

# Simulation and Visualisation for Debugging Large Scale Asynchronous Handshake Circuits

A thesis submitted to the University of Manchester  
for the degree of Doctor of Philosophy in the  
Faculty of Science & Engineering

2004

Lilian Janin

Department of Computer Science

---

# Contents

Contents .....	2
List of Figures .....	6
List of Tables .....	8
Abstract .....	9
Declaration .....	10
Copyright .....	10
Acknowledgements .....	11
<b>Chapter 1: Introduction .....</b>	<b>12</b>
1.1 Large Scale Asynchronous Handshake Circuits .....	12
1.2 Automated Circuit Synthesis .....	14
1.3 Motivations and Objectives .....	15
1.4 Contributions made by this work .....	16
1.5 Thesis Overview .....	17
1.6 Publications .....	18
<b>Chapter 2: Background .....</b>	<b>19</b>
2.1 Asynchronous Design .....	19
2.1.1 Handshake Signalling Protocols .....	21
2.1.2 Delay Models .....	22
2.1.3 Data Encodings .....	24
2.1.4 Asynchronous Difficulties .....	26
2.2 Asynchronous Specification Techniques .....	28
2.2.1 Event-Based Specification .....	28
2.2.2 State-Based Specification .....	30
2.2.3 Communicating Sequential Processes .....	30
2.2.4 Macromodules and DI Interconnect .....	31
2.3 Balsa .....	32
2.3.1 Balsa Framework .....	33
2.3.2 Balsa Language .....	33
2.3.3 Breeze Handshake Circuit .....	36
2.4 Network Graphs .....	40

---

<b>Chapter 3: Related Work .....</b>	<b>41</b>
3.1 Handshake Circuit Simulation .....	41
3.2 Debugging Asynchronous-Specific Problems .....	43
3.3 Visualisation Oriented Towards Program Comprehension .....	44
3.3.1 Knowledge Organisation by Merging Multiple Sources .....	45
3.3.2 Dynamic Visualisation .....	47
3.3.3 Information Exploration with Coordinated Views .....	50
3.4 Unified IDE for Large Scale Asynchronous Circuits .....	51
3.5 Summary .....	52
 <b>Chapter 4: Theory of Handshake Circuit Debugging .....</b>	 <b>53</b>
4.1 The Handshake Circuit Model .....	53
4.1.1 Static Model .....	53
4.1.2 Dynamic Model .....	54
4.2 Deadlocks .....	56
4.2.1 Handshake Circuit Deadlock Detection .....	57
4.2.2 Handshake Circuit Deadlock Analysis .....	59
4.3 Non-determinism .....	62
4.3.1 Metastability .....	63
4.3.2 Modelling Delays With Errors .....	64
4.3.3 Exhaustive Simulation .....	64
4.4 Activity Pattern Analysis .....	65
4.4.1 Visual Analysis .....	65
4.4.2 Automated Analysis .....	66
4.5 Circuit Optimisation – Profiling .....	67
4.6 Summary .....	67
 <b>Chapter 5: High-Performance Simulation .....</b>	 <b>68</b>
5.1 Preamble .....	69
5.1.1 Choice of the Simulation Level .....	69
5.1.2 Choice of the Handshake Protocol .....	70
5.1.3 Preliminary Statistics .....	71
5.2 Scheduler .....	72
5.2.1 A Software Model for Simulating Handshake Circuits .....	72
5.2.2 Standard Event-Driven Scheduler .....	74
5.2.3 Out-of-Order Scheduler .....	76

---

---

5.2.4 Reordering Arbitration Inconsistencies .....	83
5.3 Modelling Handshake Circuits for Speed .....	84
5.3.1 Channel Data Value Implementation .....	84
5.3.2 Premature Channel Data Storage .....	85
5.3.3 Data Sharing Between Components .....	85
5.4 Test Harnesses .....	86
5.5 Summary .....	87
<b>Chapter 6: Analysis-Oriented Simulation .....</b>	<b>88</b>
6.1 Timing Analysis .....	88
6.1.1 Determining and Adjusting Delays .....	89
6.1.2 Simulating Delays with Errors .....	90
6.1.3 Delays in Test Harnesses .....	91
6.2 Power Analysis .....	91
6.3 Source Code Position Annotation .....	92
6.4 Simulation Tracing for Offline Analysis .....	93
6.4.1 Standard Trace .....	94
6.4.2 Out-of-Order Trace .....	95
6.4.3 Pattern Analysis and Compressed Out-of-Order Trace .....	95
6.5 Summary .....	97
<b>Chapter 7: Visualisation .....</b>	<b>98</b>
7.1 Information Clustering .....	98
7.1.1 Functional Grouping .....	99
7.1.2 Control Threads .....	102
7.1.3 Data Flow .....	104
7.1.4 Test Harnesses .....	107
7.2 Multi-Source Graph View .....	108
7.2.1 Static Multiscale Structure .....	108
7.2.2 Dynamic Colour-Based Animation .....	109
7.3 Coordinated/Collaborative Views .....	110
7.3.1 Views .....	111
7.3.2 Multiple Views: Linking the Different Representations .....	113
7.4 Additional techniques .....	116
7.4.1 Dot Layout .....	116
7.4.2 Tracking Structural Changes during Design Iterations .....	117

---

---

7.5 Summary .....	119
<b>Chapter 8: Integration .....</b>	<b>120</b>
8.1 Balsa Compiler and Breeze Format .....	121
8.2 Simulation Trace .....	122
8.3 Visualisation Control Links .....	122
8.4 Summary .....	123
<b>Chapter 9: Results and Discussion .....</b>	<b>124</b>
9.1 Simulation: Boosted Compilation and Simulation Speeds .....	125
9.2 Debugging Demonstrator: The Simple Corridor Problem .....	131
9.2.1 Deadlock Handling .....	132
9.2.2 Livelock Handling .....	134
9.2.3 Non-Determinism Handling .....	135
9.2.4 Further Pattern Analysis and Trace Compression .....	136
9.2.5 Discussion .....	139
9.3 Visualisation Demonstrator: The Huge SPA Microprocessor Core .....	139
9.3.1 Merging Sources for Multiscale Graph Visualisation .....	140
9.3.2 Animated Graph .....	145
9.3.3 Coordinated/Collaborative Views .....	146
9.3.4 Discussion .....	148
9.4 Unified Debugging Environment .....	149
<b>Chapter 10: Conclusions .....</b>	<b>150</b>
10.1 Summary .....	150
10.2 Summary of Contributions .....	151
10.3 Limitations .....	152
10.4 Suggestions for Future Work .....	152
<b>Appendix A: Balsa Example Circuits with Statistics .....</b>	<b>154</b>
<b>Appendix B: Breeze Handshake Components .....</b>	<b>159</b>
<b>References .....</b>	<b>171</b>

---

## List of Figures

1.1	Synchronous and asynchronous mechanisms	13
2.1	Push and pull channel notations	20
2.2	Channel signalling protocols	22
2.3	4-phase push data validity schemes	23
2.4	STG notation	29
2.5	STG specification for a 2 input Muller C-element	29
2.6	Balsa design flow	34
2.7	Balsa language features (modulo-10 counter example)	35
2.8	Breeze handshake circuit graph (modulo-10 counter example)	37
2.9	Breeze handshake circuit netlist (modulo-10 counter example)	39
2.10	Network graph	40
4.1	Time model of the execution of a handshake circuit	54
4.2	Simplified time model of the execution of a handshake circuit	55
4.3	Elements involved in a handshake circuit deadlock	59
4.4	Deadlock analysis algorithm	60
5.1	Object-oriented view of handshake component and channel	73
5.2	Object-oriented view of a handshake circuit	75
5.3	Fork component model and scheduler's event queue	75
5.4	Pseudo handshake circuit for the equation	77
5.5	Execution order of the handshake components in figure 5.4	79
5.6	Arbitrated circuit	81
5.7	Starvation due to out-of-order arbitration	82
5.8	Three stage buffer circuit	86
6.1	Interleaved and sequential traces of threads of events	96
7.1	Abstracted functional grouping	100
7.2	Unoptimised and optimised representations of a sequence of 4 actions	102
7.3	Three control thread sets possible with a Fork component	103
7.4	The data Transferrer component	105
7.5	Pull and push Add (BinaryFunc) and Split components	106
7.6	CaseFetch component	106
7.7	Coordinated/collaborative views	112
7.8	Circuit reconfiguration strategies	118
8.1	New Balsa simulation and visualisation flow	120
9.1	Implementation model for the corridor problem	132
9.2	Deadlock with two lazy_guys in the corridor example	133
9.3	Livelock with two polite_guys in the corridor example	134
9.4	Handshake circuit of the one-place buffer example with its environment	137
9.5	Huge graph layout: SPA - Zoom 100%	141
9.6	Huge graph layout: SPA - Zoom 250%	142
9.7	Huge graph layout: SPA - Zoom 900%	143
9.8	Huge graph layout: SPA - Zoom 4500%	144
9.9	Step by step animation of a 1-place buffer	146

---

9.10	Step by step animation of a hypothetical parallel circuit	146
9.11	High level SPA animation snapshot with bars in groups	147
B.1	Continue and Halt handshake components	160
B.2	Loop handshake component	160
B.3	Sequence, Concur, Fork and WireFork handshake components	160
B.4	While handshake component	161
B.5	Bar handshake component	161
B.6	Fetch handshake component	162
B.7	FalseVariable handshake component	162
B.8	Case handshake component	162
B.9	NullAdapt handshake component	162
B.10	Encode handshake component	163
B.11	Adapt and Slice handshake components	163
B.12	Constant handshake component	164
B.13	Combine and CombineEqual handshake components	164
B.14	CaseFetch handshake component	164
B.15	UnaryFunc, BinaryFunc and BinaryFuncCont handshake components	165
B.16	ContinuePush and HaltPush handshake components	166
B.17	ForkPush handshake component	166
B.18	Call, CallMux and CallDemux handshake components	167
B.19	Passivator and PassivatorPush handshake components	167
B.20	Synch, SynchPull and SynchPush handshake components	168
B.21	DecisionWait handshake component	168
B.22	Split and SplitEqual handshake components	169
B.23	Variable and InitVariable handshake components	169
B.24	Arbiter handshake component	170

---

## List of Tables

7.1	Visualised elements per view	114
7.2	Visualised elements per source	114
9.1	Evolution of the compilation speed	127
9.2	Evolution of the simulation speed	128
9.3	Comparison of Breeze and Verilog simulators	130
9.4	Design iteration speedup	130
9.5	Trace compression results	138
A.1	Size of Balsa circuits examples	154



---

## Abstract

Recent advances in automated synthesis tools for asynchronous circuits have made possible the design of large self-timed circuits. However, these new tools are still weak in their behavioural simulation and debugging capabilities because asynchronous circuits pose different challenges and opportunities in these areas from conventional clocked circuits. Balsa is such a tool intended for the synthesis of large asynchronous circuits by using handshake circuits as an intermediate representation.

This thesis addresses new simulation and visualisation techniques for debugging large scale asynchronous circuits at the handshake circuit level. It is based on extensive behavioural simulation and large scale visualisation of handshake circuits.

A set of optimisation techniques applicable to the simulation of handshake circuits leads to a simulator four orders of magnitude faster than the previous Balsa simulator on large circuits. A visualisation system targeting program comprehension by efficiently tracking control flows is presented. It is based on two techniques: First, a graph-based view of the handshake circuit merges multiple sources of information to generate a graph viewable at any level of detail, with dynamic simulation results rendered atop it. Then, a collaborative scheme between multiple views allows the tracking of elements between views for efficient navigation.

The framework is evaluated on the largest circuit described with Balsa so far, an ARM-compatible asynchronous microprocessor.

---

## Declaration

Most of the figures and descriptions contained in Appendix B were originally described by Bardsley [5]. They have been reproduced here and updated with the permission of the author.

No other portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

## Copyright

- (1) Copyright in text of this thesis rests with the Author. Copies (by any process) either in full, or of extracts, may be made **only** in accordance with instructions given by the Author and lodged in the John Rylands University Library of Manchester. Details may be obtained from the Librarian. This page must form part of any such copies made. Further copies (by any process) of copies made in accordance with such instructions may not be made without the permission (in writing) of the Author.
- (2) The ownership of any intellectual property rights which may be described in this thesis is vested in the University of Manchester, subject to any prior agreement to the contrary, and may not be made available for use by third parties without permission of the University, which will prescribe the terms and conditions of any such agreement.
- (3) Further information on the conditions under which disclosures and exploitation may take place is available from the Head of the Department of Computer Science.

---

## Acknowledgements

It is a pleasure to thank the many people who have contributed to my happiness during these last four years and made this thesis possible.

I would like to thank first my supervisor, Dr. Doug Edwards, for his kindness, for the freedom he gave me and for always being there at the right time to guide me.

Thanks to Dr. Andrew Bardsley for the many explanations and advice about Balsa, and thanks to everybody in the APT group for their friendship.

Special thanks to my local proofreaders: Peter Riocreux, Dr. Andrew Bardsley, Dr. John Bainbridge, Dr. Luis Plana and Sam Taylor for fixing my French style as much as they possibly could, and also for being such nice colleagues and friends. I could never thank Andrew enough for reading the whole of my thesis so quickly, nor Peter for reading it twice even more quickly, saving me from the deadline.

Another set of proofreading thanks goes to my French friends: Fred Lapin Lacombe, Régis Décamps, Alexandre Klimowicz, Franck Bettinger and Nicolas Gaborit. They have been extremely helpful.

Thanks to all my friends: Franck, Nicolas, Fabrice, Vivek, Bruno, Racoon, Alexandre, Régis, Jérôme, Laurent, Arnaud, Aymeric, Nathanaël and all the others for enjoying some time with me. And a giant thank you to Lapin, who was there for me in critical times.

Finally I'd like to thank Yin Jin, my brothers and sister Florent, Marion and Jean, and my parents for their love.

# Chapter 1: Introduction

This thesis is concerned with techniques to support the debugging of large asynchronous handshake circuits by using extensive simulation and large scale visualisation methods. These techniques are applied to the Balsa asynchronous circuit synthesis framework.

## 1.1 Large Scale Asynchronous Handshake Circuits

Very Large Scale Integration (VLSI) is the current level of computer microchip miniaturisation and refers to electronic circuits on a chip containing more than hundreds of thousands of transistors. A computer microprocessor is an example of VLSI chip.

VLSI designs can be divided into two major classes: synchronous and asynchronous circuits. Synchronous circuits use a periodic signal called a *clock* to synchronise every part of the circuit. The clock signal is distributed throughout all parts (or *modules*) of a system to ensure correct timing and to synchronise the data processing mechanisms. Asynchronous circuits contain no global clock. The synchronisation required for data transfers between two modules is controlled by locally generated signals (Figure 1.1). One major class of asynchronous circuits uses a synchronisation method between modules called *handshaking*. Handshaking is used to synchronise two independent asynchronous modules for data transfer by observing the following scheme:

1. *The sender places the data onto the data wires.*
2. *The sender notifies the receiver by changing the value of a specific wire meaning:*  
*“The data on the data wires is valid, you can read it”.*
3. *The receiver receives the message and processes the data.*
4. *The receiver notifies the sender by changing the value of a specific wire meaning:*  
*“I have finished using the data”.*
5. *The sender receives the message and continues its processing.*

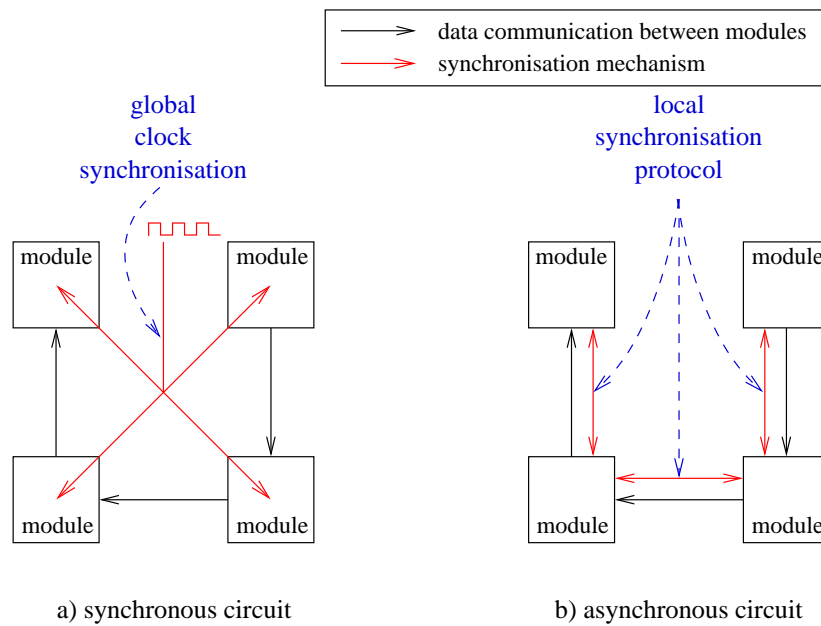


Figure 1.1: Synchronous and asynchronous mechanisms

Unfortunately, the complexity added by the local handshake protocols required in every asynchronous module makes asynchronous circuits more difficult and expensive to build in terms of circuit area and power consumption. For this reason, they have stayed a niche research area restricted to universities for many years, while synchronous methodologies and tools were developed at a high pace by companies. However, with the development of million-transistor chips, new difficulties have appeared with synchronous circuits which are more easily overcome by the asynchronous design style. The most important of these problems are:

- *Clock skew*: As the size of a chip and the number of modules increase, the clock network required to distribute the clock signal all over the chip becomes more complex. This leads to tiny differences in propagation delay known as clock skew, which can affect the correct synchronisation of the modules.
- *Non-adaptation to environmental conditions*: As environmental conditions (such as temperature, voltage, etc.) change, the propagation delays of the signals inside parts of the chip can become slower or faster. This further increases the effects of clock skew problems.

- *Electromagnetic interference:* In a synchronous circuit, all the modules get synchronised by the same clock events. This generates peaks of activity at every clock tick leading to electromagnetic interferences. In an asynchronous circuit, modules are only synchronised to their close neighbours. This tends to distribute electromagnetic emissions over time and therefore avoid the interference peaks.
- *Power management:* When a synchronous circuit has nothing to do, it can either let its clock circuitry work, thus consuming energy, or it can stop the clock in order to save this energy. Unfortunately, restarting stopped clock circuitry takes some time before stabilisation. This makes power management in synchronous chips complex. In asynchronous chips, when no processing is to be done, no activity happens at all, and thus no energy is consumed without the need for any special power management.

Noting the increasingly important qualities of asynchronous technology, a growing interest from the industry in asynchronous circuits is now stimulating research in design methodologies and the development of design tools which can be used to develop efficient asynchronous VLSI designs.

## 1.2 Automated Circuit Synthesis

Circuit synthesis refers to the process of transforming a problem description (usually in the form of a program) into a hardware circuit. Each step of the synthesis process can be made either manually or automated by using Computer Aided Design (CAD) tools. In the past, asynchronous CAD tools were limited to the design of individual small controllers [26, 41, 116], and thus were only useful for small portions of the final circuit. Some large scale asynchronous systems have been designed manually, allowing aggressive, hand-made optimisations, but at the expense of requiring a cumbersome, slow and error-prone design process. For example, the Amulet3 processor [46] took 20 man-years to complete and the Intel asynchronous instruction-length decoder chip [88] took 9 man-years, using a combination of manual techniques and academic synthesis tools for designing individual controllers. Synchronous hardware description languages (HDLs) and tools were able to handle parts of the design process, but could not tackle asynchronous specific problems such as fine-grained parallelism, deadlocks and non-determinism (see §2.1.4).

Nowadays, a few all-encompassing design methodologies and CAD tools able to handle the overall synthesis flow of asynchronous systems have emerged. The most scalable ones are syntax-directed tools [17] and can be used for VLSI designs. Other asynchronous complete CAD flows have been developed and used for large circuits such as the Caltech design methodology [115] and the NCL design flow [95, 109]. The syntax-directed model starts from a high-level abstraction, such as a concurrent program, and produces a circuit by translating each individual program construct into a corresponding sub-circuit. The proprietary Philips' *Tangram* tool [82], developed by van Berkel et al., and *Balsa* [4], developed at the University of Manchester follow this model. The work in this thesis is based on the Balsa system.

Balsa and Tangram have proven their ability to synthesise full VLSI circuits [6, 43]. However, these high-level asynchronous CAD tools suffer major deficiencies:

- lack of aggressive automatic optimisations [21, 99]
- lack of debugging capabilities.

The research work presented in this thesis deals with the last point.

## 1.3 Motivations and Objectives

The long-term aim of this work is to provide the Balsa CAD tool suite with a debugging system able to handle the peculiarities of asynchronous circuits.

The short-term motivation for this work was to help with the ongoing development of SPA, a synthesised ARM-compatible processor core designed at the University of Manchester and intended for use in third generation smartcards [84]. It is described entirely in Balsa, and the two last years of the SPA project overlapped with the two first years of this thesis. SPA is currently the largest circuit synthesised with Balsa, and is composed of about ten thousand asynchronous components, a huge number to deal with. (Making a comparison with the field of graph manipulations, graphs over a thousand nodes are considered huge). A part of the work presented here is the result of extending the capabilities of the Balsa suite with tools which are appropriate for the final validation

of SPA. In return, SPA has been used as a test case to evaluate the different aspects of this work.

The validation checks that the SPA processor passes all the 76 programs constituting the ARM architecture validation suite. In the execution of this validation suite, the most important aspect was simulation speed. The original simulation time of almost two weeks per test needed to be improved considerably. Moreover, decent simulation speed is the basis for a good debugging system as debugging information is collected during the simulation process. A fast simulator also allows repetitive tests on complex designs to be processed for design iterations and design-space exploration.

The design of a fast simulator led to the need for a tool able to trace back and pinpoint the causes of incorrect behaviour during simulation. A handshake circuit debugger was thus developed. The debugger was then extended for fixing not only problems with the simulator, but also the simulated Balsa descriptions. For this reason, the debugger was developed to recognise and treat special asynchronous failure modes such as deadlocks and non-determinism. The simulator was extended to report the information necessary for this analysis. Finally, as the size of Balsa circuits was becoming larger, a visualisation system appeared to be useful. It needed to display graphically all the information needed for the comprehension and debugging of the handshake circuit, linking all the different aspects of the circuit together: the pre-compilation Balsa code, the post-compilation handshake circuit, the dynamic simulation events, and the various results of simulation analysis such as control and data flows.

## **1.4 Contributions made by this work**

This work commenced with the idea of simulating Balsa directly at the handshake circuit level. Although this idea cannot be considered as a contribution by itself, it led to the following discoveries and achievements:

- Four orders of magnitude Balsa simulation speedup
- Ideas for debugging asynchronous-specific problems at the handshake circuit level identifying:
  - deadlocks



- non-determinism
- Pattern analysis of the out-of-order simulation trace, with direct applications to:
  - debugging of livelocks
  - compression of simulation traces
  - clustering for better visualisation
- Following the execution of a circuit for program comprehension:
  - Merging complementary sources of information together to generate a graph viewable at any level of detail
    - Colour-based graph animation to highlight handshake circuit control flows
    - Coordinating this graph view and the various views “well-known to the designer” together for an efficient element tracking and easy navigation
- The first published debugging environment for large scale asynchronous circuits.

## 1.5 Thesis Overview

Chapter 2 describes the fundamentals of asynchronous design and graph theory used in the rest of the thesis. Nomenclature and notations are introduced there. Balsa, the language and framework on which this research is based, is described.

Chapter 3 addresses other research works similar to that presented in this thesis, and exposes their differences, strengths and weaknesses.

Chapter 4 investigates the theoretical requirements for debugging asynchronous handshake circuits. Classical asynchronous problems of deadlocks, non-determinism and metastability are studied and applied to handshake circuits. Problems specific to large scale debugging are also examined, with some solutions proposed to solve the problem of the huge amount of data generated by the simulation of such systems.

Chapters 5 and 6 explore the design of a handshake circuit simulator, first by targeting high performance, then by taking into account the various requirements for gathering data for off-line analysis, visualisation and circuit debugging.

Chapter 7 examines the requirements for a visualisation system able to represent effectively the large amount of data, the fine-grained concurrency and the different steps of the synthesis process together.

Chapter 8 describes the integration of the debugging, simulation and visualisation aspects into a unified framework.

Chapter 9 presents an evaluation of the framework developed after the ideas uncovered by this thesis. It shows how the framework can be successfully employed on real-life examples.

Finally, Chapter 10 concludes the thesis by summarising the contributions and their deficiencies. Some guidelines for future possible work are suggested.

## **1.6 Publications**

The following papers have been presented at conferences:

- Debugging Tools for Asynchronous Design [58]  
(10th UK Asynchronous Forum, no peer review)
- A Visualisation System for Balsa Simulations [59]  
(12th UK Asynchronous Forum, no peer review)

The following journal paper has been published:

- Simulation and Visualisation of Asynchronous Circuits [60]  
(International Journal of Simulation: Systems, Science & Technology).

# Chapter 2: Background

This chapter provides an introduction to asynchronous design. The information presented here is intended to set the context for the description of the simulation system developed in the subsequent chapters.

First, the terminology and the major styles of asynchronous design are introduced. Then, the difficulties of metastability and deadlocks specific to asynchronous circuits are presented. The main asynchronous specification techniques are then described, concluding with a description of Balsa and handshake circuits, which form the basis of this work. Finally, graphs are introduced as a way to represent and manipulate handshake circuits.

## 2.1 Asynchronous Design

In asynchronous circuits, data is passed between modules using groups of wires, known as *channels*. These channels are unidirectional, point-to-point connections. Over the years, a number of different asynchronous channel implementations have been defined. In such channels, the data typically flows in one direction between two modules:

- The *sender* is the module that delivers data onto the channel.
- The *receiver* is the module that accepts data from the channel.

Orthogonal to this classification is the concept of *control flow*, determined by which end caused the transfer to occur:

- The *initiator* is the module that caused the transfer to begin.
- The *target* is the module that responds to the initiator.

A channel is connected to a module via a *port*. A port connected to an initiator is an *active* port while a port connected to a target is a *passive* port. As illustrated in Figure 2.1a and 2.1b, the graphical notation used to represent channels uses the filled and empty circles to denote the active and the passive partners in the handshake procedure respectively.

The relative direction of the data flow compared to the control flow determines whether the channel is classified as a *push* channel (where the sender is the initiator of the communication) or a *pull* channel (where the receiver is the initiator of the communication). These two types of channel are illustrated in figures 2.1c and 2.1d. Designers often speak of pushing or pulling data, thus implying the protocol used. A channel which does not transmit any data, and has therefore only a control part, is called a *sync* (or *nonput*) channel.

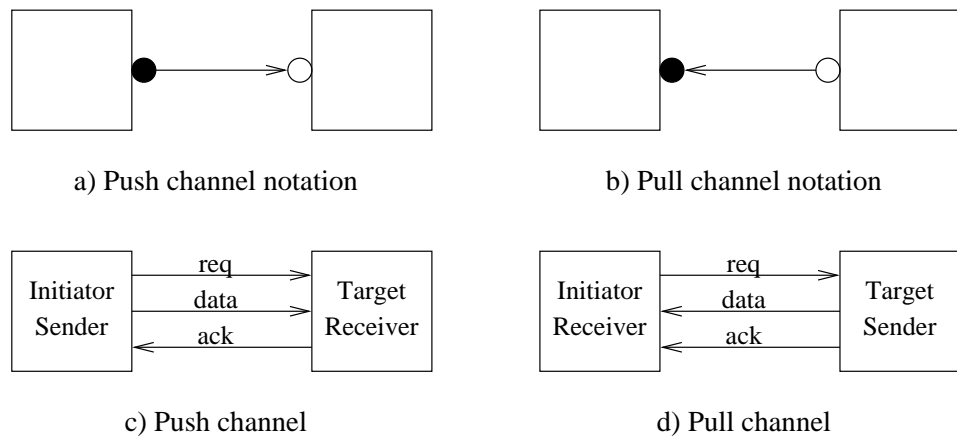


Figure 2.1: Push and pull channel notations

Which module performs which function on a channel is determined by the protocol and the transfer direction.

Most of the popular asynchronous handshaking approaches can be classified using the following three criteria, explained below:

- signalling protocol,
- delay model,
- data encoding.

### 2.1.1 Handshake Signalling Protocols

The transfer of information between two computation blocks across a channel is negotiated using a signalling protocol. Every transfer features a request action (req) where the initiator starts a transfer, and an acknowledge action (ack) allowing the target to respond. These may occur on dedicated signalling wires, or may be implicit in the data encoding used (as described below), but in either case, one event indicates data validity, and the other signals its acceptance and the readiness of the receiver to accept further data. These control signals carry all the necessary timing information to provide for proper data communication and can also be used as a mechanism for synchronising two modules without the explicit transfer of data (e.g. to implement token passing schemes or control shared resources).

The request and acknowledge may be passed using one of the two popular protocols described below: either a 2-phase transition signalling protocol (a non return-to-zero scheme) or a 4-phase level signalling protocol (a return-to-zero scheme). Conversion between the different protocols is possible [66].

#### 2-phase (transition) signalling

In the 2-phase signalling scheme, transitions on wires are interpreted as signalling events. The level of the signal is not used and a transition carries information with rising edges being equivalent to falling edges. A push channel using the 2-phase signalling protocol thus passes data using a request signal transition, and acknowledges its receipt with an acknowledge signal transition. Figures 2.2a and 2.2b illustrate the push and pull data validity schemes for the 2-phase signalling protocol. Arrows indicate causality between events.

Proponents of the 2-phase design style try to use the lack of a return-to-zero phase to achieve higher performance and lower power circuits.

#### 4-phase (level) signalling

The 4-phase signalling protocol uses the level of the signalling wires to indicate the validity of data and its acceptance by the receiver. When this signalling scheme is used to pass the request and acknowledge timing information on a channel, a return-to-zero phase

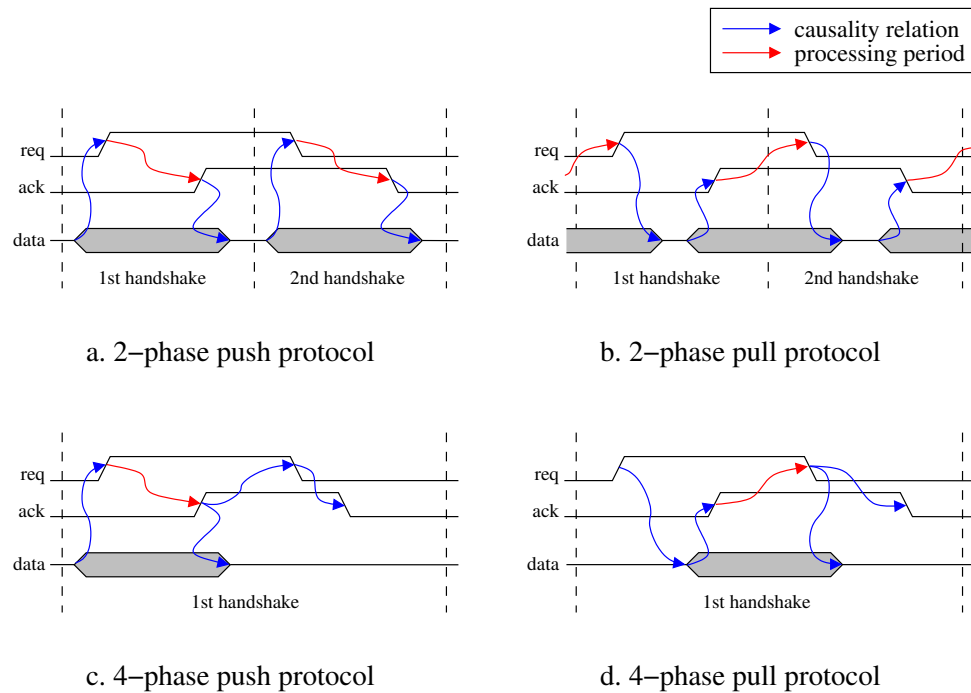


Figure 2.2: Channel signalling protocols

is necessary so that the channel signalling system ends up in the same state after a transfer as it was in before the transfer. This scheme therefore uses twice as many signalling edges per transfer than its 2-phase counterpart. Push and pull variants of the 4-phase signalling protocol are shown in figures 2.2c and 2.2d.

4-phase control circuits are often simpler than those of the equivalent 2-phase system because the signalling lines can be used to drive level-controlled latches, and the like, directly.

Data validity in 4-phase handshakes can be signalled in a number of ways, the most common of which (*broad*, *early* and *late* data validity) are illustrated with push channels in Figure 2.3.

### 2.1.2 Delay Models

Delays in digital circuits are associated with wires and gates. The effects of delays on systems are often characterised using a *delay model*. Delay models can be divided into

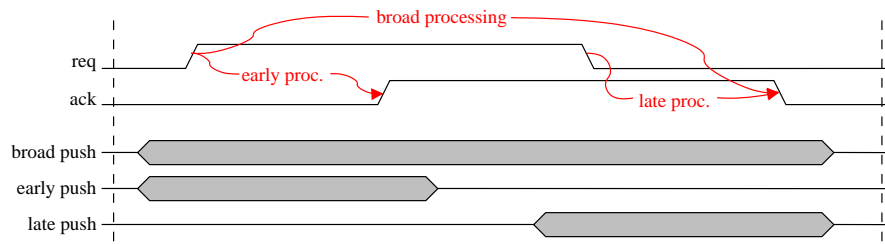


Figure 2.3: 4-phase push data validity schemes

three categories: fixed, bounded and unbounded delay models. In the fixed delay model, delays are assumed to have fixed values. In the bounded delay model, delays may have any value in a given interval. In the unbounded delay model, delays can have any finite value.

The fixed delay model is rarely used for the construction of asynchronous circuits since small variations in fixed delays could lead to significant differences in switching activity and break the model. The bounded delay model was commonly used in the early days of asynchronous design and is still used in some interconnection schemes such as the SCSI bus [94] where part of the protocol is based upon known, fixed delays allowing small variations. It is also commonly used for datapath components, where it can lead to smaller implementations. Delay variations are estimated by considering factors such as data dependence, statistical process variation, temperature and supply voltage variation. Current asynchronous VLSI designs and research efforts mostly use the unbounded delay model for the implementation of state-machines and controllers since it leads to circuits that will always operate correctly whatever the distribution of delays. This model separates delay management from the correctness issue, allowing the functionality of the circuit to be more easily verified.

Within the unbounded delay model, various design styles are commonly used, each with its own merits and problems. In order of increasing number of timing assumptions the major ones are:

**Delay-insensitive (DI) circuits**

A circuit whose operation is independent of the delays in both circuit elements (gates) and wires is said to be delay-insensitive. DI circuits require no timing constraints or assumptions to be preserved for circuit functionality to be guaranteed. Unfortunately, at the gate level, few interesting circuits conform to the delay-insensitive ideal [69]. For this reason, delay-insensitivity is most often applied to larger, more coarsely grained units, constructed using other timing regimes in order to make them easier to compose. This is the case with the handshake circuits used throughout this thesis.

**Quasi delay-insensitive (QDI) circuits**

If the difference between signal propagation delays in the branches of a set of interconnecting wires is negligible compared to the delays of the gates connected to these branches then the wires are said to form an isochronic fork [10]. Circuits created using the DI design style, augmented with the isochronic fork assumption, are said to be quasi delay-insensitive (QDI).

**Speed-independent (SI) circuits**

If wire delays in a circuit are assumed to be zero (or, in practice, less than the minimum gate delay), and the circuit exhibits correct operation regardless of the delays in any circuit elements, then the circuit is said to be speed-independent. The assumption of zero wire delay is valid for small circuits.

In general, SI circuits and QDI circuits are regarded as equivalent from the modelling point of view.

**2.1.3 Data Encodings**

A further dimension in asynchronous design is the choice of encoding scheme used for data representation where the designer must choose between a single-rail, dual-rail, or other more complex N-of-M schemes (any other encoding is possible, but these ones are the most popular ones). These alternatives are discussed in the following paragraphs.



**Single-rail encoding**

Single-rail encoding [82] uses one wire for each bit of information. The logical level of the signal represents either a logic 1 or a logic 0. This encoding is the same as that conventionally used in synchronous designs. Timing information is passed on separate request and acknowledge lines which allow the sender to indicate the availability of data and the receiver to indicate its readiness to accept more new data. This scheme is also known as the *bundled-data* approach. All single-rail encoding schemes contain inherent timing assumptions in that the delay in the signal line indicating data readiness must be no less than the delay in the corresponding data path.

Single-rail design is popular, mainly because its area requirements are similar to those of synchronous design, as is the construction of any arithmetic components using this scheme.

**Dual-rail encoding**

Dual-rail circuits [81] use two wires to represent each bit of information. Each transfer involves activity on only one of the two wires for each bit. A dual-rail circuit thus uses  $2 \times n$  signals to represent  $n$  bits of information. Timing information is implicit in the code, in that it is possible to determine when the entire data word is valid by detecting a level (for 4-phase signalling) or an event (for 2-phase signalling) on one of the two rails for every bit in the word. A separate signalling wire to convey data readiness is thus not necessary. The standard level-sensitive dual-rail data encoding technique uses the following four states for each bit of information:

- 00 – initial state, data is not valid
- 10 – transmission of a logical zero
- 01 – transmission of a logical one
- 11 – illegal state.

Once the data has been transmitted the wires must be returned to their initial state. And so, the presence of new data is indicated by a transition on one of the propagation wires. The illegal state is not used in dual-rail data encoding.

The major disadvantage of using dual-rail data representation compared to single-rail data encoding, where each wire represents one bit of binary information, is that its implementation requires twice as many wires and, as a consequence, leads to larger and more power-hungry circuits. Area overhead also comes from the large fan-in networks required to detect an event on each pair of wires in order to determine when the word is complete before being able to begin the next stage of processing.

### **N-of-M encoding**

N-of-M encodings are using groups of M wires to encode data values by considering a data valid as soon as N wires are activated.

Dual-rail encoding is an example of an N-of-M encoding scheme where  $N=1$  and  $M=2$ . Coded data systems using an N-of-M code operate correctly regardless of the distribution of delay in the wires or gates, and are thus delay-insensitive [111].

1-of-M codes are mostly used [3]. More complex codes, where  $N>1$ , use actions on more than one wire in a group to indicate one of a set of possible codes. These offer better utilisation of the available wires (for example a 2-of-7 code can transmit 4-bits of information over 7 wires in a delay-insensitive manner), but result in larger arithmetic circuits and conversion between the coded form and a single-rail scheme is more expensive than for the 1-of-M codes.

## **2.1.4 Asynchronous Difficulties**

### **Metastability and Arbitration**

Some asynchronous modules require their inputs to be mutually exclusive. For this, a special component is usually designed to provide *arbitration* between two contending asynchronous inputs. The basic circuit needed to deal with such situations is a mutual exclusion element (mutex) with two inputs and two outputs [91]. The role of the mutex is to pass the signals received on its two inputs to the corresponding outputs in such a way that at most one output is activated at any given time. If only one of the two inputs is activated, the mutex activates the corresponding output. If an input gets activated while the first input is already activated and its output selected, then the second input waits. A problem arises when both inputs are activated at the same time, or within a small time

window. The mutex' internal signals hover for an unbounded amount of time before reaching a stable state and selecting one of the outputs [27]. This problem is known as *metastability*, and the act of determining which event came first is called *arbitration*.

The condition for a mutex to go metastable is to have its two inputs activated within a small time window  $\Delta$ . The size of this time window can be determined by experiments or simulations, and a representative value for good circuit designs implemented with a 0.25 $\mu\text{m}$  fabrication process is  $\Delta = 30\text{ps}$  [96].

Once the mutex is in a metastable state, the probability of still being metastable at a given time  $t$  is:

$$P(\text{met}_t | \text{met}_{t=0}) = e^{-\frac{t}{\tau}}$$

where  $\tau$  expresses the ability of the mutex to exit the metastable state spontaneously, and  $P(\text{met}_t)$  is the probability of being metastable at time  $t$ . In the same conditions as for the time window, a representative value of  $\tau$  is  $\tau = 25\text{ps}$ .

The notions of non-determinism and race condition can also be defined. *Non-determinism* is when the next action of a system is not fully determined by its current state. This happens when a mutex arbitrarily chooses one of its outputs rather than the other. A *race condition* happens when two or more system entities potentially may be competing for resources (an arbiter component) at some time during execution.

### Deadlocks

A set of processes (which will be the handshake components of this thesis) is said to be *deadlocked* if each process in the set is waiting for an event that only another process in the set can cause.

A typical high-level example of an asynchronous processor system deadlock is as follow [96]:

1. A (non-sequential) data transfer needs access to a particular RAM block.

2. This is prevented because an instruction fetch is already using the RAM array.
3. The instruction fetch cannot complete because the instruction decoder is still busy.
4. The processor pipeline is full and is blocked by the data fetch.
5. Deadlock.

In contrast with synchronous circuits, working through deadlock problems during the design of asynchronous systems is very common. Without implicit global clock control, the control logic in an asynchronous design is more complex than in a synchronous equivalent design since each module of the design requires hardware to perform synchronisation, to wait for data, and to trigger other modules when it has produced its data. The use of explicit communications between modules increases the risk of introducing deadlocks. This problem can be introduced by design errors. Ideally, deadlocks should be detected and then avoided at a very early stage in the design process. Unfortunately, current formal validation techniques [7] cannot cope with large designs. Instead, designers use extensive simulation to give good confidence in design functionality.

## **2.2 Asynchronous Specification Techniques**

A number of specification techniques are available to the asynchronous designer. Those for which an automated synthesis route (automated translation from specification to hardware) is currently available are summarised here. For small-scale asynchronous designs, two classes of specification are commonly used: state-based and event-based. As these techniques do not scale well, they are usually used for the construction of small modules, which are then assembled together to create larger designs.

### **2.2.1 Event-Based Specification**

Petri nets [83] can be used to describe and study the behaviour of systems in terms of sequences of events, incorporating the concurrency and causality between the events.

Based upon the foundations laid by Rosenblum and Yakovlev [87], current event-based asynchronous circuit synthesis methodologies use interpreted Petri nets as the input specification with the transitions labelled with signal names [26]. These graphs are known

as Signal Transition Graphs (STG). In the STG notation, transitions describe signal activity and can model a rising signal, a falling signal or a change in level. Dependencies and causalities are represented in the STG using the notations shown in Figure 2.4. As an example of the STG specification style, Figure 2.5 shows the specification of a two input Muller C-element (which is a AND function for events, defined as follow: A transition will occur on the output only when there has been a transition on both inputs) with inputs  $a$  and  $b$  and output  $o$ . The dotted arcs show the behaviour of the circuit's environment, and the solid arcs show the behaviour of the circuit (the Muller C-element in this case). The use of Petri nets in circuit design makes it easy to describe systems which are concurrent at a very fine level. This kind of concurrent operation can be more difficult to express in state diagrams without drawing multiple diagrams or resorting to expanding the Cartesian product of the states of all the concurrent portions of a design.

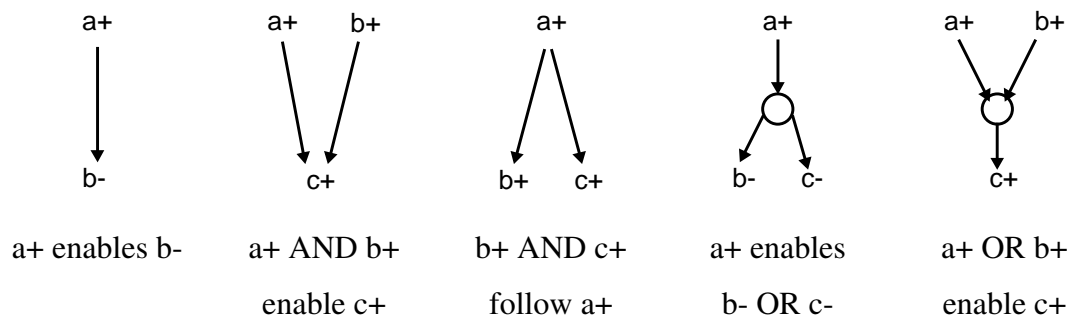


Figure 2.4: STG notation

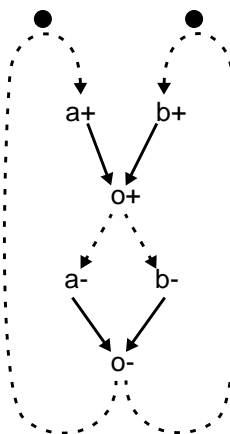


Figure 2.5: STG specification for a 2 input Muller C-element

Petri nets and state graphs also use different semantics of concurrency. The true concurrency semantics, used in Petri nets and their unfoldings, models concurrency by partial order of events, avoiding combinatorial explosion. However, the interleaving approach used in state graphs, models concurrency indirectly, or "sequentially", i.e. by means of all possible interleavings.

### **2.2.2 State-Based Specification**

Huffman state machines [54] are the classical asynchronous finite state machines. They follow the fundamental mode assumption in which the environment must wait long enough for the output data to stabilize on the circuit outputs (a bounded delay model), and the condition of only one input changing at a time. Burst-mode machines, as introduced by Stevens [98] and formalised by Nowick [79], and extended burst mode machines are a relaxation of the single input change. A design approach to building burst-mode finite state machines was proposed by Nowick et al. [80]. According to this approach:

- Each state transition can occur under a certain set of input changes (so called an input burst) so that no burst from a particular state can be a subset of another burst from the same state.
- Any state must be entered with the same set of input values.

The proposed timing mechanism allows the burst-mode finite state machine to be moved to a new state whenever the output associated with the previous state has changed, enabling the input signals to be changed. A burst-mode oriented backend for the Balsa synthesis system using the MINIMALIST tool [41] was proposed by Chelcea et al. [21].

### **2.2.3 Communicating Sequential Processes**

Communicating Sequential Processes (CSP) is a model developed by Hoare to describe concurrency by using parallel composition of processes communicating through channels [52].

Martin first suggested a synthesis method to manually compile a CSP description into DI circuits [68], where each process was synthesised as an asynchronous module, and where communications were following a handshake protocol. This work set the basis for a series

of macromodular synthesis methods, of which Balsa is an example. They are described in the following section.

### 2.2.4 Macromodules and DI Interconnect

*Macromodular* methodologies make use of pre-built modules connected together by handshaking channels to construct circuits [101]. This allows the use of a delay-insensitive style of interconnect between the modules, which is difficult to achieve at the gate level. The modules can have their own internal timing constraints (they can even be clock-driven modules) without affecting the high-level DI properties of the circuit. Other non-DI interconnection styles (e.g. single-rail protocols) can also be used in order to reduce the number of wires, but at the expense of extra-complexity in layout timing validation to ensure that bundling constraints are met.

Macromodular design styles exist for hand construction of circuits and for automated circuit synthesis. *Macromodules* were first developed by Clark at Washington University [24] during the late 1960s as a system for constructing large digital circuits which were composed of pre-built blocks of asynchronously communicating functional units: the macromodules. In his 1988 Turing Award lecture, Ivan Sutherland described an elegant approach to building elastic asynchronous pipelines called *micropipelines* [103], where the control structures were similar to those of macromodules. Micropipelines are an important design style similar to handshake circuits, but limited to the control part of the circuits. In 1989, Brunvand introduced his macromodular synthesis system [16, 17], making use of the channel-based, CSP-like programming language Occam [57, 106] to describe circuits. Descriptions are automatically synthesised into compositions of control, variable read/write and datapath macrocells implemented with 2-phase signalling with bundled data. Following this lead, *handshake circuits* were introduced by Van Berkel for use in the Tangram tool developed inside Philips [11]. The same scheme was later used with Balsa, developed at the University of Manchester by Bardsley [4].

Handshake circuits are macromodular circuits made of *handshake components* connected together via point-to-point asynchronous communication channels. Unlike micropipelines, which are only implementing control components, handshake circuits are composed of both control and data handshake components. In Balsa, these handshake

components are chosen from a cell library of about 45 predefined components parameterisable to a limited degree. Handshake circuits and components are described in the next section with the Balsa framework, and used extensively throughout this thesis.

## 2.3 Balsa

*Balsa* refers to both a framework for synthesising asynchronous circuits and the language for describing such circuits.

The Balsa system uses the *handshake circuits* macromodule-based design paradigm as an intermediate representation for synthesising Balsa designs. Handshake circuits are compiled from specifications in the Balsa language by a syntax-directed compilation scheme. The advantage of this approach is that of a transparent compilation: There is a one-to-one mapping between the language constructs in the specification and the handshake circuits that are produced. It is relatively easy for an experienced user to deduce the arrangement of the circuit that results from the original description. Moreover, incremental changes made at the language level result in predictable changes at the circuit implementation level. This is important if optimisations and design trade-offs are to be made easily at the source level and contrasts with, for example, a VHDL description in which small changes in the specification may make radical alterations to the resulting circuit.

The description of Balsa presented in this section is intended to set the context for the handshake circuit simulation-visualisation-debugging system developed in the rest of the thesis. For this reason, this description is mainly focused on the handshake circuits, which are currently used as an intermediate format in the Balsa synthesis flow. This section starts with an overview of the complete Balsa framework and design flow, followed by a presentation of the main characteristics of the Balsa language. Handshake circuits are then described. The set of macromodules (the handshake components) specific to Balsa, very useful for a deeper understanding of the contents of this thesis, is described in detail in Appendix B.

For further information on the other aspects of Balsa, a complete explanation of Balsa synthesis, including the compilation from specification to handshake circuit and the



transformation from handshake circuit to hardware, can be found in [4, 5]. A more complete description of the Balsa language can be found in the Balsa User Manual [32].

### 2.3.1 Balsa Framework

The Balsa framework is a set of tools designed for the synthesis of a Balsa language description into an asynchronous hardware circuit, with an intermediate representation using handshake circuits. As shown in Figure 2.6, the Balsa description is first compiled into a handshake circuit by the Balsa compiler ‘balsa-c’. This handshake circuit can then be transformed into a gate-level netlist by the ‘balsa-netlist’ tool. The same handshake circuit can also be used as a source for the ‘breeze2ps’ and ‘breeze-cost’ tools, and for the simulation system. The two tools generate a PostScript file of the handshake circuit graph and an area cost estimate of the circuit respectively, which can be seen as a static analysis of the handshake circuit.

Three design loops are shown in the design flow. These design loops use behavioural, gate level (functional) and final layout (timing accurate) simulation tools respectively to allow simulation to be performed at different points in the design flow. In a typical design, all three levels of simulation complexity are used as the design approaches its final version.

### 2.3.2 Balsa Language

The Balsa language was created as a source language for compiling handshake circuits, with the ambition of satisfying the *directness* property. *Directness* is the ability to map constructs of the input description to modules of the hardware implementation on a one-to-one basis. It allows the user to deduce the arrangement of the final circuit at the time of writing the original source description.

The Balsa language (following the Tangram model [9]) was designed as an imperative, channel-oriented language based on CSP (i.e. with parallel composition and channels). Its transparent syntax-directed compilation to handshake circuits satisfies directness.

Directness is preserved at every step of the synthesis process, so that the final hardware circuit has a one-to-one mapping with the original description’s structure. In the Balsa synthesis flow, the compilation process consists of two stages: A Balsa specification is

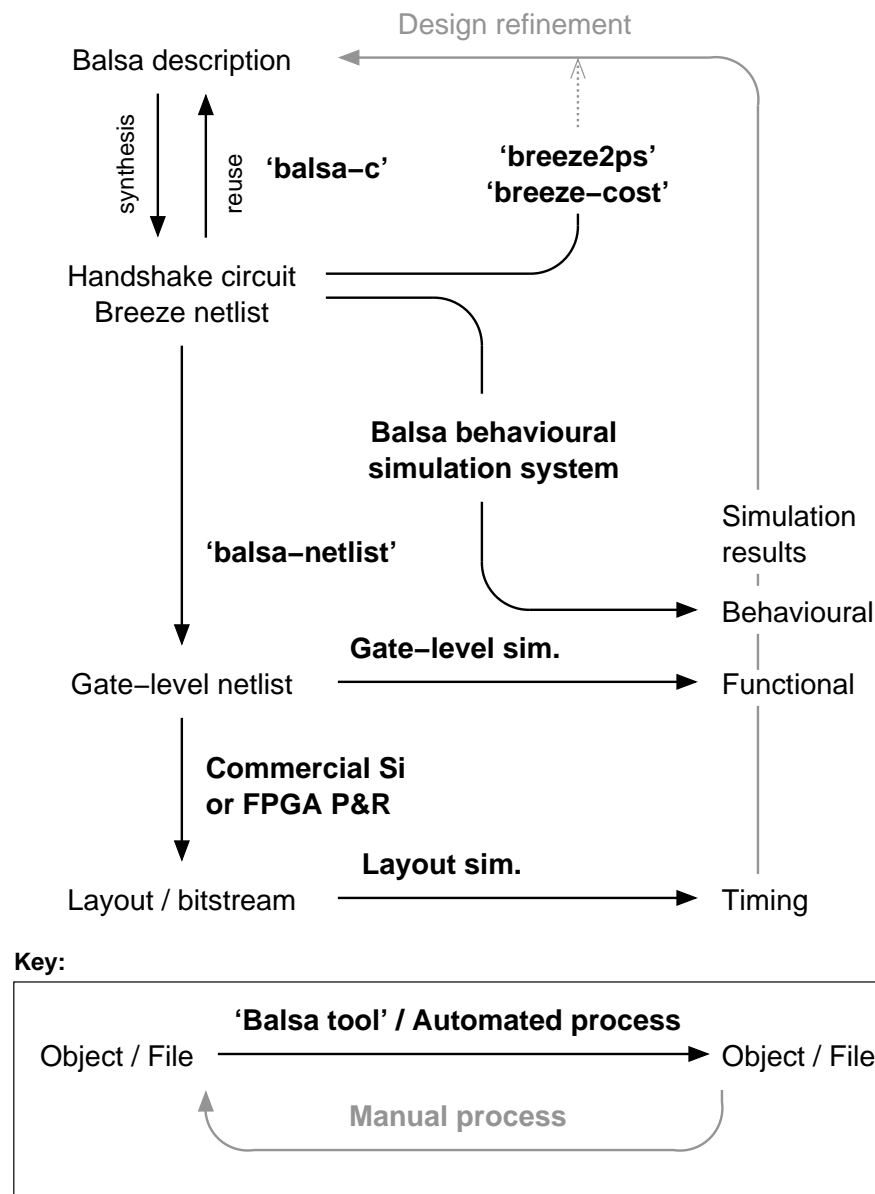


Figure 2.6: Balsa design flow

compiled into a handshake circuit, which is then transformed into an asynchronous hardware circuit. This second stage of the synthesis, as described in [4], is direct, as each macromodule making the handshake circuit is mapped to a specific piece of hardware. The only threat to directness happens during optimisation of the resulting circuit, but the Balsa tools also try to preserve directness for this operation by only using keyhole optimisations [29].

The modulo-10 counter example shown in Figure 2.7 illustrates most of the Balsa features embedded in the language:

```
import [balsa.types.basic] -- 1

type C_size is nibble -- 2
constant max_count = 9

procedure mod10 (sync aclk; output count: C_size) is -- 3
local -- 4
    variable count_reg : C_size
    variable tmp : C_size
begin
    loop -- 5
        select aclk then -- 6
            if count_reg /= max_count then -- 7
                tmp := (count_reg + 1 as C_size) -- 8
            else
                tmp := 0
            end ; -- 9
            count <- count_reg ; -- 10
            count_reg := tmp
        end
    end
end
```

Figure 2.7: Balsa language features (modulo-10 counter example)

1. Pre-compiled module inclusion. In this case `[balsa.types.basic]` defines some common types: byte, nibble, etc.
2. Type and constant declarations.
3. Procedure declaration with sync and output ports.
4. Local variables/latches.
5. Infinite repetition with `loop ... end`. Once activated a loop never terminates.
6. Passive input enclosure using `select`. The commands inside the `select` are enclosed in the handshake on `aclk`, `aclk` is effectively the activation for these commands.
7. `if ... then ... else ... end` statements.
8. Assignment, expressions and type casting.
9. Sequential composition with `;`. In the same manner, `||` specifies parallel composition

at a very fine-grained level.

**10. Output synchronising communication.** The notation for input communications is the other way around: `channel -> variable`.

A few other important features of Balsa not illustrated in this example are:

- *Structural iterations:* *for* loop constructs can be used for repeating a block's implementation.
- *Parameterised procedures:* Procedures can be declared with parameters evaluated during compilation.
- *Conditional ports and declarations:* Instantiation of procedure ports and declarations can depend on parameters evaluated during compilation.
- *Recursive procedures:* Recursive calls to parameterised procedures can be used to describe scalable circuits.
- *Shared procedures:* Procedures can be declared to be implemented as a unique circuit and be treated as a resource shared by callers.
- *Nested procedures:* Local procedures can be declared inside procedures.

### 2.3.3 Breeze Handshake Circuit

One of the major contributions of the Tangram project (which Balsa originates from) was the development of handshake circuits, an intermediate representation which both:

- abstracts low-level technology-specific details, and
- supports an elegant translation to asynchronous hardware.

A handshake circuit is built by composing a set of primitive handshake components to form a graph. Handshake circuits combine macromodular design style with delay-insensitive communications to produce a design methodology in which entire designs are described using macromodules connected together by asynchronous communication channels. Each handshake circuit instance is taken from a macromodule library cell and may be parameterised to a limited degree. There are only a small number of such cells defined in a particular handshake component set.

Balsa is made of a similar set of handshake components to Tangram. Balsa’s handshake components are called *Breeze components*, and circuits made of these components are referred to as *Breeze handshake circuits*.

A handshake component is the abstraction of an electronic circuit able to perform a specific computation. Depending on the level of abstraction desired, the component’s internals can either be shadowed as in a black box or revealed to the user. In this work, the handshake components’ internals are hidden from the outside world, and only the behavioural and timing characteristics are exposed.

*Important note:* See Appendix B for handshake component descriptions.

### Handshake Circuit Graph

Figure 2.8 shows (in black) the graph of a simple Breeze handshake circuit composed of handshake components linked by channels. This handshake circuit was obtained by compilation of the “modulo-10 counter” example shown in Figure 2.7. Some of the Balsa features enumerated in the previous section (the red numbers match the numbered comments in Figure 2.7) and illustrated at the language level are highlighted here on the graph of the handshake circuit (in red). This illustrates the directness of the compilation.

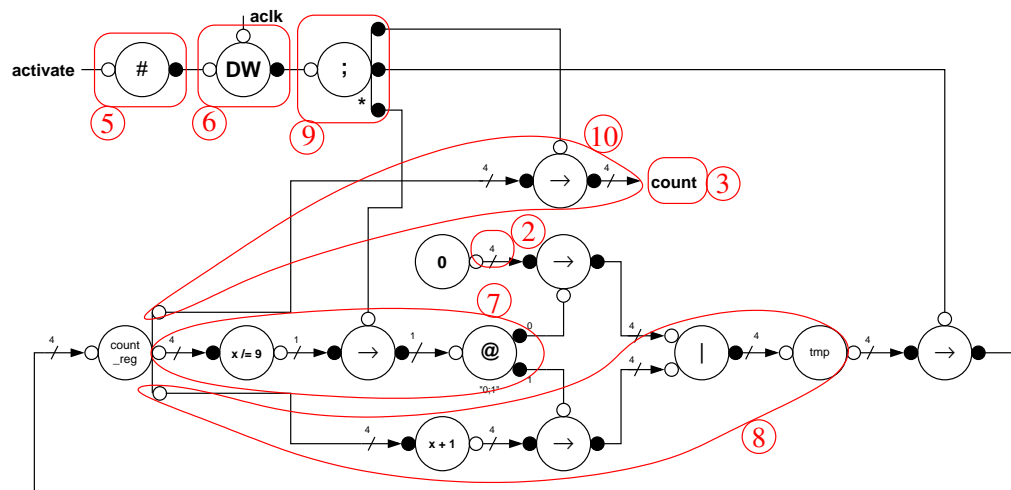


Figure 2.8: Breeze handshake circuit graph (modulo-10 counter example)

The circuit is first activated by a request signal on the *activate* channel (implicitly declared in every Balsa procedure), at the top left of the figure. This signal activates the Loop component, whose abbreviation on the graph is ‘#’ (see the list of abbreviations at the end of Appendix B) and whose behaviour is to endlessly activate the next component it is connected to. Notice that this behaviour corresponds to the *loop* construct in the original source program. The next component is a DecisionWait, which corresponds to the next line in the Balsa source code: waiting for *aclk* before continuing. The rest of the circuit continues in the same obvious translation scheme with a Sequence component and others.

Each handshake component has one or more ports through which it can be connected point-to-point to a port of another handshake component by a channel. Channels are following the specification given in §2.1, with requests flowing from the active component ports (filled circles) towards passive component ports (empty circles). Acknowledgements flow in the opposite direction to requests in the usual fashion. Where a channel carries data, the direction of that data is indicated by an arrow on that channel’s arc. The direction of data may be different from the direction of signalling to support push and pull ports and channels. The data width in bits is reported on top of the channel’s arc.

### Breeze Netlist

Figure 2.9 shows the Breeze netlist description of the handshake circuit presented in Figure 2.8. It starts with some comments, followed by global declarations for import paths and type declarations, similar to the ones declared in the original Balsa description. Each procedure of the Balsa source code is then described in the Breeze netlist as a *part* with ports, attributes, and two lists of handshake channels and handshake components. In this figure, the lists of channels and components have been truncated to show only the description of the first four components of the circuit (from the top of Figure 2.8). References to positions in the original Balsa file are available for each channel, and each component is described with a list of the channels it is connected to.

This description of the Breeze format corresponds to what it was before this work. It has since been extended, as described later in the thesis.

```

;;; Breeze intermediate file (list format)
;;; Created: Wed Aug 18 18:52:40 2004
;;; By: janinl@jabez.cs.man.ac.uk (Linux)
;;; With balsa-c version: 3.4
;;; Command: balsa-c -b -I . mod10

;;; Imports
(import "balsa.types.builtin")
(import "balsa.types.synthesis")
(import "balsa.types.basic")
;;; Types
(type "C_size" (numeric-type #f 4))
;;; Constants
(constant "max_count" 9 (numeric-type #f 4))

;;; Parts
(breeze-part "mod10"
  (ports
    (sync-port "activate" passive (at 6 1 "mod10.balsa" 0))
    (sync-port "aclk" passive (at 6 18 "mod10.balsa" 0))
    (port "count" active output (numeric-type #f 4) (at 6 41
"mod10.balsa" 0))
  )
  (attributes
    (is-procedure)
    (is-permanent)
    (at 6 1 "mod10.balsa" 0)
  )
  (channels
    (sync (at 11 2 "mod10.balsa")) ; 1
    (sync (at 12 10 "mod10.balsa") (name "aclk")) ; 2
    (push 4 (at 18 4 "mod10.balsa") (name "count")) ; 3
    (sync (at 19 14 "mod10.balsa")) ; 4
    (sync (at 18 10 "mod10.balsa")) ; 5
    (sync (at 13 4 "mod10.balsa")) ; 6
    (sync (at 17 8 "mod10.balsa")) ; 7
    (sync (at 12 3 "mod10.balsa")) ; 8
    (pull 4 (at 18 13 "mod10.balsa") (name "count_reg")) ; 9
    ;;; ... omitting some channel declarations
  )
  (components
    (component "$BrzLoop" () (1 8))
    (component "$BrzDecisionWait" (1) (8 (2) (7)))
    (component "$BrzSequence" (3) (7 (6 5 4)))
    (component "$BrzFetch" (4) (5 9 3))
    ;;; ... omitting some component declarations
  )
)

```

Figure 2.9: Breeze handshake circuit netlist (modulo-10 counter example)

## 2.4 Network Graphs

Handshake circuits are represented using network graphs. A network graph, as shown in Figure 2.10, is a collection of points, called *vertices*, and a collection of arcs, called *edges*, connecting these points. Network graphs are usually simply called *networks* or *graphs*. The latter designation is used in this thesis.

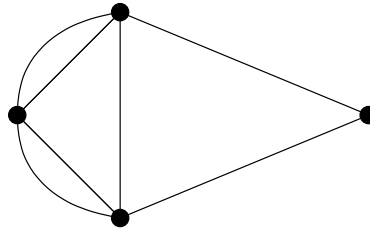


Figure 2.10: Network graph

Graphs can be extended to *control and data flow graphs* (CDFG) in order to represent and manipulate handshake circuits. In the CDFG model, edges are able to convey control events and data values. Vertices are representing processes able to respond to these events and manipulate the data values.

### Petri nets

Another graph-based model able to describe asynchronous circuits is Petri nets (see §2.2.1). Although this model is only used to describe small circuits, its graph-based representation makes it a good candidate for being compared to handshake circuit graphs, particularly in the next section where related work is explored.

Some extensions to the original model of Petri nets have been developed: The notion of delays in components can be modelled with timed Petri nets. Coloured Petri nets can be used to associate complex data structures to the communications [62]. Finally, Hierarchical Petri nets can model hierarchical systems. Although these extended models are not used for asynchronous circuit synthesis, their simulation and visualisation have some similarities with this work.



## Chapter 3: Related Work

This chapter reviews the research studies related to the one described in this thesis. This related work is partitioned into the aspects of handshake circuit simulation, debugging of asynchronous-specific problems and visualisation oriented towards program comprehension.

Handshake circuits themselves are very rarely used in the literature. It is therefore rare to find simulation, debugging and visualisation systems precisely devoted to them. That is one of the novelties of this thesis. However, the notion of handshake circuits can be extended to their origins – the CSP style of description – or to equivalent structures such as control and data flow graphs and hierarchical, timed, and coloured Petri nets. They can be further extended to other VLSI description systems and their associated hardware description languages.

### 3.1 Handshake Circuit Simulation

The simulation of a handshake circuit generated from a Balsa description happens at the behavioural level. Other HDL simulators can simulate both behavioural and structural code in VHDL and Verilog [19, 55, 70, 72], and the synthesisable subsets of these languages (or the whole language in the case of Balsa) can be implemented into detailed (lower-level) transistor-level and gate-level structures. These can be simulated by appropriate simulators [71, 90]. The lower the description level, the more precise and slow the simulation. The lower level simulators are therefore designed for different purposes than high level simulators. Then it does not make sense to compare a handshake circuit simulation to a transistor-level simulation. However, VHDL and Verilog simulations are processed at a close enough level to that of a handshake circuit to be comparable. This is particularly true for interpreted Verilog simulations. Meyer and

Vanvick, authors of the Cver Verilog simulator [72], explain that, originally, intermediate interpreted forms were so close to source that Verilog constructs could be reconstructed and printed from the interpreted data structure. This process has some similarities with the one-to-one syntax-directed translation of the Balsa programs into handshake circuits. However, the structure generated from Verilog requires a complex interpretation, while handshake circuits benefit from their very simple graph structure.

Other description methods have been created to describe systems at a higher level. A very interesting example is ARCS, an architectural level communication driven simulator designed by Nellans et al. [76]. ARCS is based on a CSP style of description and is used to describe systems at a high level by using the abstraction of communication as the fundamental unit of simulation. Because communication is essentially asynchronous at the architectural level, the simulator's architecture is almost identical to the one described in this thesis. Indeed, the components described as parts of the system are independent from each other, running concurrently, and are connected together by point-to-point CSP-style communication channels, providing local synchronisation. As in Balsa, the timing model is decoupled from the functional aspect and is made of simple delays associated to each component, independently of the complexity of the data being processed. A few differences are however distinguishable. First, although the scheduler seems to rely on a delay-insensitive behaviour of the system, it is unclear from the paper how ARCS handles the arbitration problem (as described in §2.1.4). Then, each component is modelled as a Java thread, implemented as a native Linux process. This transfers the scheduling process to the operating system's own scheduler. This is an advantage on multi-processor computers, where a single simulation can be automatically distributed between processors and properly load-balanced. However, this technique may create a very high number of processes, which would be a performance bottleneck in a non-distributed environment. This restricts scalability and performance, which are the main objectives of the Balsa simulator's scheduler described in this thesis.

Petri net (PN) simulators are comparable to handshake circuit (HSC) simulators due to their similar graph structure. PN and HSC simulators are similar in the way they handle control flows: Tokens are transmitted from places to transitions in the same way as events are transmitted between handshake components. However, HSCs can associate complex

data structures to events and can model delays. This is also feasible by using coloured and timed PN. A coloured PN simulator was originally described by Jensen [62], and recently optimised by Haagh and Hansen to handle larger descriptions [49]. However, coloured tokens require some more complex communication strategies such as *binding* and *priority queues* [49] at the simulator level, which are not necessary in the context of HSCs. Event-driven simulators of timed PN models have been written [118], equivalent to the standard version of the HSC simulator described in this thesis. Although distributed versions of the timed PN simulator have been designed [39], no optimisation of the single-threaded event-driven simulator, equivalent to the optimisations described in this thesis, have been suggested.

The closest research to this thesis work has been described ten years ago by Sutherland in [104] as *Flashback Simulation*. His description was dealing with micropipeline components, limited to the control part of the circuit. He also described an interesting analysis for reordering the arbitration inconsistencies. This thesis follows the same ideas and tries to extend them to handshake circuits and to other reordering methods.

## 3.2 Debugging Asynchronous-Specific Problems

Debugging problems such as deadlocks and non-determinism are often addressed by formal verification tools. In this domain, Petri net analysis methods and tools are well established. Verification tools can check for deadlock freeness and locate specific deadlock states [86]. However, these tools offer limited help to remove the detected deadlocks. This is explained by the fact that deadlocks are often easy to understand when they occur in small circuits: The simulation stops and it is then possible to observe where each thread was interrupted. However, this is not as easy in larger environments involving many threads.

Formal analysis methods on Petri nets also handle non-determinism. Small Petri nets can be analysed through their *reachability graph*, which contains a node for each possible state and an arc for each possible state transition. This technique is a very powerful analysis method, as it covers all the possible states. However, the reachability graph may, even for small sets, become very large. This is therefore not applicable in the case of large

circuits. Another technique, branching processes, analyses Petri net unfoldings, where it is easy to see non-determinism in larger circuits [67].

Non-determinism is also studied for detecting race conditions during the execution or simulation of distributed applications. An application tracing multiple execution paths in the case of non-deterministic behaviour, in order to detect race conditions, is described by Kamada [63]. It applies to fine-grain threads on massively parallel processors, and the debugging scheme used by this application is based on a replay facility requiring a limited amount of log information. Neri et al. describe a similar technique for ADA-based programs [77] for debugging distributed applications by using replay capabilities.

Although not employed directly for every type of analysis, other techniques for reducing the amount of traced data have been studied [1, 81, 108]. These trace compression techniques rely on pattern analysis for clustering repeated patterns. In this thesis, a straightforward technique for pattern analysis of the traced data is presented, with similar applications.

## 3.3 Visualisation Oriented Towards Program Comprehension

A variety of techniques have been suggested to assist programmers in the difficult task of program comprehension. One of these techniques, *reverse engineering*, is the process of extracting and synthesising high-level design information from a specific source (source code, compiled program, bytecode, etc.). A reverse engineer analyses the source in order to identify system components and their inter-relationships, and creates representations of the system in another form, usually at a higher level of abstraction [23]. Tilley et al. identified three basic sets of activities that are characteristic of the reverse engineering process [107]:

- *Data gathering* through static analysis of the source code or through dynamic analysis of the executing program.
- *Knowledge organisation* by organising the raw data by creating abstractions for efficient storage and retrieval.
- *Information exploration* through navigation, analysis and presentation.

According to Tilley, the exploration of software information “holds the key to program understanding”. For this reason, most of the literature found about program comprehension is focused on this last point, starting from pre-formatted information and developing methods to visualise it effectively. Fewer studies explain the process of gathering and formatting the data.

In this thesis, all three steps of the reverse engineering process are treated. Program comprehension is based on the analysis of three sources of information:

- the source code needing to be reverse engineered,
- the code compiled into a meaningful electronic circuit, and
- the dynamic information contained in the simulation trace.

This information is collected and merged to be visualised as a coherent whole in the form of a graph.

The static visualisation process is not treated in this thesis, since significant research has already been done on this subject [8, 40, 50, 102]. We decided to leave this point as a further work and use a simple layout method already implemented [44].

After the process of knowledge organisation from multiple sources, two techniques are used to represent the data effectively. First, dynamic visualisation of the execution (or simulation) of the circuit is applied atop the static graph. Then, a multiple view technique where views can communicate makes it possible to display various aspects of program execution simultaneously.

#### **3.3.1 Knowledge Organisation by Merging Multiple Sources**

A limited number of studies describe the process they employ for gathering and merging multiple sources of information for subsequent visualisation.

The most complete studies about data analysis and merging from multiple sources are those from feature analysis. Eisenbarth et al. [33] define a *feature* as a realised functional

requirement. In order to provide an understanding of how a feature is implemented in a system, they build a mapping between the system's externally visible behaviour and the relevant parts of the source code by combining dynamic and static analyses. In another study [22], conservative static analysis would yield an overestimated search space. By adding a dynamic analysis of the execution trace of the system, Chen et al. reduce this search space by considering only parts really used at runtime – though only for a particular run. Multiple dynamic analyses are used by Wilde and Scully [113] for another method of feature localisation, by comparing dynamic analyses of test cases which invoke the feature to test cases which do not. Although very interesting for data organisation, these techniques have not been applied to visualisation.

Although less complete than the previously reviewed studies for feature analysis, research on data gathering and knowledge organisation for data clustering and multiscale graph visualisation is even closer to the subject treated here. Clustering is the process of discovering groupings or classes in data, based on a chosen semantics. Clustering techniques have been referred to in the literature as *cluster analysis*, *grouping*, *clumping*, *classification*, and *unsupervised pattern recognition* [36, 74]. Two forms of clustering can be distinguished: *structure-based* clustering, which refers to clustering that uses only structural information about the graph, and *content-based* clustering, which uses the semantic data associated with the graph elements to perform clustering. An advantage of structure-based clustering is that clusters retain the structure of the original graph, which can be useful for user orientation in the graph itself. However, this class of methods often leads to the clustering of elements poorly related in their properties. Content-based clustering can yield groupings which are more appropriate for a particular application by using application-specific data and knowledge [75, 85].

By far the most common clustering approach in graph visualisation is to find clusters that are disjoint or mutually exclusive, as opposed to clusters that overlap (found by a process called *clumping*). Disjoint clusters are simpler to navigate than overlapping clusters because a visit of the clusters only visits the members once. It should be noted, however, that it is not always possible to find disjoint clusters, for instance in the case of language-oriented or semantic topologies [51]. This is one of the strengths of the techniques presented here using Balsa.

If clustering is performed by recursively applying the same clustering process to groups discovered by a previous clustering operation, the process is referred to as *hierarchical clustering* [74]. Hierarchical clustering can be used to induce a hierarchy in a graph structure that might not otherwise have a hierarchical structure. Michaud et al. have described Shrimp [73], which gathers various information sources together (source code artifacts and relationships, architectural abstractions, documentation and history information, metrics and analysis information) and visualise them together. The difference between their work and the one presented here is that they limit their study to static information, while dynamic simulation data is also used here to deduce some clusters. Hierarchical clustering is applied here to the originally flat handshake circuit.

#### **Studies for VLSI design**

Storey [100] defines a visualisation as *coherent* if the maintainer can construct from the given visualisation a mental model which corresponds to something in the real world. The advantage of visualising reverse engineered code of a VLSI design is that the code describes something concrete, the electronic circuit being built. It is then possible to use this real-world structure as an underlying base, and either transfer this structure onto the other structures being visualised, or use the other sources of information to reshape (for example by clustering) the real-world structure. No studies have been found that exploit this possibility with graphs. The visualisation system described in this thesis exploits this idea by using the to-be-built handshake circuit structure as the real-world structure onto which every other source is mapped.

#### **3.3.2 Dynamic Visualisation**

The visualisation system presented in this thesis is unique in the sense that it combines techniques not usually used together: software visualisation (Balsa is similar to classical high-level programming languages), animation on top of a network graph (handshake circuits are network graphs), and VLSI circuit simulation visualisation (handshake circuits are an intermediate form towards synthesis). These techniques are usually studied separately. They are reviewed in this section.

### **Software Visualisation**

There are two categories of dynamic software visualisation techniques: Algorithm visualisation techniques and program visualisation techniques. Algorithm visualisations [14, 15, 97] are usually hand-crafted (some instrumentation code needs to be added to the original description to drive the visualisation) and require the designer to understand the code before visualising it, making this technique infeasible for large systems or tasks involving program discovery. For this reason, such instrumented techniques are avoided in this thesis. Program visualisation techniques can be non-instrumented. In this case, they require an automated analysis of the program that deduces the elements needing to be visualised. These techniques may be scalable and used with large concurrent systems to show their dynamic execution for debugging, profiling and for understanding runtime behaviour. Systems classified in this category usually display program control and/or data structures as figures in order to facilitate the understanding of control flows and changes in data values. Sometimes, visualisation is restricted to communications between threads.

Several program visualisation techniques have been studied that are not scalable due to the way they represent elements [65, 97]. The recurring problem is that each item (process, communication link) takes a fixed amount of space on the screen, and that no clustering method is used to group these items together and visualise the system at a higher level. Other scalable visualisation systems are based on graphs, for which many techniques of clustering and hierarchical visualisation are available.

#### **Visualisation of dynamic information atop a static graph**

Dynamic representation of information on graphs is treated in this section. Some dynamic information explorers are based on dynamic changes of the graph structure, visualised by animating the transition from one layout to the next [53]. Some others – that we are interested in – keep the same graph structure and animate the change of state of elements in the graph. In this case, the computational-intensive layout stage is needed only once. The preferred method is to represent states of the graph elements with different colours [12], but other characteristics of the graph's edges and vertices can also be used to represent states: thickness of the lines, transparency, symbols, sizes (up to a certain extent, in order to keep the same graph layout). Colour-coding can be used to represent clusters either at a static level [2, 18] or for displaying how data evolves in time [78].



Basic understanding of a program's execution goes through the control flow of the program. That's how people first learn how to program. However, object-oriented and parallel designs make control flow less obvious and more complex than in sequential programs. De Pauw et al. [81] tackle this problem by suggesting a method for visualising the execution patterns of object-oriented programs with time as an x-axis. However, using the x-axis on a two-dimensional visualisation medium (computer screen) leaves only one dimension for organising the program elements, leading to some related elements being shown far from each other. Another method is to use a static graph layout to organise the program elements in two dimensions, and show the state of these elements at a specified, modifiable, instant of time [64]. Kraemer explains: "Animated displays are useful for conveying information regarding concurrency. They employ the very natural mapping of time to time, rather than the less natural time to space mapping. Events that were concurrent in the program can be shown as concurrent in the display". In this thesis, dynamic information issued from the simulation trace is visualised atop a static graph by using a colour-based animation to represent events. The visualisation of the control flows atop the static graph is intended to render efficiently the fine-grained concurrency happening in the visualised handshake circuit.

#### **Visualisation of asynchronous circuit activity**

The first attempt of visualisation of asynchronous circuit activity for Balsa was implemented on top of LARD (Language for Asynchronous Research and Development) [34], the language previously used in the simulation route of Balsa. It was possible to draw asynchronous blocks on a graphical view and change their colours when they are activated. However, this required some instrumentation code added to the original description, and it only worked for a very small number of visualised elements.

The most interesting projects are again concerning Petri nets. The MOVIE project [67], developed at the University of Newcastle, was aimed at improving visualisation support within the logic synthesis environment based on Petri nets and Signal Transition Graphs (STGs). This project investigated the development of techniques and tools which could expose the highly concurrent behavioural patterns to the designer but in such a way that he would be able to easily grasp the characteristic patterns of circuit behaviour and manipulate the model more interactively. This required better ways of visualisation of

STGs, Petri nets, and others types of graphs. Their first objective was aimed at identifying a key set of models, *based on graphs*, to represent the concurrent behaviour in circuits. State graphs were chosen for this purpose. However, while state-based models play an important role in logic synthesis of circuits in tools like Petrify, their ability to visualise asynchronous circuit behaviour is limited, mainly due to the state explosion for highly concurrent models.

Visual STG Lab [112], a visual environment for use with Petrify, is intended to improve the overall design flow for people who work with signal transition graphs. VSTGL is a graphical interface for designing and simulating signal transition graphs able to simulate the network by running Petrify directly on the graph and dynamically visualising the simulation events.

Another visualisation project, very similar to the graph-based visualisation of this thesis, is Rainbow [7]. Rainbow is a prototype hardware design framework based on Sutherland's micropipeline design style. Rainbow's Green tool is based on hierarchical structural descriptions using the micropipeline primitives and using a dataflow description style. It has schematic and textual versions, and the schematic version can be animated to follow the control and data flows inside the circuit.

The drawback of all these visualisation systems is that they are only usable to represent small asynchronous circuits.

#### **3.3.3 Information Exploration with Coordinated Views**

In VLSI design, designers are used to working with raw data: source files opened in a standard editor, execution results displayed as wave forms, etc. In the literature, some design environments are built to help designers by keeping these views they are used to and by gathering them together with, possibly, some extra functionalities.

Favre [37] describes such a collection of views representing the same information in different manners. In this application, a component-based software is represented by three techniques: First, by representing the network graph of its components. Then, by displaying a list structure containing the object-oriented hierarchical structure. And

finally, by showing the source code. Other views are also available to display either the internal representation or the external representation of a selected component. The drawback of this study is the lack of communication between views, restricted to the selection of a component.

When dynamic information is visualised and various views are representing the program's state for a given timestep, these views need to be synchronised to represent the same timestep. The Vista architecture [110] is an example of such an organisation where every view is connected to, and controlled by, a server process (called Visualization Manager). In this architecture however, views do not send any feedback to the visualization manager. They therefore do not communicate with each other either.

Software debuggers like DDD [117] or VIPS [92] follow the same idea of sharing the time variable between views: The source code view indicates the currently executed line, while another view can show the current value of some data. But DDD goes further by adding some real communication between views: From the source code view, one can select a variable and choose to add it in the other view for tracing. The other way around, other views such as the stack view and the thread view can send some feedback and cause changes in the source code view.

The best reference about collaborating views is Shrimp [73], presenting some real similarities with this thesis's work. Four sources of information are collected and represented in multiple views, some of them being network graphs representing the structure of the system. They implement a technique called *control integration*, which implies the ability for one tool to control another tool, either by directly activating a functionality or by event notification. The difference with this work is that Shrimp is targeted at exploring static information. The execution of the visualised Java program is not represented.

## **3.4 Unified IDE for Large Scale Asynchronous Circuits**

A list of tools for asynchronous design was compiled by Edwards and Toms for the ACiD Working Group [30]. Among twenty-nine reported tools oriented towards circuit

synthesis, only two of them have a graphical interface: the Balsa framework and Visual STG Lab [112].

Visual STG Lab has already been described in this chapter. It is a visual environment for design with Petri nets. However, as for all the Petri net applications, it is really applicable only to small to medium asynchronous circuit design.

It is possible to use any of the twenty nine asynchronous synthesis tools (although only a few of them are applicable to large designs) to generate gate-level or Verilog netlists. These netlists can then be used in traditional synchronous CAD tools to simulate and analyse the asynchronous circuit. However, these tools are not designed to debug asynchronous-specific problems, and more importantly, when a misbehaviour is detected at these levels, it is very difficult to trace it back to the original description. Even with Balsa, which has a direct-mapped syntax-directed compilation scheme, it is not completely obvious how to link synthesised Verilog netlist elements to Balsa constructs.

## **3.5 Summary**

This chapter has described related work in the areas of handshake circuit simulation, asynchronous-specific problems debugging and visualisation for program comprehension. It has positioned this thesis in these contexts.

# Chapter 4: Theory of Handshake Circuit Debugging

This chapter reviews the difficulties that can appear during the design of an asynchronous circuit with Balsa, the ways in which they arise and how they can be handled in a handshake circuit-oriented design environment.

A model of asynchronous handshake circuits is first introduced to describe the different situations and algorithms encountered throughout the thesis. Two sections then describe approaches for treating the problems of deadlocks, non-determinism and metastability, recurrent in asynchronous circuits. Some techniques are then presented to structure the large amount of information contained in the simulation trace by using activity pattern recognition. Finally, the issue of optimising an asynchronous design through profiling methods is raised.

## 4.1 The Handshake Circuit Model

### 4.1.1 Static Model

Previously, a handshake circuit has been defined as a set of handshake components connected together by asynchronous communication channels.

Let  $K$  be the set of handshake components  $\{k_1, k_2, \dots, k_n\}$  and  $C$  the set of communication channels  $\{c_1, c_2, \dots, c_m\}$ . Any handshake circuit is uniquely identified by a pair  $(K, C)$  of handshake components and communication channels.

The relationship between handshake circuits and graphs is immediate: The handshake circuit  $(K, C)$  can be associated with the graph  $G = (K, C)$  where  $K$  and  $C$  are respectively

the graph's vertices and edges. Graph properties can therefore be applied easily to handshake circuits.

### 4.1.2 Dynamic Model

Activity in a handshake circuit can be modelled as a series of asynchronous events: changes of state of the communication channels and computations processed by the handshake components. In the following description, these are referred to as *events* and *actions* respectively.

The exercise of a handshake circuit results in a series of instantaneous events produced by processes and channels. Each of these events leads to the execution of a piece of program (usually called callback), referred to as an *action*. Actions take a certain amount of time to be carried out before causing new events: As illustrated in Figure 4.1, an event arriving in a component triggers the execution of an action, which in turn, after predefined delays, sends new events onto the output channels. When a channel receives an event, the time taken for the signal to flow across the wire to the next component is modelled as a channel

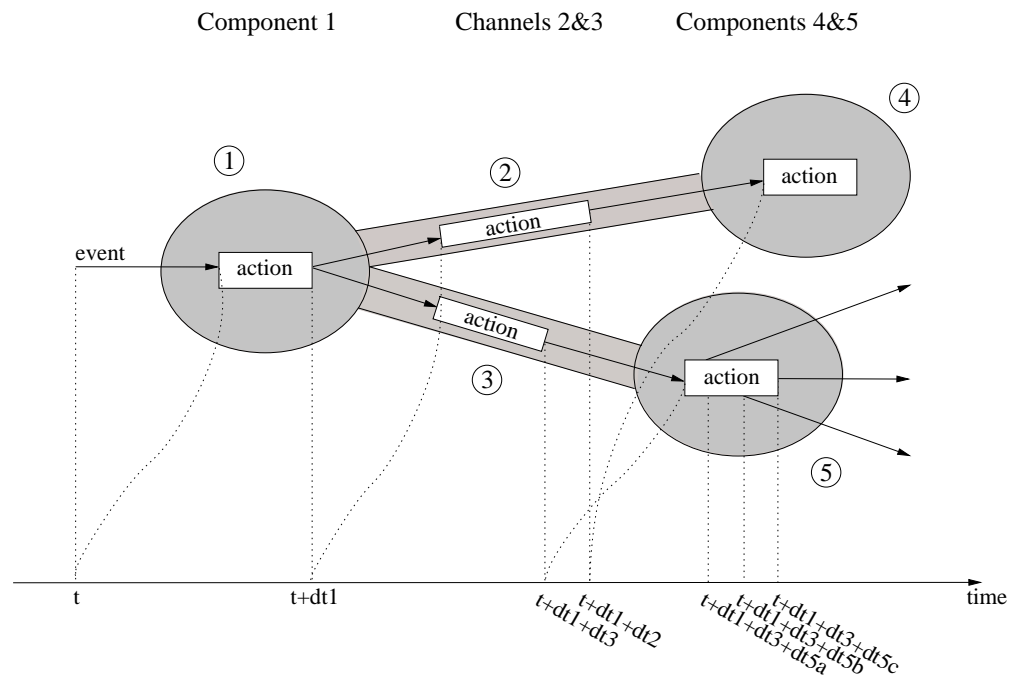


Figure 4.1: Time model of the execution of a handshake circuit

action taking the corresponding amount of time before generating an event to the destination component. Channel actions are not used for anything other than delaying the next event to model the wire's delay. Events are always instantaneous and generate one and only one action, whereas actions take some time to be executed and can generate any number of events (including none). Moreover, channel actions always send a unique event to the destination component, whereas component actions can generate any number of events at any time during the execution of the action (see component5's action in Figure 4.1). The delaying action of the channels can be included inside the components' actions for a simplified execution model, as shown in Figure 4.2. The reduced number of events and callbacks should lead to more efficient implementations and easier debugging, as less information needs to be analysed.

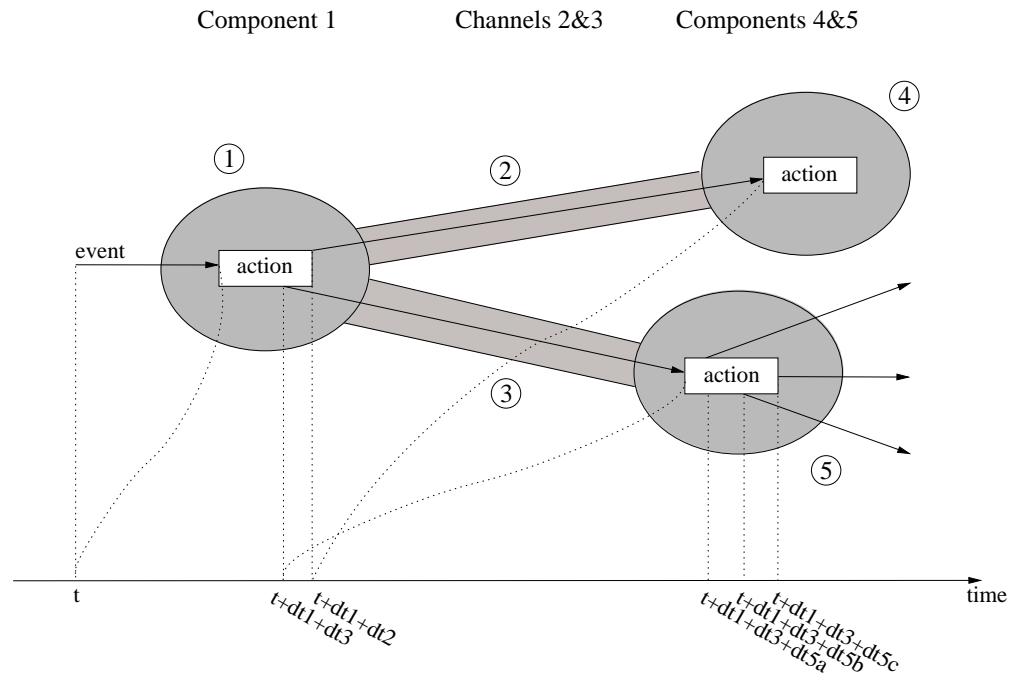


Figure 4.2: Simplified time model of the execution of a handshake circuit

The timestamp of an event is the sum of the delays in all the actions processed to reach this event since the beginning of the simulation. A small error in the delays can therefore make a big difference to the timestamps at the end of the simulation. In the same manner, during the simulation, as during the execution of the real asynchronous hardware, there is no clock to keep everything synchronised. This imprecision will be important when the

problems of non-determinism and metastability are considered. The *trace* of an execution (or more likely simulation) of a handshake circuit is defined as being the set of events associated with their timestamps from the start to the end of the execution.

Finally, in order to comply with Chandy and Misra's use of processes [20], a *process* is defined as the set of actions associated with the receipt of events in a handshake component. One and only one process is associated with each handshake component. Chandy's processes can therefore be seen as being equivalent to the handshake components in the model discussed previously. On receipt of an event, a process executes the appropriate action depending on the current state of the process, and updates its state. A process is said to be *stopped* if no event has been received (initial state) or all events have been acknowledged (back to initial state). A process is said to be *running* if it is executing an action. A process is said to be *waiting* if it is not running and is not stopped (it is then waiting for an event).

## 4.2 Deadlocks

A set of processes is said to be deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

The following properties were enunciated by Chandy and Misra as *Deadlock conditions* [20]: A set of processes  $h$  in a network is said to be (error) deadlocked at some stage of the computation if and only if

- *termination condition*: not all the processes in  $h$  are stopped and
- *executability condition*: no process in  $h$  is running and
- *closure condition*: if  $h_i$  in  $h$  is waiting on edge  $e$ , and  $e$  is incident on  $h_j$ , then  $h_j$  is in  $h$ .

In a generic environment (not necessarily asynchronous), where an event is the response to a process's request for accessing a critical section or a resource, the conditions for deadlock are [25]:

- **Mutual exclusion**: Resources cannot be shared.



- **No preemption:** Resources cannot be forcibly taken from processes.
- **Hold and wait:** Processes request resources incrementally, and hold on to what they have got.
- **Circular wait:** Circular chain of waiting, in which each process is waiting for a resource held by the next process in the chain.

In a handshake circuit environment, the first two conditions for deadlock are always true, dictated by the structure of the circuit: Handshake components cannot be shared and cannot be forcibly taken. The other two conditions may happen.

The consequence of any deadlock is the premature end of the circuit execution. Unfortunately, the end of a circuit execution does not necessarily mean that a deadlock has occurred. For example, an asynchronous processor executing the Halt instruction [42] fully stops its execution, yet can still be reactivated at a later time by an external interrupt. The problem here is to determine whether or not the processor is to be considered *fully stopped*: The interrupt may not be received for a very long time, however the circuit is not really fully stopped as it can resume its execution at the receipt of the interrupt signal. This kind of false end of the execution will be referred to as a *temporary deadlock*, and detailed more extensively later in this section. From this point in the thesis, this false end should not and will not be considered as a total end in a circuit's execution.

### 4.2.1 Handshake Circuit Deadlock Detection

Apart from the false end situation, handshake circuits have only two ways of stopping their execution: either the acknowledgment of the main control signal or a deadlock. The former indicates the successful completion of the simulation, and is characterised by the absence of any pending control signal in the circuit. A real circuit built in hardware and ending in such a manner would need to be reactivated (reset) before being able to operate again. The deadlock situation indicates that the activity has been stopped due to a missing acknowledge/request event. Unfortunately, this does not reveal whether the missing event is due to a normal or to an erratic behaviour.

Different types of deadlocks can be distinguished, eventually leading to different actions of the simulator:

- *Valid deadlock.* This deadlock arises when a circuit designed to run forever (e.g. a pipeline circuit) has processed all the available input data. The circuit has correctly sent a request on its input data port, but never received any answer, leading to the deadlock situation. This is the normal and only way for the circuit to finish when it has consumed every test data. The simulator should then stop without indicating an error.
- *Temporary deadlock.* Not really a deadlock, this situation arises when the external environment (test harnesses, or other simulators in the case of a co-simulation) is taking a very long simulation time to process its data, and thus appears to be dead from the point of view of the simulator. In this situation, the simulator should wait until an external event is available. This is not precisely a deadlock, as “temporary” indicates that the deadlock situation will be solved after an undefined period of time. This type of deadlock can be avoided if processes are able to indicate a minimal potential timestamp of their next event, as proposed by Chandy and Misra [20].
- *Error deadlock.* This type of deadlock is due to a real error in the high-level description, and requires the simulator to stop and generate a complete enough description of the handshake components and channel states for debugging. It generally occurs before all the test harness data has been consumed, but can also happen between the moment the last data has been taken and the normal termination of the simulation.
- *Error in co-simulation deadlock.* This is a high-level deadlock between two or more co-simulation systems. The problem is that each simulator has its own local view of the whole circuit, and thus cannot detect individually such deadlocks. A process tracking high-level communications is necessary.

The difficulty is to be able to detect the correct type of deadlock in any situation. The solution proposed here is to specify the expected behaviour of the circuit through its interface ports, linked to a properly described test harness.

The simplest case is a single simulation without any port other than the activation (reset) port. In this special situation, a deadlock will be immediately reported as an error deadlock, as any other type of deadlock requires the presence of other input ports.

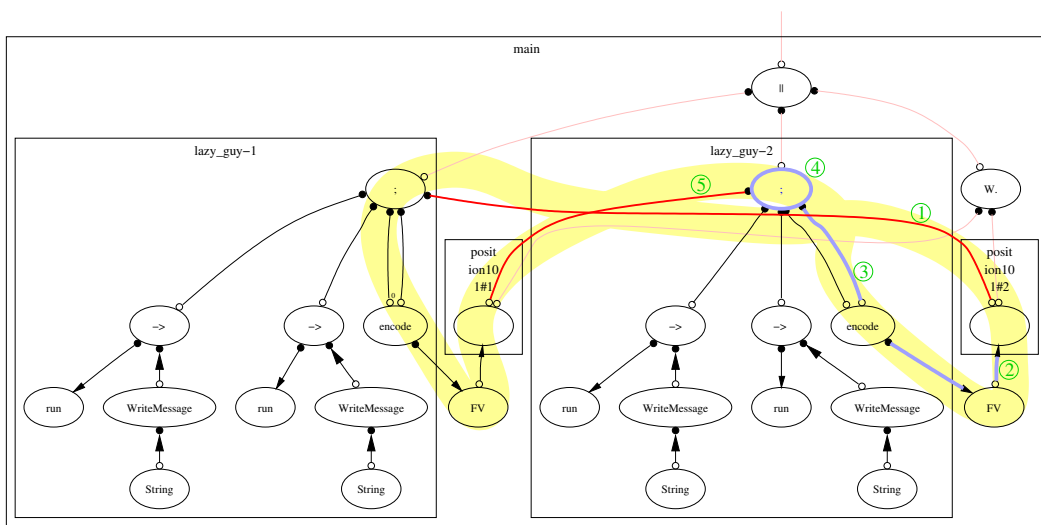


Figure 4.3: Elements involved in a handshake circuit deadlock

The set of *deadlocked processes* is referring to the set used in the definition of a deadlock. It is the set of processes such that each process in the set is waiting for an event that only another process in the set can cause. This set is unique by construction when only one deadlock is present. In the rare case of more than one deadlock, the union of all the sets satisfying the definition is itself verifying it and can be used to analyse all the deadlocks together. In the example shown in Figure 4.3, this set of processes and the channels linking them together are highlighted in yellow.

*Deadlocked channels* are those on which a request has been sent by the initiator of the communication, but has not been processed by the target. They are often the last channels activated before a simulation stops in a deadlock. They are drawn in red in Figure 4.3.

By starting from one deadlocked channel, four sets of involved components and channels can be iteratively extracted from the set of deadlocked processes and their channels. The longer red channel of Figure 4.3 is used in the example as a starting point for the algorithm.

Figure 4.4 describes the two parts of the deadlock analysis algorithm.

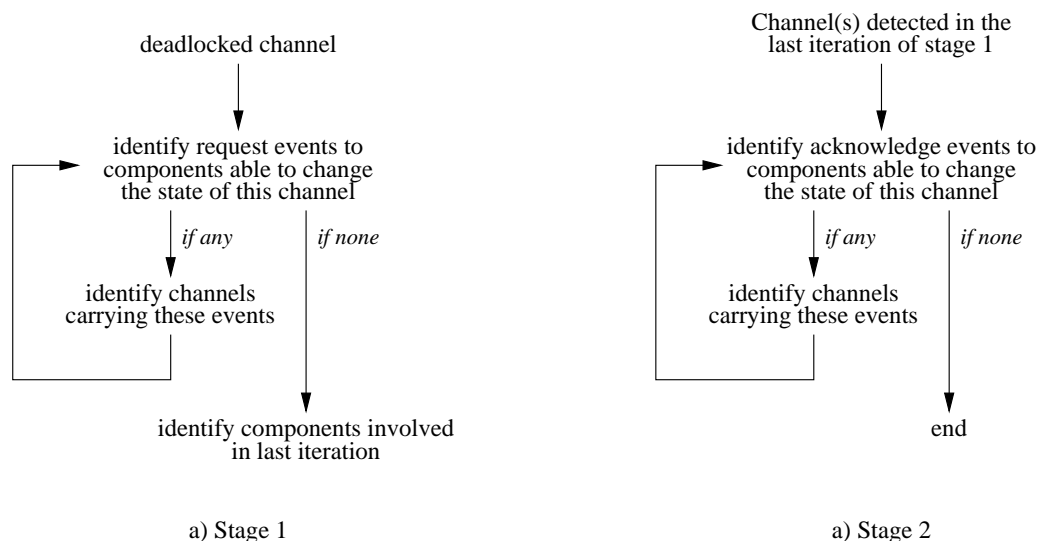


Figure 4.4: Deadlock analysis algorithm

The first part of the deadlock analysis algorithm presented here starts from one deadlocked channel and tries to find its causes. This is a recursive algorithm. It first identifies which request events would lead to the acknowledgment of the deadlocked channel. Then, recursively, it identifies which events would lead to the generation of these required events. The algorithm stops when a detected required event needs to be generated by a process which is actually waiting for an acknowledge event. In Figure 4.3, the string of channels corresponding to the required events is drawn in blue.

At the end of this first part of the algorithm, three involved channels or sets thereof (although in practice each of these sets are often reduced to one element) are detected as being of most importance, and one involved set of components is used to start the second part of the algorithm:

- The analysed deadlocked channel itself, which indicates the last action performed by the handshake circuit (the red channel indicated by a green ‘1’ in Figure 4.3).
- The first channel detected by the algorithm (the first channel of the blue string, indicated by a green ‘2’ in Figure 4.3), which indicates the closest required action that has not happened.
- The last channel(s) detected by the algorithm (the blue channel connected to the blue component, indicated by a green ‘3’ in Figure 4.3), which indicates the earliest unrealised action(s) that would have been necessary to avoid the deadlock.
- The component(s) supposed to deliver an event onto this last channel(s), but which was actually busy, waiting for another event (component drawn in blue in the figure, indicated by a green ‘4’ in Figure 4.3).

The second part of the algorithm is used to determine why this last component was waiting for another event, and was therefore not in a ready state. For this, the involved events are again searched recursively, by starting from the event the component was waiting for and following the path of required events until a deadlocked event is reached. In the example of Figure 4.3, the deadlocked event is reached at the first iteration of the algorithm, and corresponds to the red channel connected to the blue component (indicated by a green ‘5’ in Figure 4.3).

This last detected channel is believed to indicate the main reason for the deadlock. This channel and the three other important (sets of) channels detected by the first part of the algorithm are the most important ones for understanding a deadlock, and are successfully used in practice. This is illustrated in the results chapter (Chapter 9).

## 4.3 Non-determinism

A non-deterministic system does not always have a single, uniquely defined next action, but a choice between several next actions.

An advantage in the analysis and design of a synchronous system is that the state in the next cycle can be determined entirely from the current state. This *may* also be true in an asynchronous system, but the timing freedom means that this is not the only choice of action. Within a small asynchronous state machine it is possible to achieve the same behaviour with internal transitions ordered differently (e.g. the inputs to a Muller C-element can change in any order) and this is also true on a macroscopic level. On the other hand, because of non-determinism caused by tiny variations and arbitration, repeated executions of the same asynchronous design for the same inputs may give different outputs.

In a simulation at the handshake component level, a repeatable behaviour can be enforced if the following conditions hold:

- Inputs from external environment are the same.
- Initial values for the different components of the system are the same.
- Individual processes are deterministic.

The first two conditions can easily be realised. However, the last one depends on the structure of the simulation system and is therefore an unavoidable cause of non-determinism.

In Balsa, the only process which is a source of non-determinism is the handshake component called *Arbiter* (the only component which contains a mutex), used to guarantee the mutual exclusion of two passive input channels' communications by

passing a single communication at a time onto one of the two active output channels [4]. Arbiters are explicitly introduced by the Balsa “arbitrate ... end” construct. Designers always try to minimise the use of arbitration. However, the simulator should be able to handle the few cases where arbiters are required and non-determinism problems can occur.

During the execution of the real handshake circuit, non-determinism can also arise due to some changes in the delays of circuit components and wires (due to thermal effects, cross-talk between wires, supply voltage noise, process variations, etc.). Correctness in delay-insensitive circuits is not affected by these changes, and again, with Balsa, the Arbiter is the only component suffering from its non-DI behaviour. Some other Breeze handshake components have non-DI internal construction (e.g. the Case component), but the construction mechanism of the Breeze handshake circuits ensures that all critical inputs will always arrive one at a time in a correct order.

The simulation of an asynchronous circuit must therefore handle the Arbiter’s non-determinism in two ways: handle metastability, and handle bounded delays (delays with allowed small variations).

### 4.3.1 Metastability

The non-deterministic behaviour of the Arbiter appears when both of its inputs are activated within a short period of time. When this happens, signals in the Arbiter’s circuit hover for an unbounded period of time before reaching a stable state. This is *metastability* (see §2.1.4).

When metastability is encountered, the delay and the final output choice of the Arbiter are undefined. However, the simulator has to choose a subsequent set of actions. A random behaviour, or different deterministic behaviours can be used: always choosing the first input, or always the second one; starting with the first input and switching; choosing the same input as the one received last, or alternating; etc.

Although less important, a delay also has to be chosen. It should be greater or equal to the normal arbiter delay, but does not influence much the final result. A delay including error can be used, as developed in the next section.

### 4.3.2 Modelling Delays With Errors

Timing estimations of the simulation at the handshake circuit level are quite poor and far from what happens in the real circuit. Two requests arriving at the same time in an Arbiter during the handshake circuit simulation – leading to a non-deterministic behaviour – may have arrived quietly one after the other on the real hardware. In the same way, but even more problematic, two requests arriving at the same time on the real hardware circuit could arrive at different times in the simulator, avoiding the important detection of the non-deterministic situation.

In order to detect such cases, two solutions can be used. The first one is to work with time windows: When a communication is received on one of its inputs, the Arbiter component waits for a possible request on its other input during a specified amount of time before being able to decide if its behaviour should be deterministic (one request received during the lapse of time) or non-deterministic (two received requests). The choice of delay is critical: Too short a delay would miss the detection of some non-deterministic situations, whereas too long a delay would lead to false detections of non-deterministic situations.

The second solution consists of modelling delay errors: Actions' delays may be modelled as  $delay \pm error$ . Of course, the same problem as before happens again: Too small delay errors would miss the detection of some non-deterministic situations, whereas too large errors would lead to false detections of non-deterministic situations.

### 4.3.3 Exhaustive Simulation

When a non-deterministic situation occurs during the execution of a circuit, one next action is randomly chosen amongst two possible actions. It is impossible for a simulator to know the issue of the choice in the real circuit execution.

One way to mitigate this is to allow the user to choose the desired behaviour: The current implementation of the simulator makes a default choice, and eventually a checkpointing



system allows the designer to rewind a simulation to the non-deterministic points and manually define the desired behaviour before restarting the simulation at this point.

However, the only way to fully solve the problem is for the simulator to simulate every possible action: The solution is to create two branches of the simulation every time a non-deterministic situation occurs (Breeze arbiter components only have two inputs). This method requires a way to avoid the creation of  $2^n$  simulation branches, which can be done by exploiting the fact that, most of the time, two branches differ only during a certain amount of time before merging back to an identical state. This requires a way of analysing different branches fairly quickly in order to join them back when their execution becomes identical. These short local variations in behaviour with flows merging back into a common behaviour are very hard to spot. This last idea has not been implemented, although it would be a way to reflect every possible behaviour of the VLSI circuit.

## **4.4 Activity Pattern Analysis**

Activity pattern analysis is employed here as a technique for structuring the large amount of information contained in the simulation of handshake circuits. This clustering technique is mostly based on simulation trace information. This research is intended to meet our needs for basic debugging of large designs.

### **4.4.1 Visual Analysis**

The idea is to display the information in such a manner that patterns can be visually detected. The simulation trace is used as a source of information to display the space and time dimensions of the channels' activity information in a single view: time on the x-axis and space on the y-axis, as illustrated in the lower view (Timeline behaviour view) of Figure 7.7, page 112. The challenge is to fit the information in the view in such a manner that specific patterns are made visible. For example, repeating patterns are hoped to be visually detected easily on this view.

Patterns may include a large number of channels (in the space dimension) and long intervals (in the time dimension). It is therefore necessary to fit in this view as much information as possible on each axis. Time is a continuous value, and thus can be zoomed

in or out easily. This is not the case with handshake channels, which are discrete components. However, zoom in and out actions can find their equivalence with the various methods of groupings discussed later in §7.1.

After experimentation, it was felt that in order to visualise patterns, the space axis must be entirely visible at any time (no scrolling must be required to view some channels). The very efficient method of grouping channels by procedure is a good way to obtain a small number of groups on the space axis. This method allows every channel to fit together in the same view. The time axis does not need to be entirely visible at all time as it is more dependent on the length of the visualised pattern.

#### 4.4.2 Automated Analysis

A feasible automated analysis concerns the detection of *debugging templates*, situations that the debugger recognises and can help solve more or less automatically.

A single *template* is analysed here: the repetition of identical patterns in the simulation trace file. No debugging action other than reporting the results of the analysis is envisaged. However, Erbacher shows in [35, §3.1.2] that the same kind of repeating pattern analysis can lead to other interesting applications:

*“Thistlewaite and Johnson [108] describe an environment that attempts to reduce the number of primitive events in a trace file by deducing compound events, representing higher level concepts, from the trace file. The environment also recognizes adjacent repetitions of the same compound event. This environment greatly reduces the complexity of trace files but is limited in that it may remove details of importance.”*

The automated detection of repetitions of activity patterns can easily be used to report useful information about livelocks: which activity pattern is repeated and where and when it started. A simple algorithm can be employed: Starting from the end of the simulation trace, looking for corresponding repetitions of the channel activity at regular interval of times in direction of the start of the simulation trace. “Livelock Handling” on page 134 illustrates this result.

The detection of repeating patterns can also be used to compress the simulation trace, as will be developed later in this thesis.

## 4.5 Circuit Optimisation – Profiling

At some point in the design process, the circuit is working – i.e. it exhibits the correct functionality – but a faster and lower energy consuming system would obviously be welcome. It is time to look for performance bottlenecks and power-hungry modules. This can be carried out by profiling delays and consumption of the modules in the circuit.

Profiling is performed by associating a value of interest (delay, power consumption, etc.) to each event of a simulation, and integrating these values over specified areas and specified periods of time. This statistical method allows the grouping of thousands of individually insignificant values into a few meaningful high-level values. A strategy involving tracing the simulation and analysing the obtained traces is usually employed. This has the advantage of letting the user decide *a posteriori* which areas and periods of time he wishes to integrate the profiling values over. The difficulty with this method is that trace instrumentation always comes at a cost (execution time and storage space). The trace set therefore needs to be optimised according to the performance profiling problem being solved in order to minimise the effect of tracing on the system's performance.

## 4.6 Summary

An algorithm to detect and analyse deadlocks from a handshake circuit simulation has been described, in order to identify the involved handshake channels. The Arbiter component has been identified as being responsible for the other asynchronous-specific problems of non-determinism and metastability. A couple of methods – modelling delays with errors and exhaustive simulation – have been suggested for improving the handling of these two problems. A straightforward method for activity pattern recognition has then been presented to structure the large amount of information contained in the simulation trace. Finally, the issue of optimising an asynchronous design through profiling methods has been raised.

# Chapter 5: High-Performance Simulation

The two most popular hardware development languages used for synchronous design, Verilog and VHDL, were originally designed for simulation purposes. This made the process of synthesising such languages to hardware systems a tricky task, with only a subset of each of these languages being synthesisable. Balsa, on the other hand, is a synthesis-oriented language. This has the disadvantage of making the route to simulation less obvious and perhaps less efficient. This chapter exposes the results of the research for making Balsa simulation as efficient as possible.

The need for a fast simulation route mainly comes from the iterative style of development usually associated with Balsa: Design space exploration is performed in Balsa by making changes to the source Balsa language description and design iteration is used to evaluate the effects of these changes. For this to be an effective technique, a simulator must be fast and able to reflect the speed and structure of the real circuit.

The new simulation system for Balsa has been developed around two goals:

- speed: necessary for practical design iteration and validation,
- ease of design analysis: to provide the designer with relevant information for debugging and optimising his circuit.

This chapter focuses on the performance aspect of the simulator, while the analysis aspect is described in the next chapter. First, a preamble explains the choice of simulating at the handshake circuit level and exposes a few properties of the handshake circuits in use here. Then follows the description of the simulator itself and the solutions adopted to optimise simulation speed, first with out-of-order processing of scheduled events and then by

developing a set of techniques specific to handshake circuits. Finally, it is related how test harness descriptions for Balsa circuits have evolved towards better integration with the Balsa description for improved performance.

## 5.1 Preamble

### 5.1.1 Choice of the Simulation Level

Simulation of a Balsa description can be performed at several distinct levels of abstraction [4]:

- language level behavioural simulation
- handshake circuit simulation
- gate level simulation
- switch and analogue extracted layout based simulation.

The lower the level of abstraction is, the more precise the simulation will be but more parameters are required to set it up. In addition to this, a few pros and cons can be specifically associated with each level of abstraction.

The two lower simulation levels correspond to the simulation of the netlist generated from the handshake circuit by the synthesis tool at the gate level or later as an extracted layout. Their main advantage is to provide more precise timing simulations as well as to enable estimations of effects such as electro-magnetic emissions, but at the cost of some additional simulation processing time. Gate level and layout simulators are already available as synchronous tools, and can be used with Balsa-synthesised circuits, although without any automatic way for the simulator to refer to the Balsa source code for error reports or flow analysis. Both of these features are important for debugging purposes.

A language level behavioural simulation presents the advantage of providing an easy access to variable values and structures for inspection and debugging purposes. However, the complexity of Balsa descriptions (parameterised procedures, structural iterations, conditional ports, test harnesses, etc.) would be transferred to the simulation process. The major advantage of a behavioural simulation at the Balsa language level is technology

independence: Handshake protocol and data encoding (the way the data and the request and acknowledge signals are encoded on wires) do not have to be chosen prior to the simulation. Unfortunately, in the context of circuit design and debugging, one aim of the simulation is to observe the evolution of control and data flows inside the circuit, which differ according to the technology chosen. A technology-independent simulation would therefore be of limited use.

At the handshake circuit level, the simplistic flat structure of handshake circuits is a real advantage: A handshake circuit can be modelled by a simple graph structure of handshake components where the component set is well defined and does not change for every modification in the language. Moreover, the syntax-directed compilation process, ensuring a one-to-one correspondence between Balsa constructs and handshake components, makes it easy for the simulator to refer back to the Balsa source code for error reports or flow analysis. It also implies a simpler architecture not only for the simulator but for the whole simulation framework (simulation, visualisation, debugging).

In summary, simulating at the handshake circuit level provides the following advantages:

- simple simulator (a small set of only 40 to 50 standard handshake components is used by the Balsa compiler, linked together by easily simulated channels),
- good possibilities of circuit analysis exploiting data and control flows,
- one-to-one correspondence with the Balsa source code.

This is a good compromise between the direct simulation of the high-level Balsa description and the simulation of the low-level synthesised netlist. However, one may prefer simulating at a lower level (gates or layout) for the increased precision. As stated previously, this is still possible through the use of conventional circuit simulation tools.

### **5.1.2 Choice of the Handshake Protocol**

As the simulation of handshake circuits is technology dependent, the handshake protocol and data encoding have to be defined prior to simulation. The Balsa framework is constructed in such a way that different back-end technologies and implementation styles

can be used for synthesis and the simulation framework must therefore take these options into account where appropriate.

Data encoding defines how request and acknowledge signals are mixed with data to define the way a handshake channel will be synthesised as a set of wires. The techniques developed in this chapter are designed for single-rail data encoding because, compared to dual-rail, single-rail requires fewer events (and therefore leads to a faster simulation) and these events are more meaningful when visualising or debugging the circuit.

Both the 2-phase and 4-phase single-rail handshake protocols can be implemented as libraries of handshake components used by the simulator. The former provides a better speed as half as many events are flowing in the circuit. However, the latter provides more information about the flow of data thanks to its return-to-zero phase and reflects in a better way real asynchronous circuits, which often use 4-phase encodings.

Switching from 2-phase to 4-phase is a good means for designers to choose between speed and fidelity at the simulation level.

### 5.1.3 Preliminary Statistics

Some preliminary statistics on handshake circuits are necessary to understand the methods employed to optimise their simulation. These figures are obtained from the examples used in the results chapter (Chapter 9):

- The average (mean) number of ports per component goes from 2.5 for small circuits to 3.2 for large ones.
- In large circuits, for components with parameterised port counts, the average (mean) fan-in/fan-out (i.e. length of input and output arrays (e.g the output array of a fork component is the list of its outputs)) is 3.0 with:
  - $\text{Probability}(\text{fan}=2) = 48\%$ ,
  - $\text{Probability}(\text{fan}=3) = 42\%$ .

## 5.2 Scheduler

Handshake circuits, as control data flow graphs, can seem very easy to simulate: Events are generated by components, travel over the graph's arcs, and activate the next components to start appropriate actions. And the process is repeated. In fact, an efficient simulator can be based exactly on this behaviour. However, with handshake circuits, some interesting optimisations can be applied, in particular by taking advantage of the delay-insensitive nature of most handshake circuits.

This section first describes a software model for simulating handshake circuits, then details the basic event-driven scheduler style that can be used to simulate any handshake circuit, and follows on an out-of-order scheduler optimised for exploiting the delay-insensitivity in handshake circuits.

### 5.2.1 A Software Model for Simulating Handshake Circuits

At the handshake circuit simulation level, the behaviour of handshake components is very simple to model and the key is to abstract channel behaviour into two events (request and acknowledgment) for a 2-phase protocol, or four events for a 4-phase protocol.

Each handshake component is modelled as an object with properties and methods, where properties define the current simulated state of the component and methods implement the actions being raised when input ports are activated. This model is illustrated in Figure 5.1. Channels are the medium of the handshake protocol, and as such they must be able to transmit protocol events (requests/acknowledgments) in appropriate directions and, in the case of data channels, hold a value. The interface between components and channels is modelled by ports. Ports can be implemented either as distinct objects, or partly in both the component and the channel objects. Reducing the number of objects usually tends to make for faster designs. The two-parts implementation has therefore been chosen, as illustrated in the figure: On the component's side, a port would be implemented as a reference to the connected channel, while on the channel's side it would be a reference to the component object.

Methods in channels implement wire actions, which model signal propagation delays. An optimised model is used here which bypasses channel actions by letting components call



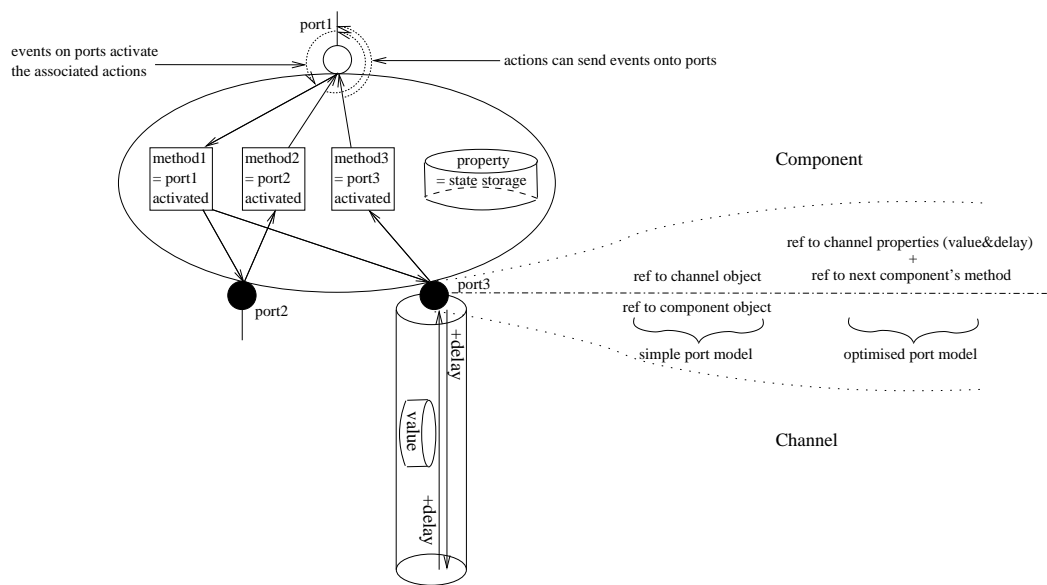


Figure 5.1: Object-oriented view of handshake component and channel

directly the next component's method. Delay associated to channels can easily be included in the calling component's delay. For further optimisation, the value held by a channel could be stored on the component's side, removing the need for accessing any channel structure. This, however, implies a less tidy implementation of the different structures, where channels are linked to two components being only implemented in one of them.

### Note on software optimisation

Action handlers are extremely often called. They must therefore be stored in a contiguous memory space to make best use of the instruction cache. Furthermore, handlers can be sorted by frequency of execution in order to group the most often called ones together.

The handshake circuit is modelled as a set of handshake components, each component being a structure containing its current state and some pointers to the handlers and structures of other components that can be called. In the same manner as with the handlers, the best organisation is to group connected components together and keep the size of their structure as small as possible to make best use of the data cache.

### 5.2.2 Standard Event-Driven Scheduler

The Balsa simulator is based on a standard event-driven scheduler where a single event queue contains the events waiting to be processed sorted by timestamp. The execution of the first event of the queue leads to its removal from the queue and the activation of the method associated to the event's component and port. This action executes some code specific to the component's behaviour and schedules the activation of the component's output ports by inserting new events in the scheduler's queue. An event sent to an active port is a request to the next component, while an event sent to a passive port is an acknowledge sent to the previous component.

Figure 5.2 reveals some details of the simulation of the handshake circuit of a 1-place buffer. Handshake components are represented as oval shapes with their passive and active ports respectively represented as white and black circles. Rectangles inside components are representing the actions executed when ports are activated: Incoming arrows to an action's rectangle indicate which ports are leading to the execution of this action, while outgoing arrows indicate which ports may be fired (i.e. events inserted into the scheduler's event queue) by this action. Also, dotted lines indicate the internal causality of events. Finally, the database symbol indicates a property storage specific to the component, usually to save a state or value. The simulation of the circuit can be retraced by starting at the top of the figure and by following arrows and channels (Note: An arrow pointing to a port from inside a component should only be followed to the next component. Never follow it by other arrows inside the same component).

In this circuit, each action inserts in the queue a unique event, which, as soon as the current action is finished, is dequeued and executed as the next action. Interesting behaviours are seen in the Loop component, which never acknowledges its passive port, and in the Variable component for its use of storage.

In the previous, single-threaded, circuit, the role of the scheduler's queue is not clear. Figure 5.3 shows how the multi-threaded Fork component is implemented following the same scheme: When the input is activated, two events are added to the scheduler's event queue, and the acknowledgment of the input only happens when both outputs have received their own acknowledgment events. A local counter is used inside the component

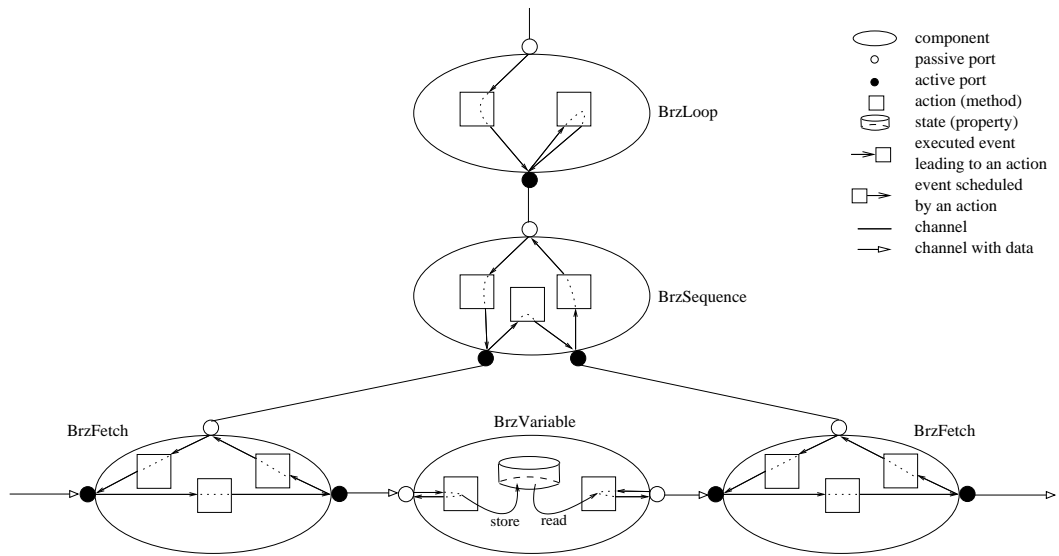


Figure 5.2: Object-oriented view of a handshake circuit

to keep track of how many output events have been received and the resulting event is inserted into the scheduler's queue only when the counter reaches two (or reaches the number of outputs for a more generic  $n$ -output fork). Also shown on the figure is the progress of the scheduler's event queue during the simulation of the fork.

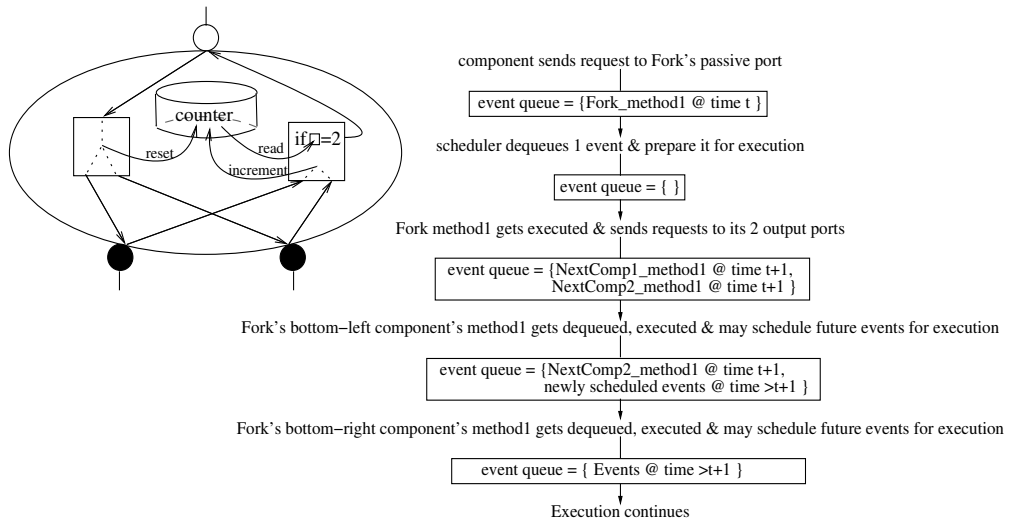


Figure 5.3: Fork component model and scheduler's event queue

This example illustrates why the event queue is intensively used during the simulation process: Each executed action typically comes from an event that has been processed (and removed) from the event queue only a few time steps earlier. Therefore, each action which is executed (i.e. each handshake component port that is activated) requires a push and a pop of an event on/from the event queue. Actions are usually extremely concise and quick to execute (for example, the  $n-1$  first input requests on a rendez-vous point are only used to increment a counter and check whether it reaches  $n$ ;  $n=2$  in the example shown in Figure 5.3). The event queue manipulations therefore consume a large proportion of processing time. The next section investigates a way to improve this.

### 5.2.3 Out-of-Order Scheduler

Asynchronous handshake circuits described in Balsa present the important property of being delay-insensitive when coupled with a DI protocol. The simulator assumes a single-rail protocol, which is non-DI in hardware. However, the manner in which it is implemented can ensure the DI property of the communications: in simulation, the arrival of data can be event-modelled and ensured to reach the destination component before the request or acknowledge event it is bundled with. In these conditions the protocol is DI.

The advantage of DI circuits at the scheduler level is their ability to be executed “really asynchronously”: an activated component can wait as long as it wants before being processed without changing the behaviour of the circuit. This corresponds to the same situation as if the component’s activation wire takes a very long time to transmit the event, which is not a problem in a DI environment.

A simulator for DI circuits can partly ignore the strict order of simulated time. It can “flash back” to simulate events that in simulated time actually occurred earlier: In simulating any one of several concurrent paths, the simulator can proceed along the path even if some other path would, in reality, have acted first. This act of loosening the relationship between simulation time and simulated time improves the simulator performance by reducing the number of occasions on which the scheduler must refer to its event queues. This has been described ten years ago by Sutherland as *Flashback Simulation* [104]. His description was dealing with micropipeline components, limited to the control part of the circuit. This is used here and extended to handshake circuits. He also described a simple

method for reordering the arbitration inconsistencies, which is extended in the next section.

Reusing Sutherland's terms, *simulation time* refers to the time at which the simulator performs an operation, whereas *simulated time* refers to the time at which the simulated event would have occurred in the world being simulated. The simulation and simulated times of an event A are respectively noted  $s'n(A)$  and  $s'd(A)$ .

An example is given in figure 5.4 to calculate the formula  $\frac{-(x+1)}{2} + \frac{2}{x-1}$ . The idea shown there is that a row-by-row simulation corresponds to the standard scheduler where the simulation time follows the simulated time, while a column-by-column simulation corresponds to the out-of-order scheduler. This is a small and obvious example; However, larger DI circuits are following the same scheme where the simulation time does not always have to increase monotonically.

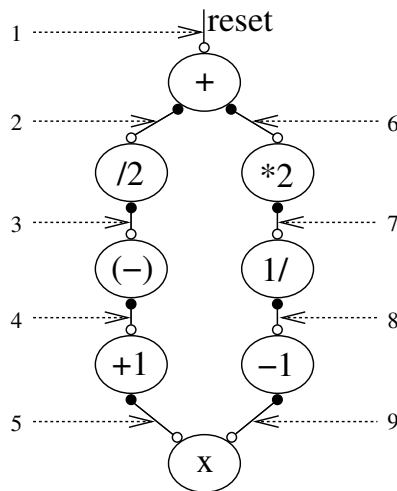


Figure 5.4: Pseudo handshake circuit for the equation  $\frac{-(x+1)}{2} + \frac{2}{x-1}$

In this example, the only timing constraints are given by causality between components: Component '+' needs to be activated before being able to generate requests to components '\*2' and '/2', leading to the relations  $s'n(req_1) < s'n(req_2)$  and  $s'n(req_1) < s'n(req_6)$ , and on the way back both acknowledges need to be received before generating the main acknowledge, meaning  $s'n(ack_2) < s'n(ack_1)$  and

$s'n(ack_6) < s'n(ack_1)$ . The same causality relations apply to the other components, leading to the final constraints

$$\begin{aligned} s'n(req_k) &< s'n(req_{k+1}) \\ s'n(req_5) &< s'n(ack_5) \\ s'n(ack_{k+1}) &< s'n(ack_k) \end{aligned} \quad 1 \leq k < 5$$

and

$$\begin{aligned} s'n(req_1) &< s'n(req_6) \\ s'n(req_k) &< s'n(req_{k+1}) \\ s'n(req_9) &< s'n(ack_9) \\ s'n(ack_{k+1}) &< s'n(ack_k) \\ s'n(ack_6) &< s'n(ack_1) \end{aligned} \quad 6 \leq k < 9$$

The second point is that, in real life, components on the same row would probably be executed at similar times:

$$\begin{aligned} s'd(req_k) &\approx s'd(req_{k+4}) \\ s'd(ack_k) &\approx s'd(ack_{k+4}) \end{aligned} \quad 2 \leq k < 5$$

A standard event-driven simulator - where the simulation time follows monotonically the simulated time - would therefore simulate the handshake components line by line, thus interleaving the execution of both columns. A better simulator (in the same way as a real person reading the diagram) would execute the first column, and then the second one, thus making a better use of data locality (better use of the cache memory) and fewer accesses to the event queue. This is the idea behind the out-of-order scheduler.

Note that this is different from speculative execution, as no risk is taken: Only events whose simulation will not affect future events are processed using this method. No *roll back* of the system is ever required.

In order to achieve this result, the standard event-driven scheduler can be simplified: The time queue necessary to execute in timestamp order (and thus line by line) is not useful

anymore, and the direct execution without time queue processes the column serially: Each component requests the data from the next component, and this request is directly processed in the order shown in figure 5.5.

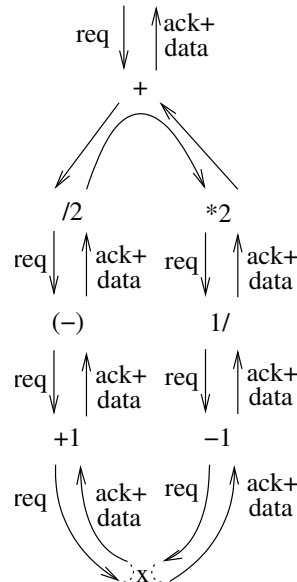


Figure 5.5: Execution order of the handshake components in figure 5.4

Due to the extremely simple actions processed by the handshake components, the time spent in the standard scheduler represents a high proportion of the total simulation time. The impact of this out-of-order scheduler on the overall simulation speed will be measured in the results chapter.

Note: Take care not to mix up this out-of-order scheduler with multithread simulation: Both can execute the handshake circuit column by column and benefit from this, but for different reasons. In a multithread simulation, the simulated time of each component still follows monotonically the local simulation time of each thread.

### Special Situations

In both the real and the out-of-order simulated circuits, the Add component has to wait for its two inputs to acknowledge before being able to carry on its work. This rendez-vous point is one of the most important aspects of DI circuits, as it synchronises paths that might be arbitrarily delayed, whether it be in the real hardware circuit or in the out-of-order simulator.

### **Simulating rendez-vous points**

When the simulator reaches an input of a rendez-vous element, there are two possibilities. First, it may not yet have simulated the other paths up to the rendez-vous point. In this case, the execution of this path must wait until these other paths have completed. The execution therefore switches to some other task. Second, it may already have simulated the other paths up to the other inputs of the rendez-vous element. In this case, it may simulate the output of the rendez-vous element directly, without any reference to the event queue.

The simulated time of every input must be recorded by the rendez-vous point, and at the reception of the last input, the latest simulated time of all the inputs corresponds to the simulated time of the execution of the rendez-vous's action. Notice, however, that the simulator may have recorded the simulated times out of order, and needs then to record the latest timestamp of input arrived.

On the way back, the acknowledge signal received from the output port leads to  $n$  acknowledges being sent onto the input ports. The optimisation described here leads to store into the event queue only  $n-1$  of these ack events and execute the last one directly without going through the queue and therefore saving a push/pop cycle.

The improvement from the ordered scheduler to the out-of-order one is caused by the less intensive use of the event queue: Whereas the former was pushing (and pulling) into this queue  $n+1$  events ( $n$  acks and 1 req), the latter only needs to push  $n-1$  ack events, which may lead to a considerable increase in performance, as  $n=2$  most of the time in practice.

### **Simulating arbiters**

An arbiter is special because the order of simulated time is important. The simulator may be written to choose to only compute the output of the arbiter if it has computed all contending inputs. However, arbiters often respond to a single input, and others may never arrive. It is therefore improper to wait for them before computing the output.

Two solutions can be employed: The first one is to use the first simulated input to generate the output, without taking into account the fact that another input may be simulated later



and yet have an earlier simulated time. The second solution consists of reordering the inputs of the arbiter, and is treated in the next section.

Processing the inputs of the arbiter in an out-of-order fashion may seem to lead to a completely wrong outcome. This is actually not the case, as illustrated by the example in figure 5.6. The shared read-only memory can be accessed by either of the two processors in a completely independent manner, and still the final behaviour will be correct. This is a valid DI behaviour.

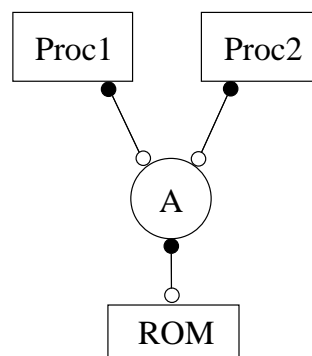


Figure 5.6: Arbitrated circuit

The first reaction is of course to see the problem that could arise in the case of a read-write memory: If proc1 is writing some data that is to be read at a later time by Proc2, an out-of-order simulation might give the wrong result. The mistake with this situation is not only in the out-of-order scheduler, but in the way this asynchronous system is designed: It is built on timing assumptions, and precisely on the assumption that the asynchronous block Proc1 will fire before Proc2. Such a case of *non-DI behaviour* will certainly fail the out-of-order arbitration.

Another obstacle concerns the *fairness* of the arbitration, i.e. the ability of the arbiter not to always choose the same input, thus preventing *starvation*. In fact, the arbiter described here, using the first simulated input to generate the output, could be absolutely unfair even in simple situations. Such a situation is illustrated in Figure 5.7 (an equivalent behaviour would be obtained without the DecisionWait component. The arbiter’s outputs could be directly connected to the “run” components, whose behaviour is to acknowledge without

delay any incoming request): Both the arbiter's inputs are continuously requested, and when the arbiter chooses an input and forwards the signal to the corresponding output, this signal is directly acknowledged by the rest of the circuit. In the case of an out-of-order simulation, the cycle Loop-Arbiter-DecisionWait-Continue-DecisionWait-Arbiter-Loop is uninterrupted and is repeated indefinitely. Only one of the two threads is uninterruptedly simulated, starving the other.

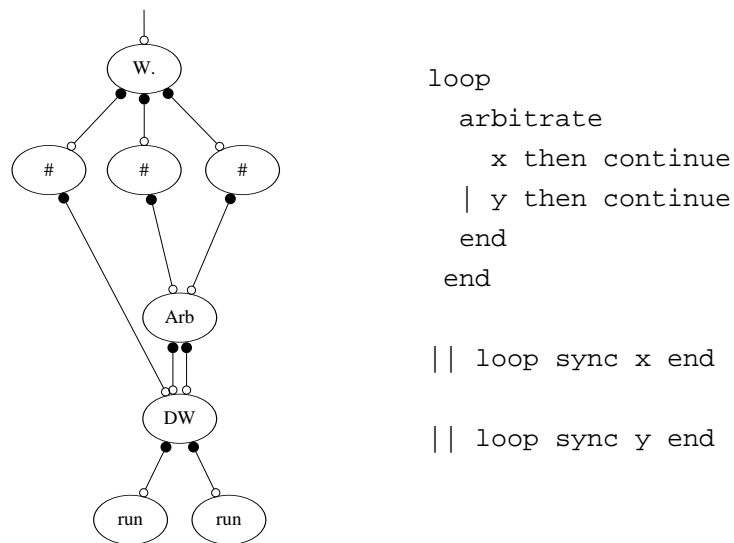


Figure 5.7: Starvation due to out-of-order arbitration

Yet another problem with this arbitration is that even when global behaviour is correct, handshake events may happen in a different order than what they would be in a real execution of the circuit. Although this is not important for circuit validation, in a debugging context, out-of-order messages prevent the correct analysis of the different flows of data and control. As a consequence of this, this special scheduler cannot be used for thorough debugging.

Reordering the inputs of the arbiter component would ensure the correctness of the timestamps relative to real circuit execution. The next section explores this issue, while trying not to affect the performance gained by the out-of-order execution of the other (DI) handshake components.

### 5.2.4 Reordering Arbitration Inconsistencies

The problem of processing the arbiter's action in function of the out-of-order input events is identical to the problem posed by parallel discrete event simulation (PDES), where events computed on different processors with their own local clock have to be synchronised when reaching a common execution point.

All the solutions proposed for PDES can be employed to solve the arbitration issue: Conservative methods will not compute the output of the arbiter before a necessary set of events with earlier timestamps has completely been processed, whereas optimistic methods will compute the output immediately and rollback in case of wrong decision.

#### Conservative Methods

The simulator may correctly compute the output of an arbiter based on only a single input time. It may do so only if it can prove that it will not, later on in simulation time, compute an event that would have arrived earlier in simulated time than the event it has already computed for the single arbiter input. This may require the simulator to work through all of the events on the event queue that are earlier in simulated time than the arrival time of an arbiter input before computing the output of the arbiter. Null messages, as proposed by Chandri and Misra [20] in the case of parallel simulation can also be used to transmit indications of minimal possible event timestamps on the inputs of the arbiter.

#### Optimistic Methods

In the optimistic methods, an arbiter handles the arriving events aggressively assuming all the events are safe. However, when a message arrives, its timestamp may be less than of some events already executed. It then rollsback by *Time Warping* [61]. During a rollback the current state of the whole simulation is modified to return to a correct old state. It is obvious that this procedure, which involves states and events saving and restoring, demands a large amount of memory and may be extremely slow if rollbacks are too often needed.

### Chosen Solution

As arbiters are not extremely frequent in an asynchronous design (in comparison to the number of other components), the conservative method is chosen for its simplicity. A timestamp-ordered queue is used for arbiters' request events. Events of this queue are processed only when the first (out-of-order, unordered) queue is empty. This ensures that all earlier events have already been processed.

This solution provides two additional benefits. First, each execution of an event from the ordered queue provides an indication of minimal timestamp for trace reordering. Then, the same queue can also be used to print console and file messages in order.

## 5.3 Modelling Handshake Circuits for Speed

This section describes a set of techniques aimed at optimising handshake circuit simulation.

### 5.3.1 Channel Data Value Implementation

In Balsa, data values carried on channels are not limited to a particular maximum number of bits. Values wider than the maximum integer width supported by the host's architecture must therefore be encoded specifically in order to process data copies and operations (e.g. add, xor, not) efficiently. For this purpose, the GNU MP library [47] has been used to represent these long data types. The unbounded integer data type defined by this library is *mpz\_t* and is referred to as *mpint*. Both long values (larger than the host's maximum integer width) and short ones can be encoded as mpints. However, applying operations on mpints is a lot slower than applying the same operations on hosts' integers (even when the mpints are encoding short numbers). For this reason handshake channels are implemented in such a manner that they can either contain an integer or reference an mpint. Handshake components must therefore be able to handle data operations on both integers and mpints. This has led to an unfortunate complication of the handshake component implementations. Some components such as BinaryFunc not only have to handle integers and mpints, but all the combinations thereof: int+int, int+mpint, mpint+int and mpint+mpint). These components do, however, benefit greatly in increased simulation speed.

The gain in speed between an all-mpint and a mixed int-mpint implementation varies from 0% to 30%, depending on the circuit. These figures are obtained from the examples used in the results chapter (Chapter 9).

### 5.3.2 Premature Channel Data Storage

In order to pass to the scheduler as short events as possible, the data value defined when a channel's data becomes valid does not necessarily need to be passed together with the *data\_valid* signal. Since no process is supposed to access the data value while it is invalid, it is correct to set this value in the data channel earlier and keep sending the *data\_valid* signal at the correct time. In this manner, no data ever needs to be associated to any scheduled event. Channels' data values are changed immediately by the components, while signals *dataOn* and *dataOff* are sent at the correct simulation time

### 5.3.3 Data Sharing Between Components

As a handshake circuit is simulated, some data can be observed being replicated identically over long threads of data channels, thus consuming resources (memory and CPU time) for each copy of the data. One way to optimise this behaviour is to change all these identical copies of the same data to one unique piece of data referenced in each of the places. This section investigates where and when this optimisation is feasible.

Figure 5.8 shows the handshake circuit for a three stage buffer. Each stage contains two data channels linked by a transferrer component.

In this example, data enters from the left-hand side of the circuit, follows the horizontal datapath and exits to the right. It would be nice to avoid copying five times the same data from channel to channel and replace this by a direct copy from the first channel to the last one. Unfortunately, this is impossible as the three stages shown in the figure operate in parallel and each can (and will) carry different data (That actually is the purpose of a three stage buffer). It is however possible to apply data sharing to each Transferrer component. This optimises this particular example to require only two data copies instead of five.

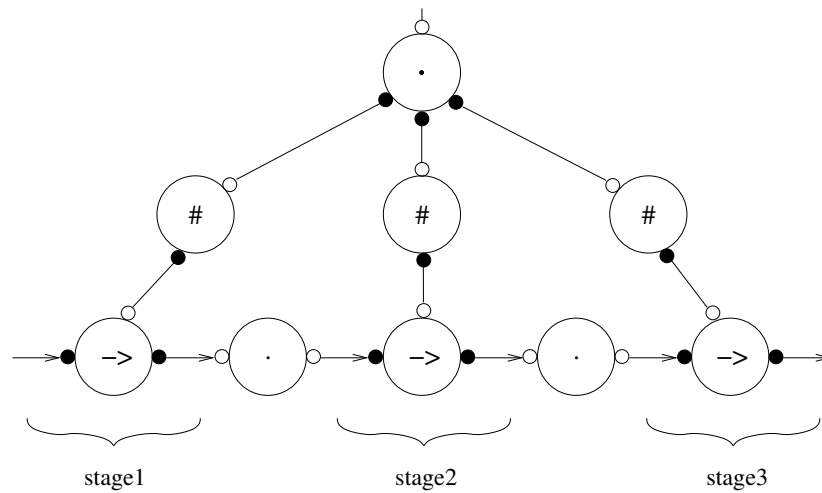


Figure 5.8: Three stage buffer circuit

The validity of this optimisation is verified: As the Transferrer does not have any storage capability, the output data starts being valid only after the input data is valid and always finishes being valid before the other one.

A few components can benefit from the same optimisation: FalseVariable components and Constant components. The gain in speed obtained in practice is lower than 1%, thus negligible. However, this optimisation comes at no cost in the complexity of the simulator. This is a space rather than a speed optimisation.

## 5.4 Test Harnesses

Test harnesses generally require access to the computer's resources (output to screen or files, input from files), which were not originally part of the Balsa HDL. In previous Balsa simulation setups, they were described using LARD, the language originally used at the University of Manchester for modelling the behaviour of asynchronous circuits.

Unfortunately, the synchronisations required to co-simulate LARD and Balsa descriptions, added to the slow simulation speed of LARD, were increasing the simulation time considerably. Furthermore, simulations of LARD and Balsa were visualised by different software tools, making it difficult for the user to observe both of them together.

Some special test harness components have therefore been designed closer to the Balsa level, available for simulation without any loss of speed, and provided with a direct interface in the Balsa visualisation system. These components were originally specially integrated for the simulation of the SPA processor and provide read and write accesses to files and a console; a specific memory component simulates a configurable memory.

These components were first linked to the Balsa description of the circuit via a separate test harness description containing the association of each port of the circuit to a test harness component. They have then evolved to a more integrated structure where test harness components can be addressed directly in the Balsa description of the circuit. They can be described in any programming language, are dynamically loaded for simulation and ignored for synthesis. They also work in Verilog simulations.

## **5.5 Summary**

High simulation speed is intended to make programming by design iteration possible and reduces considerably the time necessary for validating a design by using extensive simulation.

An out-of-order simulator designed for high simulation speed has been described. Further improvements towards high simulation speed have been suggested through techniques specific to the simulation of handshake circuits.

# Chapter 6: Analysis-Oriented Simulation

The previous chapter dealt with techniques for accelerating the simulation of handshake circuits, useful for exploring the design space by successive iterations. During this exploration, the state of the circuit as well as a number of properties and statistics are necessary for the designer to understand how the circuit is behaving during the simulation and therefore to detect misbehaviours. This chapter deals with those aspects of simulation necessary to report any information useful for this kind of analysis.

Usually, designers do not expect a simulation at the handshake circuit level to be very precise. They are used to taking advantage of high level simulators for their speed, and relying on lower level simulators for more precise analyses of what is happening in their design. The simulator designed here follows this idea by favouring speed over analysis capabilities. The motivation of this chapter is thus to search for methods which increase the analysis capabilities of the simulator without affecting the simulation speed. It is explained how timing and power analyses can be implemented at the handshake circuit level. Then it is described how references to source code positions are important for the analysis of the simulation results and how they can be specified. Finally, accepting to trade some simulation performance in favour of analysis benefits, the simulation trace, medium of the information between the simulation and analysis systems, is described.

## 6.1 Timing Analysis

Delay estimates at the handshake component level are intended to help detect performance bottlenecks inherent in an architecture and to provide a basis for circuit optimisation. Delays in components are used by the simulator to compute timestamps of events and therefore determine the order and the speed of the different operations being



simulated. Because the timestamp of an event is equal to the sum of all the delays encountered since the beginning of the simulation until this event, errors in delays accumulate at each step of the simulation. These errors can grow very large, leading to imprecise timing analysis, and sometimes to non-deterministic simulations following different paths to what would happen in a real circuit. For this reason, the precision in the estimation of component delays is crucial.

This section approaches this problem from three different angles. First, a brief view of getting precise estimations of the delays is presented. Then, modelling delays with errors is described. Finally, the problem of how to ignore delays beyond the boundaries of the circuit is tackled.

### **6.1.1 Determining and Adjusting Delays**

The lower the level of simulation, the more precise the timing information. At the handshake circuit level, delays cannot be as precise as at a lower simulation level due to the simplified model in use, which uses an average timing value for each type of handshake components and a fixed delay (if any) for all the channels. During the design of the simulator, each of these average timing values must be calculated in order to be used as fixed values during the simulation. Unfortunately, this aspect of the simulator is not part of the research presented here. It is left as a future work, as complex methods could be used to achieve a better degree of precision.

On a simple scheme, the computation of delays can be done by synthesising and simulating each handshake component at a lower level and measuring the delays obtained. On a more complex and more precise scheme, a co-simulation between the simulation at the handshake circuit level and a simulation at a lower level could provide precise measurements of the delay of each component and each wire. These measurements could be obtained on a small bootstrap simulation and be used for longer simulations.

### 6.1.2 Simulating Delays with Errors

The model of delays with errors introduced in §4.3.2 can be used in the simulator for improving the timing analysis of handshake circuits. The implementation consists of replacing the single *time* value of each timestamp by a tuple (*time*, *error*) or (*min\_time*, *max\_time*), meaning that the event can occur at any time in the interval [*time*-*error*; *time*+*error*] or [*min\_time*; *max\_time*]. The latter of these two notations is preferred because it requires less computation.

The introduction of this model with errors requires modifications in the behaviour of some components in order to compute correctly the output timestamps.

When a component is activated by an event, it will execute a handler which may schedule some new events after certain delays. Such delays are also modelled with an error as (*min\_delay*, *max\_delay*), and the scheduled event will obtain the new timestamp tuple (*min\_time*+*min\_delay*, *max\_time*+*max\_delay*).

When a component is waiting for multiple inputs before executing an action, with the previous model it just has to count the number of inputs received and execute the action when all the inputs are activated, the timestamp of the output being based on the timestamp of the last input received. With the new model, the component needs to use the timestamp tuples (*min\_time<sub>i</sub>*, *max\_time<sub>i</sub>*) of every input in order to compute the output timestamp ( $\max(i = 0, i < n, \text{min\_time}_i)$ ,  $\max(i = 0, i < n, \text{max\_time}_i)$ ).

When a component is waiting for one input amongst multiple inputs, the situation is more complicated. Such a situation only happens with the Arbiter component, which only has two inputs. When an Arbiter component's input gets activated at timestamp (*min\_time<sub>1</sub>*, *max\_time<sub>1</sub>*), the action associated to this input must not be executed before the simulator certifies that no other input can get activated with a *min\_time* smaller or equal to *max\_time<sub>1</sub>*. A metastability window can also be taken into account, as explained in §4.3.1. However, once every desired delay has been taken into account, processing the input events in the correct order is exactly the same problem as the one solved earlier in §5.2.4, when reordering the arbitration inconsistencies in the out-of-order scheduler.

### 6.1.3 Delays in Test Harnesses

In the most recent versions of Balsa, test harnesses can be described using the Balsa language in the same manner as for the main circuit. This poses problems of delays: Sometimes the environment needs to be infinitely fast in order not to affect the timing measurements of the main circuit by having to wait for slow inputs; sometimes the environment needs to be slow to model correctly the delays experienced at the boundaries of a very fast main circuit.

A non-invasive solution at the description level (i.e. the Balsa description of a circuit and its environment do not need to be changed) is to automatically detect test harnesses parts in the easy cases and adopt the “infinitely fast” model, which corresponds to removing every delay in the handshake components making the test harness circuit. Easy cases for the detection of test harnesses correspond to all the automatically generated test harnesses, which always have a structure easily recognisable.

Another solution is to extend the Balsa language with some non-synthesisable timing constructs. Procedures marked as *timeless* will have every delay in their handshake components removed, while a special *delay(int)* call can be used to add a specific delay during the simulation of the handshake circuit by the intermediate of a special handshake component. However, this second modification is outside the scope of this work.

## 6.2 Power Analysis

Since asynchronous circuits are often oriented towards low power and low electromagnetic interferences, being able to analyse such properties is necessary.

Power analysis is done in a similar manner to the timing analysis described above: An average power value is assigned to each type of handshake component by using the results of a lower level simulation. The difference with timing analysis is that power analysis can be done post-simulation, based on the simulation trace. By using the simulation trace, the power analysis tool can integrate the values assigned to each component over different periods of time and different sets of components. Moreover, these consumption values can be changed a posteriori and reflected directly on the power estimates simply by reprocessing the integration calculus. For timing analysis, the delay values assigned to

each component are needed during the simulation to compute the timestamps of events and their ordering.

This model is not realistic, as power should be calculated based on transition counts and loading. However, although not as precise as the profiling information obtainable at lower levels of simulation, the estimations provided by this model can be sufficient for detecting the major consuming areas in a circuit.

## 6.3 Source Code Position Annotation

When an error such as a deadlock is detected during simulation, the guilty channel can be easily pointed out by the simulation trace. The designer/programmer of the circuit is then usually interested in knowing the position corresponding to this channel in the source code. This can be obtained by making the compiler annotate each channel with its corresponding position in the source code. Unfortunately, this “easy” feature does not fulfil all the requirements: Often, a block (procedure) of Balsa source code is called multiple times, leading to multiple implemented instances of the corresponding sub-circuit. In each of them, channels will refer to the same piece of code without distinction. It is therefore required to include a hierarchy of callers that identify each channel in an explicit and unique way. As an illustration, let's consider the following pseudo-Balsa source code:

```

1   Proc sub_common
2       ... <chan X> ...
3   end proc

4   Proc sub1
5       ...
6       sub_common
7       ...
8   end

9   -- Main Part
10      sub1 |
11      sub_common |
12      sub_common

```

Channel X in sub\_common will be instantiated three times. Whenever a bug is detected during the simulation, it is insufficient to know that “channel X at line 2” is the cause of it. A useful description of the channel’s location is needed, including the backtrace of

called procedures with their positions in the code, such as: (dot-separated format with “feature@line\_number” items)

- main@10 . sub1@6 . sub\_common@2 . X,
- main@11 . sub\_common@2 . X, or
- main@12 . sub\_common@2 . X.

This is done by including a hierarchy of callers-callees in the description of the circuit. With the above example, it would lead to any description equivalent to the following one:

```
#0: main procedure
#1: sub1 is called by #0 at line 10
#2: sub_common is called by #0 at line 11
#3: sub_common is called by #0 at line 12
#4: sub_common is called by #1 at line 6
```

And the three channels X would be described as:

```
channel X described at line 2 under context #4
channel X described at line 2 under context #2
channel X described at line 2 under context #3
```

Such a description is used in the new Breeze handshake circuit netlist description, with the caller-callee items being referred to as *call-contexts*.

## 6.4 Simulation Tracing for Offline Analysis

The simulation trace is used as part of the Balsa design flow to transmit information from the simulation to the analysis and visualisation tools (called *clients*).

The aim of a simulation trace is to transmit all the required information while keeping the overhead as low as possible. The first issue is therefore for the simulator to know what information is required by the clients. A simulation trace basically consists of a report of the events occurring during simulation, which correspond to the channel activity inside the handshake circuit. Therefore, if the clients are able to indicate which channels they are interested in, the simulator can generate a shorter simulation trace for this subset of channels.

The second issue is to keep the overhead due to tracing as low as possible. Three sources of overhead can be distinguished: the extra instructions used for gathering the traced data (the instrumentation points), the process of formatting this data into the desired format and the process of writing the result to disk. In the simplest case, the events are intercepted before execution and sent to the trace file with little formatting. The overhead is then mostly due to the amount of traced data (proportional to the number of events), which rapidly gets large. This data can be compressed on-the-fly before being written to the file, but while this reduces the amount of data needed to be transferred, the compression process takes a significant amount of time to be executed and uncompression processes are also necessary on the client side.

The third issue concerns the communication medium. The choice of a file instead of a more direct communication link such as a socket or a pipe presents the following advantages:

- Multiple clients can read simultaneously, while requiring only one write process.
- Generation and analysis can be decoupled, for delayed/offline analysis.
- The data can be analysed repeatedly off-line to find errors in the program execution.
- If the file is written on a network file system, the simulator and the different clients can run on different machines, for performance or multiple display benefits.

The disadvantages of using a file as an intermediate medium are a loss of speed due to writing on a physical medium, made worse by the fact that trace files are generally extremely large.

### **6.4.1 Standard Trace**

Classical event-driven circuit simulation trace formats (e.g. the Value Change Dump (VCD) format[56]) consist of:

- a header and definitions section describing the static parts of the simulated circuit and introducing some convenient abbreviations to reference these parts in the second section;
- a list of events, reported in increasing timestamp order.

The structure of the Breeze simulation trace follows this scheme: The header section lists all the channels of the circuit with their properties: width, source code position and a unique identifier (a sequential number). The main section consists of a timestamp-sorted list of channel activity, which includes the request/acknowledge signal transitions and changes in the data.

The different clients reading the trace file can easily reconstruct the complete state of the circuit at any given time from the reported events.

### **6.4.2 Out-of-Order Trace**

When the out-of-order simulation scheduler is used, events in the simulator are not happening in timestamp order. On the clients' side, however, it is always necessary to have a sorted list of events or a way to sort them before being able to process them. The traced events therefore need to be sorted by timestamp either by the simulator at the time of writing or by each client at the time of reading. An intermediate program inserted just after the simulator's output and before the clients could be also used to sort the trace file.

During the sorting operation, an event can only be written into the output file when it is certain that no other event with an earlier timestamp will arrive later. The sorting process must therefore either wait for the end of the whole simulation, or rely on the simulator to provide some information about a minimal global time of the system.

This minimal global time is already calculated by the simulator when no event other than non-deterministic choice events (due to Arbiter components) are scheduled (see §5.2.4). Unfortunately, this can take a long time before obtaining such an information, if at all in the case of a circuit without Arbiter. Luckily, the computation of the minimal global time can be forced, for example at regular interval of real time, in order to process the sorting algorithm and output those events which happened before the current minimal global time.

### **6.4.3 Pattern Analysis and Compressed Out-of-Order Trace**

Compressing the traced data before writing allows less data to be written in the trace file, resulting in less writing overhead. In the case of compression of the standard trace,

however, the overhead gained in writing less data is lost in having to keep the data in memory buffers before compression instead of sending it directly to the trace file. In the case of the out-of-order trace, all the data already needs to be kept inside buffers in order to be reordered. It is therefore an opportunity for compressing the data at the same time with little extra-overhead (by choosing an appropriate compression method).

The main motivation for compressing out-of-order traces is that their particular structure is appropriate for compression: During out-of-order simulation, handshake components are directly calling their successors, which leads to threads of components being executed sequentially in the same order every time they are called. When more than one thread of components are executed in parallel in the real circuit, they are simulated sequentially with the out-of-order scheduler, while they are interleaved (sorted by timestamp) with the standard scheduler. This is sketched in Figure 6.1.

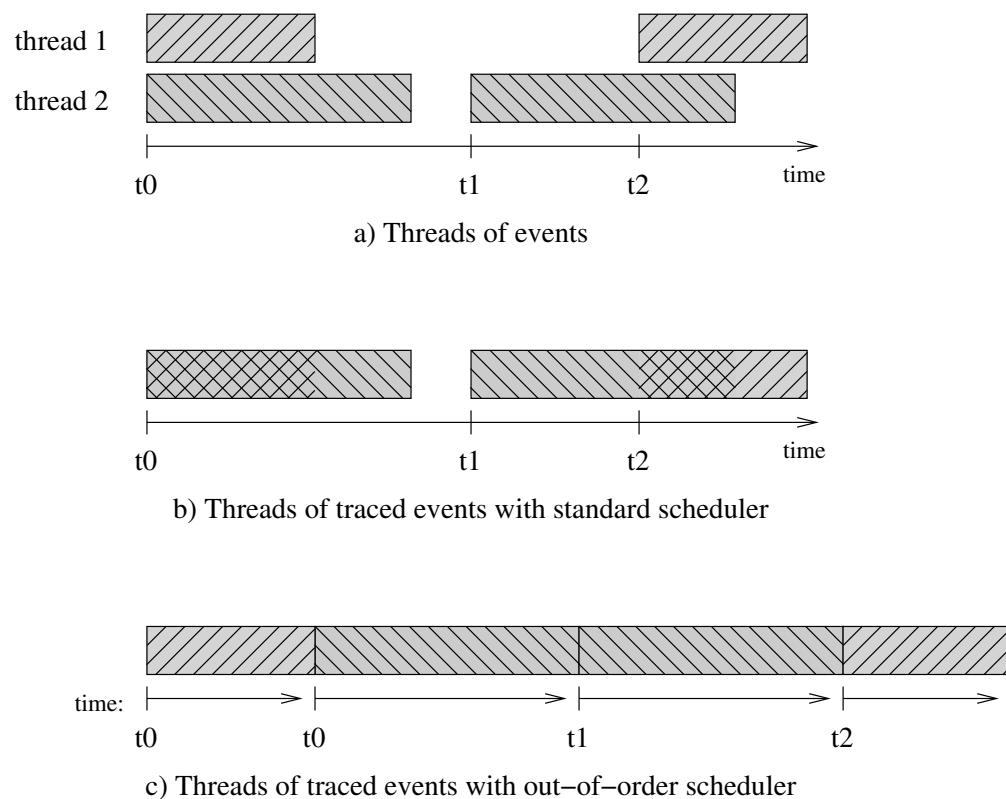


Figure 6.1: Interleaved and sequential traces of threads of events



The fact that threads of events are not interlaced with each other makes them very easy to be detected and extracted. During a long simulation, the same thread may be frequently repeated many times identically due to the loops in the circuit description. Very often the only things that are changing between two occurrences of the same thread are data values. It is, however, rare that a same thread is repeated many times with the same data. The pattern analysis/compression algorithm should therefore recognise repeated patterns by their control flow, and keep data values as secondary information specific to each occurrence.

## **6.5 Summary**

This chapter presented techniques for increasing the analysis capabilities of the handshake circuit simulator described in the previous chapter without affecting its simulation speed. Timing and power analyses, followed by specifications of source code positions were treated. Finally, the simulation trace has been described, with an emphasis on the simulation trace generated by the out-of-order scheduler. This out-of-order trace can easily be used for pattern analysis and compression.

# Chapter 7: Visualisation

Chifosky and Cross II give the following definition of reverse engineering [23]: “Reverse engineering is the process of analyzing a subject system to identify the system’s components and their interrelationships and create representations of the system in another form or at a higher level of abstraction”.

In this sense the visualisation system presented in this section is a reverse engineering tool used for program understanding. It extracts important information from all the available sources and builds a dynamic and interactive representation of the system in order to help designers understand the structure and behaviour of the handshake circuit.

Three visualisation techniques are developed in this chapter. First, three sources – the Balsa source description, the compiled handshake circuit, and the simulation trace – are analysed and assembled together to construct a graph structure combining the qualities of each source of information. This structure is intended to be visualisable at any level of detail. The second technique adds the time dimension to the previously obtained picture. It animates the static graph using a colour-based representation of the simulation trace events. Finally, other simpler views are taken from a typical design environment and are connected together following a collaborative scheme, allowing them to exchange data for efficient inter-view element tracking and navigation.

## 7.1 Information Clustering

The simulation of a handshake circuit reveals a huge amount of information. In order to be understandable by a human user, this information needs to be structured into meaningful groups. Starting from the raw handshake circuit graph, this section shows how

clusters can be formed by using other related sources of information and how they can be used for understanding and debugging asynchronous circuits.

This section describes a number of different methods for grouping handshake circuit elements, either by proximity (functional grouping), by threads (control threads and data flows) or by behavioural properties (test harness isolation).

All these methods correspond to static allocations of the groups: Although they may need some information taken from the simulation in order to be determined, the groups do not change dynamically during the simulation.

### 7.1.1 Functional Grouping

The network of handshake components is derived from the high level description of the circuit in Balsa, which itself is organised by structural information such as procedures, functions, instruction blocks and local variables. Given the close relationship (due to transparent syntax-directed translation) between the Balsa description and the generated handshake circuit, it is logical to try to transfer this high-level structure onto the lower-level handshake circuit in order to partition this huge network into more manageable chunks.

Figure 7.1 shows how functional grouping dramatically improves the visualisation of an (abstract) set as small as ten components. The representation on the right not only shows an additional structure, but also suggests a similarity between the three groups (this should be used cautiously, as the similarity may not be true at the description level). Not shown on this figure is the recursive aspect of functional grouping: The Balsa description contains procedures and local sub-procedures inside these procedures (as well as other local structures such as functions, instruction blocks and local variables), resulting in nested groups and sub-groups in the graph of the handshake circuit. Experiments on large circuits (the SPA example, §9.3) show that functional grouping applied to procedures and functions usually divides a graph of  $n$  elements into groups of about  $\sqrt{n}$  elements. A reasonably large network of ten thousand handshake components would therefore be divided into groups of about hundred elements. However, this number is an average:

Many groups will be smaller than a hundred elements, while a few groups could reach thousand items or more, which is still too large for an efficient visualisation.

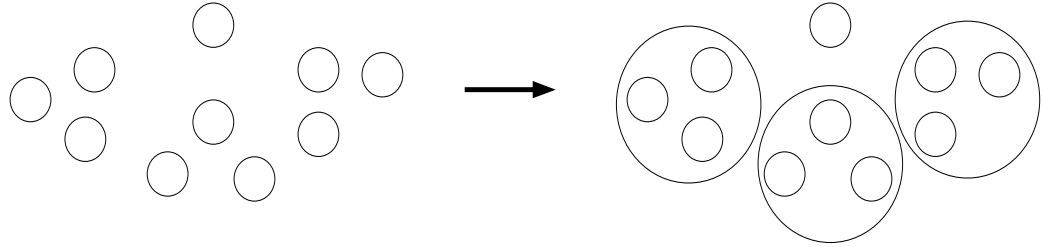


Figure 7.1: Abstracted functional grouping

When functional grouping is applied to every instruction block (i.e. blocks of instructions contained between language keywords, such as the division: `if <block> then <block> else <block> end`), the original circuit gets divided into small groups of usually less than five elements. This unfortunately results in a huge, and thus difficult to manage, quantity of nested groups: The clusters themselves are wasting the entire visualisation area and are as difficult to organise as the original flat graph. Such clustering can however be applied by limiting the clustering depth or by setting a minimum number of elements per group.

### Clustering Optimisation

The Balsa language is composed of many block structures (see BNF in appendix of the Balsa manual [32]). Those can be used to cluster channels and components at a very fine granularity: At the lowest level, groups are made of only a few channels and components.

The structure of the Balsa source code is reported in the Breeze file by an extension of the Balsa→Breeze compiler: A new section expresses the hierarchical structure of the nested Balsa blocks/structures with a unique identifier for each of them, these identifiers being used by the handshake channels and components as a means to indicate their location.

A few optimisations in this structure are useful. They concern structures with undefined number of elements, such as the sequence and parallel structures, which are recursively defined in the Balsa language (and parsed as such by the Balsa parser). For instance, the Balsa block

```
action1 ; action2 ; action3 ; action 4
```

would be parsed and internally stored (due to the compiler's construction) as

```
action1 ; ( action2 ; ( action3 ; action 4 ) ).
```

A direct output would thus look like:

```
Sequence Block
  action1
  Sequence Block
    action 2
    Sequence Block
      action3
      action4
    End Sequence
  End Sequence
End Sequence
```

The following optimisation would generally be considered better:

```
Sequence Block
  action1
  action2
  action3
  action4
End Sequence
```

Figure 7.2 shows what could be the graphical representations of the unoptimised and optimised forms of the sequence example. The optimised form provides better readability by placing every action at a common level and avoids wasting space with unnecessary block frames.

### **Variable Groups**

Two more functional groupings concern variables.

First comes the case where exactly one Breeze Variable handshake component is associated to a Balsa variable. Breeze Variable components have, by design, only a single write port. When the Balsa code contains more than one writer to the same variable, a tree of CallMux (data merges) and Encode components is used to combine writers from all

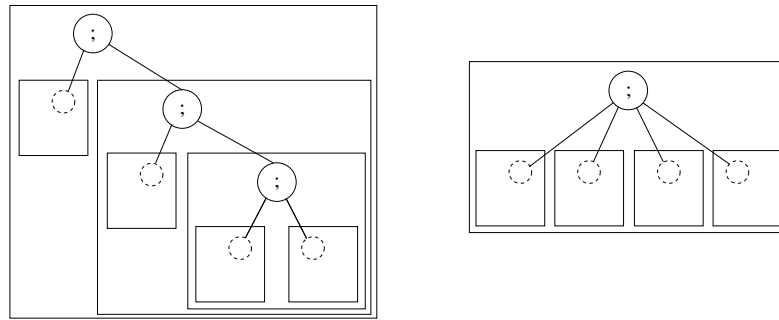


Figure 7.2: Unoptimised and optimised representations of a sequence of 4 actions

sources into one. In the visualisation, this tree has no reason to belong to the same group as one of the writers. It should then appear next to the Variable component, hence the grouping. The result of this grouping does not have any significant consequence on the number of groups and the number of components inside groups. However, the components grouped together by this method have a strong relationship, which improves greatly the visualisation, as such groups may transparently be reduced to single elements representing variables with many write ports.

Sometimes, Balsa variables are distributed into more than one Breeze Variable component. This is for processes needing only a few bits of a variable to avoid reading the whole variable (and therefore holding this resource). When the whole variable is required, a tree of Combine components is used to reconstitute the data from its parts. In the same way as with the writers' tree, this tree of components may be clustered with the Variable component.

### 7.1.2 Control Threads

Considering a handshake circuit, a control thread is a set of contiguous communication channels in which only one channel is allowed to change state at a time.

The control threads of a handshake circuit can be chosen to form a partition of its set of communication channels, but this partitioning is not unique: Figure 7.3 shows an obvious case of three possible choices of control threads for a Fork component. The first solution makes use of three threads whereas the other two only need two threads to partition this

circuit part. All partitions are thus not equal and can have different numbers of threads, which raises the questions:

- What makes a thread “good” (or “useful”)?
- Is a thread of maximum size better than any of its sub-threads?
- Is a partition of minimum number of threads best?

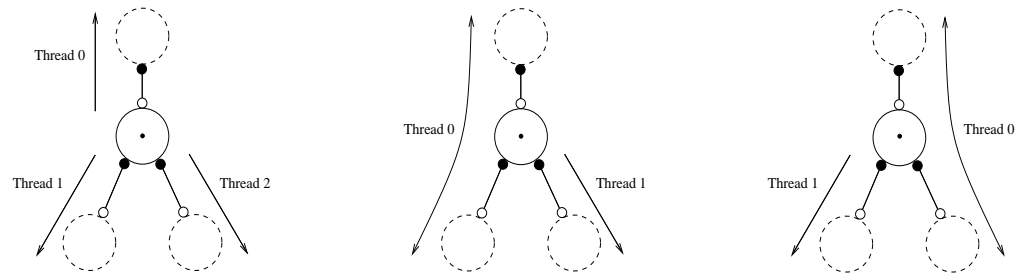
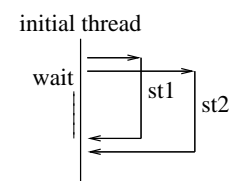


Figure 7.3: Three control thread sets possible with a Fork component

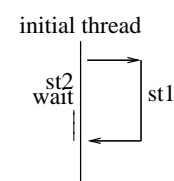
Before answering the previous questions, a different view of control threads is necessary: A control thread as defined previously, also corresponds exactly to the idea of threads of execution in a high-level language such as Balsa. Using the example of Figure 7.3: The original source code in Balsa would be a parallel statement such as “statement1 || statement2”. When asked to write a lower-level code that simulates this behaviour, three solutions are obvious:

```
thread t1 = create_thread (statement1)
thread t2 = create_thread (statement2)
wait_for (t1, t2)
```



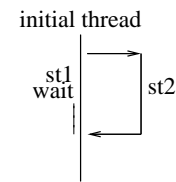
or

```
thread t1 = create_thread (statement1)
execute (statement2)
wait_for (t1)
```



or the symmetric

```
thread t2 = create_thread (statement2)
execute (statement1)
wait_for (t2)
```



Which solution is best? It depends on many parameters: the architecture on which the threads will be executed, the habits of the programmer in writing software, his habits and preferences for the visualisation of threads during debugging, etc.

**Is a thread of maximum size better than any of its sub-threads?**

**Is a partition of minimum number of threads best?**

Answering “Yes” to these two questions would lead to favour the two last partitions to the first one in the Fork example. Yet, one can think of a situation where the first solution with three threads may be preferred. This is the case when both threads 2 and 3 are doing exactly the same thing: There is no reason to justify the grouping of only one of them with thread 1 while leaving the other thread independent.

**What makes a thread “good” (or “useful”)?**

Threads have been introduced to solve the problem of visualising too many components simultaneously by grouping them together. They are therefore directly related to the visualisation theory and to the notion of perception, which implies aspects of beauty and efficiency such as proportions and symmetry: Processes doing equivalent jobs should have an equivalent representation. This agrees with the previous idea of preferring a symmetric distribution of the threads in the case of a Fork component with two equivalent branches. Therefore, a partition of threads can be qualified as “good” when its visualisation reflects accurately the behaviour of the different parts of the circuit.

### 7.1.3 Data Flow

Communication channels are composed of both a control part and a data part. It is therefore possible to apply the same theory to data flow as used with control threads. The main difference is that, whereas all channels contain a control part, only some of them are



carrying data. The partition of a circuit by data flows is then incomplete, restricted to data channels. Yet, one can consider that in a circuit, data channels are the most important channels as they carry the information. It is therefore interesting to provide a grouping scheme dedicated to these important elements.

The data Transferrer component (or simply Transferrer), represented in Figure 7.4, is the simplest and most used handshake component dealing with data channels. It transfers on demand the data contained on one channel to another. The data flow associated with this component is obvious, and runs from left to right in a single flow.

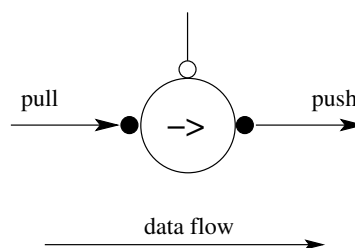


Figure 7.4: The data Transferrer component

Other data processing components are not as simple as the Transferrer: Figure 7.5 shows two versions of what would be an Add component (the real components having this functionality are more generic ones called `BinaryFunc` and `BinaryFuncPush` components) and a Split component. The two Add components have identical data flows, with their two data inputs merging into one output. They differ by their control schemes, the first one being a control fork and the second one a control merger. On the other hand, the Pull Add and the Split components share the same control scheme while having opposite data directions: data merge for the Add component and data fork for the Split component.

More annoying is the `CaseFetch` component, which does not follow the simple rule “every data output is connected to every data input”, and therefore requires special handling during the detection of data flow. Past this difficulty, the data flow inside this component is of the simplest form, as shown in Figure 7.6.

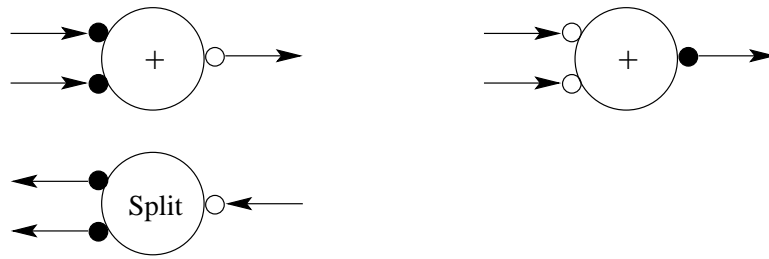


Figure 7.5: Pull and push Add (BinaryFunc) and Split components

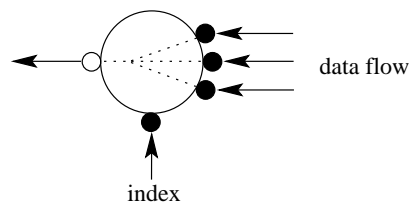


Figure 7.6: CaseFetch component

This analysis of data flow is based on the same model as control threads, resulting in some kinds of *data threads* with forks, in the case of Split components, and with merges, in the case of BinaryFunc components (for example). The data flow of a data channel is defined as the union of the data threads starting from this channel and connected together tail-to-head by common components. These data flows can be used for finding the range of action of a particular data. A backward data flow can also be defined as the union of the data threads connected together tail-to-head and ending at a specified data channel. This can be used for finding the causes leading to a particular data.

The main problem with data flow sets is that they can grow to a very large size. This can be improved by using the simulation trace to follow only paths which are actually taken during the simulation. This is particularly effective with some components such as the CaseFetch component. When this component is activated, it requests a value onto its *index* channel, and this value is then used to indicate which input needs to be used to fetch a value which will then be forwarded to the main port. Many inputs can usually be discarded when the simulation shows that they are not used during the analysed period of

time, thus simplifying the data flows. This optimisation also works, but to a lesser extent, with other components.

### **Notes on control and data flows**

A distinction can be made between control flow-based threads which always result in threads having a unique instantiation and data flow-based threads which can have multiple instances running at the same time. For example, the data flow thread going from one end of a pipeline to the other end can hold multiple values flowing at the same time and taking the same route. Each value flow can be seen as a distinct instance of the same defined thread.

The visualisation of flows of data and of control states is useful for debugging as it allows the designer to verify that data and threads of execution proceed correctly inside the circuit. Observing the flow of data and the state of control can also be useful for optimising the circuit, since the designer can detect if some control flows are not finishing as early as expected. This happens for example when some control components are “wasting time” waiting for the return-to-zero phases of other components in a 4-phase protocol.

Finally, mapping the clustered graph reflecting the structure of the Balsa description onto the layout of the graph of the handshake circuit is useful as it allows low-level information such as data flow to be visualised on the high-level Balsa structure.

### **7.1.4 Test Harnesses**

The specific problem of information clustering concerning test harnesses is that they are supposedly part of the environment but, with the new version of Balsa, they are appearing in the handshake circuit at the same level as the main circuit. Even Balsa libraries to use within test harnesses are written with the Balsa language itself. The difficulty is therefore to detect those tests which are now embedded inside the handshake circuit as normal Balsa code in order to treat them differently from the main circuit. For example, at the simulation level, test harnesses are often better simulated as timeless circuits (the environment delays do not have to influence the tested circuit). At the visualisation level,

tests harnesses connected to inputs and outputs of the main circuit are better visualised by input/output components with specific interfaces for controlling them.

## 7.2 Multi-Source Graph View

The clustering techniques exposed in the previous section can be used to visualise efficiently the information, first as a static graph, and then as an animation atop the static graph.

### 7.2.1 Static Multiscale Structure

Combining multiple sources of information offers the following benefits:

- This information can be used to cluster some components from one source by using the information from another source. Clustering allows the number of elements to be processed at a time to be reduced by processing them by group instead of individually.
- Sources usually used to visualise at different scales are combined to make a graph viewable at any scale.
- More clues are available to reconstitute the designer's mental image of the circuit.

A static view of the handshake circuit is constructed from the hierarchical graph obtained after clustering. This graph view enjoys the above-mentioned benefits.

#### **Multiscale visualisation**

The clustering techniques exposed in the previous section not only have the beneficial effect of separating a huge number of handshake elements into fewer manageable groups. They also have the important consequence of transferring the structure of each analysed source of information onto the handshake circuit. The resulting hierarchical structure exhibits the advantages of all the source structures.

One important implication comes from the fact that different sources of information are usually used to visualise the structure and behaviour of a circuit at different scales (or level of detail): The hierarchy of Balsa procedures gives a high-level representation of the

description, data and control flows can show an intermediate level, while traced events happen on handshake channels at a low-level. Therefore, the structure obtained after clustering can be visualised at any scale and always shows useful information: From the global view of the circuit to the lowest level, the main components of the circuit can be distinguished, followed by the (sometimes recursive) high-level implementations of the modules, and on until the detailed implementation of the modules, precise enough to visually understand their behaviour. This is illustrated in the results chapter (Chapter 9).

### **Fitting the user's mental image**

The following references to the designer's mental image are integrated in the graph view:

- The written Balsa source code.
- The generated handshake circuit – a good designer will be able to anticipate (and *will* anticipate) the generated circuit when writing Balsa code.
- The execution trace: During the design of a circuit, the designer always anticipates the amount of information flowing on the different channels/buses and bases the circuit's architecture on this information.
- The visualisation software interface with the user (treated further in this chapter): Human interaction can be used to designate important regions of the circuit, to correct software guesses about the architecture, etc.

All the elements coming from the designer's mind to create the circuit are therefore gathered together in the graph view, hopefully making for an intelligible representation.

### **7.2.2 Dynamic Colour-Based Animation**

The last section generated a representation of the circuit structure by organising the components of the circuit in an easily readable way. Based on the simulation trace, the role of the animation module is to add further information to the static picture in order to represent the data and control flows, and the changing activity of the components during the simulation.

This is achieved by marking the handshake circuit graph with colour annotations: Each component or channel's state is represented by a colour, and the circuit is animated as the simulation system updates the states of the components and channels.

The advantage of such an animation system is its ability to show all the information available from the Balsa description and from the execution of the simulation of the system, and then let the user decide what he wants to focus on. Debugging is made easier through the visualisation of the parallel activity: Every thread of execution of a simulation can be shown simultaneously, and the observer can focus on one specific thread, observe its activity, and can easily observe its merging with another thread or its splitting into two threads. Moreover, every thread is ensured not to overlap with any other in the visualisation area, whereas they often overlap on a source file description.

This animation system also provides some interesting debugging features for deadlocks and livelocks. When a deadlock situation arises, the program stops, leaving the guilty components in a specific colour and the trace of the components before them in another colour, making it less difficult to debug. In a livelock situation, the colours can be observed circling in an endless loop, but while this identifies the components involved it does not indicate the entry condition.

Moreover, the one-to-one correspondence between the Balsa description and the visualised handshake components makes it easy to link any error located on the visualised circuit with its corresponding location in the Balsa description.

## **7.3 Coordinated/Collaborative Views**

Until now, the visualisation system was used to visualise multiple sources of information together in a single view. The benefits of such a view have been presented. However, in order to be useable and intuitive, a visualisation system must also take into consideration what the user wants to see. Most of the time, the user/designer wants to continue to use the same style of design he always used. In the case of asynchronous design with Balsa, the views usually consist of a text editor containing the source code, and a waveform viewer to analyse the results of the simulation. In order to make these views even more

useful, a collaboration scheme is suggested in this section to track the visualised elements and navigate efficiently from one view to another.

### **7.3.1 Views**

The main view (Figure 7.7, top-left) is accompanied by a number of other views, representing the designed system at other levels:

#### **Source Code View**

The source code view (Figure 7.7, top-right) is a text viewer showing the Balsa source code. The designer's preferred text editor can also be used. Although this is the simplest of all the views when used individually, this is the most difficult to link bi-directionally to the rest, as simple text viewers are not generally designed to display anything but text or to forward keyboard/mouse events.

#### **GTKWave**

GTKWave [48] (Figure 7.7, middle-left) is an external program used to display waveforms of the handshake channels. It is directly and bi-directionally linked to the handshake circuit visualisation system, which provides the user with an interface to select which channels are to be displayed in GTKWave. In return, GTKWave can be used to select some channels and periods of time of interest over which some processing actions can be executed by the visualisation system, such as calculating the power consumption of a sub-circuit over a certain period of time.

#### **Verilog Description (generated from Balsa)**

The Verilog description (Figure 7.7, middle-right) is generated from the Balsa description and corresponds to a direct translation from handshake circuit to Verilog.

#### **Time Line Behaviour View**

The time line behaviour view (Figure 7.7, bottom) shows the activity of clusters of channels (y-axis) over time (x-axis). This view is intended to help the user detect repeating patterns visually. The advantage of this method is that patterns do not have to be exactly identical to look similar. Detecting non-exact pattern repetitions is very hard to

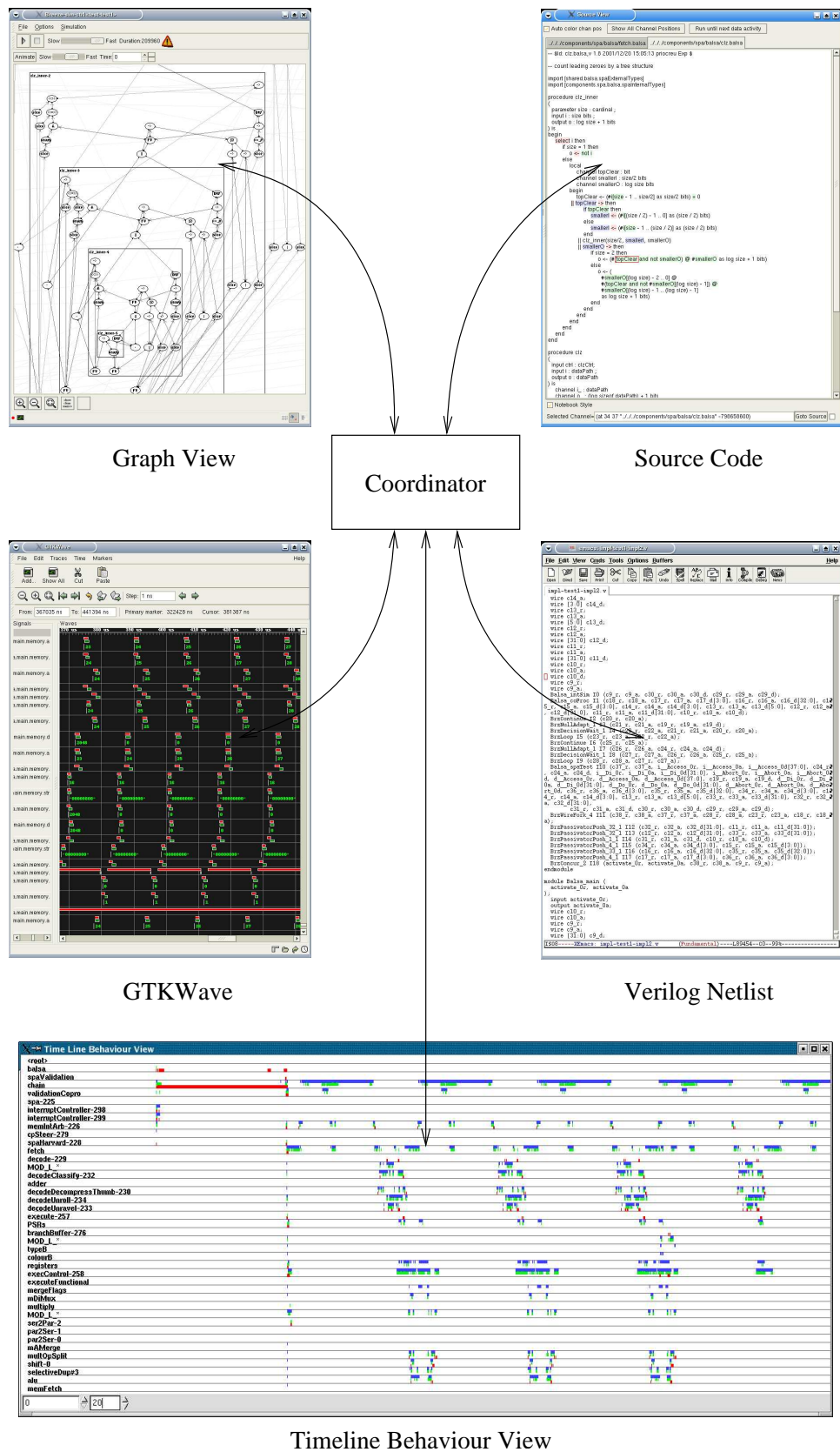


Figure 7.7: Coordinated/collaborative views



automate. In the Timeline Behaviour View, Figure 7.7, a pattern “non-exactly” repeated four times can be identified.

### Test Harnesses

An interface is provided for visualising the activity of test harness components. Input from files and output to files and console are displayed together with the channel activity. The special test harness Memory component gets a more complex interface where it is possible to visualise and edit the contents of the simulated memory.

## 7.3.2 Multiple Views: Linking the Different Representations

A number of different views of the design have been described so far. In each of these views, a subset of the whole information set (source code, handshake circuit, simulation trace) is represented. This section explains how the different representations of a same item in different views are linked together. This linking is used for synchronising views when a component’s properties are changed in one of them. More importantly, it also allows the user to switch efficiently from one view to another, for instance: going from a source code statement to the corresponding channel in the handshake circuit view and vice versa.

Table 7.1 shows what information is exposed in each view. Table 7.2 shows where this information comes from.

The visualised elements can be categorised into three groups:

- The *simple* elements originally contained in the Breeze and simulation trace files: procedures, ports, channels, components, time and events. They are visualised in most of the views.
- The *compiled* elements: data and control flows, states. These are generated after analysis of the simple elements.
- The Balsa statements, present only in the original Balsa source code and visualised only in the source code view.

The following elements are used as links between views:

Graph View	Element List View	Channel Selection List
procedures ports channels components data&control flows events, states	procedures ports channels components	channels

Timeline, Menus	Timeline Behaviour View	GtkWave
time	procedures time groups of events	time channels events, states

Source Code View	Verilog Backend
Balsa statements variables blocks structures procedures	channels components

Table 7.1: Visualised elements per view

Balsa Source File	Breeze File	Simulation Trace	Verilog
Balsa statements variables blocks structures procedures	procedures ports channels components	time events	channels components

Table 7.2: Visualised elements per source

### **Source Code Elements**

Source code elements (or statements) constitute the special case of this section: They are not represented directly but are “cross-referenced” by other elements via their position in the source code files. Balsa source code statements are first referenced in Breeze files as channel positions. This work extends this to any component that can be associated with a channel, such as handshake components and simulation events. The source code position is the invisible medium of the association between Balsa statements and visualised elements.

### **Handshake Channels**

Handshake channels are first generated in Breeze files. They are directly and fully visualised in most of the views and are therefore the preferred means for going from one view to another. They are the most fine-grained components of handshake circuits, and as such provide efficient ways of manipulating and associating the different views at a low level of the design. In addition to themselves being the link between different views, handshake channels contain the source code position of the Balsa statement they have been compiled from. This allows every view containing channels to report references to the original source code, necessary for correctly reporting errors to aid the debugging process.

### **Handshake Components**

Handshake components are the base execution blocks of handshake circuits. They can usually be logically associated to Balsa operations. The link between a handshake component and the corresponding source code is implemented by the intermediate of handshake channels. Components are connected to channels, among which one channel can always be seen as ‘more special’ than the others: Variable components have a unique write port, BinaryFunc components have a unique output, most components have a unique activation port, etc. The association between a component and the source code is made through this special channel.

### **Procedures**

Procedures are described in Balsa and compiled as groups of handshake channels and components. Their coarse grain allows the user to manipulate large circuits at a high level before going into the details of handshake channels and components.

### **Simulation Time**

Simulation time is an easy parameter to deal with: A single number allows to specify the current simulation time in a view and see the correspondence in other views. The only slight difficulty is due to the asynchronous nature of the circuits being designed, which let events happen at any time instead of at regular intervals of time as would be the case with synchronous circuits.

## **7.4 Additional techniques**

This section describes two additional techniques used during the visualisation process: a technique for laying out the static graph used in §7.2, and a method for tracking structural changes during design iterations.

### **7.4.1 Dot Layout**

One step missing in this visualisation framework is the static layout of the graph. As much research has been previously carried out on this subject, this stage has been skipped and a very simple technique using already available tools has been employed. The Dot tool from the graphviz package is used here [44, 45].

Dot is a freely available tool developed by AT&T that can layout network graphs on a plane. It works with nested clustered structures. Its main advantage is its ready availability, which makes it possible to obtain a first method of layout with little effort. Unfortunately, Dot suffers some disadvantages. The most problematic one is its inability to process too large graphs. Other inconveniences in Dot are the slow processing speed and the poor final shape of the laid out graph, which is almost always overly developed either in width or in height and almost never nicely square.

The only solution has been to keep the graphs fed into Dot below a certain size (determined by trial and error). Two ways of restraining the size of the graphs present themselves: Partitioning the flat graph into a set of smaller graphs, or using the nested clusters' information to recursively layout the subgraphs by applying dot on each of them. The first idea quickly appears to break down: The way dot places the interface ports to the circuit, the connections between partitions are visually very unpleasant.

### **Recursive Dot Layout**

It is possible to apply Dot recursively on the nested clusters of the graph in a bottom-up fashion. The nested structure is first traversed top-down and Dot is called on the way up in order to recursively calculate the area occupied by each group at the next (higher) level. The main advantage of this technique is that identical graphs are laid out in an identical manner, whereas applying Dot on a nested graph (in one operation, not recursively) leads to subgraphs laid out differently, in order to accommodate the edges between subgroups. Identical representations for identical graphs helps recognising repeated instantiations of the same procedures.

## **7.4.2 Tracking Structural Changes during Design Iterations**

When a Balsa description is modified and recompiled, the generated handshake circuit reflects those changes. Due to the syntax-directed nature of the compilation, small modifications in the Balsa description are translated into small changes in the resulting handshake circuit. This important property makes the visualisation system a very interesting tool if it manages to show the transformation from one version of the circuit to the next one: Students can conveniently see the result of source code modifications onto the handshake circuit and designers can keep their visualisation system and debugging environment open even as the circuit evolves and is recompiled.

Given two versions of a Balsa source description, the visualisation of one of them with reference to the other can be considered in two ways: Either the Balsa compiler can analyse both sources and output the changes in addition to the new handshake circuit (figure 7.8-left), or both sources can be normally compiled into two handshake circuits

which are later compared and sent to the visualisation system (figure 7.8-right). In the latter case, the differentiation process can be integrated within the visualisation system.

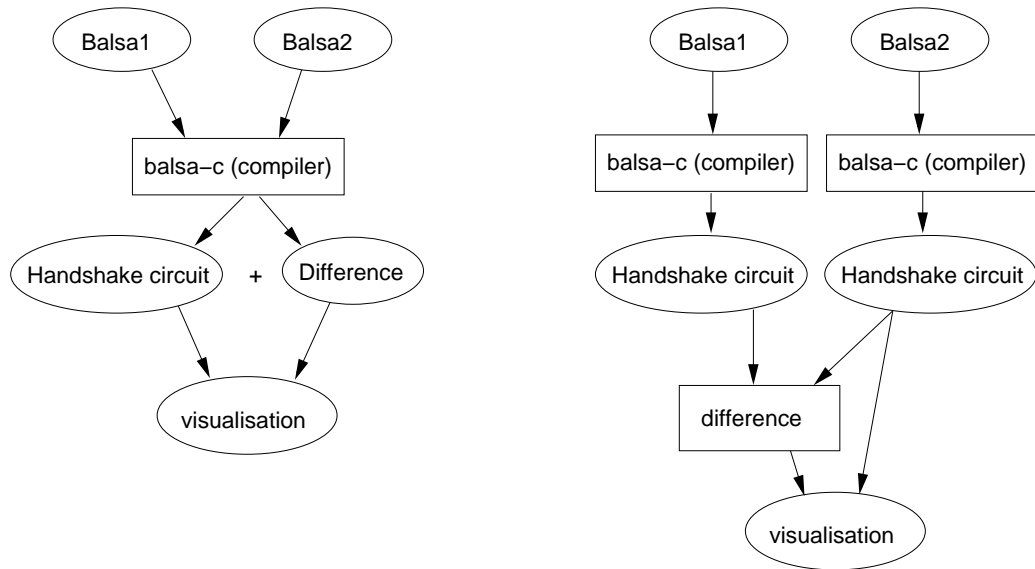


Figure 7.8: Circuit reconfiguration strategies

The latter case gathers all the advantages:

- The Balsa source code always has a more complex structure than its handshake circuit. It is then advantageous to parse it only once. When considering a complete design process with a series of source code modifications and recompilations, the first strategy forces parsing of each Balsa file twice: once when compared to the previous version and once when compared to the next version. The second strategy can keep the compiled handshake circuit in a file after the first compilation in order to use it later for comparison with the next version.
- The Balsa compiler is a complex piece of software that is better kept untouched.

The visualisation of the changes requires some additional techniques. Some research [53, 114] successfully employed structural animations from one graph to another, with smooth transitions in between.

Structure tracking has not been implemented in the current framework, and has therefore not been evaluated in this thesis. It is however believed that it would benefit greatly the design iteration process, and for this reason has been described here.

## 7.5 Summary

Three visualisation techniques are developed in this chapter. First, three sources (the Balsa source description, the compiled handshake circuit and the simulation trace) are analysed and assembled together to construct a graph structure combining the qualities of each source of information. The obtained structure is viewable at any level of detail. The second technique adds the time dimension to the previous picture. It animates the static graph using a colour-based representation of the simulation trace events. Finally, other simpler views are taken from a typical design environment and are connected together following a collaborative scheme, allowing them to exchange data for efficient inter-view element tracking and navigation.

## Chapter 8: Integration

This chapter covers the integration of the results issued from the previous chapters into a unified framework.

The main component of this framework is the handshake circuit visualisation system, designed after the research presented in the previous chapter. As seen previously, the visualisation is based on multiple sources of information. These sources – contained in the Breeze and Balsa files and the simulation trace – are therefore provided as inputs to the visualisation system. The handshake circuit simulator (Chapters 5 and 6) is designed as a separate process and connected to the rest via a trace file. Finally, the debugging techniques presented in Chapter 4 are integrated into the visualisation system, as they require user interaction and visualisation of the results, both available at the visualisation system level. These new components are added to the Balsa flow as shown in Figure 8.1.

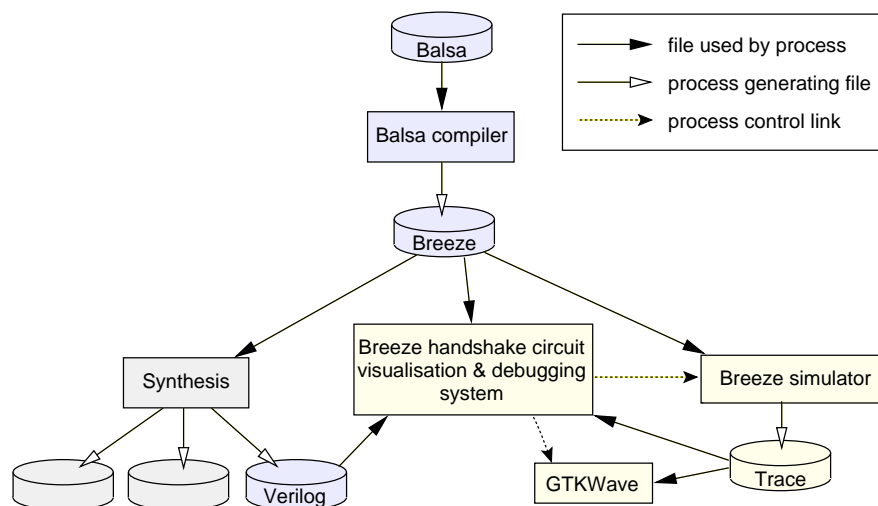


Figure 8.1: New Balsa simulation and visualisation flow



The external application GTKWave is used as a component of the visualisation system and its integration is effected via a link giving full control of GTKWave to the visualisation system. In the same way, a control link allows the visualisation system to run and administrate the simulator process. From the users' point of view, all the interactions are thus gathered in one place. The rest of this chapter deals with the necessary additions to make these elements cohabit efficiently and transparently.

## 8.1 Balsa Compiler and Breeze Format

The Breeze file, medium of the handshake circuit structure, has been extended in two ways to accommodate the visualisation requirements. First, some knowledge about the original Balsa structure has been added to the Breeze structure via the use of *call-contexts*. Then, as the Breeze file is now acting as an input for multiple tools, a linking process is useful to gather descriptions compiled into multiple Breeze files into a single file.

### Call-contexts

Call-contexts are used to save in the Breeze file the hierarchical structure of the original Balsa description. In Balsa, a procedure call to a non-shared procedure means that the callee's circuit will be included, after compilation into a handshake circuit, inside the caller's circuit. When a Breeze file is flattened, either for optimisation purposes or after linkage, the callee's lists of handshake components and channels are concatenated to the caller's lists in order to be compatible with the original Breeze format. The knowledge about caller and callee is therefore lost. To solve this problem, a list of *call-contexts* is appended at the end of the flattened procedure's description in the Breeze file, containing an identifier, some debugging information about the called procedure, and a pointer to the callee's call-context. These call-contexts therefore form a tree representing the architecture of the circuit at the procedure level. To each handshake channel in the Breeze file is added a reference to one of these call-contexts by means of their identifiers, in order to indicate which procedure it was originally described in.

### Handshake Circuit Linker

When a circuit is described by multiple Balsa files, each of them is compiled separately by the Balsa compiler and one Breeze file is generated for each Balsa file. Applications

needing to exploit the entire circuit, either for synthesis, visualisation or simulation, are required to regroup the information contained inside all the Breeze files before being able to carry on with their work. This is called *linking*. The linking process can either be left to each application needing to use the handshake circuit, or be done once for all at the end of compilation. The first solution was previously used when synthesis was the only really used route from a Breeze file. It avoided the storage of the linked Breeze file. However, for multiple simulation, visualisation and synthesis, letting every process link the same Breeze files consumes more resources than it can save. The linking process has therefore been moved at the end of the compilation stage. It has actually also been rewritten to process large handshake circuits more efficiently.

## 8.2 Simulation Trace

As seen previously, the simulation trace is a file used to convey the simulation results from the simulator to the visualisation system and to the external application GTKWave. As with most of the simulation traces, the quantity of traced data can get very large. Two techniques for reducing the amount of traced information are used in this system.

### Reducing the amount of traced information

The first idea corresponds to the pattern analysis described in §6.4.3. The amount of storage required is reduced by exploiting the repetitions of groups of traced events during the simulation.

The second idea is to reduce the amount of information at its source, by offering the possibility to the user to select the required traced channels. Letting the user choose which channels are important works well for designs smaller than a few dozens of channels, or for small focused parts in large designs.

## 8.3 Visualisation Control Links

Control links are used by the visualisation system to launch, send commands to, query the state of and terminate other processes. Via these control links, the user interface of external processes can be integrated in one place: the visualisation system's user interface. The implementation of these links uses UNIX pipes, which are more adapted

than files for point-to-point communications between processes. Two external processes are linked to the visualisation system: the Breeze simulator and GTKWave.

### **Visualisation-Simulation Link**

The control interface of the simulator can be divided into two parts: simulation time control and debugging actions.

Current simulation time can be set, stepped forward/back in time, stepped to the next visible event or automatically increased at adjustable speed.

Traced channels can be changed at any time during a simulation. Channel-based breakpoints can be set and removed.

### **Visualisation-GTKWave Link**

The role of GTKWave is to display the activity of a handshake circuit on a channel basis. Each visualised channel is displayed horizontally with the state of the channels visualised from left to right as shown in Figure 7.7, page 112. Such a method is very useful because it is well-known to designers. However, its disadvantage is the overwhelming amount of information likely to be displayed in the case of large designs.

The control link allows the designer to choose which channels are to be displayed in GTKWave via the visualisation system's interface. They can therefore select/unselect channels in any convenient view, such as the handshake circuit graph view or the Balsa source code view.

## **8.4 Summary**

The simulation, visualisation and debugging techniques developed in this thesis are integrated into a unified framework. The visualisation system is established as the main element of the framework, and other processes are controlled by the visualisation process via control pipes. All the functionalities of the various processes are gathered under the visualisation system's user interface.

## Chapter 9: Results and Discussion

The aim of this chapter is to present the results of this research work in the form of contributions, and to evaluate them on real-life examples as a proof of adequacy and a base for discussion. Each contribution is associated with an evaluation method, which intends to measure the validity of the contribution.

### **Simulation:**

#### **Four orders of magnitude Balsa simulation speedup**

The new handshake circuit simulator is compared to the previous Balsa simulator on a set of circuits of different sizes. It is also evaluated against Verilog simulators, which were previously the most efficient simulation route for design iterations with Balsa. The evaluation reflects the speedup observed by the designers of SPA [84] in the course of their work.

### **Debugging:**

#### **Ideas for debugging asynchronous-specific problems at handshake circuit level**

Ideas for debugging the asynchronous-specific problems of deadlocks and non-determinism at the handshake circuit level are validated on a case-study example.

#### **Easy pattern analysis of the out-of-order simulation trace**

The same case-study example is used to illustrate the readiness of the out-of-order simulation trace for easy pattern analysis and extraction, and shows its successful application to livelock debugging and simulation trace compression.

**Visualisation:****Following the execution of a circuit for program comprehension**

Program comprehension is evaluated on SPA, the largest circuit designed with Balsa. The evaluation is done at the three stages studied in this thesis:

- Merging complementary sources of information together to generate a graph viewable at any level of detail;
- Colour-based graph animation to highlight handshake circuit control flows;
- Coordinating this graph view and the various views “well-known to the designer” together for efficient element tracking and easy navigation.

For each stage, specific examples are illustrating the functional correctness of the visualisation system. Designers feedback is reported when available.

**Simulation and visualisation for debugging large scale asynchronous handshake circuits:****First debugging environment for large scale asynchronous circuits**

The unification of the tools developed after the above mentioned contributions led to the first debugging environment for large scale asynchronous circuits.

## **9.1 Simulation: Boosted Compilation and Simulation Speeds**

The most important and useful result coming out of this research is the formidable speedup of the compilation and simulation processes. This was originally the main aim of this work in order to help with the parallel development of SPA, and it led to achievements beyond expectations.

Balsa was always aimed at synthesising large asynchronous circuits. Unfortunately the route to simulation (i.e. the compilation of the Balsa description into the structure being simulated) for large designs was originally long and the simulator very slow. The results presented here show the speed up obtained for the compilation and simulation of different sizes of circuits, but it must be kept in mind that the largest of these circuits were primarily targeted. In short, a typical large Balsa design whose compilation and simulation were

previously taking respectively 2 hours 18 minutes and 12 days 22 hours 15 minutes is now compiled in 3.8 seconds and simulated in 56.9 seconds.

This section shows the evolution of the compilation and simulation speeds on a few selected examples. The Balsa descriptions of some of these examples and their sizes in terms of handshake channels and components are given in Appendix A. SPA being an ARM-compatible processor core, the ARM architecture validation suite comprising 76 programs was used during the development of SPA. After simulation of the complete validation suite, one of these validation programs called “undefs\_v5” was observed to represent the average case, based on simulation time. It is used here as a representative of the whole test suite. Another short ARM program, called “hello world” (not part of the validation suite) is used to represent short simulations of a large design.

One difficulty was to find two versions of the Balsa toolkit able to compile and simulate identical circuit descriptions, regardless of the changes which have happened to the Balsa language in the last four years. It was even more of a challenge to find a usable description of SPA recent enough to reach an interesting size and a working condition, and still not using too recent Balsa features such as new language constructs or builtin types, in order to be compiled with the older version of Balsa. The CVS version of SPA from the 1<sup>st</sup> February 2002<sup>1</sup> satisfied these criteria, and was able to be compiled both by the Balsa toolkit from the 12<sup>th</sup> March 2002 and by the most recent toolkit from the 1<sup>st</sup> September 2004. From this recent framework, the three versions of the simulator described in §5.2.2, §5.2.3 and §5.2.4 are evaluated.

Two computers have been used for this evaluation. The first one is an AMD Athlon 900 MHz with 512 MB of RAM running Linux. The second one is an Ultra Sparc II, 500 MHz with 2 GB of RAM running Solaris.

### Compilation

The compilation speedup obtained here is not due to any work or research, but only to the idea of simulating Balsa directly at the handshake level. The original simulation route required a 3-stage compilation: Balsa was first compiled into a handshake circuit, which

---

1. SPA 1<sup>st</sup> February 2002 patched with multiply.balsa 1<sup>st</sup> June 2002

was in turn transformed into a behavioural language (LARD [34]) description, finally compiled into bytecode ready to be simulated. The new simulation system takes the handshake circuit directly as its input, thus needing only the first stage of compilation from the old route. The only new requirement concerns descriptions distributed in multiple files, for which linking all the handshake sub-circuits together is required either at the end of the compilation or at the beginning of the simulation. It is included here in the compilation timings.

Tables 9.1 shows how the large circuit of SPA gets compiled two thousand times faster, while smaller circuits only get a speedup of about four. This is explained by an initialisation process taking between half a second and a second in the new test-harness generator, which consumes a significant proportion of the total time in small examples but is insignificant in large ones.

	Old compiler <sup>a</sup>	New compiler <sup>b</sup>	Compilation speedup
1-place buffer	3.4s	0.85s	4
Corridor	4.1s	0.9s	4.5
SPA	2h 18min	3.8s	<b>2179</b>
SPA on Solaris	4h 20min	22s	709

Table 9.1: Evolution of the compilation speed

a. Balsa & LARD compilers 12<sup>th</sup> March 2002

b. Balsa compiler 10<sup>th</sup> June 2004

In addition to these timing results, the memory used by compilers peaks at 200 MBytes with the LARD compiler versus 25 MBytes with the Balsa compiler. This is however an unimportant issue, as most computers already had more than 200 MBytes of memory in the early days of Balsa.

Another aspect worth considering is that the figures reported here correspond to the compilation of the entire circuit, which rarely happens: In the case of SPA, most of the time during the design process, only one file amongst the 23 source files is modified and only this file and those depending on it need to be recompiled. Still, the improved compilation speed has been welcomed as it reduced the compilation of a single file from

six minutes in average to less than a second. This was very useful as it immediately permitted iterative design experiments by trial and error.

The last row of the table shows that comparable results are obtained when compiling SPA on a Sparc Solaris machine. The smaller speedup is explained by slower hard drive accesses on this machine: Many large files are read and written during the compilation, leading to a negligible proportion of slow hard drive accesses compared to 4h20min, but important compared to 22s.

### Simulation

Simulation results (Table 9.2) follow the same scheme as those reported for compilation, with even better speedups: Small circuits are now simulated one to two orders of magnitude faster, while large ones get accelerated by more than four orders of magnitude.

	LARD simulator <sup>a</sup>	Breeze simulator <sup>b</sup> ( <i>speedup</i> )	Out-of- order simulator <sup>c</sup> ( <i>speedup</i> )	Reordered simulator <sup>c</sup> ( <i>speedup</i> )
1-place buffer with test-harness, 10000 data	19s	1.15s (17)	0.445s (43)	0.450s (42)
1-place buffer without I/O, 10000 loops	1min 23s	0.55s (151)	0.094s (883)	0.094s (883)
SPA running Hello World	1h 21min 12s	1.06s (4596)	0.64s (7612)	0.62s (7858)
SPA running Big Hello World (modified hw with 253 characters)	14h	7.5s (6720)	2.95s (17085)	2.74s (18394)
SPA running undefs_v5	12days 22h 15min	2min 44s (6810)	55.0s (20307)	56.9s ( <b>19629</b> )
SPA undefs_v5 on Solaris	91days 10h 40min <sup>d</sup>	16min 30s (7981)	6min 0s (21947)	6min 15s (21069)

Table 9.2: Evolution of the simulation speed

a. LARD simulator 12<sup>th</sup> March 2002

b. Breeze standard event-driven simulator 1<sup>st</sup> September 2004

c. Breeze out-of-order and reordered out-of-order simulators 1<sup>st</sup> September 2004

d. Estimated total time based on a two days execution



This is explained by the fact that LARD, the language used in the previous simulation system, was based on a time-driven scheduler whose CPU consumption was proportional to the total number of handshake components in the design (at each timestep the scheduler was checking for any change in the inputs of most components). The simulation speed was therefore reasonable for small designs but was reduced dramatically for medium to large designs.

The most remarkable figure in this table is the speedup obtained by the reordered out-of-order simulator when simulating the SPA microprocessor with the typical-sized ARM validation test “undef\_v5”. The speedup of more than four orders of magnitude is the one which has been observed in general during the development of SPA.

This significant improvement had a direct influence on the development style used with Balsa. Associated with the faster compilation, multiple design iterations were then possible in a short period of time.

This speedup also changed the simulation route used with Balsa: Prior to this, it was more advantageous to synthesise Balsa circuits to Verilog netlists and simulate these. Now, the direct simulation at the handshake circuit level is one to two orders of magnitude faster than the Verilog simulation route, making it worth using (even without considering the better debugging capabilities available at the handshake circuit level, which cannot be provided at the Verilog level). Table 9.3 reports and compares timings when simulating the SPA undefs\_v5 test with the following simulators: the old LARD simulator (to show that it was not worth being used), the new Breeze handshake circuit simulator and the two most popular non-commercial Verilog simulators: Icarus Verilog which compiles the Verilog description, and Cver which reads the Verilog description at the beginning of the simulation process for interpretation. Most popular commercial Verilog simulators are licensed with prohibition against benchmarking, and for that matter prohibit disclosure of any information. They are therefore not included in these results.

	compilation	linking & initialisation	simulation	Total
Old LARD simulator (reminder)	2h 18min	6min 40s	12days 22h 8min	13days 32min...
Breeze simulator <sup>a</sup>	3.8s	1.7s	55.2s	1min 0.9s
Icarus Verilog Simulator <sup>b</sup>	3min 9s	3min 30s	12min 25s	19min 4s
Cver Verilog simulator <sup>c</sup>	3min 9s	1min 20s	3h 2min 17s	3h 6min 46s
Speed ratio Breeze/Icarus	50	124	13	<b>19</b>
Speed ratio Breeze/Cver	50	47	198	<b>184</b>

Table 9.3: Comparison of Breeze and Verilog simulators <sup>d</sup>

- a. Breeze handshake circuit simulator, with reordered out-of-order scheduler  
b. Icarus Verilog Simulator (compilation = Balsa->netlist + netlist->internal format)  
c. Cver Verilog simulator (compilation = Balsa->netlist + netlist->internal format)  
d. Simulations of SPA running the undefs\_v5 ARM validation test

### Design iteration

Table 9.4 reports some delays more representative of a real life situation during the design of SPA using design iterations. It is considered here that only 10% of the files need recompiling, and that the simulation will stop (usually as it detected a bug) at 1% of its total execution time.

	compilation	linking/initialisation	simulation	Total
Old LARD simulator (reminder)	13min 48s	6min 40s	1day 7h 0min 48s	1day 7h 21min...
Breeze simulator	0.4s	1.7s	0.55s	2.65s
Icarus Verilog Simulator	18.9s	3min 30s	7.5s	3min 56.4s
Cver Verilog simulator	18.9s	1min 20s	1min 49s	3min 27.9s
Speed ratio Breeze/Icarus				<b>89</b>
Speed ratio Breeze/Cver				<b>78</b>

Table 9.4: Design iteration speedup

## 9.2 Debugging Demonstrator: The Simple Corridor Problem

In order to demonstrate the resolution of deadlocks, livelocks and non-determinism, and prove the ability of these techniques to link the problems back to their source code, an interesting and amusing problem has been implemented in Balsa: the behaviour of two people meeting face to face in a narrow (two lanes) corridor and wanting to go pass each other.

Depending on the behaviour of each person, these three problematic cases of asynchronous systems are illustrated:

- Livelock: Each moves aside to let the other pass, but they end up swaying from side to side without making any progress because they always move the same way at the same time.
- Deadlock: Each waits for the other to move aside.
- Non-determinism: During precise simulation of the livelock case, if no special synchronisation is made, the two processes will not stay synchronised infinitely because of delay changes due to external factors (temperature, etc.). They will eventually get out of the livelock.

In the following, the two individual behaviours are called “lazy\_guy” and “polite\_guy”. The lazy\_guy always stops when somebody arrives in front of him and waits until he leaves. The polite\_guy always moves: If somebody arrives in front of him, he tries to let him pass by “changing lane”. The two-lanes corridor is modelled as four positions as shown in figure 9.1.

The full source code for this example is available in Appendix A.3. The interesting part of the implementation of lazy\_guy are described here, to serve as a reference when used in this section:

```
1 procedure lazy_guy (parameter name:String; output my_pos : bit;  
                     sync waitfor_pos_in_front_empty) is  
2 begin  
3   my_pos <- 1;
```

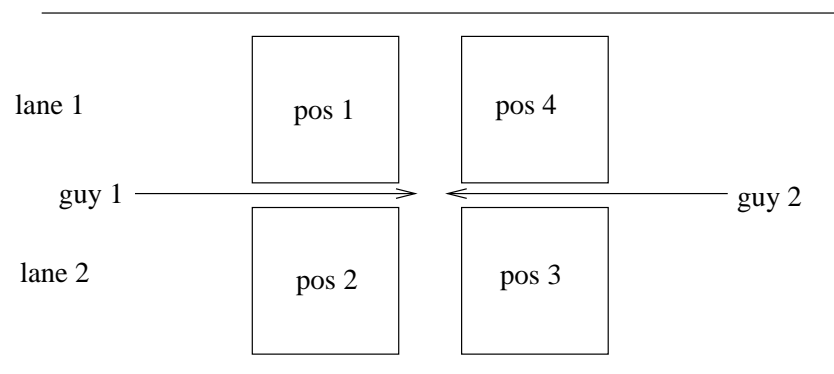


Figure 9.1: Implementation model for the corridor problem

```

4  print "lazy guy ", name, ": Somebody in front of me... I'll wait that
he moves aside...";
5  sync waitfor_pos_in_front_empty;
6  print "ok I can go";
7  my_pos <- 0
8  end

```

### 9.2.1 Deadlock Handling

Let's put two lazy\_guys in the same lane and run the simulation of the circuit:

```

lazy guy A: Somebody in front of me... I'll wait that he moves aside...
lazy guy B: Somebody in front of me... I'll wait that he moves aside...
Deadlock.

```

It is obvious what is happening: Both processes are waiting for the other process to free its position. The interesting part is how the deadlock is detected, signalled and explained by the simulation and visualisation system: Would this help to solve real-life situations where the user does not have a clue why the circuit deadlocked?

The deadlock is detected by the absence of events to be processed by the simulator. Figure 9.2 shows the last state of the simulation: Pink channels represent the (uninteresting) activation tree while red channels show the (interesting) last events which happened in the visualised circuit. These blocked channels are therefore causing the deadlock. The interface graph-source code viewer reveals that they both point to the same source code statement: line 5 of the source code, the “sync waitfor\_pos\_in\_front\_empty” statement.

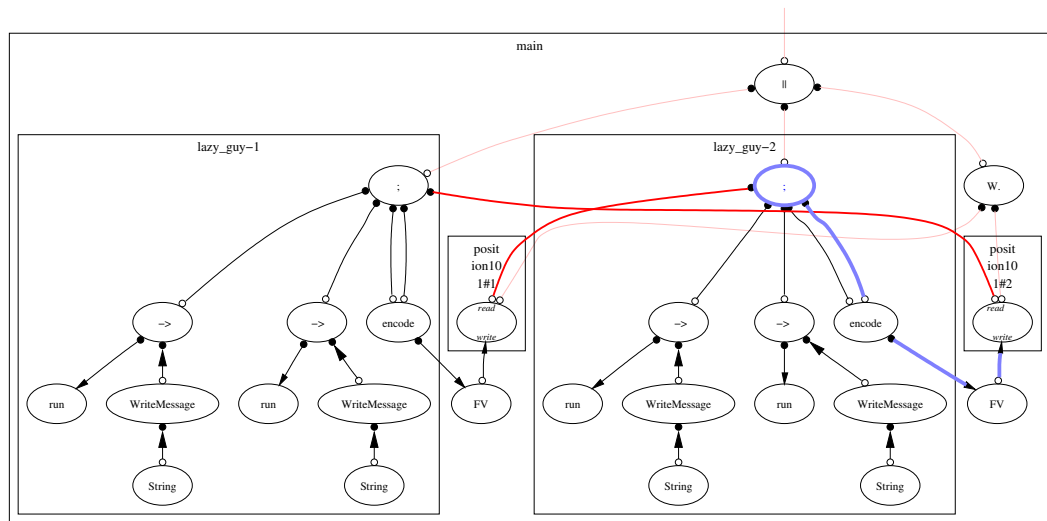


Figure 9.2: Deadlock with two lazy\_guys in the corridor example

The algorithm described in §4.2.2 has been implemented for finding the causes of only one deadlocked channel. However, this does not reduce the effectiveness of the process, which can be repeated for as many deadlocked channels as necessary, the union of the results being the equal to the result which would have been obtained by analysing the whole set of deadlocked channels at once. By applying this algorithm to the left-most blocked channel, the string of blue blocking channels is highlighted. This shows that an event to the *write* port of the *position* module would solve the deadlock. This further shows that this event should have been generated by the blue Sequence ‘;’ component, itself blocked by the second red blocked channel. Unsurprisingly, the first blue channel required to solve the deadlock points to the source code position at line 7: “my\_pos <- 0”.

The analysis of the second red blocked channel with the same algorithm results in a symmetric result, proving that the two red channels are deadlocked together, and that each of them, executing code at line 5, prevents the other one from *writing* the required value onto the *position* component – at line 7.

It has been illustrated that the algorithm suggested in this thesis can help discover important channels for the understanding of deadlocks.

### 9.2.2 Livelock Handling

Livelocks are not usually targeted in debugging environments because they do not happen frequently and are not hard to debug by hand. This research did not focus on debugging these problems. However, the following discussion is useful for introducing the next section about non-determinism handling.

Let's put two polite\_guys in the same lane and run the simulation of the circuit:

```
polite guy B: Somebody in front of me! Let's move aside to let him pass
polite guy A: Somebody in front of me! Let's move aside to let him pass
polite guy A: Still somebody in front of me! let's move back then
polite guy B: Still somebody in front of me! let's move back then
polite guy B: Somebody in front of me! Let's move aside to let him pass
polite guy A: Somebody in front of me! Let's move aside to let him pass
... <Livelock>
```

We notice an infinite repetition of the same output messages, suggesting a livelock. The simulator is unfortunately not detecting livelocks automatically: It would be too expensive to check for state repetitions continuously during the simulation. On the other hand, livelock detection can be made by the analysis/visualisation system: In the same way as humans detect livelocks by observing repetitions in the outputs of the circuit, the pattern analysis tool is able to detect repetitions of patterns in the handshake circuit's activity trace. Figure 9.3 shows the high-level behaviour view obtained with this circuit's trace. A pattern repeated three times can be clearly seen. The analysis tool also detects this pattern and indicates that a pattern of length 60100 is repeated from time 14801 to the end of the trace - "Pattern Start = 18201, Length = 60100, 17 repetitions; Split pattern can be followed back until time 14801/14901". The start and end points of the first occurrence

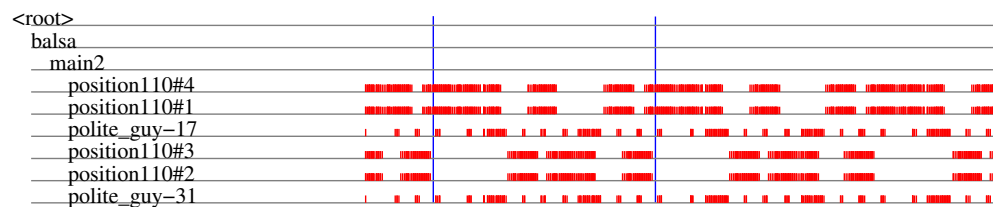


Figure 9.3: Livelock with two polite\_guys in the corridor example

of the pattern are represented by two blue lines on the figure. The source code view shows that this pattern corresponds exactly to the bodies of the two `polite_guys`' main loops.

The limitation of this example is that except for the initialisation tree and the few initialisation assignments, all the channels in the circuit are detected as being part of the livelock, which does not help at all for the debugging. However, on larger designs, this method is able to detect sub-circuits containing the involved channels. Manual analysis can then be performed on these smaller domains.

### 9.2.3 Non-Determinism Handling

The previous livelock example was based on the assumption that both `polite_guys` take exactly the same amount of time to perform their identical actions. However, this assumption may not be true when the circuit is implemented in hardware: Process and temperature variations, at the very least, can make delays vary, and parts of the circuit synthesised from the same high-level description eventually show different delays.

Without any delay variations, the simulation of the previous circuit was deterministically running forever in a simple livelock: The possible sources of non-determinism, the Arbiter components, were always called in turn at distinct timestamps. When delay variations are taken into account, a more complex behaviour emerges: In the same way as a difference of speed of the human polite guys in a real corridor would eventually get them out of synchronisation and solve their problem, different delays in the handshake circuit, modelled here by delays with errors, show that a possible resolution of the livelock could eventually happen.

With the handshake component delays defined as before and with an error set to 1% of each delay, the simulation shows that in the best-case situation the livelock situation would be lost during the sixth loop of the repeating pattern: This happens when every delay happening on the first `polite_guy`'s path is 1% slower than its definition, while every delay happening on the second `polite_guy`'s path is 1% faster than its definition. The non-deterministic situation corresponds to when the Arbiter defined in the *Position* component has a choice between which of these two operations is to be executed first: either write the position of the `polite_guy` in front to its new position or read if the position in front of the

current polite\_guy is occupied or not. The former will terminate the livelock, while the latter will let it continue until the next non-deterministic write/read choice.

The interesting thing to notice in this example is that the non-deterministic situation is desirable to get out of the livelock problem, whereas non-determinism is usually an unexpected behaviour leading to bugs. In all cases its early detection benefits the designer.

### 9.2.4 Further Pattern Analysis and Trace Compression

This section intends to show that the out-of-order simulation is particularly suited for subsequent pattern analysis of the simulated events.

The very small one-place buffer example already used in §9.1 is used here again because it is small enough to be able to reproduce a meaningful part of its simulation trace. The source code of this example is available in Appendix A.2 and the corresponding handshake circuit is reproduced in Figure 9.4. The buffer circuit with environment is used here. It contains three threads: the buffer itself, an input thread and an output thread.

The out-of-order simulation of this circuit generates the trace file available in Appendix A.4. In this trace, the “XXX” keyword ending each line indicates that the out-of-order scheduler had to obtain the next event from the event queue instead of having it directly indicated by the previously simulated component. What is called "Easy pattern analysis/extraction" in this thesis is the ability to use these threads of events directly as patterns and observe their repetitions.

In the present example, the following patterns are directly extracted by this method:

```
Pattern 1: 2-requp 31-requp 32-requp 3-requp 23-requp 24-requp 25-
          requp 26-requp 27-requp 26-ackup 25-ackup 27-ackup 27-
          reqdown 25-reqdown 24-ackup 24-1 26-reqdown 26-ackdown 25-
          ackdown 27-ackdown 27-requp 25-requp 24-ackdown 24-requp
          26-requp 1-requp
Pattern 2: 17-reqdown 16-ackup 16-reqdown 18-reqdown 18-ackdown 17-
          ackdown 17-requp 16-ackdown 16-requp 18-requp 19-requp 21-
          ackup 20-ackup 20-reqdown 21-reqdown 21-ackdown 20-ackdown
          20-requp 21-requp 22-requp
```



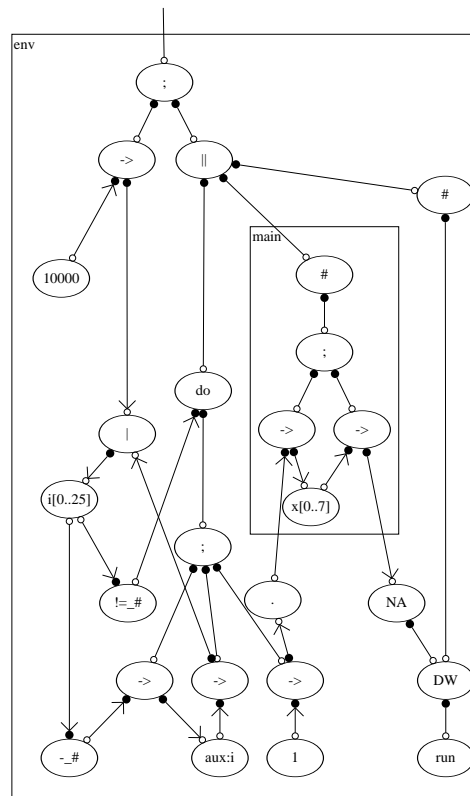


Figure 9.4: Handshake circuit of the one-place buffer example with its environment

```

Pattern 3: 17-requp 16-ackdown 16-requp 18-requp 19-requp 21-ackup
20-ackup 20-reqdown 21-reqdown 21-ackdown 20-ackdown 20-
requp 21-requp 19-ackup 19-reqdown 19-ackdown 13-ackup 11-
ackup 27-ackup 27-reqdown 11-reqdown 15-ackup 15-1 13-
reqdown 13-ackdown 11-ackdown 27-ackdown 27-requp 11-requp
15-ackdown 15-requp 13-requp 12-ackup 14-ackup 14-1 10-
ackup 9-ackup 9-reqdown 10-reqdown 12-reqdown 12-ackdown
14-ackdown 14-requp 10-ackdown 9-ackdown 9-requp 10-requp
12-0 18-ackup 17-ackup 2-reqdown 31-reqdown 31-ackdown 30-
ackdown 30-requp 2-ackdown

Pattern 4: 6-requp 4-requp 28-ackdown 28-requp 29-requp 31-ackup 30-
ackup 30-reqdown 2-ackup

Pattern 5: 6-reqdown 4-reqdown 28-ackup 28-reqdown 29-reqdown 29-
ackdown 4-ackdown 6-ackdown 5-ackdown 5-requp 7-requp 8-
requp

Pattern 6: 17-reqdown 16-ackup 16-reqdown 18-reqdown 18-ackdown 17-
ackdown 2-requp 31-requp 32-requp 32-ackup 32-reqdown 32-2
29-ackup 4-ackup 6-ackup 5-ackup 5-reqdown 7-reqdown 7-
ackdown

```

Pattern 7: 6-reqdown 4-reqdown 28-ackup 28-reqdown 29-reqdown 29-ackdown 4-ackdown 6-ackdown 5-ackdown 5-requp 7-requp 7-ackup

The following trace of patterns is obtained:

```
1@1 2@32 3@62 4@70 5@32 6@40 7@62 3@70 4@47 7@55 6@77 3@85 4@62 7@70
6@92 3@100 4@77 7@85 6@107 3@115 ...
```

where "x@y" means pattern x started at time y.

The livelock is visible in the pattern trace as the succession of threads 7-6-3-4, with a time period of 15. Something interesting to notice is that during the first iterations, the threads have not been simulated in the same order as in the rest: the succession of threads 6-7-3-4 appears at time 62.

### Trace compression results

The simulation of the livelock, which is allowed to run for 10 000 iterations, led to the generation of 7 patterns. The average length of a pattern is 24 events, and a manual analysis of these patterns reveals that the repeating block forming the livelock consists of exactly 4 patterns. These 4 successive patterns are repeated in the exact same order 9 999 times.

This manual analysis of a trace file representing almost a million events, reduced by such a simple pattern analysis process, demonstrates the potential for analysis offered by these extracted patterns.

Table 9.5 shows the sizes of the traces obtained by the different versions of the Breeze simulator. it should be noted that none of the files make use of size-optimised notations. The resulting compression rate is thus only a rough estimation intended to show the applicability of the idea.

	# of reported items	Trace size (in bytes)
Uncompressed standard ordered trace	960 036 events	29 684 798

Table 9.5: Trace compression results

	# of reported items	Trace size (in bytes)
Uncompressed out-of-order trace	960 036 events	40 902 323
Compressed trace, file 1: pattern descriptions	7 patterns	1 208
Compressed trace, file 2: pattern trace	40 002 patterns	400 020
Compression ratio		<b>1:75 ~ 1:100</b>

Table 9.5: Trace compression results

### 9.2.5 Discussion

The ideas and algorithms suggested in this thesis to debug deadlocks and non-deterministic situations in a handshake circuit have been implemented and applied on a small example. This small-scale application proved the ability of these techniques to link the problems back to the source code. Although this experiment was applied here on a small example, this does not mean that the illustrated results are only working on small scale problems. Deadlock debugging has been successfully used with the large SPA example by the author for detecting bad implementations of handshake components. However, the amount of information and the number of steps become too large to be clearly represented here.

Similarly, the adequacy of the out-of-order simulation trace for detecting patterns has been exposed on a very small example, but is scalable and its direct application, the simulation trace compression, is able to process large simulation traces such as SPA's.

## 9.3 Visualisation Demonstrator: The Huge SPA Microprocessor Core

SPA is an asynchronous ARM-compatible microprocessor core entirely described in Balsa. As stated previously, it is the largest circuit synthesised with Balsa so far, and the SPA description is compiled into a handshake circuit comprising around ten thousand elements. This figure can be related to the field of graph manipulations, where graphs over a thousand nodes are considered huge.

This section illustrates the contributions concerning the visualisation of large handshake circuits. As said earlier, this visualisation system is oriented towards program

comprehension, by aiming at helping the user “follow the execution of the circuit”. For this purpose, a strategy featuring three stages has been studied:

- Merging complementary sources of information together to generate a graph viewable at any level of detail.
- Animating the graph to highlight the control and data flows present in the circuit.
- Coordinating this graph and the various views “well-known to the designer” together for efficient element tracking and easy navigation.

### 9.3.1 Merging Sources for Multiscale Graph Visualisation

The aim of this section is to show that the graph structure obtained by the merging algorithms developed by this research is suitable for multiscale visualisation. However, printing the structure requires the use of a layout algorithm, whose quality will impact on the result. Fortunately, an extremely simple layout algorithm is used here. Obtaining a good picture at different levels of detail in these conditions would therefore prove that the visualised structure was appropriately organised. This would attest to the quality of the merging algorithms.

The version of SPA visualised here is made of 8 118 handshake components and 13 268 handshake channels. Studies show that graphs of this size are difficult to layout [53]. Figure 9.5 shows the layout obtained after functional grouping (§7.1.1) and recursive layout (§7.4.1). The operation of functional grouping generated 118 nested groups, represented in the graph by rectangles, while handshake components and channels are respectively represented by ellipses and lines. This first picture is pleasant.

Technical features are available for the manipulation of the static graph: zooming and panning, moving components and filtering. Filtering can be observed in this figure where long arcs are more transparent than short ones. Multiscale visualisation of the same graph is illustrated in the series of Figures 9.5, 9.6, 9.7 and 9.8. Each level of zoom properly shows different characteristics of the circuit. Figure 9.5 shows the global view of the circuit, and its main components can be distinguished. Figure 9.6 shows the recursive implementations of two modules: shift and clz (count leading zeroes). Figure 9.7 shows a more detailed view of the decodeUnroll module and its submodules. And Figure 9.8

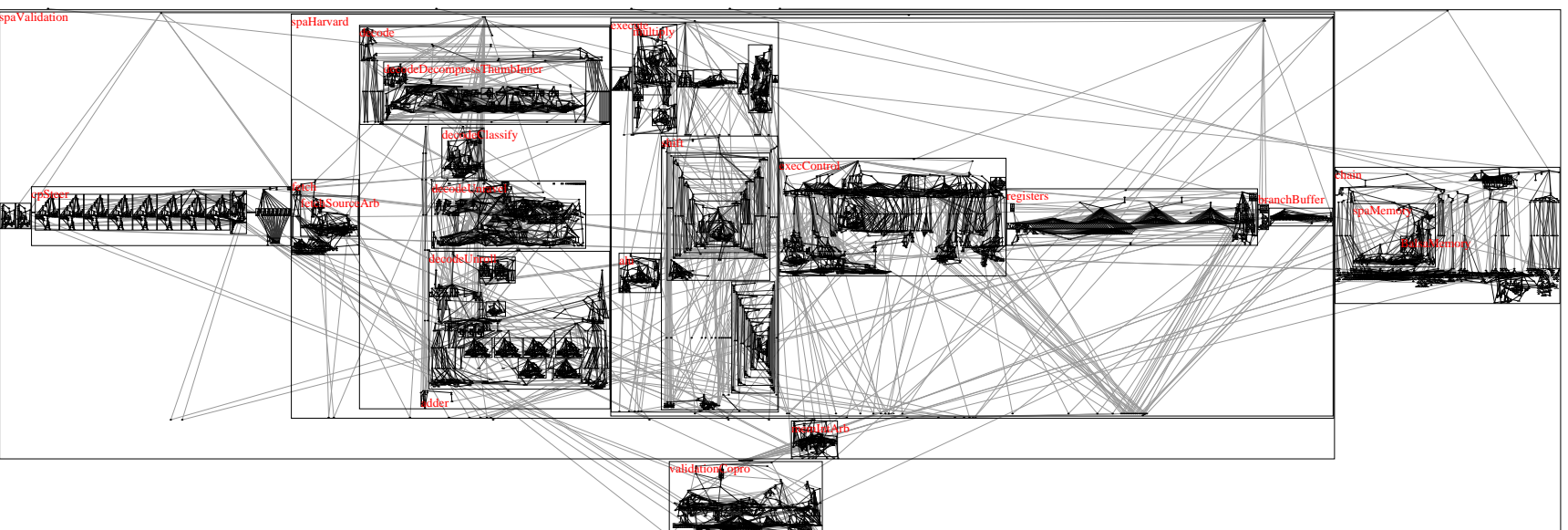


Figure 9.5: Huge graph layout: SPA - Zoom 100%

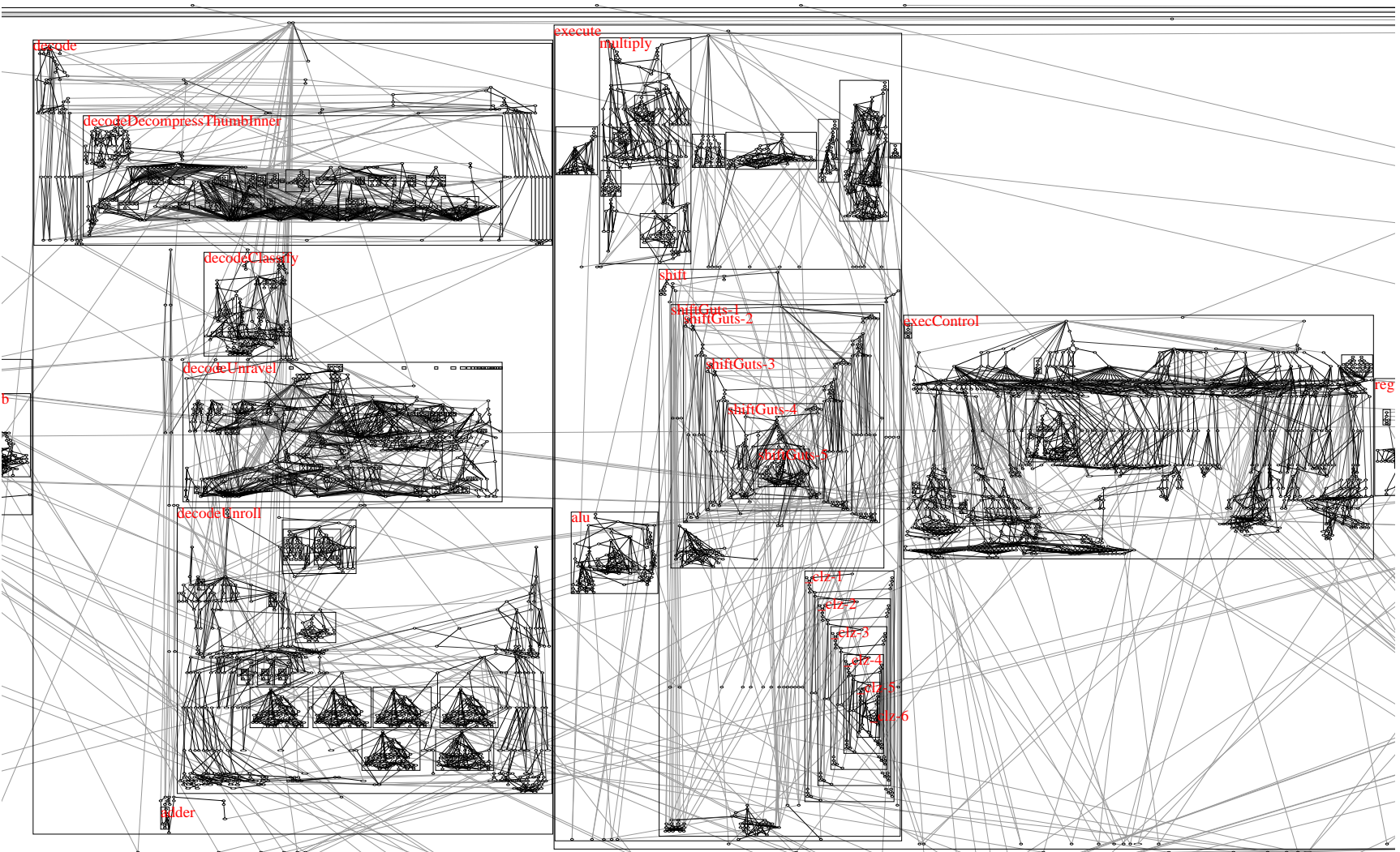


Figure 9.6: Huge graph layout: SPA - Zoom 250%



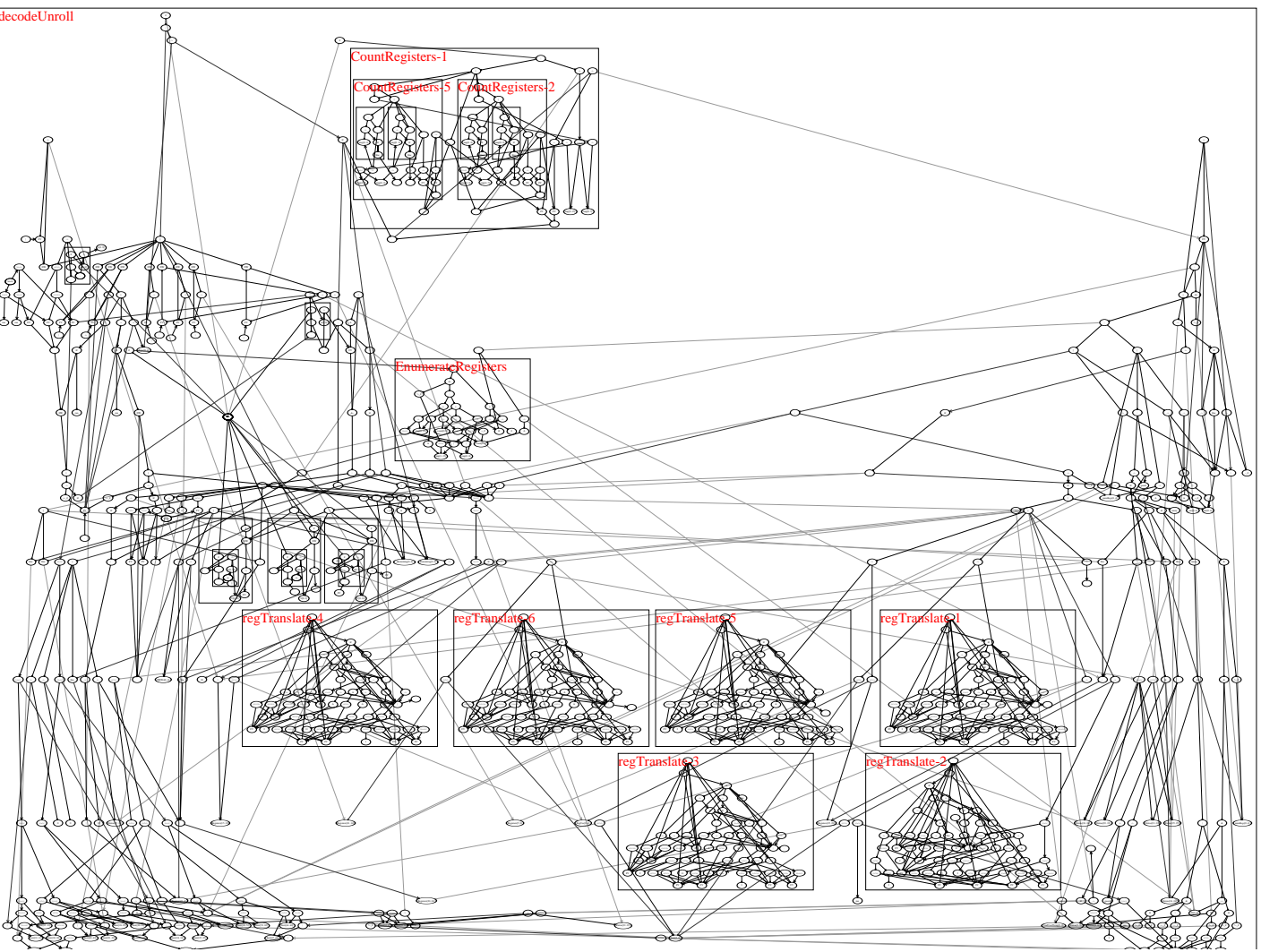


Figure 9.7: Huge graph layout: SPA - Zoom 900%

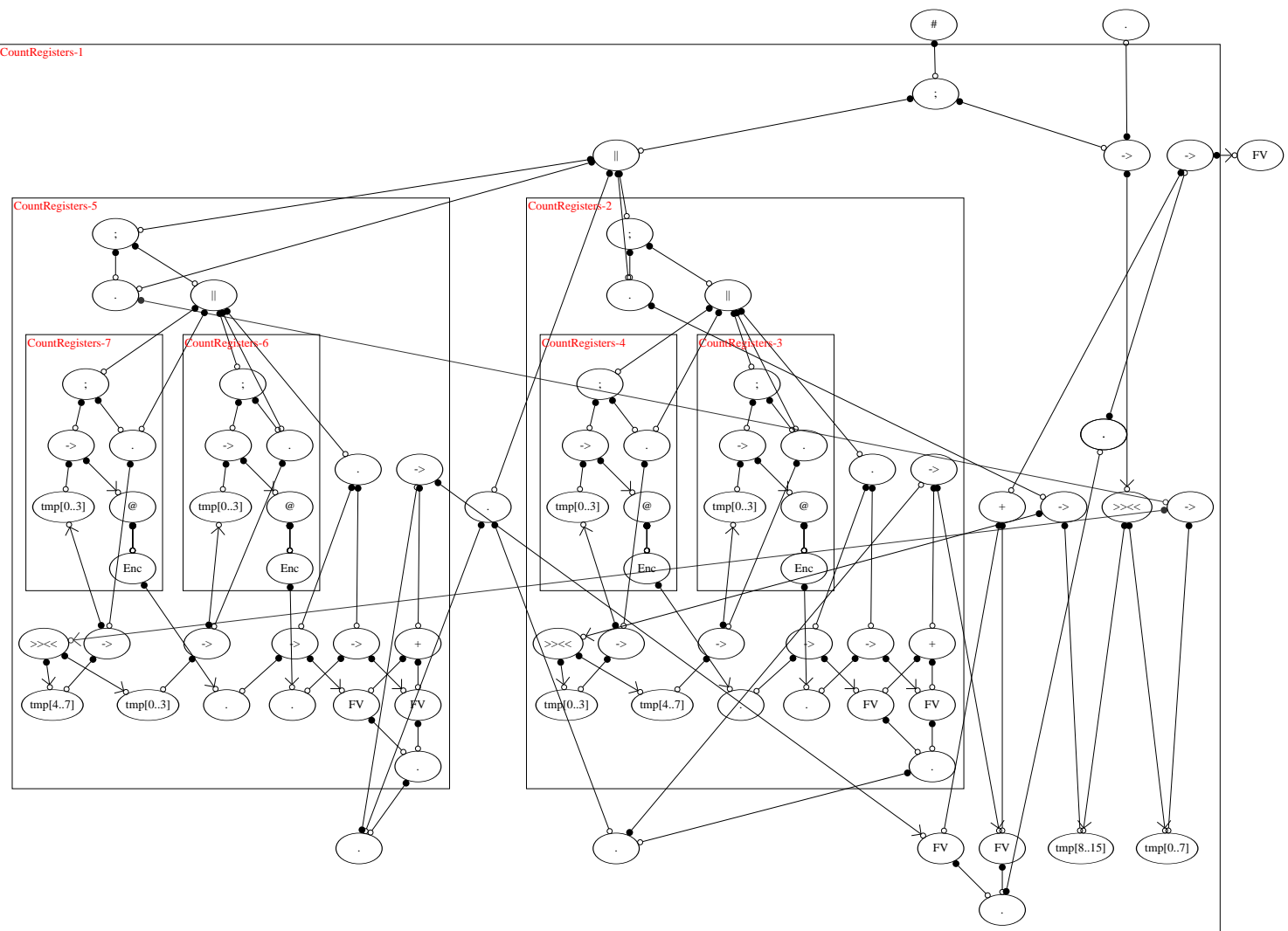


Figure 9.8: Huge graph layout: SPA - Zoom 4500%



shows the detailed implementation of a few modules, intelligible enough for an experienced reader to understand their behaviour.

The appropriate visualisation of characteristics of the circuit at different levels of detail attests to the quality of the graph structure.

### 9.3.2 Animated Graph

This stage is of most importance for helping the user follow the execution of the circuit. Control flows are animated by highlighting the corresponding graph edges according to the events reported in the simulation trace.

Figure 9.9 shows a step-by-step animation of the activity present in a very small part of the SPA processor: a 1-place buffer used to store a value in a buffer and transmit it to the next processing stage. The associations between handshake channels and pseudo-source code are represented, thus indicating the meaning of each coloured event. In this animation, a 2-phase protocol is used where requests are represented by a thick red highlighting scheme and acknowledgments by a thick blue one. Thin red channels represent channels which have been activated at a previous timestep and have not been acknowledged yet.

By following the same scheme, concurrent flows and their interactions can be easily observed. For example, Figure 9.10 shows the same circuit as above, but where the Sequence component has been replaced by a Fork component. It should be noted that the Balsa compiler would not generate such a circuit unless forced (by the user) to do so: the concurrent read and write accesses to the variable are non-deterministic.

For visualising activity in large scale handshake circuits, the animation must be ported to a higher level than handshake components. The method used is to visualise activity in groups of components (i.e. modules, like those represented by rectangles in Figures 9.5-9.8) by a coloured activity bar whose height is proportional to the number of events happening inside the group. Unfortunately, the results only give an imprecise idea that *some* activity is happening in the group. More research on this matter is planned as future work. A snapshot of the execution unit during a simulation of SPA can be seen in Figure



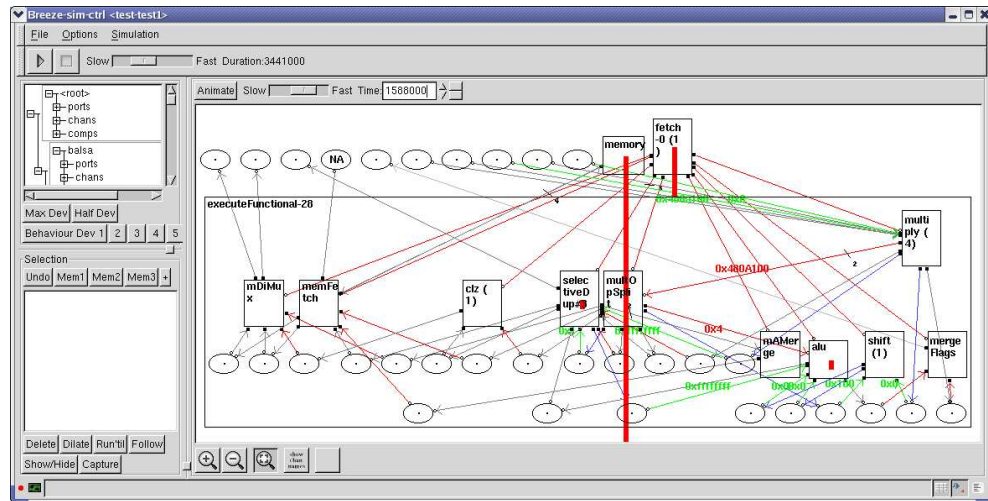


Figure 9.11: High level SPA animation snapshot with bars in groups

these views usually consist of a text editor containing the source code, and a waveform viewer to analyse the results of the simulation. In order to make these views even more useful, they are collaborating to track visualised elements from one view to another. This has been illustrated in Figure 7.7, page 112: An element (handshake component, channel, procedure, etc.) chosen in one view can be used to navigate to another view at the element's position.

This collaborating scheme had been long awaited for three particular applications:

### 1. Balsa source code ↔ Handshake circuit view

This link is useful to understand how a particular source code is translated into a handshake circuit, and the other way around to find out which source code element corresponds to a particular handshake channel (useful, for example, when a channel is highlighted by the simulation after a deadlock). Before collaborative views were available, the designer's knowledge of the handshake circuit compilation process guided this task. It was also possible to read the channel's number on the handshake circuit's graph, look it up in the Breeze file, where the source code position is written. However, this was tedious and only practical for small circuits (this was the process with the DMA controller of the AMULET3 processor [6]).

This situation has been successfully used by designers studying the Balsa compiler. They were able to understand which handshake channels and components resulted from the compilation of various Balsa constructs.

## **2. Handshake circuit view ↔ Waveform viewer**

Displaying the waveform associated to a specific channel was previously almost impossible. The channel to be viewed in the waveform viewer had to be practically guessed. In the same manner as above, the channel number could have been used as an intermediate for going from one view to the other. However, it was tedious to look for the correct channel in the middle of hundreds of generated channels. This was only acceptable for very small designs, and guessing the channels by looking at their waveform activity was easier than searching for specific channels. This task is now scalable to thousands of channels.

## **3. Balsa source code ↔ Handshake channels at the Verilog level.**

This association was requested by the designers of SPA needing to pursue their debugging at the Verilog level. This feature has been very useful and has received excellent feedback. Designers never went back to use their previous method where they were hand-drawing the handshake circuits and annotating the channel numbers to look for in the Verilog file.

### **9.3.4 Discussion**

This section has evaluated the ability of the visualisation system to handle very large designs and to represent the information in a comprehensible way. The large Balsa description, handshake circuit and simulation trace have been merged to generate a graph structure viewable at different levels, while still showing useful information. The colour-based animation, although imprecise at a high level, is able to display simulation events either at a high level or at a very fine level, showing precisely the concurrency in the circuit. Finally, the coordinated/collaborative views scheme has been used by the original designers of SPA for a month and received excellent feedback.

## 9.4 Unified Debugging Environment

The contributions evaluated above are brought together into a unified environment for debugging large asynchronous handshake circuits by using extensive simulation and large scale visualisation methods.

This is the first integrated environment for debugging large-scale asynchronous circuits. Previously, the only existing environments were either to debug small-scale asynchronous circuits or large-scale synchronous ones. An integrated environment around Balsa allows the first stages of the large asynchronous circuit design not to require the use of synchronous tools. Behavioural simulation and debugging can be done at the handshake circuit level, leading to better error reports, easier debugging and more efficient design space exploration.

### **Educational Tool**

Educational qualities have been observed, mainly due to the visualisation of handshake circuits, oriented towards program comprehension.

Depending on the level of capabilities of the student, the toolkit developed here can help teach and learn different aspects of asynchronous circuits:

- Asynchronous circuit bases can be introduced easily to students by visualising step by step simulations of simple circuits (as in Figure 9.9).
- Newcomers to Balsa can get a good understanding of the syntax-directed compilation process, which is useful to start describing asynchronous circuits with Balsa, by visualising the handshake circuits corresponding to different Balsa constructs and using the interface to see associations between handshake channels and source code. In the same way, more experienced Balsa designers can perfect their understanding of the compilation and optimisation tasks, which is a requirement to describe efficient circuits.
- The scalable visualisation system can be used for presenting the architecture of an asynchronous system in front of groups of people.

# Chapter 10: Conclusions

## 10.1 Summary

This thesis is concerned with techniques to support the debugging of large asynchronous handshake circuits by using extensive simulation and large-scale visualisation methods. These methods are applied to the Balsa asynchronous circuit synthesis framework.

A set of optimisation techniques applicable to the simulation of handshake circuits leads to a simulator four orders of magnitude faster than the previous Balsa simulator on large circuits. This makes programming by design iteration possible and reduces considerably the time necessary for validating designs at the behavioural level by using extensive simulation.

A visualisation system oriented towards program comprehension is presented. It is able to merge and represent in a single view different sources of information related to handshake circuits: the original Balsa source code, the compiled static handshake circuit and the dynamic simulation trace. This results in a very useful graph structure, viewable at any level of detail and showing the evolution of the control flows present in the circuit during the simulation. The visualisation system is also based on a structure of coordinated/collaborative views. In order to provide a familiar environment to the designer, the usual *source code*, *wave form* and *post-synthesis Verilog* views are integrated in the environment. Their usefulness is improved by a collaboration scheme allowing the user to track elements from one view to another. The navigation between the various sources of information is considerably enhanced. This enables, in particular, an efficient tracking of the control flows from the simulation trace to the source code or to the post-synthesis Verilog structure, leading to an easy and precise comprehension of the handshake circuit's structure.

Between the simulation and visualisation processes, a thin layer of analysis is presented. Based on both the static and the dynamic structures of the handshake circuit, some ideas for debugging the asynchronous-specific problems of deadlocks, livelocks and non-determinism are suggested. The problems are correctly analysed and their causes are identified, giving the user the opportunity to relate the misbehaviours to their precise locations in the original source code.

During the analysis of the simulation trace, it is discovered that the out-of-order structure of the trace makes it an excellent candidate for pattern analysis. From this observation, a simple application of trace compression is presented.

The unification of the above-mentioned techniques leads to the first debugging environment for large scale asynchronous circuits. This framework is based on handshake circuits and is integrated into the existing Balsa framework.

## 10.2 Summary of Contributions

### **Simulation:**

- Four orders of magnitude Balsa simulation speedup

### **Debugging:**

- Ideas for debugging asynchronous-specific problems at handshake circuit level
- Easy pattern analysis of the out-of-order simulation trace

### **Visualisation:**

- Following the execution of a circuit for program comprehension
  - Merging complementary sources of information together to generate a graph viewable at any level of detail
  - Colour-based graph animation to highlight handshake circuit control flows
  - Coordinating this graph view and the various views “well-known to the designer” together for an efficient element tracking and an easy navigation

### **Simulation and visualisation for debugging large scale asynchronous handshake circuits:**

- First published debugging environment for large scale asynchronous circuits

## 10.3 Limitations

The major limitations of the final framework concern areas in which the research work undertaken does not address a problem. These cases are clearly identified in this thesis.

First, timing and power analyses at the handshake circuit level are quasi worthless as they are now. The reason is that the delay and consumption values assigned to each handshake component have been chosen arbitrarily. In order to be useful, these delays must be determined by a precise method, possibly based on measurements made at a lower level of simulation.

Then, the graph layout process is quite slow. This is emphasised by the necessity to recompute the layout of the visualised graph every time a group is developed or shrunk.

Finally, although the concurrent activity is well rendered in the handshake circuit graph view, no particular method has been found for representing efficiently the same information in the source code view. This therefore limits source code debugging to a single thread at a time. Moreover, even a single thread is not displayed very efficiently in the source code view.

## 10.4 Suggestions for Future Work

Some research work arising from the research presented in this thesis has already been started:

- *Distributed simulation*: A distributed version of the handshake circuit simulator is currently being undertaken by Theodoropoulos and Tsirogiannis at the University of Birmingham [31].

Future work could address the limitations mentioned in the previous section:

- *More precise timing and power analyses*: Delay and power consumption values of each handshake component should be better estimated. A possible solution is to simulate them at a lower level of simulation and extract the required values, as has been suggested in §6.1.1.



- *Better and faster layout*: Force-directed placement [28] has the advantage of a compact representation. However, it usually is a slow method not representing identical subgraphs in an identical manner. Other layout methods could be investigated [38]. They could be coupled with the “tracking structural changes” technique suggested below.
- *Visualisation of multiple threads on top of the source code*.

Extensions to the actual framework are also possible:

- *Tracking structural changes*: Tracking structural changes between two handshake circuits could help visualise the effect of changes in a Balsa description during design iterations. This idea has been described in §7.4.2. This could also be used to obtain a faster layout, by having to place only the new components after changes to a Balsa description.
- *Co-simulation*: A basic interface is already in place for the (deprecated) co-simulation of Balsa with the LARD asynchronous behavioural language. Co-simulation of the handshake circuit with a simulation at a lower, more precise level could be used for dynamic correction of the handshake components’ delay and power consumption estimations.
- *Simulation trace patterns*: The patterns obtained from the out-of-order simulation trace offer new opportunities for organising data for debugging and visualisation applications.
- *Checkpointing*: The state of a simulation process could be saved at regular intervals. This would allow the simulation to run without generating a simulation trace, and later generate the simulation trace on demand for any specified time interval for debugging purposes.
- *Search features*: A collection of search methods is necessary for efficient navigation among large amounts of information, as demonstrated by some program comprehension studies [13].
- *Asynchronous design teaching tool*: The educational qualities of the current framework, although undeniable (see §9.4), have not been evaluated per se. A complete work of evaluation on real classes could be considered.

# Appendix A: Balsa Example Circuits with Statistics

## A.1 Sizes

	# channels	# components
1-place buffer + test-harness, 10000 data	$8 + 37 = 45$	$5 + 26 = 31$
1-place buffer without I/O + environment, 10000 loops	$8 + 24 = 32$	$5 + 20 = 25$
Corridor	276	192
SPA 1 <sup>st</sup> february 2002	7453	4653
SPA 2004	13268	8118

Table A.1: Size of Balsa circuits examples

## A.2 Source Code for the 1-Place Buffer Example

```
import [balsa.types.basic]

-- main procedure used with I/O test-harness
procedure main (input inp:byte; output out:byte) is
  variable x : byte
begin
  loop
    inp -> x;
    out <- x
  end
end

-- environment used in the "without I/O" case
procedure env is
  variable i : 26 bits
  channel x,y : byte
begin
  i := 10000;
```

---

```

    loop while i /= 0 then
        x <- 1;
        i := ((i-1) as 26 bits)
    end ||
    loop
        select y then continue end
    end ||
    main (x,y)
end

```

## A.3 Source Code for the Corridor Example

```

import [balsa.types.basic]

procedure delay_move is
    variable x:byte
begin
    x := 0
end

type opl is enumeration OP_read, OP_set_wfz end

procedure position (input write:bit; output read:bit;
                    input op:opl; sync wait_for_zero) is
    variable value : bit
    variable flagb, flagb2 : bit
    variable value2 : bit
    variable flag, flag2 : bit
    sync c2
begin
    begin
        value := 0;
        flagb := 0;
        loop
            arbitrate write then
                value := write
            |
                op then
                case op of
                    OP_read      then flagb := 1
                | OP_set_wfz then flagb2 := 1
                end
            end;
            if flagb then
                read <- value;
                flagb := 0
            else
                if value = 0 and flagb2 = 1 then
                    sync c2;
                    flagb2 := 0
                end
            end
        end
    end
end ||!

loop
    select wait_for_zero then
        if value = 1 then
            flagb2 := 1;
            sync c2
        end
    end
end

```

---

```

    end
end

procedure real_lazy_guy (parameter name:String; output my_pos : bit;
                        sync waitfor_pos_in_front_empty) is
begin
    my_pos <- 1;
    print "lazy guy ", name, ": Somebody in front of me... I'll wait that he moves
aside...";
    sync waitfor_pos_in_front_empty;
    print "ok I can go";
    my_pos <- 0
end

procedure lazy_guy (parameter name:String; sync chan4wfz, chan5wfz;
                   output chan4op, chan5op : op1; output chan1w, chan2w : bit;
                   input chan4r, chan5r : bit) is
    variable flag : bit
begin
    flag := 0;
    if flag then
        sync chan4wfz;
        chan4op <- OP_set_wfz;
        chan5op <- OP_set_wfz;
        chan2w <- 0;
        chan4r -> flag;
        chan5r -> flag
    end;
    real_lazy_guy (name, chan1w, chan5wfz)
end

procedure polite_guy (parameter name:String; sync chan4wfz, chan5wfz;
                     output chan4op, chan5op : op1; output chan1w, chan2w : bit;
                     input chan4r, chan5r : bit) is
    variable pos4, pos5, flag : bit
begin
    flag := 0;
    if flag then
        chan1w <- 0 ; sync chan4wfz;
        chan2w <- 0 ; sync chan5wfz
    end;
    loop
        chan1w <- 1;
        chan5op<-OP_read; chan5r->pos5;
        if pos5 = 1 then
            print "polite guy ", name, ": Somebody in front of me! Let's move aside to
let him pass"
        else
            print "a: got him!";
            chan1w <- 0;
            halt
        end;
        chan2w <- 1;
        chan1w <- 0;
        chan4op<-OP_read; chan4r->pos4;
        if pos4 = 1 then
            print "polite guy ", name, ": Still somebody in front of me! let's move
back then"
        else
            print "b: got him!";
            chan2w <- 0;
            halt
        end
    end
end

```

---

---

```

    end;
    chan1w <- 1;
    chan2w <- 0
  end
end

procedure main2 is
  variable pos1,pos2,pos3,pos4,pos5 : bit
  channel chan1w, chan1r : bit
  channel chan2w, chan2r : bit
  channel chan4w, chan4r : bit
  channel chan5w, chan5r : bit
  channel chan1op : op1
  channel chan2op : op1
  channel chan4op : op1
  channel chan5op : op1
  sync chan1wfz
  sync chan2wfz
  sync chan4wfz
  sync chan5wfz
begin
  -- polite_guy (pos2, pos4) ||
  -- polite_guy (pos4, pos2)

  position (chan1w, chan1r, chan1op, chan1wfz) ||
  position (chan2w, chan2r, chan2op, chan2wfz) ||
  position (chan4w, chan4r, chan4op, chan4wfz) ||
  position (chan5w, chan5r, chan5op, chan5wfz) ||

  lazy_guy ("A", chan4wfz, chan5wfz, chan4op, chan5op,
    chan1w, chan2w, chan4r, chan5r)
  ||
  lazy_guy ("B", chan2wfz, chan1wfz, chan2op, chan1op,
    chan5w, chan4w, chan2r, chan1r)

end -- procedure main

```

## A.4 Out-of-Order Livelock Simulation Trace

This is the simulation trace generated by the out-of-order scheduler with the one-place buffer and its environment (source code in §A.2):

```

@1: 2-requp 31-requp 32-requp 3-requp 23-requp 24-requp 25-requp 26-requp
    27-requp 26-ackup 25-ackup 27-ackup 27-reqdown 25-reqdown 24-ackup 24-
    reqdown 26-reqdown 26-ackdown 25-ackdown 27-ackdown 27-requp 25-requp
    24-ackdown 24-requp 26-requp 1-requp 0-requp XXX
@32: 17-reqdown 16-ackup 16-reqdown 18-reqdown 18-ackdown 17-ackdown 17-
    requp 16-ackdown 16-requp 18-requp 19-requp 21-ackup 20-ackup 20-
    reqdown 21-reqdown 21-ackdown 20-ackdown 20-requp 21-requp 22-requp 0-
    requp XXX
@62: 17-requp 16-ackdown 16-requp 18-requp 19-requp 21-ackup 20-ackup 20-
    reqdown 21-reqdown 21-ackdown 20-ackdown 20-requp 21-requp 19-ackup
    19-reqdown 19-ackdown 13-ackup 11-ackup 27-ackup 27-reqdown 11-reqdown
    15-ackup 15-reqdown 13-reqdown 13-ackdown 11-ackdown 27-ackdown 27-
    requp 11-requp 15-ackdown 15-requp 13-requp 12-ackup 14-ackup 14-

```

---

```

reqdown 10-ackup 9-ackup 9-reqdown 10-reqdown 12-reqdown 12-ackdown
14-ackdown 14-requp 10-ackdown 9-ackdown 9-requp 10-requp 12-requp 18-
ackup 17-ackup 2-reqdown 31-reqdown 31-ackdown 30-ackdown 30-requp 2-
ackdown 0-requp XXX
@32: 6-requp 4-requp 28-ackdown 28-requp 29-requp 31-ackup 30-ackup 30-
reqdown 2-ackup 0-requp XXX
@40: 6-reqdown 4-reqdown 28-ackup 28-reqdown 29-reqdown 29-ackdown 4-
ackdown 6-ackdown 5-ackdown 5-requp 7-requp 8-requp 0-requp XXX
@62: 17-reqdown 16-ackup 16-reqdown 18-reqdown 18-ackdown 17-ackdown 2-
requp 31-requp 32-requp 32-ackup 32-reqdown 32-ackdown 29-ackup 4-
ackup 6-ackup 5-ackup 5-reqdown 7-reqdown 7-ackdown 0-requp XXX
@70: 6-reqdown 4-reqdown 28-ackup 28-reqdown 29-reqdown 29-ackdown 4-
ackdown 6-ackdown 5-ackdown 5-requp 7-requp 7-ackup 0-requp XXX
@47: 17-requp 16-ackdown 16-requp 18-requp 19-requp 21-ackup 20-ackup 20-
reqdown 21-reqdown 21-ackdown 20-ackdown 20-requp 21-requp 19-ackup
19-reqdown 19-ackdown 13-ackup 11-ackup 27-ackup 27-reqdown 11-reqdown
15-ackup 15-reqdown 13-reqdown 13-ackdown 11-ackdown 27-ackdown 27-
requp 11-requp 15-ackdown 15-requp 13-requp 12-ackup 14-ackup 14-
reqdown 10-ackup 9-ackup 9-reqdown 10-reqdown 12-reqdown 12-ackdown
14-ackdown 14-requp 10-ackdown 9-ackdown 9-requp 10-requp 12-requp 18-
ackup 17-ackup 2-reqdown 31-reqdown 31-ackdown 30-ackdown 30-requp 2-
ackdown 0-requp XXX
@55: 6-requp 4-requp 28-ackdown 28-requp 29-requp 31-ackup 30-ackup 30-
reqdown 2-ackup 0-requp XXX
@77: 6-reqdown 4-reqdown 28-ackup 28-reqdown 29-reqdown 29-ackdown 4-
ackdown 6-ackdown 5-ackdown 5-requp 7-requp 7-ackup 0-requp XXX
@85: 17-reqdown 16-ackup 16-reqdown 18-reqdown 18-ackdown 17-ackdown 2-
requp 31-requp 32-requp 32-ackup 32-reqdown 32-ackdown 29-ackup 4-
ackup 6-ackup 5-ackup 5-reqdown 7-reqdown 7-ackdown 0-requp XXX
@62: 17-requp 16-ackdown 16-requp 18-requp 19-requp 21-ackup 20-ackup 20-
reqdown 21-reqdown 21-ackdown 20-ackdown 20-requp 21-requp 19-ackup
19-reqdown 19-ackdown 13-ackup 11-ackup 27-ackup 27-reqdown 11-reqdown
15-ackup 15-reqdown 13-reqdown 13-ackdown 11-ackdown 27-ackdown 27-
requp 11-requp 15-ackdown 15-requp 13-requp 12-ackup 14-ackup 14-
reqdown 10-ackup 9-ackup 9-reqdown 10-reqdown 12-reqdown 12-ackdown
14-ackdown 14-requp 10-ackdown 9-ackdown 9-requp 10-requp 12-requp 18-
ackup 17-ackup 2-reqdown 31-reqdown 31-ackdown 30-ackdown 30-requp 2-
ackdown 0-requp XXX
@70: 6-requp 4-requp 28-ackdown 28-requp 29-requp 31-ackup 30-ackup 30-
reqdown 2-ackup 0-requp XXX
@92: 6-reqdown 4-reqdown 28-ackup 28-reqdown 29-reqdown 29-ackdown 4-
ackdown 6-ackdown 5-ackdown 5-requp 7-requp 7-ackup 0-requp XXX
@100: 17-reqdown 16-ackup 16-reqdown 18-reqdown 18-ackdown 17-ackdown 2-
requp 31-requp 32-requp 32-ackup 32-reqdown 32-ackdown 29-ackup 4-
ackup 6-ackup 5-ackup 5-reqdown 7-reqdown 7-ackdown 0-requp XXX
@77: 17-requp 16-ackdown 16-requp 18-requp 19-requp 21-ackup 20-ackup 20-
reqdown 21-reqdown 21-ackdown 20-ackdown 20-requp 21-requp 19-ackup
19-reqdown 19-ackdown 13-ackup 11-ackup 27-ackup 27-reqdown 11-reqdown
15-ackup 15-reqdown 13-reqdown 13-ackdown 11-ackdown 27-ackdown 27-
requp 11-requp 15-ackdown 15-requp 13-requp 12-ackup 14-ackup 14-
reqdown 10-ackup 9-ackup 9-reqdown 10-reqdown 12-reqdown 12-ackdown
14-ackdown 14-requp 10-ackdown 9-ackdown 9-requp 10-requp 12-requp 18-
ackup 17-ackup 2-reqdown 31-reqdown 31-ackdown 30-ackdown 30-requp 2-
ackdown 0-requp XXX
...

```

The four last blocks (blocks are going from one XXX to the next) are repeated until the end of the program (This example is not a real livelock: A loop counter stops the simulation after 1000 iterations).

# Appendix B: Breeze Handshake Components

The material contained in this appendix (notations, figures and descriptions) were originally described in chapter 3 of Bardsley's Ph.D. Thesis [5]. They have been reproduced here and updated with the permission of the author.

## B.1 Notation

Each of the handshake components described in this section is accompanied by a description of that component's behaviour. This behaviour is expressed in a notation invented by Bardsley [5], which is a modified form of van Berkel's *handshake circuit calculus* [11].

## B.2 Activation driven control components

The control components provide the events used by other components to sequence their activities. Each control component has a passive sync *activation* port and optionally a number of active sync *output activation* ports. Connecting the output activation port of a component to the activation port of another allows control trees to be constructed in which activity at the leaf ports is controlled by a single collective activation port on the root component. Activity on the output activations is enclosed within handshakes on the activation port and so leaf activity is enclosed within handshakes on the root component's activation. These components are used primarily to implement command composition in handshake circuit HDLs through activation triggered sub circuits connected to control components' output activation channels. The Balsa control components are: Continue, Halt, Loop, Sequence, Concur, Fork and WireFork.

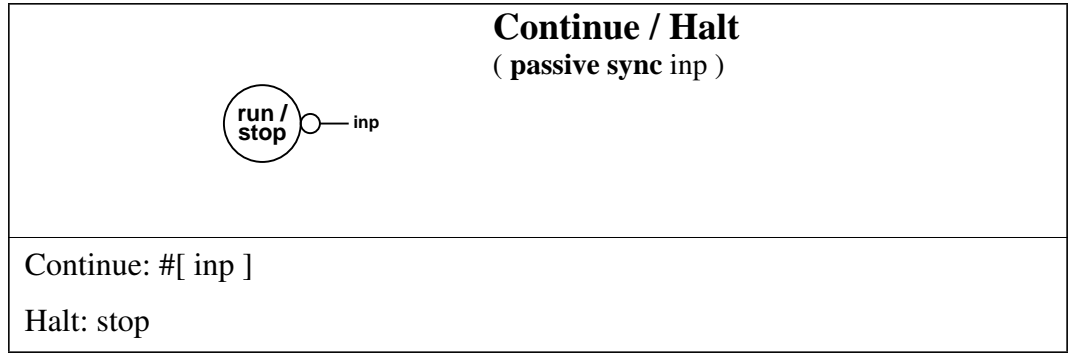


Figure B.1: Continue and Halt handshake components

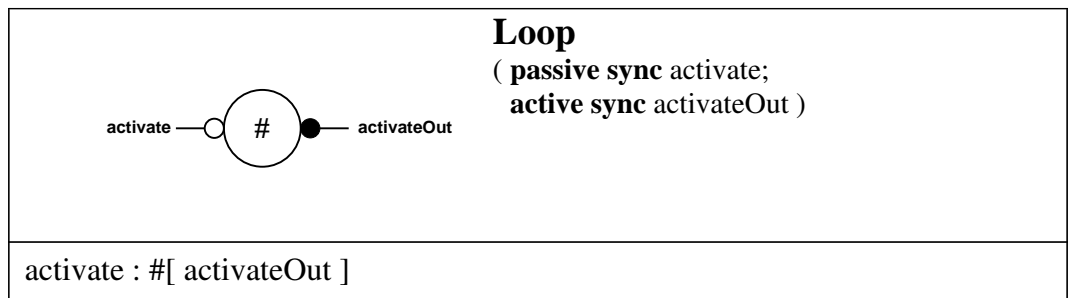


Figure B.2: Loop handshake component

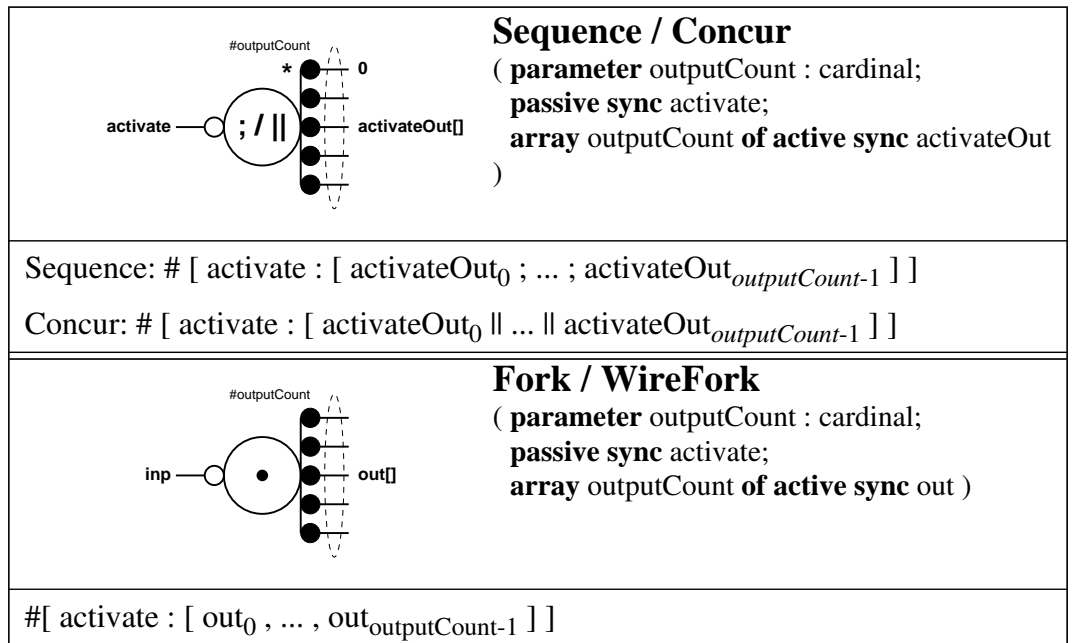


Figure B.3: Sequence, Concur, Fork and WireFork handshake components



## B.3 Control to datapath interface components

A small number of components allow control sync channels to interact with data transactions. The *transferrer* is the most common of these components, it controls the transfer of data from an active input port to an active output port under the control (and enclosure) of a passive activation port. Components with activations implementing looping and condition control operations as well as the Case component (which translates data values on a passive input activation port into activity on one of a number of active sync ports) also fall in this component class. The complete set of components is: While, Bar, Fetch, FalseVariable, Case, NullAdapt, Encode.

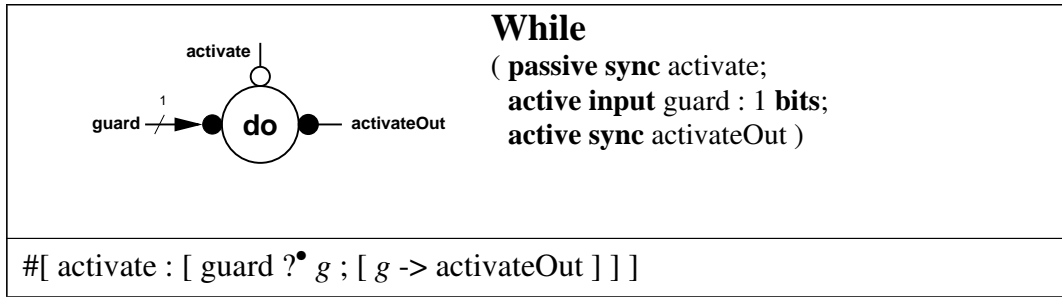


Figure B.4: While handshake component

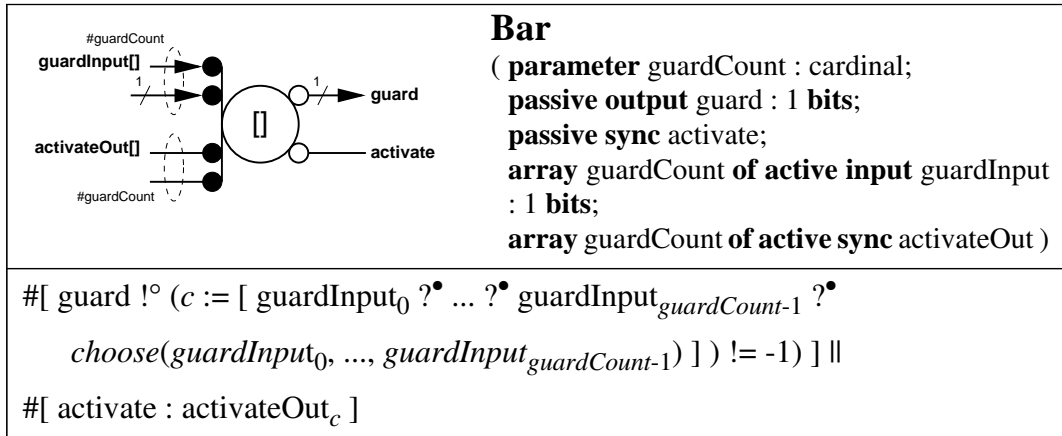


Figure B.5: Bar handshake component

## B.4 Pull datapath components

Compiled data operations (+, -, ...) in Balsa consist of a sync channel meeting a transferrer causing a result to be requested from a tree of pull datapath components implementing the

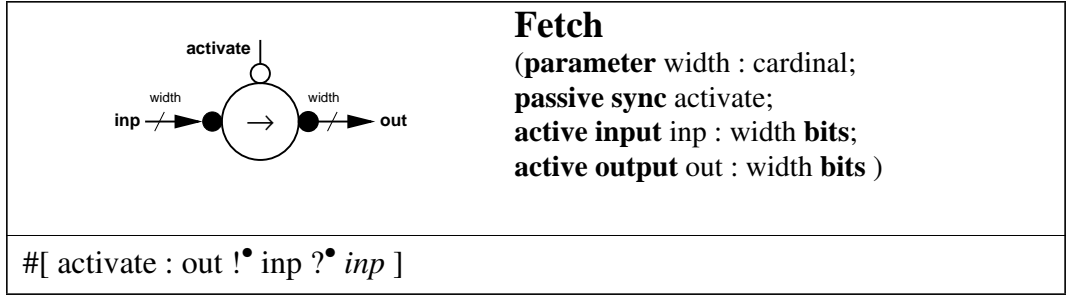


Figure B.6: Fetch handshake component

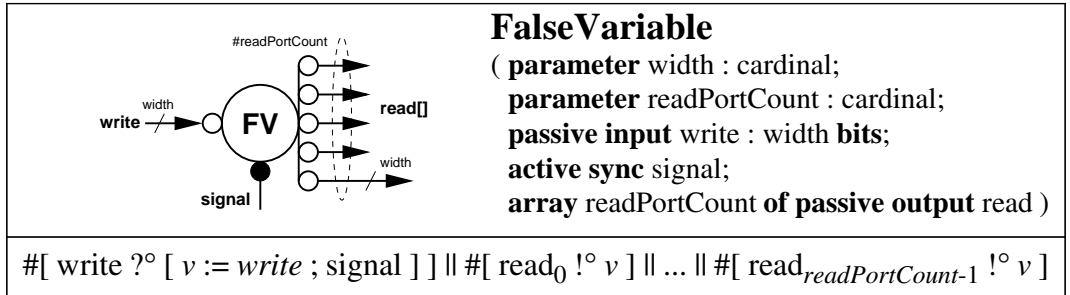


Figure B.7: FalseVariable handshake component

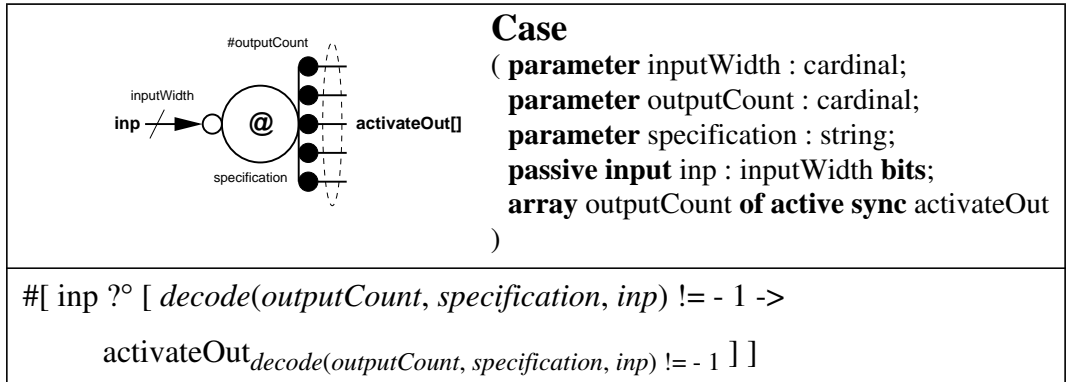


Figure B.8: Case handshake component

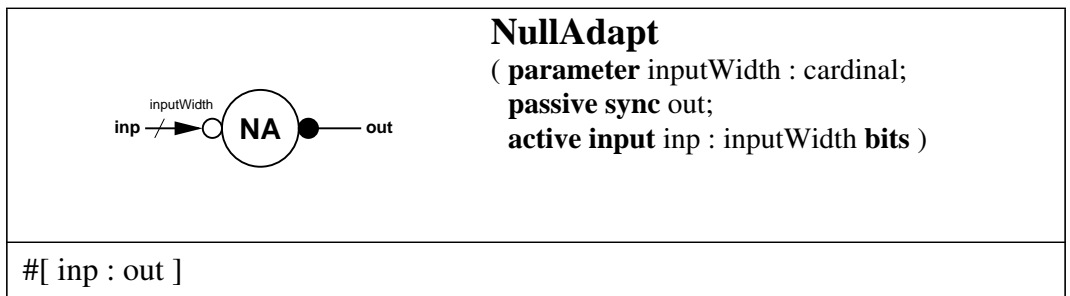


Figure B.9: NullAdapt handshake component

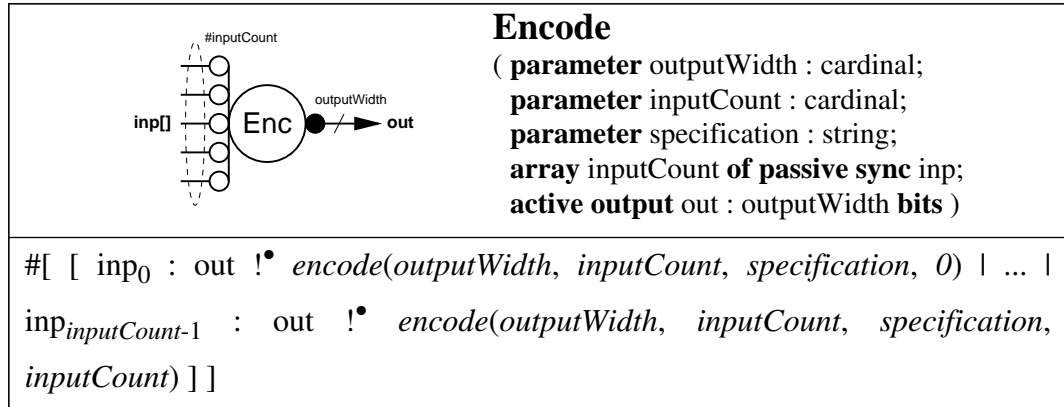


Figure B.10: Encode handshake component

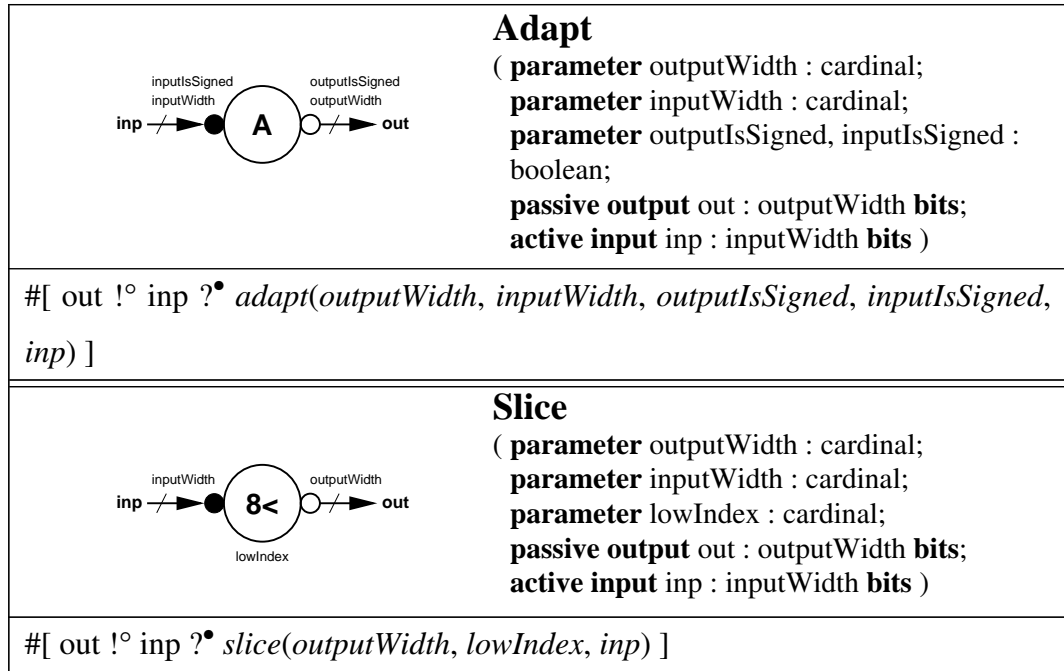


Figure B.11: Adapt and Slice handshake components

required function and pushing that result onto an output channel or into a variable (variables are the components which implement HDL level variables as latches). The pull datapath components form an activation driven tree in the same way as control components but with variables or input channels at the leaves. The activations of these components are pull ports with the incoming request flowing (and forking) towards the leaves of the tree with the result flowing (and joining) back to the root forming the result acknowledgement. The datapath components are: Adapt, Slice, Constant, Combine, CombineEqual, CaseFetch, UnaryFunc, BinaryFunc and BinaryFuncConstR.

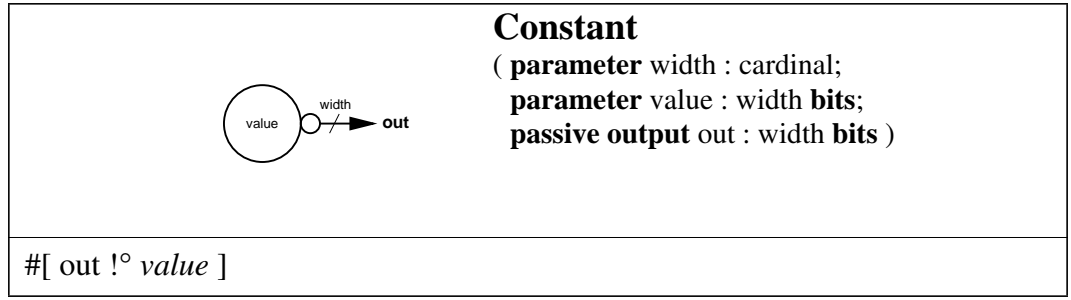


Figure B.12: Constant handshake component

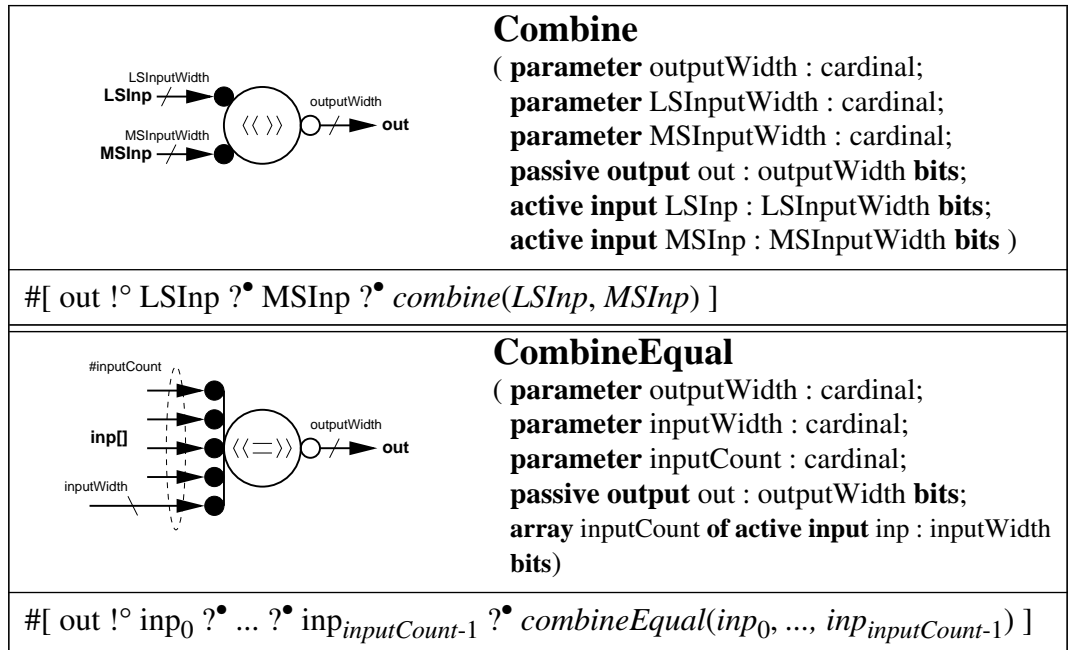


Figure B.13: Combine and CombineEqual handshake components

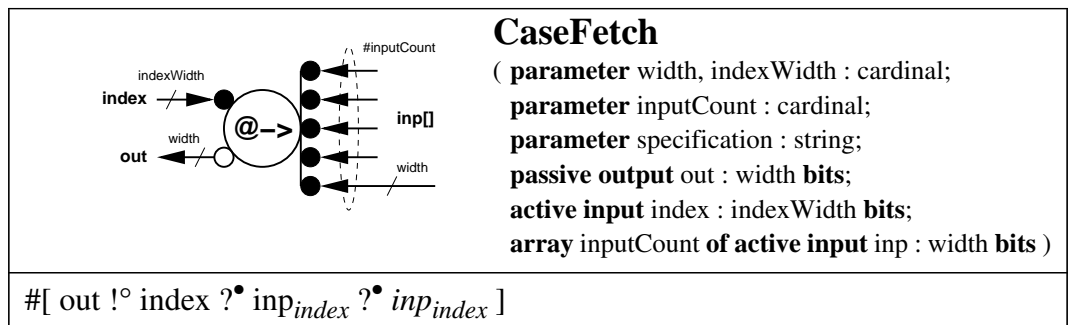


Figure B.14: CaseFetch handshake component

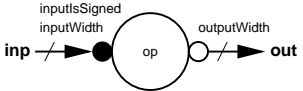
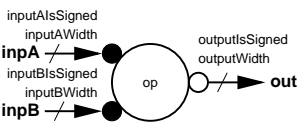
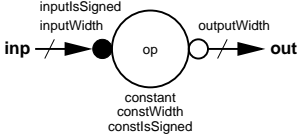
	<p><b>UnaryFunc</b>  ( <b>parameter</b> outputWidth : cardinal;  <b>parameter</b> inputWidth : cardinal;  <b>parameter</b> op : UnaryOperator;  <b>parameter</b> inputIsSigned : boolean;  <b>passive output</b> out : outputWidth <b>bits</b>;  <b>active input</b> inp : inputWidth <b>bits</b> )</p>
<p>#[ out !° inp ?° op(outputWidth, inputIsSigned, op, inp) ]</p>	
	<p><b>BinaryFunc</b>  ( <b>parameter</b> outputWidth : cardinal;  <b>parameter</b> inputAWidth : cardinal;  <b>parameter</b> inputBWidth : cardinal;  <b>parameter</b> op : BinaryOperator;  <b>parameter</b> outputIsSigned : boolean;  <b>parameter</b> inputAIsSigned : boolean;  <b>parameter</b> inputBIsSigned : boolean;  <b>passive output</b> out : outputWidth <b>bits</b>;  <b>active input</b> inpA : inputAWidth <b>bits</b>;  <b>active input</b> inpB : inputBWidth <b>bits</b> )</p>
<p>#[ out !° inpA ?° inpB ?° op(outputWidth, outputIsSigned, inputAIsSigned, inputBIsSigned, op, inpA, inpB) ]</p>	
	<p><b>BinaryFuncConstR</b>  ( <b>parameter</b> outputWidth : cardinal;  <b>parameter</b> inputWidth : cardinal;  <b>parameter</b> constWidth : cardinal;  <b>parameter</b> op : BinaryOperator;  <b>parameter</b> outputIsSigned : boolean;  <b>parameter</b> inputIsSigned : boolean;  <b>parameter</b> constIsSigned : boolean;  <b>parameter</b> constant : constWidth <b>bits</b>;  <b>passive output</b> out : outputWidth <b>bits</b>;  <b>active input</b> inp : inputWidth <b>bits</b> )</p>
<p>#[ out !° inp ?° op(outputWidth, outputIsSigned, inputIsSigned, constIsSigned, op, constant, inp) ]</p>	

Figure B.15: UnaryFunc, BinaryFunc and BinaryFuncCont handshake components

## B.5 Connection components

This class includes components used to connect together channels of the same sense, provide synchronisation between multiple channels and combine the activity of a number of channels to allow multiplexing and resource sharing. This class also includes variables as they occupy the same positions in a handshake circuit as other types of channel

connection component. Other than variables, the connection components in a handshake circuit are the only components whose presence isn't explicitly described in the HDL source for that handshake circuit. This is because they are usually present as glue to implement HDL level channels and in particular, the multicast nature of Balsa channels. The greater part of connection components implementations consist of just port-to-port wire connections. For this reason, optimising and combining connection components gives us better control of the location of troublesome wire forks which can cause wire load and drive strength management problems in implementation.

The collection of synchronising and resource sharing connection components is mostly borrowed from the Tangram component set with the addition of parameterised arrayed ports. The connection components are: ContinuePush, HaltPush, ForkPush, Call, CallMux, CallDemux, Passivator, PassivatorPush, Synch, SynchPull, SynchPush, DecisionWait, Split, Arbiter and Variable.

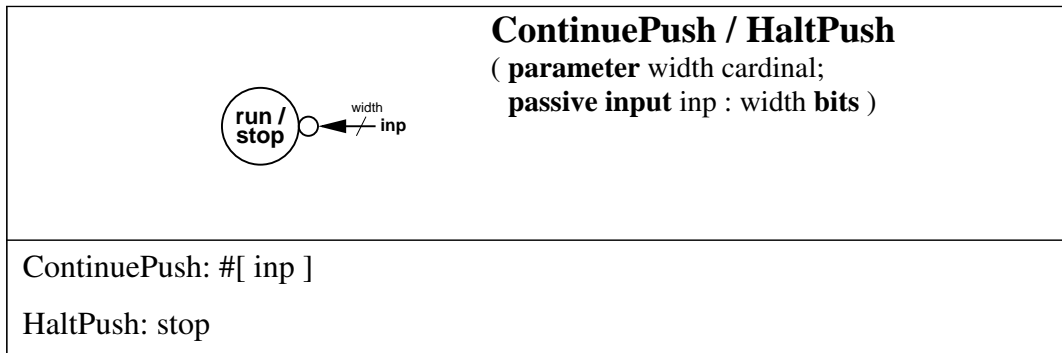


Figure B.16: ContinuePush and HaltPush handshake components

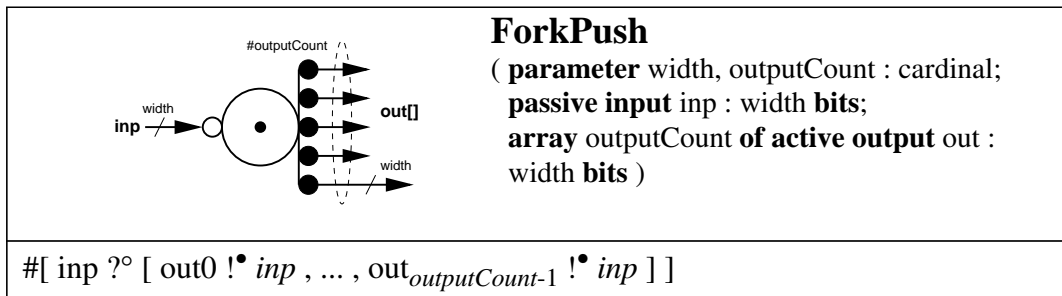


Figure B.17: ForkPush handshake component

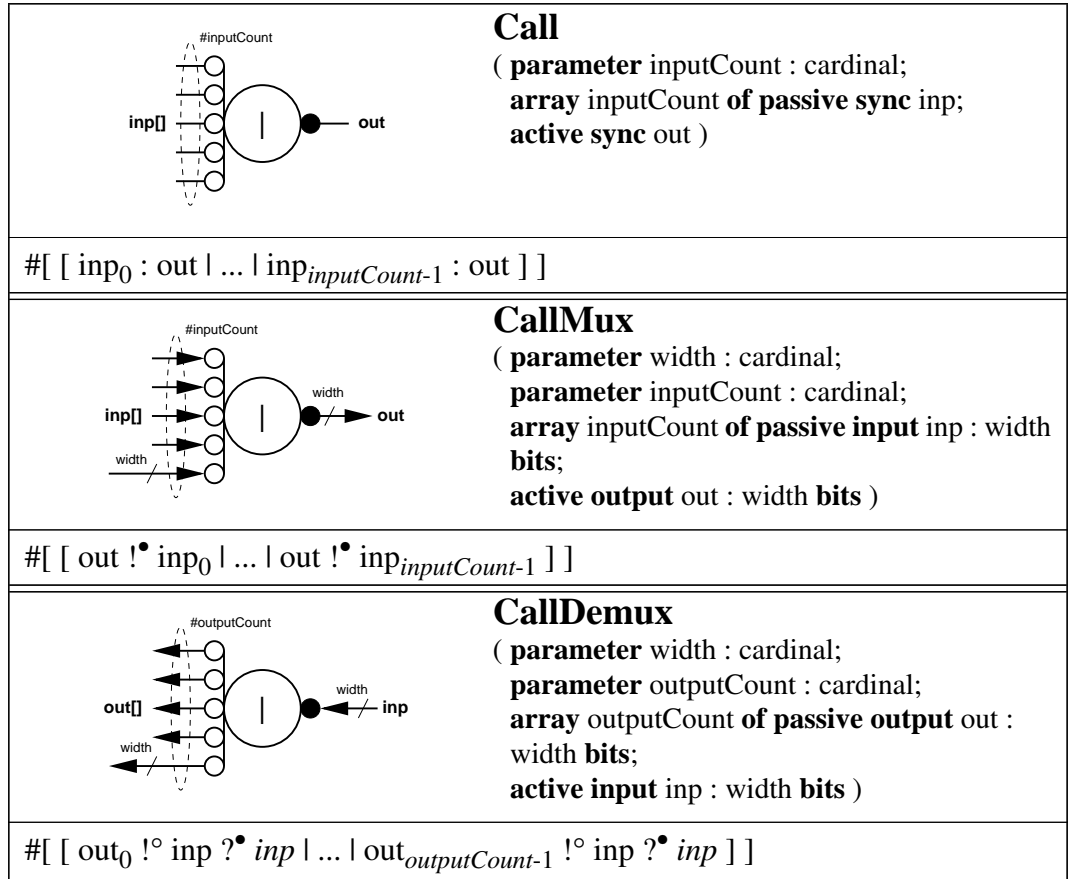


Figure B.18: Call, CallMux and CallDemux handshake components

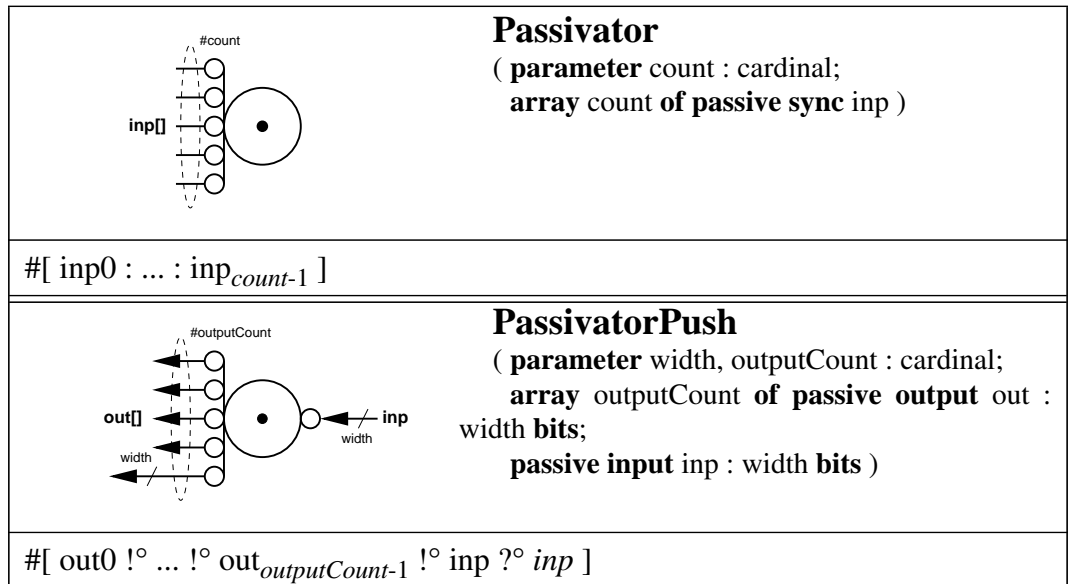


Figure B.19: Passivator and PassivatorPush handshake components

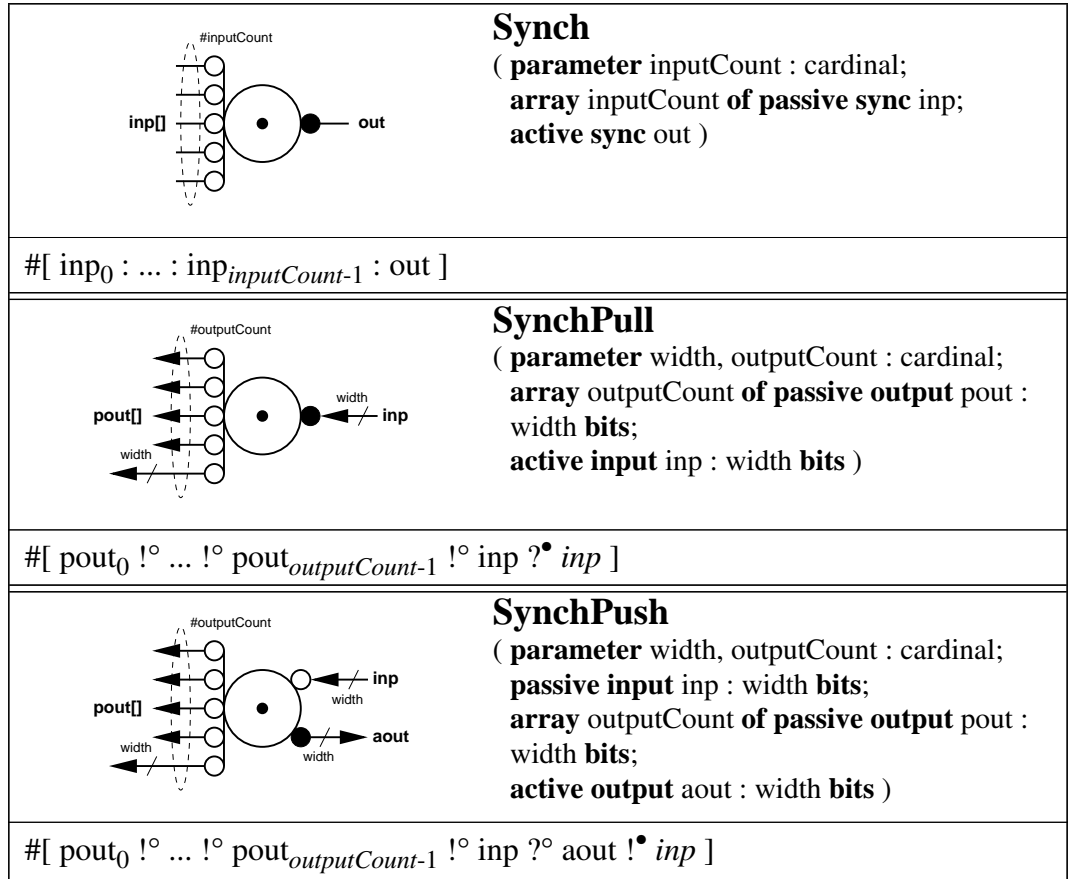


Figure B.20: Synch, SynchPull and SynchPush handshake components

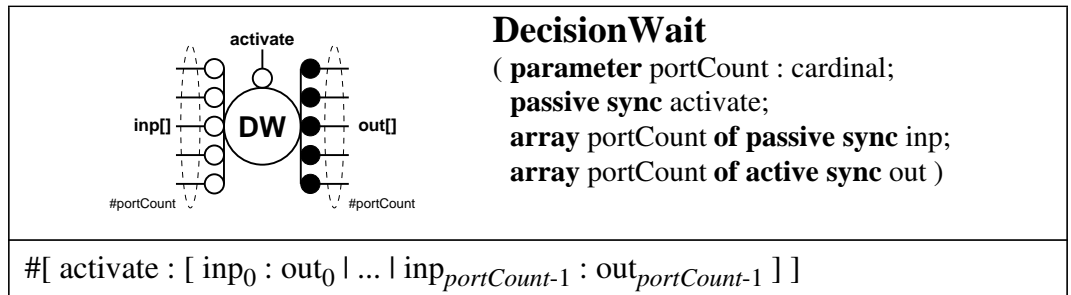


Figure B.21: DecisionWait handshake component



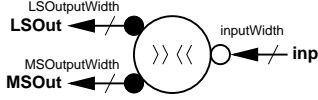
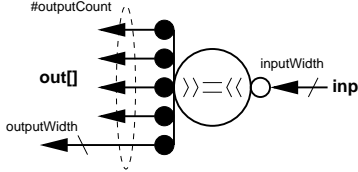
	<p><b>Split</b>  ( <b>parameter</b> inputWidth : cardinal;  <b>parameter</b> LSOutputWidth : cardinal;  <b>parameter</b> MSOutputWidth : cardinal;  <b>passive input</b> inp : inputWidth <b>bits</b>;  <b>active output</b> LSOut : LSOutputWidth <b>bits</b>;  <b>active output</b> MSOut : MSOutputWidth <b>bits</b> )</p>
<pre>#[ inp ?° [ LSOut !° bitfield(0, LSOutputWidth-1, inp)          MSOut !° bitfield(LSOutputWidth, inputWidth-1, inp) ] ]</pre>	
	<p><b>SplitEqual</b>  ( <b>parameter</b> inputWidth : cardinal;  <b>parameter</b> outputWidth : cardinal;  <b>parameter</b> outputCount : cardinal;  <b>passive input</b> inp : inputWidth <b>bits</b>;  <b>array</b> outputCount <b>of active output</b> out :  outputWidth <b>bits</b> )</p>
<pre>#[ inp ?° [ out<sub>0</sub> !° bitfield(0, outputWidth-1, inp)    ...          out<sub>outputCount-1</sub> !° bitfield(inputWidth-outputWidth, inputWidth-1, inp) ] ]</pre>	

Figure B.22: Split and SplitEqual handshake components

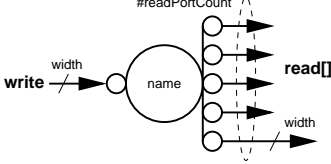
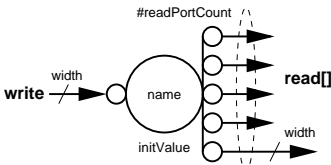
	<p><b>Variable</b>  ( <b>parameter</b> width, readPortCount : cardinal;  <b>parameter</b> name : string;  <b>passive input</b> write : width <b>bits</b>;  <b>array</b> readPortCount <b>of output</b> read : width  <b>bits</b> )</p>
<pre>#[ write ?° v := write ]    #[ read<sub>0</sub> ! v ]    ...    #[ read<sub>readPortCount-1</sub> !° v ]</pre>	
	<p><b>InitVariable</b>  ( <b>parameter</b> width, readPortCount : cardinal;  <b>parameter</b> name : string;  <b>parameter</b> initValue: width <b>bits</b>;  <b>passive input</b> write : width <b>bits</b>;  <b>array</b> readPortCount <b>of output</b> read : width  <b>bits</b> )</p>
<pre>v := initValue ; #[ write ?° v := write ]    #[ read<sub>0</sub> ! v ]    ...    #[ read<sub>readPortCount-1</sub> !° v ]</pre>	

Figure B.23: Variable and InitVariable handshake components

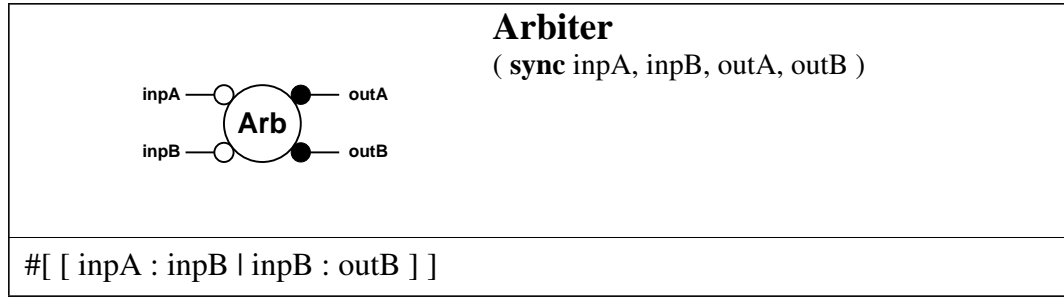


Figure B.24: Arbiter handshake component

## B.6 Breeze handshake components indexed by name

Component Name - Abbreviation - Page	Component Name - Abbreviation - Page
Adapt ..... A ..... 163	Fork ..... • ..... 160
Arbiter ..... Arb ..... 170	ForkPush ..... • ..... 166
Bar ..... [] ..... 161	Halt ..... halt ..... 160
BinaryFunc ..... <op> ..... 165	HaltPush ..... halt ..... 166
BinaryFuncConstR ..... <op> ..... 165	InitVariable ..... <var.name>... 169
Call .....   ..... 167	Loop ..... # ..... 160
CallDemux .....   ..... 167	NullAdapt ..... NA ..... 162
CallMux .....   ..... 167	Passivator ..... • ..... 167
Case ..... @ ..... 162	PassivatorPush ..... • ..... 167
CaseFetch ..... @-> ..... 164	Sequence ..... ; ..... 160
Combine ..... « » ..... 164	Slice ..... 8< ..... 163
CombineEqual ..... «=» ..... 164	Split ..... » « ..... 169
Concur .....    ..... 160	SplitEqual ..... »=« ..... 169
Constant ..... <value> ..... 164	Synch ..... • ..... 168
Continue ..... run ..... 160	SynchPull ..... • ..... 168
ContinuePush ..... run ..... 166	SynchPush ..... • ..... 168
DecisionWait ..... DW ..... 168	UnaryFunc ..... <op> ..... 165
Encode ..... Enc ..... 163	Variable ..... <var.name>... 169
FalseVariable ..... FV ..... 162	While ..... do ..... 161
Fetch ..... -> ..... 162	WireFork ..... W ..... 160

# References

- [1] Antonioli, G., di Penta, M., A Distributed Architecture for Dynamic Analyses on User-Profile Data. 8<sup>th</sup> Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04), March 2004.
- [2] Ball, T., Eick, S., Software Visualization in the Large. IEEE Computer, Vol. 29(4), pp. 34-43, April 1996.
- [3] Bainbridge W.J., Furber S., Delay Insensitive System-on-Chip Interconnect Using 1-of-4 Data Encoding. Proceedings Async 2001, pp. 118-126, IEEE Computer Society Press (ISSN 1522-8681 ISBN 0-7695-1034-4), March 2001.
- [4] Bardsley, A., Balsa: An Asynchronous Circuit Synthesis System. Master Thesis, Department of Computer Science, The University of Manchester, 1998.
- [5] Bardsley A., Implementing Balsa Handshake Circuits. Ph.D. Thesis, Department of Computer Science, The University of Manchester, 2000.
- [6] Bardsley, A., Edwards D.A., Synthesising an Asynchronous DMA Controller with Balsa. Journal of Systems Architecture 46, pp. 1309-1319, 2000.
- [7] Barringer, H., Fellows, D., Gough, G.D., Jinks, P., Marsden, B. and Williams, A., Design and Simulation in Rainbow: A Framework for Asynchronous Micropipeline Circuits. Proceeding of the European Simulation Symposium, Genoa, Italy, 1996.
- [8] di Battista, G., Eades, P., Tamassia, R., Tollis, I.G., Graph Drawing: Algorithms for the Visualization of Graphs. Prentice Hall, Upper Saddle River, NJ, 1999.
- [9] van Berkel, K., et al., The VLSI-Programming Language Tangram and its Translation into Handshake Circuits. Proceedings of the conference on European design automation, pp. 384-389, 1991.

- 
- [10] van Berkel, K., Beware the Isochronic Fork. *Integration* 13(2), pp. 103-128, June 1992.
- [11] van Berkel, K., *Handshake Circuits - An Asynchronous Architecture for VLSI Programming*. Cambridge International Series on Parallel Computers, Cambridge University Press, Cambridge, 1993.
- [12] Brandes, U., Corman, S.R., Visual Unrolling of Network Evolution and the Analysis of Dynamic Discourse. *Information Visualization* 2(1), pp. 40-50, 2003.
- [13] Brooks, R., Towards a Theory of the Comprehension of Computer Programs. *International Journal of Man-Machine Studies*, 18, pp. 543-554, 1983.
- [14] Brown, M.H., Exploring Algorithms Using Balsa-II. *IEEE Computer*, Vol. 21(5), pp. 14-36, May 1988.
- [15] Brown, M.H., Zeus: A System for Algorithm Animation and Multi-View Editing. *Proc. IEEE Workshop on Visual Languages*, pp. 4-9, October 1991.
- [16] Brunvand, E., Sproull, R.F., Translating Concurrent Programs into Delay-Insensitive Circuits. *Proc. ICCAD*, IEEE Computer Society Press, pp. 262-265, November 1989.
- [17] Brunvand, E., Translating Concurrent Communicating Programs into Asynchronous Circuits. Ph.D. Thesis, Carnegie Mellon University, 1991.
- [18] Burd, E., Chan, P., Duncan, I., Munro, M., Young, P., Improving Visual Representations of Code. University of Durham, Computer Science Technical Report, 1996.
- [19] Cadence Design Systems, Verilog-XL Reference Manual. December 1994.
- [20] Chandy, K.M., Misra, J., Deadlock Absence Proof for Networks of Communicating Processes. *Information Processing Letters*, 9, 4, pp. 185-189, November 1979.
- [21] Chelcea, T., Bardsley, A., Edwards, D.A., Nowick, S.M., A Burst-Mode Oriented Back-End for the Balsa Synthesis System. *Design, Automation and Test in Europe Conference and Exhibition*, March 2002.
- [22] Chen, K., Rajlich, V., Case Study of Feature Location Using Dependence Graph. *Proc. Int. of the 8<sup>th</sup> int. workshop on Program Comprehension*, pp. 241-249, June 2000.
-

- 
- [23] Chifosky, E.J., Cross, J.H., II, Reverse Engineering and Design Recovery: A Taxonomy. IEEE software, pp.13-17, January 1990.
- [24] Clark, W.A., Macromodular Computer Systems. AFIPS Conference Proceedings: Spring Joint Computer Conference, 1967.
- [25] Coffman, E.G., Elphick, M.J., Shoshani, A., System Deadlocks. ACM Computing Surveys, Vol. 3(2), pp. 67-78, June 1971.
- [26] Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A., Petrify: A Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers. IEICE Transactions on Informations and Systems, pp. 315-325, March 1997.
- [27] Couranz, G.R, Wann, D.F., Theoretical and Experimental Behaviour of Synchronizers Operating in the Metastable Region. IEEE Transactions on Computers 24(6), pp. 604-616, June 1975.
- [28] Eades, P., Huang, M.L., Navigating Clustered Graphs using Force-Directed Methods. Journal of Graph Algorithms and Applications: Special Issue on Selected Papers from 1998 Symp. Graph Drawing, Vol. 4(3), pp. 157-181, 2000.
- [29] Edwards, D.A., Bardsley, A., Balsa: An Asynchronous Hardware Synthesis Language. The Computer Journal, Vol. 45(1), pp. 12-18, 2002.
- [30] Edwards, D.A., Toms, W.B., Design, Automation and Test for Asynchronous Circuits and Systems. Information Society Technologies (IST) Programme, Concerted Action Thematic Network Contract, IST-1999-29119, 2nd Edition, Feb 2003.
- [31] Edwards, D.A., Theodoropoulos, G., Kwiatkowska, M., An Integrated Framework for Formal Verification and Distributed Simulation of Asynchronous Hardware Case for Support. Research Project Proposal, 2003.
- [32] Edwards, D.A., Bardsley, A., Janin, L., Toms, W., Balsa: A Tutorial Guide. version 3.4.1, Department of Computer Science, The University of Manchester, May 2004.
- [33] Eisenbarth, T., Koschke, R., Simon, D., Aiding Program Comprehension by Static and Dynamic Feature Analysis. Proceedings of the IEEE International Conference on Software Maintenance, 2001.

- 
- [34] Endecott P.B., Furber, S.B., Modelling and Simulation of Asynchronous Systems using the LARD Hardware Description Language. Proceedings of the 12<sup>th</sup> European Simulation Multiconference, Manchester, pp. 39-43, June 1998.
- [35] Erbacher, R.F., Visual Assistance for Concurrent Processing, Ph.D. Thesis, Institute for Visualization and Perception Research, University of Massachusetts, 1998.
- [36] Everitt, B., Cluster Analysis, First edition, Heinemann Educational Books Ltd, 1974. Fourth edition, ISBN 0340761199, 2001.
- [37] Favre, J.-M., Cervantes, H., Visualization of Component-based Software. Laboratoire LSR-IMAG, University of Grenoble, France, 2002.
- [38] Feng, Q., Algorithms for drawing clustered graphs. Ph.D. Thesis, Department of Computer Science and Software Engineering, The University of Newcastle, Australia, 1997.
- [39] Ferscha, A., Adaptive Time Warp Simulation of Timed Petri Nets. IEEE Transactions on Software Engineering, Vol. 25(2), March/April 1999.
- [40] Fruchterman, T.M.J., Reingold, E.M., Graph Drawing by Force-Directed Placement. Software - Practice and Experience, Vol. 21(11), pp. 1129-1164, 1991.
- [41] Fuhrer, R., Nowick, S., Theobald, M., Jha, N., Plana, L., MINIMALIST: An Environment for the Synthesis and Verification of Burst-Mode Asynchronous Machines. Technical Report CUCS-020-99, Columbia University, 1999.
- [42] Furber, S.B., Garside, J.D., Riocreux, P., Temple, S., Day, P., Liu, J., Paver, N.C., AMULET2e: An Asynchronous Embedded Controller. Proceedings of the IEEE, volume 87, number 2, pp. 243-256, February 1999.
- [43] van Gageldonk, H., Baumann, D., van Berkel, K., Gloor, D., Peeters, A., Stegmann, G., An Asynchronous Low-Power 80C51 Microcontroller. Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, pp. 96-107, 1998.
- [44] Gansner, E.R., North, S.C., An Open Graph Visualization System and its Applications to Software Engineering. Software Practice and Experience, 1999.
- [45] Gansner, E.R., Koutsofios, E., North, S.C., Dot User's Manual. AT&T Labs, February 2002.

- 
- [46] Garside, J.D., Bainbridge, W.J., Bardsley, A., Clark, D.M., Edwards, D.A., Furber, S.B., Liu, J., Lloyd, D.W., Mohammadi, S., Pepper, J.S., Petlin, O., Temple, S., Woods, J.V., AMULET3i - An Asynchronous System-on-Chip. Proceedings of Async'2000, IEEE Computer Society Press, pp. 162-175, April 2000.
- [47] Granlund, T., GNU MP: The GNU Multiple Precision Arithmetic Library. Reference manual, Free Software Foundation, Inc., 1991.
- [48] GTKWave Electronic Waveform Viewer.  
URL: <http://www.cs.man.ac.uk/apt/tools/gtkwave/>
- [49] Haagh, T.B., Hansen, T.R., Optimising a Coloured Petri Net Simulator. Master Thesis, University of Aarhus, Department of Computer Science, December 1994.
- [50] Harel, D., Koren, Y., A Fast Multi-Scale Method for Drawing Large Graphs. Graph Drawing: 8th International Symposium (GD'00), pp. 183-196, 2000.
- [51] Herman, I., Melançon, G., Marshall, M.S., Graph Visualization and Navigation in Information Visualization: A Survey. IEEE Transactions on Visualization and Computer Graphics, Vol. 6(1), pp. 24-43, 2000.
- [52] Hoare, C.A.R., Communicating Sequential Processes. Communications of the ACM 21 (8), pp. 666-677, 1978.
- [53] Huang, M.L., Eades, P., Wang, J., On-Line Animated Visualization of Huge Graphs Using a Modified Spring Algorithm. Journal of Visual Languages and Computing, 9, pp. 623-645, 1998.
- [54] Huffman, D.A., The Synthesis of Sequential Switching Circuits. J. Franklin Institute, March/April 1954.
- [55] Icarus Verilog.  
URL: <http://www.icarus.com/eda/verilog/>
- [56] IEEE Std 1364-1995, IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language - Description, 1995.
- [57] INMOS Ltd., Occam 2 Programming Manual. Series in Computer Science, Prentice-Hall International, 1989.
- [58] Janin, L., Edwards, D.A., Debugging Tools for Asynchronous Design. 10<sup>th</sup> UK Asynchronous Forum, July 2001.

- 
- [59] Janin, L., A Visualisation System for Balsa Simulations. 12<sup>th</sup> UK Asynchronous Forum, June 2002.
- [60] Janin, L., Bardsley, A., Edwards, D.A., Simulation and Visualisation of Asynchronous Circuits. International Journal of Simulation: Systems, Science & Technology, September 2003.
- [61] Jefferson, D.R., Virtual Time. ACM Transactions on Programming Languages and Systems, 7(3), pp. 404-425, July 1985.
- [62] Jensen, K., Coloured Petri Nets : Basic Concepts, Analysis Methods, and Practical Use. 2<sup>nd</sup> ed., Vol.1, Berlin : Springer, ISBN 3540609431, 1996.
- [63] Kamada, T., Yonezawa, A., A Debugging Scheme for Fine-Grain Threads on Massively Parallel Processors with a Small Amount of Log Information - Replay and Race Detection. PSLS, pp. 108-127, 1995.
- [64] Kraemer, E., Stasko, J.T., Issues in Visualization for the Comprehension of Parallel Programs. Third Workshop on Program Comprehension, IEEE Computer Society Press, pp. 116-127, 1994.
- [65] Kusalik, A.J., Prestwich, S.D., Visualizing Parallel Logic Program Execution for Performance Tuning. JICSLP, 1996.
- [66] Liu, J., Arithmetic and Control Components for an Asynchronous System. Ph.D. Thesis, Department of Computer Science, The University of Manchester, 1997.
- [67] Madalinski, A., Bystrov, A., Khomenko, V., Yakovlev, A., Visualization and resolution of coding conflicts in asynchronous circuit design. Proc. Design, Automation and Test in Europe (DATE), IEEE Computer Society Press, March 2003.
- [68] Martin, A.J., Compiling Communicating Processes into Delay-Insensitive VLSI Circuits. Distributed Computing, 1(4), pp. 226-234, 1986.
- [69] Martin, A.J., The Limitations to Delay-Insensitivity in Asynchronous Circuits. 6<sup>th</sup> MIT Conference on Advanced Research in VLSI, pp. 263-278, MIT Press, 1990.
- [70] Mentor Graphics, Modelsim SE User's Manual. 2001.
- [71] Meta-Software Inc., Hspice User's Manual. June 1987.
- [72] Meyer, S., Vanvick, A., A Verilog HDL Virtual Machine. Internal paper, Pragmatic C Software.



- 
- [73] Michaud, J., Storey, M.-A., Müller, H., Integrating Information Sources for Visualizing Java Programs. Proc. International Conference on Software Maintenance, IEEE, pp. 250-259, 2001.
- [74] Mirkin, B., Mathematical Classification and Clustering. Kluwer Academic Publishers, 1996.
- [75] Mukherjea, S., Foley, J.D., Hudson, S., Visualizing Complex Hypermedia Networks through Multiple Hierarchical Views. Human Factors in Computing Systems, CHI'95 Conference Proceedings, ACM Press, pp. 331-337, 1995.
- [76] Nellans, D., Kadaru, V.K., Brunvand, E., ARCS - An Architectural Level Communication Driven Simulator. Proceedins of the 14<sup>th</sup> ACM Great Lakes symposium on VLSI, pp. 73-77, April 2004.
- [77] Neri, D., Pautet, L., Tardieu, S., Debugging Distributed Applications with Replay Capabilities. TRI-Ada, pp. 189-195, 1997.
- [78] Neufeld, E., Kusalik, A.J., Dobrohoczki, M., Visual Metaphors for Understanding Logic Program Execution. Graphics Interface '97, pp. 114-120, 1997.
- [79] Nowick, S.M., Automatic Synthesis of Burst-Mode Asynchronous Controllers. Ph.D. Thesis, Stanford University, 1993.
- [80] Nowick, S.M., et al, MINIMALIST: An Environment for the Synthesis, Verification and Testability of Burst-Mode Asynchronous Machines. Tech. Report CUCS-020-99, Columbia University Computer Science Dept., July 1999.
- [81] de Pauw, W., Lorenz, D., Vlissides, J., Wegman, M., Execution Patterns in Object-Oriented Visualization. Proceedings Conference on Object-Oriented Technologies and Systems, pp. 219-234, 1998.
- [82] Peeters, A.M.G., Single-Rail Handshake Circuits. Ph.D. Thesis, Eindhoven University of Technology, June 1996.
- [83] Petri, C.A., Fundamentals of a Theory of Asynchronous Information Flow. Proc. of IFIP Congress 62, pp. 386-390, 1963.
- [84] Plana, L.A., Riocreux, P.A., Bainbridge, W.J., Bardsley, A., Garside, J.D., Temple, S., SPA - A Synthesisable Amulet Core for Smartcard Applications. Proceedings of Async'2002, Manchester, pp. 201-210, April 2002.

- 
- [85] Risch, J.S., Rex, D.B., Dowson, S.T., Walters, T.B., May, R.A., Moon, B.D., The STARLIGHT Information Visualization System. Proceedings of the IEEE Conference on Information Visualization, IEEE CS Press, pp. 42-49, 1997.
- [86] Roig, O., Cortadella, J., Pastor, E., Verification of Asynchronous Circuits by BDD-Based Model Checking of Petri Nets. Proceeding of the 16th International Conference on Application and Theory of Petri Nets, pp. 274-391, Turin, June 1995.
- [87] Rosenblum, L. Ya., Yakovlev, A.V., Signal graphs: from self-timed to timed ones. Proc. of the Int. Workshop on Timed Petri Nets, IEEE Computer Society Press, pp. 199-207, Turin, July 1985.
- [88] Rotem, S., Stevens, K., Ginosar, R., et al., RAPPID: An Asynchronous Instruction Length Decoder. Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, pp.60-70, 1999.
- [89] Rugaber, S., Program Comprehension for Reverse Engineering. AAAI Workshop on AI and Automated Program Understanding, pp. 106-110, July 1992.
- [90] Salz, A., Horowitz, M., IRSIM: An Incremental MOS Switch-Level Simulator. Proceedings of the 26th ACM/IEEE conference on Design automation, pp.173-178, June 1989.
- [91] Seitz, C., System Timing , Chapter 7 of Introduction to VLSI Systems by Mead, C, Conway, L., Addison Wesley, Second Edition, 1980.
- [92] Shimomura, T., Isoda, S., VIPS: A Visual Debugger for List Structures. Proc. Computer Software and Applications Software, pp. 530-537, 1990.
- [93] Sloot, P.M.A., Modelling and Simulation, Proceedings of the 1994 CERN School of Computing, CERN, 1994.
- [94] Small Computer System Interface (SCSI), American National Standards Institute, 1986.
- [95] Smith, R., Ligthart, M., High-Level Design for Asynchronous Logic. ASP-DAC, February 2001.
- [96] Sparsø, J., Furber, S., Principles of Asynchronous Circuit Design: A Systems Perspective. Kluwer Academic Publishers, 2001.

- 
- [97] Stasko, J.T., Tango: A Framework and System for Algorithm Animation. IEEE Computer, Vol. 23(9), pp. 27-39, September 1990.
- [98] Stevens, K.S., Practical Verification and Synthesis of Low Latency Asynchronous Systems. Ph.D. Thesis, University of Calgary, Alberta, Canada, September 1994.
- [99] Stevens, K., et al., An Asynchronous Instruction Length Decoder. IEEE Journal of Solid-State Circuits, 2001.
- [100] Storey, M.-A., A Cognitive Framework For Describing and Evaluating Software Exploration Tools. PhD Thesis, Computing Science, Simon Fraser University, Canada, 1998.
- [101] Stucki, M.J., Ornstein, S.M., Clark, W.A., Logical Design of Macromodules. AFIPS Spring Joint Computer Conference, pp. 357-364, 1967.
- [102] Sugiyama, K., Tagawa, S., Toda, M., Methods for Visual Understanding of Hierarchical System Structures. IEEE Trans. Syst. Man Cybern., 11(2), pp. 109-125, 1981.
- [103] Sutherland, I. E., Micropipelines. Communications of the ACM 32(6), pp. 720-738, June 1989.
- [104] Sutherland, I.E., Flashback Simulation. Research Report SunLab 93:0285, Sun Microsystems Laboratories, Inc., August 1993.
- [105] Theobald, M., Nowick, S.M., Transformations for the Synthesis and Optimization of Asynchronous Distributed Control. Proceedings of the 38<sup>th</sup> conference on Design automation, 2001.
- [106] Theodoropoulos, G.K., Tsakogiannis, G.K., Woods, J.V., Occam: An Asynchronous Hardware Description Language. 23<sup>rd</sup> EUROMICRO Conference, 1997.
- [107] Tilley, S.R., Paul, S., Smith, D.B., Towards a Framework for Program Understanding. WPC'96: 4<sup>th</sup> workshop on program comprehension, Berlin, Germany, pp.19-28, march 1996.
- [108] Thistlewaite, P., Johnson, C., Towards Debugging and Analysis Tools for Kilo-Processor Computers. Fujitsu Scientific and Technical Journal, Vol. 29(1), pp. 32-40, 1993.

- 
- [109] Theseus Logic Inc.  
URL: <http://www.theseus.com>
- [110] Tuchman, A., Jablonowski, D., Cybenko, G., Run-Time Visualization of Program Data. Proc. IEEE Conference on Visualization, pp. 255-261, October 1991.
- [111] Verhoeff, T., Encyclopedia of Delay-Insensitive Systems. Eindhoven University of Technology, The Netherlands, 1995-1998.  
URL: <http://edis.win.tue.nl/edis.html>
- [112] Visual STG Lab.  
URL: <http://vstgl.sourceforge.net/>
- [113] Wilde, N., Scully, M.C., Software Reconnaissance: Mapping Program Features to Code. Software maintenance: Research and Practice, Vol. 7, pp. 49-62, 1995.
- [114] Wittenburg, K., Sigman, E., Visual Focusing and Transition Techniques in a Treeviewer for Web Information Access. Proc. Visual Languages '97, Capri, Italy, pp. 20-27, Sept 1997.
- [115] Wong, C.G., Martin, A.J., High-Level Synthesis of Asynchronous Systems by Data-Driven Decomposition. Proc. 40<sup>th</sup> Design Automation Conference (DAC), June 2003.
- [116] Yun, K., Dill, D., Automatic Synthesis of 3D Asynchronous Finite-State Machines. ICCAD, 1992.
- [117] Zeller, A., Lutkehaus, D., DDD - A Free Graphical Front-End for UNIX Debuggers. SIGPLAN Notices, Vol. 31(1), pp. 22-27, 1996.
- [118] Zuberek, W.R., Event-Driven Simulation of Timed Petri Net Models. 33<sup>rd</sup> Annual Simulation Symposium, pp. 91-98, April 2000.