfigure a component model can be done based on the demands of applications, a malfunction in a component or the thermal characteristics of system component. The reconfigurability feature poses several issues to the formal framework such as how well is reconfigurability supported by the existing models, and how to reason about temperature (hot spots), and its effects on reconfigurability. Thus, is it possible to extend the introduced power aware action systems to reason temperature, for instance, to use the

instantaneous power model to detect the hot spots in the system.

At the moment, there exist several design steps that could benefit from the tool support. This is long and tedious process, and therefore, in our lab, to obtain a solution for the lack of tool support is to integrate our framework into existing tools that have been developed for other languages, and to develop new ones only when the existing tools are unsuitable or the translation between the design languages is not reasonable. The first steps have already been taken to investigate the Action System approach to system design with SystemC [57, 58]. This approach, however, does not include the power aware design framework, which requires tool development atleast for the abstract area complexity and power models. Naturally the optimal solution would be to integrate these two approaches at some point of the design cycle. Moreover, the experience gained form the SystemC approach can be used in the development of the power modeling tools in Action Systems.

Bibliography

[1] *ACM/SIGDA 1985-1995 benchmark sets.* http://www.cbl.ncsu.edu/benchmarks/. (consulted 24th June 2009).

[2] *BuDDy Decision Diagram Package.* http://sourceforge.net/projects/buddy/. (consulted 24th June 2009).

[3] *CUDD: CU Decision Diagram Package.* http://vlsi.colorado.edu/~fabio/CUDD/. (consulted 24th June 2009).

[4] *Esterel Technologies.* http://www.esterel-technologies.com/. (consulted 24th June 2009).

[5] *Esterel Verification Environment (XEVE).* http://www-sop.inria.fr/meije/verification/Xeve/. (consulted 24th June 2009).

[6] *International Technical Roadmap for Semiconductors.* http://www.itrs.net. (consulted 24th June 2009).

[7] *MVSIS: Logic Synthesis and Verification.* http://embedded.eecs.berkeley.edu/Respep/Research/mvsis/. (consulted 24th June 2009).

[8] *REALJava Benchmark Results.* http://vco.ett.utu.fi/~teansa/REALResults/. (consulted 24th June 2009).

[9] *Tilera.* http://www.tilera.com/. (consulted 24th June 2009).

[10] *Verification Interacting with Synthesis (VIS).* http://embedded.eecs.berkeley.edu/research/vis/. (consulted 24th June 2009).

[11] Jean-Raymond Abrial, Egon Brger, and Hans Langmaack. Formal methods for industrial applications – specifying and programming the steam boiler control. In *Lecture Notes in Computer Science*, volume 1165. Springer-Verlag, 1996.

[12] Sheldon B. Akers. Binary decision diagrams. *IEEE Transaction on Computers*, C-27(6):509–516, 1978.

[13] Ferd E. Anderson, J. Steve Wells, and Eugene Z. Berta. The core clock system on the next generation itanium ™processor. In *IEEE International Solid-State Circuit Conference*, 2002.

[14] Ralp-Johan Back and Joakim von Wright. Trace refinement of action systems. In *In Proceedings of 5th International Conference on Concurrency Theory (CONCUR'94)*, volume 836 of *LNCS*, pages 367–384, 1994.

[15] Ralph-Johan Back. *On the Correctness of Refinement Steps in Program Development*. PhD thesis, Åbo Akademi, 1978.

[16] Ralph-Johan Back. Refinement calculus, part ii: parallel and reactive programs. In *In Proceedings of Stepwise Refinement of Distributed Systems:Models, Formalisms, Correctness*, volume 430 of *LNCS*, pages 67–93, 1989.

[17] Ralph-Johan Back and Reino Kurki-Suonio. Decentralization of process nets with centralized control. In *In Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 131–142, 1983.

[18] Ralph-Johan Back, Luigia Petre, and Ivan Paltor Porres. Generalizing action systems to hybrid systems. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1926 of *Lecture Notes in Computer Science (LNCS)*, pages 202–213, 2000.

[19] Ralph-Johan Back and Kaisa Sere. Action systems with synchronous communication. In *Proceedings of the IFIP TC2/WG2.1/WG2.2/WG2.3 Working Conference on Programming Concepts, Methods and Calculi*, pages 107–126, 1994.

[20] Ralph-Johan Back and Kaisa Sere. From modular systems to action systems. In *In Proceedings of Formal Methods Europe*, LNCS, 1994.

[21] Ralph-Johan Back and Kaisa Sere. Superposition refinement of reactive systems. *Formal Aspects of Computing*, 8(3):324–346, 1996.

[22] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.

[23] Luca Benini and Giovanni De Micheli. *Networks on Chips*. Morgan Kaufmann, 2006.

[24] Albert Benveniste and Paul Le Guernic. Hybrid dynamical systems theory and the signal language. *IEEE Transactions on Automatic Control*, 35(5):535–546, 1990.

[25] Giuseppe Bernacchia and Marios C. Papaefthymiou. Analytical macromodeling for high-level power estimation. In *In Proceedings of IEEE International Conference on Computer Aided Design*, pages 280–283, 1999.

[26] Gérard Berry and Laurent Cosserat. The esterel synchronous programming language and its mathematical semantics. In *In Seminar of Concurrency*, volume 197 of *LNCS*, pages 389–448, 1985.

[27] Gérard Berry and Ellen Sentovich. Multiclock esterel. In *In Correct Hardware Design and Verfication Methods*, volume 2144 of *LNCS*, pages 110–125, 2001.

[28] Niraj Bindal, Timothy Kelly, Nicholas Velastegui, and Ken L. Wong. Scalable sub-10ps skew global clock distribution for a 90nm multi-ghz ia microprocessor. In *IEEE International Solid-State Circuits Conference*, 2003.

[29] Randal E. Bryant. Graph.based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, C-35(8):677–691, 1986.

[30] Randal E. Bryant. On the complexity of vlsi implementations and graph presentations of boolean functions with application to integer multiplication. *IEEE Transaction on Computers*, 40(2):205–213, 1991.

[31] Saif Ali Butt, Stefan Schmermbeck, Jurij Rosenthal Alexander Pratsch, and Eike Schmidt. System level clock tree synthesis for power optimization. In *Design, Automation and Test in Europe Conference*, pages 1–6, 2007.

[32] Paul Caspi, Nicolas Halbwachs, Daniel Pilaud, and Johan Plaice. Lustre: A declarative language for programming synchronous systems. In *In Proceedings of the 14th Symposium on Principles of Programming Languages (POPL'87)*, pages 178–188, 1987.

[33] Anantha Chandrakasan, William Bowhill, and Frank Fox, editors. *Design of A High-Performance Microprocessor Circuits*. IEEE Press, 2001.

[34] Yen-Kuang Chen and S. Y. Kung. Trend and challenges on a system-on-chip designs. *Journal of Signal Processing Systems*, 53:217–229, 2008.

[35] Kwang-Ting Cheng and Vishwani Agraval. An entropy measure for the complexity of boolean functions. In *In Proceedings of the Design Automation Conference (DAC)*, pages 302–305, 1990.

[36] Sam Chitwood and Ji Zheng. Ir-drop in high-speed packages and pcbs. *Printed Circuit Design and Manufacture*, 2005.

[37] Robert W. Cook and Michale J. Flynn. Logical network cost and entropy. *In IEEE Transaction on Computer*, 22(9):823–826, 1973.

[38] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms - Second Edition*. The MIT Press and McGraw-Hill, 2001.

[39] William J. Dally and John W. Poulton. *Digital Systems Engineering*. Cambridge University Press, 1998.

[40] Edgar W. Dijkstra. *A Discipline of Programming*. Prentice-Hall International, 1976.

[41] Gilles Dubost, Sylvie Granier, and Gerard Berry. An esterl-based formal specification methodology for power manager development. In *In Sophia Antipolis forum on MicroElectronics*, 2007.

[42] Phillip J. Restle et.al. A clock distribution network for microprocessors. *IEEE Journal of Solid-State Circuits*, 36(5):792–799, 2001.

[43] Eby G. Friedman. Clock distribution networks in synchronous digital integrated circuits. *IEEE*, 89(5):665–692, May 2001.

[44] Subodh Gupta and Farid N. Najm. Power macromodeling for high level power estimation. *IEEE Transactions on VLSI Systems*, 8:18–29, 2000.

[45] Subodh Gupta and Farid N. Najm. Energy and peak-current per cycle estimation at rtl. *IEEE Transactions on VLSI Systems*, 11(4):525–537, 2003.

[46] Shmuel Katz. ”a superposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2), 1993.

[47] Michael Keating, David Flynn, Robert Aitken, Alan Gibbons, and Kaijian Shi. *Low-Power Design Methodology Manual for System-on-Chip Design*. Springer, 2007.

[48] Eduardo Kellerman. A formula for logical network costs. *IEEE Transactions on Computers*, 17(9):881–884, 1968.

[49] Nam Sung Kim, Todd Austin, David Blaauw, Krisztián Flautner, Jie S. Hu, Mary Jane Irwin, Mahmut Kandemir, and Vijaykrishnan Narayanan. Leakage current: Moore’s law meets static power. *IEEE Computer*, 36(12):68–75, 2003.

[50] Sébastien Lafond. *Simulation of Embedded Systems for Energy Consumption Estimation*. PhD thesis, Åbo Akademi, 2009.

[51] Paul Landman. High-level power estimation. In *In Proceedings of International Symposium on Low-Power Electronics and Design*, pages 29–35, June 1996.

[52] Chih-Hung Lee, Chin hung Sy, Shih-Hsu Huang, Chih-Yuan Lin, and Tsai-Ming Hsieh. Floorplanning with clock tree generation. In *Proceedings of 2005 IEEE international Symposium on Circuits and Systems*, volume 6, pages 6244– 6247, 2005.

[53] Pasi Liljeberg, Johanna Tuominen, Sampo Tuuna, Juha Plosila, and Jouni Isoaho. *Interconnect-Centric Design for Advanced SoC/NoC*, chapter Self-Timed Methodology and Techniques for Noise Reduction in NoC Interconnect, pages 285–313. Kluwer Academic Publishers, 2004.

[54] Dake Liu and Christer Svensson. Power consumption estimation in cmos vlsi chips. *IEEE Journal of Solid State Circuits*, 29(6):663–670, 1994.

[55] Diana Marculescu, Radu Marculescu, and Massoud Pedram. Information theoretic measures of energy consumption at register transfer level. *1996*, 15:599–610, IEEE Transaction on Computer-Aided Design.

[56] Renu Mehra and Jan Rabaey. Behavioral level power estimation and exploration. In *In Proceedings of First International Workshop on Low Power Design*, pages 197–202, 1994.

[57] Tomi Metsälä, Tomi Westerlund, and Juha Plosila. Introducing action systems class hierarchy to systemc modeling. Technical Report 946, Turku Centre for Computer Center, 2009.

[58] Tomi Metsälä, Tomi Westerlund, Seppo Virtanen, and Juha Plosila. Rigorous communication modeling at transaction level with systemc. In *The Third International Conference on Software and Data Technologies, ICSOFT2008*, pages 246–251, July 2008.

[59] Jon Meyer and Troy Downing. *Java Virtual Machine*. O'Reilly and Associates, 1997.

[60] Mohammad Reza Mousavi, Paul Le Guernic, Jean-Pierre Talpin, Sandeep Kumar Shukla, and Twan Basten. Modeling and validating globally asynchronous design in synchronous frameworks. In *In Proceedings of the Conference on Design Automation and Test in Europe (DATE'04)*, IEEE Computer Society Press, pages 384–389, 2004.

[61] David E. Muller. Complexity in electronic switching circuits. *IRE Transactions on Electronic Computers*, 5:15–19, 1956.

[62] Klaus Dieter Müller, Karlheinz Kirsch, and Karl Neusinger. Estimating essential design characteristics to support project planning for asic design management. In *In Proceedings of IEEE International Conference on Computer-Aided Design*, pages 148–151, 1991.

[63] Farid N. Najm. A survey of power estimation techniques in vlsi circuits. *IEEE Transaction on VLSI Systems*, 2:446–455, 1994.

[64] Mahadevamurty Nemani and Farid N. Najm. Towards a high-level power estimation capability. *IEEE Transaction on Computer-Aided Design*, 15:588–598, 1996.

[65] Mahadevamurty Nemani and Farid N. Najm. High-level power estimation and the area complexity of boolean functions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):697–713, 1999.

[66] Nicholas Pippenger. Information theory and the complexity of boolean functions. *Mathematical System Theory*, 10:129–167, 1977.

[67] Juha Plosila. *Self-Timed Circuit Design - The Action System Approach*. PhD thesis, University of Turku, 1999.

[68] Juha Plosila, Pasi Liljeberg, and Jouni Isoaho. Modeling and refinement of an on-chip communication. In *Formal Methods and Engineering: 7th International Conference on Formal Engineering Methods*, volume 3785/2005 of *LNCS*, pages 219–234, 2005.

[69] Jan Rabaey, Anantha Chandrakasan, and Borivoje Nikolić. *Digital Integrated Circuits - A Design Perspective*. Prentice Hall, 2nd edition, 2003.

[70] Ramesh Radhakrishnan, Narayan Vijaykrishnan, Lizy Kurian John, Anand Sivasubramaniam, Juan rubio, and Jyotsna Sabarinathan. Java runtime systems: Characterization andarchitectural implementations. *IEEE Transactions on Computers*, 50(2):131–145, 2001.

[71] Anand Raghunathan, Sujit Dey, and Niraj K. Jha. Register transfer level estimation techniques for switching activity and power consumption. In *In Proceedings of International Conference on Computer-Aided Design*, pages 158–165, November 1996.

[72] Basant Rajan and R.K. Shyamansundar. Multiclock esterel: a reactive framework for asynchronous design. In *In Proceedings of the 14th IEEE International Parallel and Distributed Processing Symposium (IPDPS'00)*, pages 201–210, 2000.

[73] Mauno Rönkko. *Stepwise Development of Hybrid Systems.* PhD thesis, Åbo Akademi, 2001.

[74] Mauno Rönkkö and Anders Ravn. Action systems with continuous behavior. In *Hybrid Systems V*, volume 1567 of *Lecture Notes in Computer Science (LNCS)*, pages 304–323, 1999.

[75] Tero Säntti. *A Co-Processor Approach for Effecient Java Execution in Embedded Systems.* PhD thesis, University of Turku, 2008.

[76] Tiberiu Seceleanu. *Systematic Design of Synchronous Digital Circuits.* PhD thesis, Åbo Akademi, 2001.

[77] Emil Sekerinski and Kaisa Sere. A theory of prioritizing composition. *The Computer Journal*, 39(8):701–712, 1996.

[78] Kaisa Sere and Marina Waldén. Data refinement of remote procedures. *Formal Aspects of Computing*, 12(4):278–297, 2000.

[79] Claude E. Shannon. The synthesis of two-terminal switching circuits. *Bell System Technical Journal*, 28(1):59–98, 1949.

[80] Jens Sparsøand Steve Furber. *Principles of Asynchronous Circuit Design - A System Perspective.* Kluwer Academic Publishers, 2001.

[81] Jean-Pierre Talpin, Paul Le Guernic, Sandeep Kumar Shukla, Frédéric Doucet, and Rajesh Gupta. Polychrony for formal refinement-checking in a system level design methodology. In *In Proceedings of the 3rd IEEE International Conference on Application of Concurrency to System Design (ACSD'2003)*, pages 9–19, 2003.

[82] Johanna Tuominen, Tero Säntti, and Juha Plosila. Comparative study of synthesis for asynchronous and synchronous cache controllers. In *Proceedings of the 24th IEEE Norchip Conference, Linköping, Sweden*, pages 11–14, 20 - 21 November 2006.

[83] Johanna Tuominen, Tomi Westerlund, and Juha Plosila. Formal power analysis of on-chip communication. In *Proceeding of the Brazilian Symposium on Formal Methods (SBMF)*, pages 87–102, 2007.

[84] Johanna Tuominen, Tomi Westerlund, and Juha Plosila. Feasiblity report on formal area complexity estimation. Technical Report 907, Turku Centre for Computer Science, 2008.

[85] Johanna Tuominen, Tomi Westerlund, and Juha Plosila. Power aware system refinement. *Electronic Notes in Theoretical Computer Science*, 201C:223–253, 2008.

[86] Tomi Westerlund. *Time Aware Modeling and Analysis of Systems-on-Chip*. PhD thesis, University of Turku, 2007.

[87] Tomi Westerlund and Juha Plosila. Formal, time aware modeling of communication channels for vlsi systems. In *Proceeding of the Brasilian Symposium on Formal Methods (SBMF)*, pages 27–42, 2006.

[88] Qing Wu, Qinru Qiu, Massoud Pedram, and Chin-Shun Ding. Cycle accurate macro-model for rt-level power analysis. *IEEE Transactions on VLSI Systems*, 6(4):520–528, 1998.

Turku Centre for Computer Science
TUCS Dissertations

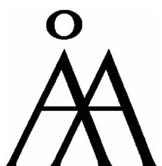
87. **Tuomas Hakkarainen**, On the Computation of the Class Numbers of Real Abelian Fields
88. **Elena Czeizler**, Intricacies of Word Equations
89. **Marcus Alanen**, A Metamodeling Framework for Software Engineering
90. **Filip Ginter**, Towards Information Extraction in the Biomedical Domain: Methods and Resources
91. **Jarkko Paavola**, Signature Ensembles and Receiver Structures for Oversaturated Synchronous DS-CDMA Systems
92. **Arho Virkki**, The Human Respiratory System: Modelling, Analysis and Control
93. **Olli Luoma**, Efficient Methods for Storing and Querying XML Data with Relational Databases
94. **Dubravka Ilić**, Formal Reasoning about Dependability in Model-Driven Development
95. **Kim Solin**, Abstract Algebra of Program Refinement
96. **Tomi Westerlund**, Time Aware Modelling and Analysis of Systems-on-Chip
97. **Kalle Saari**, On the Frequency and Periodicity of Infinite Words
98. **Tomi Kärki**, Similarity Relations on Words: Relational Codes and Periods
99. **Markus M. Mäkelä**, Essays on Software Product Development: A Strategic Management Viewpoint
100. **Roope Vehkalahti**, Class Field Theoretic Methods in the Design of Lattice Signal Constellations
101. **Anne-Maria Ernvall-Hytönen**, On Short Exponential Sums Involving Fourier Coefficients of Holomorphic Cusp Forms
102. **Chang Li**, Parallelism and Complexity in Gene Assembly
103. **Tapio Pahikkala**, New Kernel Functions and Learning Methods for Text and Data Mining
104. **Denis Shestakov**, Search Interfaces on the Web: Querying and Characterizing
105. **Sampo Pyysalo**, A Dependency Parsing Approach to Biomedical Text Mining
106. **Anna Sell**, Mobile Digital Calendars in Knowledge Work
107. **Dorina Marghescu**, Evaluating Multidimensional Visualization Techniques in Data Mining Tasks
108. **Tero Säntti**, A Co-Processor Approach for Efficient Java Execution in Embedded Systems
109. **Kari Salonen**, Setup Optimization in High-Mix Surface Mount PCB Assembly
110. **Pontus Boström**, Formal Design and Verification of Systems Using Domain-Specific Languages
111. **Camilla J. Hollanti**, Order-Theoretic Mehtods for Space-Time Coding: Symmetric and Asymmetric Designs
112. **Heidi Himmanen**, On Transmission System Design for Wireless Broadcasting
113. **Sébastien Lafond**, Simulation of Embedded Systems for Energy Consumption Estimation
114. **Evgeni Tsivtsivadze**, Learning Preferences with Kernel-Based Methods
115. **Petri Salmela**, On Commutation and Conjugacy of Rational Languages and the Fixed Point Method
116. **Siamak Taati**, Conservation Laws in Cellular Automata
117. **Vladimir Rogojin**, Gene Assembly in Stichotrichous Ciliates: Elementary Operations, Parallelism and Computation
118. **Alexey Dudkov**, Chip and Signature Interleaving in DS CDMA Systems
119. **Janne Savela**, Role of Selected Spectral Attributes in the Perception of Synthetic Vowels
120. **Kristian Nybom**, Low-Density Parity-Check Codes for Wireless Datacast Networks
121. **Johanna Tuominen**, Formal Power Analysis of Systems-on-Chip

Turku
Centre *for*
Computer
Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www.tucs.fi

University of Turku

- Department of Information Technology
- Department of Mathematics



Åbo Akademi University

- Department of Information Technologies

Turku School of Economics

- Institute of Information Systems Sciences

ISBN 978-952-12-2357-0
ISSN 1239-1883

Johanna Tuominen Formal Power Analysis of Systems-on-Chip

Johanna Tuominen Formal Power Analysis of Systems-on-Chip